Elmer FEM
open source multiphysical simulation software

Elmer non-GUI Tutorials

CSC – IT Center for Science

January 20, 2026

Elmer non-GUI Tutorials

About this document

The Elmer non-GUI Tutorials is part of the documentation of Elmer finite element software. Elmer non-GUI Tutorials gives examples on the use of Elmer in different fields of continuum physics. Also coupled problems are included.

These tutorials were written before ElmerGUI was written (i.e. before year 2008) and they don't make use of it. See the Elmer GUI Tutorials for information how to set up the cases using ElmerGUI. These cases are not really that well fitted as tutorials for beginners but they may be useful for power users who want to understand more deeply how models in Elmer are set up.

The present manual corresponds to Elmer software version 26.1.

Latest documentations and program versions of Elmer are available (or links are provided) at `http://www.csc.fi/elmer`.

Copyright information

Contents

Tutorial 1

Eigenvalue analysis of an elastic beam

Directory: ElasticEigenValues
Solvers: StressSolve, EigenSolve
Tools: ElmerGrid,Editor
Dimensions: 3D, Steady-state

Case definition

A homogenous, elastic silicon beam of dimensions 1 m length, 0.1 m height and 0.2 m width is supported on its both ends (boundaries 1 and 2). A beam has the density 2330 kg/m^3, Poisson ratio 0.3 and Young's modulus 10^{11} N/m^2. The problem is to calculate the eigenvalues of the beam. Mathematically the equation to be solved is

$$-\rho\omega^2\phi = \nabla \cdot \tau(\phi)$$

where ρ is the density, ω^2 is the eigenvalue, ω is the angular frequency, ϕ is the corresponding vibration mode and τ is the stress tensor.

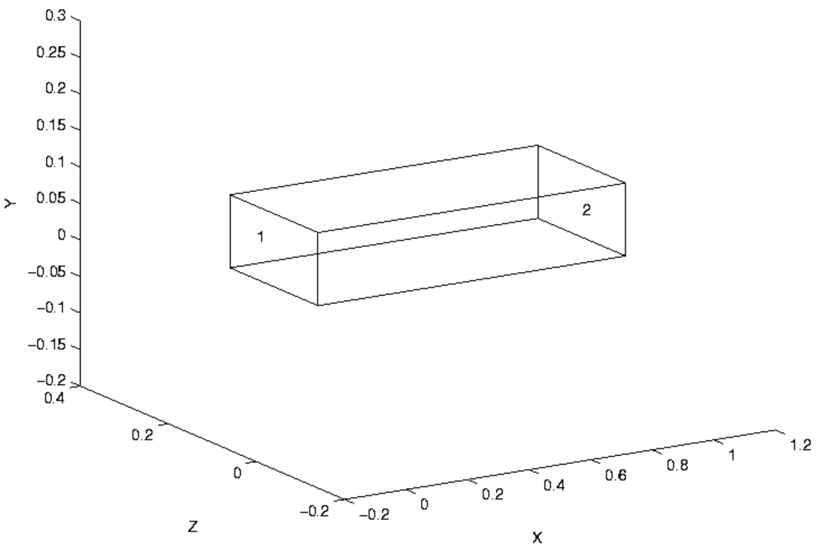

Figure 1.1: Beam.

Solution procedure

The mesh has been created by using Gambit software and it consists of 2500 elements. The mesh can be converted to Elmer format with ElmerGrid with the command

```
ElmerGrid 7 2 mesh.FDNEUT
```

This command creates the directory which contains the Elmer mesh files.

```
Header
  Mesh DB "." "mesh"
  Include Path ""
  Results Directory ""
End
```

A steady-state three-dimensional analysis is defined in the simulation section.

```
Simulation
  Coordinate System = "Cartesian 3D"
  Coordinate Mapping(3) = 1 2 3
  Simulation Type = "Steady State"
  Steady State Max Iterations = 1
  Solver Input File = "eigen_values.sif"
  Output File = "eigen_values.dat"
  Post File = "eigen_values.vtu"
End
```

The geometry of the problem is simple and it includes only one body and material.

```
Body 1
  Equation = 1
  Material = 1
End

Material 1
  Youngs Modulus = 100e9
  Poisson Ratio = 0.3
  Density = 2330
End
```

The problem is solved according to linear elastic theory and due to that stress analysis is set to true.

```
Equation 1
  Stress Analysis = True
End
```

In the solver section `Stress Analysis` is selected. In addition, the value of the keyword `Eigen Analysis` has to be set to true. The keyword `Eigen System Values` defines the number of the computed eigenvalues. The problem also is possible to solve with iterative solver but we have used direct solver in this example.

```
Solver 1
  Equation = "Stress Analysis"
  Eigen Analysis = Logical True
  Eigen System Values = Integer 5
  Linear System Solver = "direct"
  Variable = "Displacement"
  Variable Dofs = 3
  Linear System Iterative Method = "BiCGStab"
  Linear System Max Iterations = 1000
  Linear System Convergence Tolerance = 1.0e-08
  Linear System Abort Not Converged = True
  Linear System Preconditioning = "ILU0"
  Linear System Residual Output = 1
  Steady State Convergence Tolerance = 1.0e-05
  Nonlinear System Convergence Tolerance = 1.0e-05
  Nonlinear System Max Iterations = 1
  Nonlinear System Newton After Iterations = 3
  Nonlinear System Newton After Tolerance = 1.0e-02
  Nonlinear System Relaxation Factor = 1
  Linear System Precondition Recompute = 1
End
```

The beam is supported on its both ends and therefore displacements are set to zero in all directions.

```
Boundary Condition 1
  Target Boundaries(1) = 1
  Displacement 1 = 0
  Displacement 2 = 0
  Displacement 3 = 0
End

Boundary Condition 2
  Target Boundaries(1) = 2
  Displacement 1 = 0
  Displacement 2 = 0
  Displacement 3 = 0
End
```

After that, the problem is ready to solve.

An anisotropic model

The same problem can also be solved as an anisotropic problem which causes a couple of changes in the sif-file. First, it is reasonable to rename the files in the simulation section

```
Solver Input File = "eigen_values_aniso.sif"
Output File = "eigen_values_aniso.dat"
Post File = "eigen_values_aniso.vtu"
```

For anisotropic material Young's modulus has to be redefined as a matrix. In this case the matrix is defined as follows

```
Youngs Modulus
Size 6 6
    Real  200e9  60e9   60e9   0      0      0
          60e9   200e9  200e9  0      0      0
          60e9   60e9   200e9  0      0      0
          0      0      0      80e9   0      0
          0      0      0      0      80e9   0
          0      0      0      0      0      80e9
    End
```

No more changes are needed in the sif-file.

Results

Both the eigenvalues of the isotropic and the eigenvalues of the anisotropic model are shown below in Elmer outputs. Figure 1.2 presents the computed eigenvectors of the beam with the isotropic model. The formula $\omega = 2\pi f$ have been used in calculating frequencies (f) (Table 1.1). According to the results the anisotropic model yielded greater eigenvalues with these values of Young's modulus.

```
EigenSolve: Computed Eigen Values:
EigenSolve: --------------------------------
EigenSolve:            1        (16737546.4275755,0.00000000000000D+000)
EigenSolve:            2        (48175589.4544061,0.00000000000000D+000)
EigenSolve:            3        (99674749.0526558,0.00000000000000D+000)
EigenSolve:            4        (110392974.959463,0.00000000000000D+000)
EigenSolve:            5        (253947166.278411,0.00000000000000D+000)
```

Isotropic model.

```
EigenSolve: Computed Eigen Values:
EigenSolve: --------------------------------
EigenSolve:            1        (29608629.8775828,0.00000000000000D+000)
EigenSolve:            2        (88782964.0905879,0.00000000000000D+000)
EigenSolve:            3        (198583949.415515,0.00000000000000D+000)
EigenSolve:            4        (205085884.544046,0.00000000000000D+000)
EigenSolve:            5        (480903841.387323,0.00000000000000D+000)
```

Anisotropic model.

Table 1.1: Computed frequencies.

step	isotropic	anisotropic
1	651.127 Hz	866.023 Hz
2	1104.673 Hz	1499.633 Hz
3	1588.959 Hz	2242.809 Hz
4	1672.210 Hz	2279.229 Hz
5	2536.249 Hz	3490.191 Hz

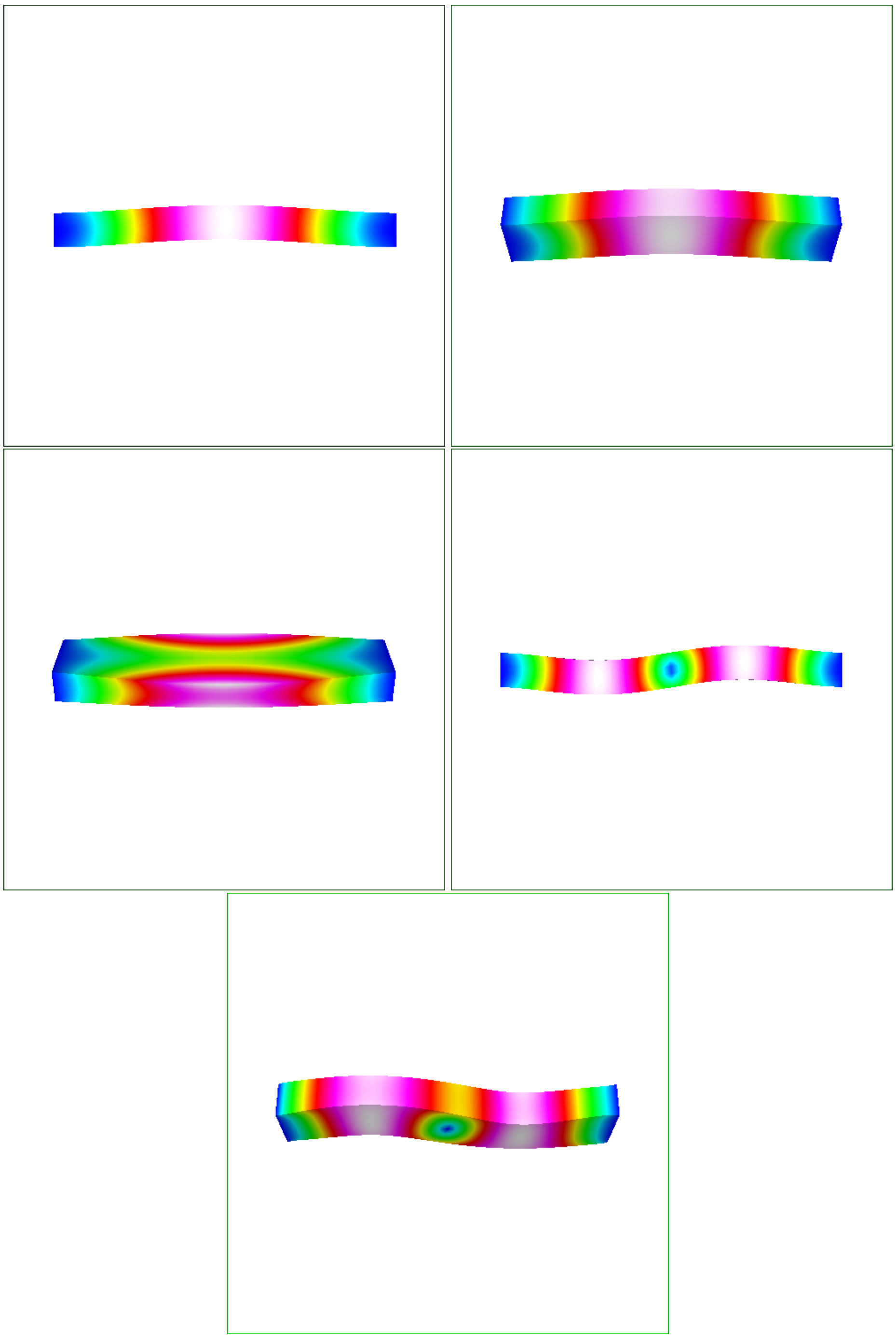

Figure 1.2: Eigenvectors

Tutorial 2

Flow through a hole – determining the acoustic impedance

Directory: FlowResistance
Solvers: FlowSolve
Tools: ElmerGrid, editor
Dimensions: 3D, Steady-state

Note: This test case is available as consistency tests `FlowResNoslip` and `FlowResSlip`. This may be outdated in parts. For example, it is not necessary to use any special unit system, and also the computation of forces is now more accurate.

Case definition

The problem at hand consists of finding the resistance that a fluid faces as it is forced to flow through a hole. The flow resistance is stated by the ratio of pressure drop over the hole and the input velocity. In micro-system modeling, the hole resistance is often needed to analyse the gas damping effect in perforated structures. Here, the contribution of the holes is homogenized over the perforated structure based on a single hole resistance. For homogenization in Elmer, the specific acoustic impedance is used to represent the flow resistance. Specific acoustic impedance z_h is defined as

$$z_h = \frac{p}{v} = \frac{F}{vA_h}, \tag{2.1}$$

where F is the net force due to gas on the moving surface, v is the velocity of the gas on the same surface, and A_h is the area of the moving surface. The calculation is best performed in a unit cell of the geometry.

In order to the homogenization to be possible, the dependence of input velocity and the net force should be linear. Further, there should not be a phase change between these two quantities. These conditions are satisfied when the flow is incompressible. In a linear case, the fluid flow can be treated with the linear form of Navier-Stokes equations called the Stokes equation

$$\rho\frac{\partial\vec{u}}{\partial t} - \nabla\cdot(2\eta\bar{\bar{\varepsilon}}) + \nabla p = \rho\vec{f}, \tag{2.2}$$

where $\vec{u}$ is the unknown velocity field, p is the pressure, η is the viscosity of the fluid, $\rho\vec{f}$ is a body force and $\bar{\bar{\varepsilon}}$ is the linearised strain tensor. Note, that the stationary version of the above equation can be used in homogenization calculations.

The condition for Stokes equation to apply is that the Reynolds number Re of the problem should be small

$$Re = \frac{\rho UL}{\eta}, \tag{2.3}$$

where ρ is density of the fluid and U and L are, respectively, the velocity and length scales of the problem.

The issue of compressibility is more difficult to answer. A classical condition for the compressibility is that the Mach number Ma of the problem should be small

$$Ma = \frac{U}{a} < 0.3, \tag{2.4}$$

where a is the speed of sound in the gas in operating conditions and the value 0.3 is often stated limit for a small Mach number (actually, the condition is that Ma^2 has to be small). Also the frequency and amplitude of the oscillations of the system have an effect on the validity of the linearity and incompressibility assumptions, since they affect the velocity scale of the problem.

However, also other factors have an effect on the compressibility of the gas. In micro-systems, the viscous effects on pressure, or even temperature changes, can affect the density of the gas. A condition for viscous pressure changes is that Ma^2/Re has to be small, and for temperature, in addition, that the Prandtl number Pr may not be too large

$$Pr = \frac{\eta c_p}{k}, \tag{2.5}$$

where c_p is the heat capacity (*ie.* specific heat) in constant pressure and k is the thermal conductivity.

The conditions listed here for the flow to be approximately incompressible are only an overview and the validity of incompressibility assumption should be considered in each case separately. In micro-systems, refer for example to the article M. Gad-el-Hak, J. Fluids Eng., 121, 5–33, 1999. Additionally, it is advisable to perform numerical checks on the issue.

One final point on the applicability of the Stokes (or Navier-Stokes) equations is the effect of gas rarefaction. If the dimensions of the problem are very small the continuity assumption may not be valid anymore. The importance of the gas rarefaction effects are given by the Knudsen number Kn

$$Kn = \frac{\mathcal{L}}{L}, \tag{2.6}$$

where $\mathcal{L}$ is the mean free path of the gas molecules. The mean free path depends inversely on ambient pressure, which has to take into account in stating the Knudsen number. For Knudsen numbers close to and less than 1, slip boundary conditions should be used.

To summarize, the motivation of this tutorial is to perform a linear incompressible simulation of fluid flowing through a hole. The wake for the flow is a constant velocity boundary condition for a boundary before the hole. On the same boundary, the force caused by the fluid is recorded. These two quantities can then be used to determine the specific acoustic impedance of a single hole. The constant velocity boundary condition may be interpreted as describing a moving wall with small displacement. In this particular tutorial, a symmetrical quadrant of a square-shaped hole is used.

Solution procedure

The solution for the problem is found by solving iteratively the Stokes equation. Nonlinear iterations are not needed, since the problem is linear.

The computational mesh should include enough free space after the hole so that any artificial effects due to the boundaries of the mesh are avoided. In this tutorial, the geometry is created and meshed using the ElmerGrid program by the command `elmergrid 1 2 hole.grd`. The default mesh consists of about 12000 nodes and 10500 eight-noded hexahedrons.

The header section of solver input file includes only the location of the mesh files.

```
Header
  Mesh DB "." "hole"
End
```

In the simulation section, a steady-state three-dimensional analysis is defined.

```
Simulation
  Coordinate System = Cartesian 3D
  Simulation Type = Steady State
  Steady State Max Iterations = 1
  Output File = "flow.result"
  Post File = "flow.vtu"
End
```

The geometry contains only one body and no body forces or initial conditions are present. The body section reads thus as follows.

```
Body 1
  Equation = 1
  Material = 1
End
```

For solving the flow patterns the Navier-Stokes solver is used but the nonlinearity through convection is switched off in the equation block. Also, solvers for the fluidic force and saving data are enabled.

```
Equation 1
  Active Solvers(3) = Integer 1 2 3
  NS Convect = False
End
```

Just a single iteration of the Navier-Stokes solver is needed, since the equation is linear. This can be verified by switching the number of nonlinear iterations to a value more than one, and observing the change in solution between iteration steps.

```
Solver 1
   Equation = Navier-Stokes
   Variable = Flow Solution
   Variable DOFs = 3
   Linear System Solver = Iterative
   Linear System Iterative Method = BiCGStab
   Linear System Preconditioning = ILU0
   Linear System Max Iterations = 200
   Linear System Convergence Tolerance = 1.0e-08
   Stabilize = True
   Nonlinear System Convergence Tolerance = 1.0e-05
   Nonlinear System Max Iterations = 1
   Nonlinear System Newton After Iterations = 3
   Nonlinear System Newton After Tolerance = 1.0e-08
   Nonlinear System Relaxation Factor = 1.0
   Steady State Convergence Tolerance = 1.0e-05
End
```

The fluidic force solver needs to be run only once, after the flow solution is finished. With the keyword `Calculate Viscous Force` it is possible to define whether the viscous forces of the fluid are included in the force or not. If this is set to false, only the pressure integral is calculated.

```
Solver 2
  Exec Solver = After All
  Equation = Fluidic Force
  Procedure  ="FluidicForce" "ForceCompute"
  Calculate Viscous Force = True
End
```

The final solver is used to save data from the analysis. With the following definitions, the input velocity and the net force on the input boundary as well as the area of the boundary are written into a file called `flowdata.dat`.

```
Solver 3
  Exec Solver = After All
  Equation = SaveScalars
  Procedure = "SaveData" "SaveScalars"
  Filename = "flowdata.dat"
  Save Variable 1 = Velocity 3
  Save Coordinates(1,2) = 0.0 0.0
End
```

The fluid is defined to be air. Note the Elmer MEMS units used.

```
Material 1
  Name = Air
  Density = 1.293e-12
  Viscosity = 1.67e-5
End
```

Finally, the boundary conditions. BC 1 defines the input boundary, where also the fluidic force is calculated. BCs 2 and 4 define the symmetry boundaries, BC 3 defines the no-slip conditions for the walls, and BC 5 defines an open boundary.

```
Boundary Condition 1
  Target Boundaries = 4
   Velocity 1 = 0.0
   Velocity 2 = 0.0
   Velocity 3 = 1.0e3
   Calculate Fluidic Force = True
End

Boundary Condition 2
  Target Boundaries(2) = 8 10
   Velocity 2 = 0.0
End

Boundary Condition 3
  Target Boundaries(4) = 1 2 3 7
   Velocity 1 = 0.0
   Velocity 2 = 0.0
   Velocity 3 = 0.0
End

Boundary Condition 4
  Target Boundaries(2) = 6 9
   Velocity 1 = 0.0
End

Boundary Condition 5
  Target Boundaries = 5
  Pressure = 0.0
End
```

Slip boundary conditions

The same simulation can also be performed using slip boundary conditions. These are appropriate, as stated in introduction, when the Knudsen number is between 10^{-3} and 1. The slip boundary condition implemented in Elmer is of first order

$$S \cdot \vec{u} = \overline{\overline{\sigma}} \cdot \vec{n}, \tag{2.7}$$

where S is a vector containing the slip coefficients s_i for each velocity component, μ is the viscosity, and $\overline{\overline{\sigma}}$ is the stress tensor. For Newtonian fluids and for tangential directions of the boundary this gives

$$s_i u_i = \mu \frac{\partial u_i}{\partial n}, \tag{2.8}$$

where s_i and u_i refer to the same tangential component of the slip coefficient and the flow velocity.

The value of the slip coefficient is related to the mean free path of the gas molecules λ. For example, Maxwell's first order slip boundary condition may be used (as in *e.g.* A. Beskok, *Num. Heat Transfer,* B, 40, 451–471, 2001):

$$u_i = \frac{2-\sigma_v}{\sigma_v}\lambda \frac{\partial u_i}{\partial n}, \tag{2.9}$$

where σ_v is the tangential momentum accommodation coefficient, which models the momentum exchange of gas molecules and the surface. The accommodation coefficient is dependent on the gas and on the surface, and recent measurements give a result of $\sigma_v \simeq 0.80$ for various monoatomic gases such as Argon in contact with prime Silicon crystal.

The slip coefficient of Elmer can finally be written as

$$s_i = \frac{\mu}{\lambda}\frac{\sigma_v}{2-\sigma_v}. \tag{2.10}$$

The mean free path is defined as

$$\lambda = \frac{\mu}{\rho}\sqrt{\frac{\pi M}{2RT}}, \tag{2.11}$$

where ρ is density, M is the molar mass, T is the temperature, and $R = 8.3145$ J/mol K is the molar gas constant.

In the Elmer analysis, only a few changes in the sif-file are needed to make the slip conditions active. The flow force boundary conditions have to be turned on and the numerical value of the slip coefficient has to be defined on each boundary (here s =2e-4 is used for air). Further below is a list of the Boundary Condition blocks. Note that there are more BCs than in the no-slip simulation, since a separate condition is needed for surfaces oriented differently in space.

Generally, a normal-tangential orientation scheme for the boundary conditions are needed, since the surfaces are not necessarily having a normal vector pointing in one of the coordinate directions. This would be done for each such boundary by the line

```
  Normal-Tangential Velocity = True
```

after which the Velocity component 1 points to the normal direction and the other components to the tangential directions.

```
! Definitions for slip boundary conditions:
Boundary Condition 1
  Target Boundaries = 4
   Flow Force BC = True
   Slip Coefficient 1 = 2e-4
   Slip Coefficient 2 = 2e-4
   Velocity 3 = 2.0e3
   Calculate Fluidic Force = True
End

Boundary Condition 2
  Target Boundaries(2) = 8 10
   Velocity 2 = 0.0
End

Boundary Condition 3
```

```
  Target Boundaries(2) = 2 3
   Flow Force BC = True
   Velocity 3 = 0.0
   Slip Coefficient 1 = 2e-4
   Slip Coefficient 2 = 2e-4
End

Boundary Condition 4
  Target Boundaries(2) = 6 9
   Velocity 1 = 0.0
End

Boundary Condition 5
  Target Boundaries = 5
  Pressure = 0.0
End

Boundary Condition 6
  Target Boundaries = 1
   Flow Force BC = True
   Velocity 1 = 0.0
   Slip Coefficient 2 = 2e-4
   Slip Coefficient 3 = 2e-4
End

Boundary Condition 7
  Target Boundaries = 7
   Flow Force BC = True
   Velocity 2 = 0.0
   Slip Coefficient 1 = 2e-4
   Slip Coefficient 3 = 2e-4
End
```

Results

The computation takes about 200 cpu seconds on an AlphaServer with 1 GHz central processor when trilinear elements are used (historical results). The results for two different input velocities taken from the file `flowdata.dat` are summarised in Table 2.1. Also the specific acoustic impedance z_h is calculated in the table. The results of slip and no-slip simulations are also compared. Note that for the force, only the component perpendicular to the surface should be used since the other components cancel out due to symmetry. The values in the table are again given in Elmer MEMS units (these units are numerically favourable in small dimensions and were used historically in MEMS projects).

Table 2.1: Results of flow simulations for two input velocities

v	slip model	F_z	z_h
$1.0 \cdot 10^3$	no-slip	36.13	$1.45 \cdot 10^{-3}$
$2.0 \cdot 10^3$	no-slip	72.25	$1.45 \cdot 10^{-3}$
$1.0 \cdot 10^3$	slip	29.30	$1.17 \cdot 10^{-3}$
$2.0 \cdot 10^3$	slip	58.60	$1.17 \cdot 10^{-3}$

The identical values obtained for the specific acoustic impedance in Table 2.1 prove by no means that the flow in reality is linear, since this was the assumption and the simulation performed can and should not reveal any nonlinear behavior. The results indicate, though, that allowing slip on the boundaries reduces the resistance that the fluid faces. This example shows that in micro-systems, as the dimension of the smallest flow channel is in the range of a micrometer, it is reasonable to use slip boundary conditions for the velocity.

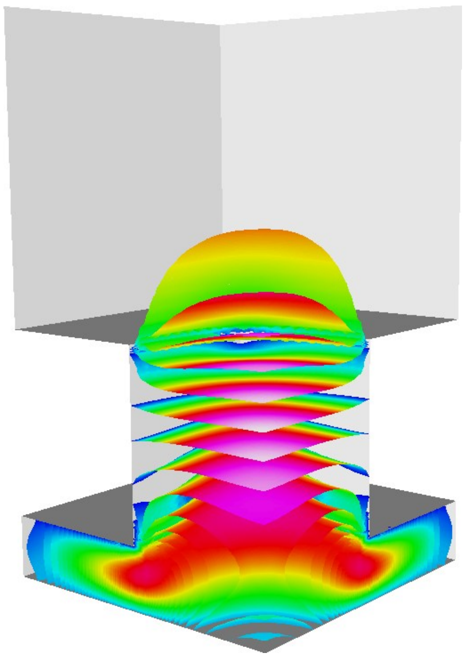
Figure 2.1: The linear flow results.

Finally, a picture of the results obtained with no-slip conditions is presented. The Fig. 2.1 shows a lot of pressure isosurfaces which are coloured using the absolute value of the velocity.

Note: it seems that the results with the current code are not exactly the same. However, we didn't invest where the small discrepancy might come from.

Tutorial 3

Electrostatics

Directory: Electrostatics
Solvers: StatElecSolve, ElectricForce
Tools: ElmerGrid, editor
Dimensions: 3D, Steady-state

Case definition

This case presents solving the Poisson equation for electric potential and calculating appropriate derived quantities, such as capacitance, based on the result. The geometry studied is a symmetric quadrant of a plane capacitor having a rectangular hole in another plate. A setting of this kind can be used to study the effects of geometrical features on the capacitance and on the electrostatic force, which both are meaningful quantities for coupled simulations in *e.g.* microsystems.

Solution procedure

The mesh is constructed using ElmerGrid with the following command

```
ElmerGrid 1 2 elmesh.grd
```

The mesh is extended above the hole to avoid undesired boundary effects. The geometry is presented in the Figure 3.1

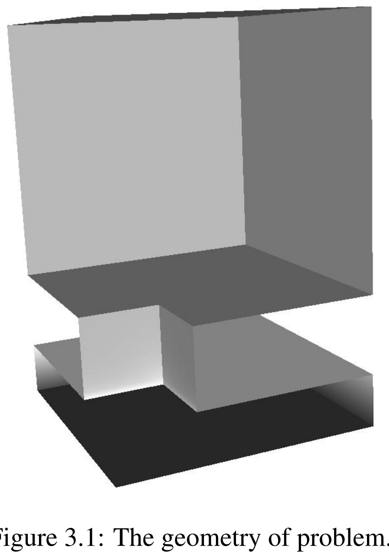

Figure 3.1: The geometry of problem.

The simulation problem includes a single body, and thus one material and one equation set, as well as three solvers. The solvers are used to compute the electric potential and related quantities, to calculate the electric force, and to save relevant data into a file. This tutorial is defined in Elmer MEMS units. The sif-file is presented below.

```
Check Keywords Warn

Header
  Mesh DB "." "elmesh"
End
```

Only a single steady state iteration is needed, since the Poisson equation is linear.

```
Simulation
  Coordinate System = Cartesian 3D
  Simulation Type = Steady State
  Steady State Max Iterations = 1
  Output File = "elstatics.result"
  Post File = "elstatics.vtu"
End
```

The permittivity of vacuum has to be defined in the Constants section.

```
Constants
  Permittivity Of Vacuum = 8.8542e-12
End

Body 1
  Equation = 1
  Material = 1
End
```

Electric energy density is added into the results in Equation section. This allows energy density to be visualised. Here the visualization is done with now obsolete ElmerPost but you would probably rather use Paraview (or some other software that can handle VTU files). Note also, that calculating electric flux (or the electric displacement field) is disabled in the Solver 1 block. Further, the potential difference used in calculating the capacitance of the system has to be defined in this section. This should be the same as the boundary conditions define for the capacitance calculation to be sensible.

```
Equation 1
  Active Solvers(2) = 1 2
  Calculate Electric Energy = True  ! (default False)
End

Solver 1
  Equation = Stat Elec Solver
  Variable = Potential
  Variable DOFs = 1
  Procedure = "StatElecSolve" "StatElecSolver"
  Calculate Electric Field = True  ! (default True)
  Calculate Electric Flux = False  ! (default True)
  Potential Difference = 1.0e6
  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Max Iterations = 200
  Linear System Convergence Tolerance = 1.0e-07
  Linear System Preconditioning = ILU1
  Linear System ILUT Tolerance = 1.0e-03
  Nonlinear System Max Iterations = 1
  Nonlinear System Convergence Tolerance = 1.0e-4
  Nonlinear System Newton After Tolerance = 1.0e-3
  Nonlinear System Newton After Iterations = 10
  Nonlinear System Relaxation Factor = 1
  Steady State Convergence Tolerance = 1.0e-4
End
```

The static electric force solver does not need a lot of information:

```
Solver 2
  Equation = Electric Force
  Procedure = "ElectricForce" "StatElecForce"
End
```

Finally, some data is saved in file scalars.dat in working directory.

```
Solver 3
  Exec Solver = After All
  Equation = SaveScalars
  Procedure = "SaveData" "SaveScalars"
  Filename = "scalars.dat"
End
```

Only the relative permittivity of the material has to be defined.

```
Material 1
  Relative Permittivity = 1
End
```

The boundary conditions include the values of electric potential (voltage) and indication on which boundary the electric force should be calculated. On all the other boundaries a natural boundary condition is used, basically stating that the electric flux through these boundaries is zero.

```
Boundary Condition 1
  Target Boundaries = 4
  Potential = 0.0
  Calculate Electric Force = True
End

Boundary Condition 2
  Target Boundaries = 3
  Potential = 1.0e6
End
```

Results

The results obtained for capacitance and electric force are compared to those of a complete plane capacitor. For a plane capacitor, the capacitance is

$$C = \varepsilon_r \varepsilon_0 \frac{A}{d}, \tag{3.1}$$

and the electrostatic force is

$$F_e = \frac{1}{2}\varepsilon_r \varepsilon_0 \frac{A}{d^2}\Phi^2, \tag{3.2}$$

where ε_r is the relative permittivity, ε_0 is the permittivity of vacuum, A is the area of a capacitor plate, d is the separation of the capacitor plates, and Φ is the potential difference between the plates.

Table 3.1: Comparison of numerical results to analytic values

	simulation	analytic	ratio
Capacitance	$2.1361 \cdot 10^{-10}$	$2.2136 \cdot 10^{-10}$	0.965
Electric Force	$1.0406 \cdot 10^{2}$	$1.1068 \cdot 10^{2}$	0.940

The results of the simulation as well as the comparison to the complete plane capacitor values are shown in Table 3.1 (in Elmer MEMS units). Note that the fringe fields on capacitor edges are not calculated. This would require much larger mesh extending outside the capacitor boundaries.

Finally, a picture of the results is presented. The Figure 3.2 shows the isosurfaces of the electric potential with the color marking the strength of the electric field. From the picture it is clearly seen that the electric field is constant between the plates except for the proximity of the hole which causes weakening of the field magnitude. There are also strong electric fields at the edges of the hole.

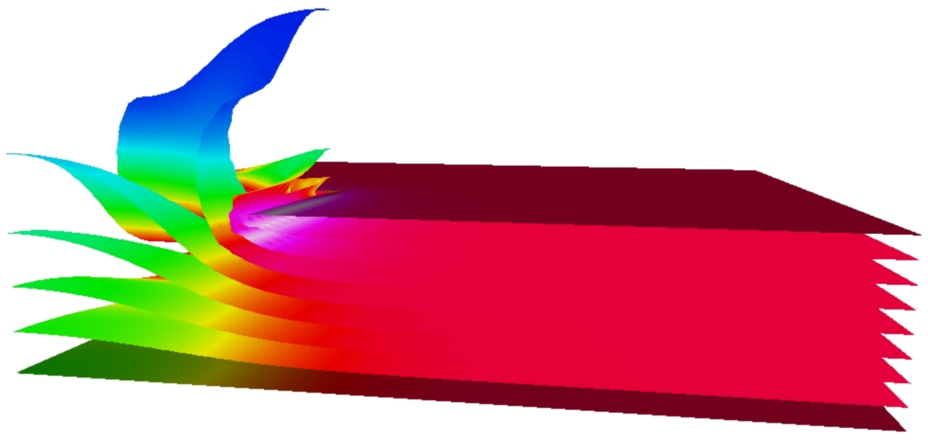
Figure 3.2: Isosurfaces of the potential coloured with electric field magnitude.

Tutorial 4

Induction heating of a graphite crucible

Directory: InductionHeating
Solvers: StatMagSolve
Tools: ElmerGrid, editor
Dimensions: 2D, Axi-Symmetric

Case definition

At high temperatures the most practical method to heat up the crucible is by electromagnetic induction. The induction coil generates an alternating current that flows through the crucible. The Ohmic resistance encountered by this current dissipates energy, thereby directly heating the crucible via internal heat generation.

The tutorial case is a simple axi-symmetric crucible that could be used, for example, to grow silicon carbide (SiC) by the sublimation method. The crucible is made of dense graphite and isolated by porous graphite. At the bottom of the crucible there is some SiC powder. The physical properties of the material are given in Table 4.1. The dimensions of the induction heating crucible are given in Table 4.2. Additionally, the powder thickness is 1.0 cm and there are 10 spirals in the coil. The frequency of induction heating f is 50 kHz and the current I is 10 A. The permeability of the space is $4\pi 10^{-7}$ if the other variables are in SI-units.

Solution Procedure

At low frequencies the free charges may be neglected and the induction heating problem may be solved in terms of an magnetic vector potential. The proper solver to do this is `StatMagSolver`. However, the induction heating problem can only be modeled if the helicity of the coil is neglected and an average current density is assumed. This current density may be computed easily when the area of the coil is known $j_0 = nI/A$, where A is the coil area.

The mesh for this problem may easily be created by ElmerGrid. The provided mesh is quite sufficient for this case but for higher frequencies the mesh should be tuned to solve the thin boundary layers. The computational mesh is created from file `crucible.grd` by the command

```
ElmerGrid 1 2 crucible
```

Table 4.1: Material parameters of the crucible

material	ε	κ [W/mk]	σ (1/Ωm)
graphite	0.7	10.0	2.0E4
insulation	0.9	1.0	2.0E3
powder	0.5	25.0	1.0E4

Table 4.2: Dimensions of the crucible

body part	r_{inner}	r_{outer}	h_{inner}	h_{outer}
graphite	2.0	2.5	6.0	8.0
insulation	2.5	4.0	8.0	12.0
coil	5.0	5.5		8.0

The mesh consists of 5 different bodies which need 4 different materials sets. Only on set of boundary conditions are required for the external boundary. Thus the header information of the command file is as follows

```
Header
  Mesh DB "." "crucible"
  Include Path ""
  Results Directory ""
End
```

In the `Simulation` section the coordinate system and time dependency is set, among other things. Also we know that the equation is linear and therefore only one steady state iteration is requited. If the electric properties depend on the magnitude of the field several iterations are required.

```
Simulation
  Coordinate System = "Axi Symmetric"
  Simulation Type = Steady State
  Steady State Max Iterations = 1
  Output File = "crucible.result"
  Post File = "crucible.vtu"
End
```

In the `Constants` section the permittivity of vacuum must be given.

```
Constants
  Permittivity Of Vacuum = 8.8542e-12
End
```

In the differential equation for the magnetic vector potential the source the is the current density. Thus, it is given in the `Body Force` section.

```
Body Force 1
  Current Density = 2.5e5
End
```

In the `Body` section the different bodies are assigned with correct equation sets and material parameters, for example

```
Body 3
  Name = "Insulation"
  Equation = 1
  Material = 2
End
```

In the `Equation` block all the relevant solvers are set to active.

```
Equation
  Name = "Vector Potential Equation"
  Active Solvers = 1
End
```

The only solver in this simple tutorial is the solver for the magnetic vector potential. Look for the relevant model manual for information about the options. Here the equation is solved iteratively and the local Joule heating and magnetic flux are computed as a postprocessing step. The Joule heating is scaled so that the total heating power is 3.0 kW. This option may be used when the total heating efficiency is known. The nonlinear solver parameters are not really needed as the material parameters are constant. Sometimes the parameters may depend on the magnetic field and thus the nonlinear problem must be solved iteratively.

```
Solver 1
  Equation = Potential Solver
  Variable = Potential
  Variable DOFs = 2

  Angular Frequency = Real 50.0e3
  Calculate Joule Heating = Logical True
  Calculate Magnetic Flux = Logical True
  Desired Heating = Real 3.0e3

  Procedure = "StatMagSolve" "StatMagSolver"
  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Max Iterations = 300
  Linear System Convergence Tolerance = 1.0e-10
  Linear System Preconditioning = ILU1
  Linear System ILUT Tolerance = 1.0e-03
  Linear System Residual Output = 1
  Nonlinear System Max Iterations = 1
  Nonlinear System Convergence Tolerance = 1.0e-6
  Nonlinear System Relaxation Factor = 1
  Steady State Convergence Tolerance = 1.0e-6
End
```

In the `Material` sections all the necessary material parameters are given, for example

```
Material 2
  Name = "Insulation"
  Electric Conductivity = 2.0E3
End
```

The magnetic field must vanish at infinity. Unfortunately the computational domain is bounded and therefore the infinite distance becomes very finite. A proper distance may be checked by gradually increasing it until no change in the result occurs.

```
Boundary Condition 1
  Target Boundaries = 1
  Potential 1 = Real 0.0
  Potential 2 = Real 0.0
End
```

Results

With the given computational mesh the problem is solved in a few seconds. With the 20 072 bilinear elements the heating efficiency is 16.9 W. The corresponding results are shown in Fig. 4.1.

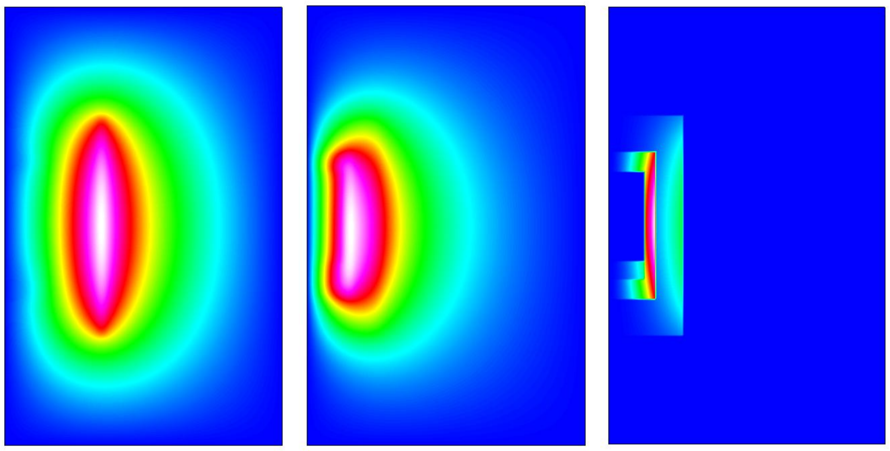

Figure 4.1: Induction heating of a simple crucible. a) in-phase component of the vector potential b) out-of-phase component of the vector potential c) Joule losses in the conductors

Tutorial 5

Thermal actuator driven with electrostatic currents

Test: ThermalActuator
Directory: ThermalActuator
Solvers: StatCurrentSolve, HeatSolve, StressSolve
Tools: ElmerGrid, editor
Dimensions: 3D, Steady-state

Case definition

The tutorial introduces a micro mechanical thermal actuator as shown in Fig. 5.1. A static electric current is driven through the actuator. The power loss due to the resistance of the actuator is transformed into heat which in turn causes thermal stresses into the structure. The electric current thus results in deformation of the actuator. In industry, such an actuator might be used to control the position of a micromechanical component.

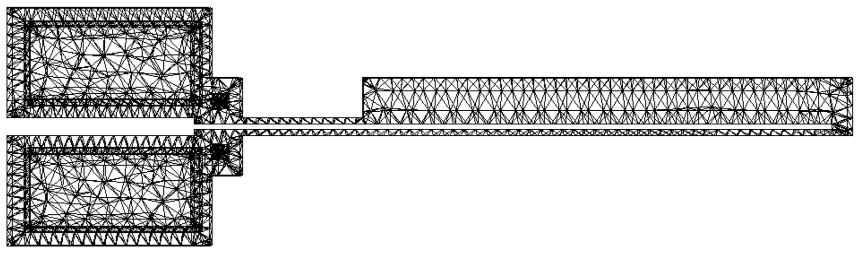

Figure 5.1: The geometry of the actuator.

Solution procedure

The problem is solved by first iterating the electrostatic current solver and heat equation until both are converged. The temperature distribution is then used as a load for stress analysis solver which calculates the actual deformation of the structure. The electric conductivity of the actuator depends on the temperature and thus the electrostatic - thermal problem is coupled in both directions.

The computational mesh for this particular tutorial is created by using Ansys software. The details of the mesh are written into files called `ExportMesh` by a certain Ansys macro and converted to Elmer format by the ElmerGrid program. The command to use is

```
ElmerGrid 4 2 ExportMesh -order 1.0 0.1 0.001 -o thermal
```

The above command reads in the Ansys mesh files, arranges the mesh nodes in a reasonable way and saves the mesh in Elmer format in a directory called `thermal`.

The geometry of the problem includes only one body and material. Boundary conditions are defined on the actuator legs, which are kept at constant electric potential, temperature and position. Thus, only Dirichlet boundary conditions are used.

The header and simulation blocks of the solver input file are

```
Header
  Mesh DB "." "thermal"
End

Simulation
  Coordinate System = Cartesian 3D
  Simulation Type = Steady State
  Steady State Max Iterations = 30
  Output Intervals = 1
  Output File = "actuator.result"
  Post File = "actuator.vtu"
End
```

An initial condition for temperature is defined in order to ease the convergence of the iterative solvers. Also, a body force for the heat equation solver defining the Joule heating is needed. These both have to be declared in the body section as follows:

```
Body 1
  Equation = 1
  Material = 1
  Initial Condition = 1
  Body Force = 1
End
```

The solution procedure requires the use of three solvers: Static current solver, heat equation solver and the stress analysis solver. The equation block below defines that these solvers are used.

```
Equation 1
  Active Solvers(3) = Integer 1 2 3
  Calculate Joule Heating = True
End
```

The solver blocks define the parameters of the respecting solvers. The static current conduction problem is tackled by an iterative conjugate gradient method (CG). For heat equation, a stabilized biconjugate gradient method is used. The coupled problem of these two solvers is difficult since the static current calculated heats the structure on each step, and the rise of temperature makes the current conduction more and more difficult. To overcome this problem, a relaxation factor of 0.5 is defined for the heat equation solver.

```
Solver 1
  Equation = Stat Current Solver
  Procedure = "StatCurrentSolve" "StatCurrentSolver"
  Variable = Potential
  Variable DOFs = 1
  Calculate Volume Current = True
  Calculate Electric Conductivity = True
  Linear System Solver = Iterative
  Linear System Iterative Method = CG
  Linear System Preconditioning = ILU3
  Linear System Max Iterations = 300
  Linear System Convergence Tolerance = 1.0e-8
  Nonlinear System Max Iterations = 1
  Nonlinear System Convergence Tolerance = 1.0-6
  Nonlinear System Newton After Iterations = 3
  Nonlinear System Newton After Tolerance = 1.0e-12
  Nonlinear System Relaxation Factor = 1.0
  Steady State Convergence Tolerance = 1.0e-6
End

Solver 2
   Equation = Heat Equation
   Variable = Temperature
   Variable DOFs = 1
   Linear System Solver = Iterative
   Linear System Iterative Method = BiCGStab
   Linear System Preconditioning = ILU1
   Linear System Max Iterations = 350
   Linear System Convergence Tolerance = 1.0e-9
   Nonlinear System Max Iterations = 1
   Nonlinear System Convergence Tolerance = 1.0e-07
   Nonlinear System Newton After Iterations = 3
   Nonlinear System Newton After Tolerance = 1.0e-12
   Nonlinear System Relaxation Factor = 0.5
   Steady State Convergence Tolerance = 1.0e-07
End
```

For stress analysis, a direct solver is used instead of an iterative solver. It is often difficult for the iterative solver to find a solution for a structure that contains parts with varying stiffness properties, which is obviously the case here (try the iterative solver and see!). The stress analysis solver is called first only after the coupled iteration of two previous solvers is complete. This is possible since the deformation of the structure is so small that it does not change the current density distribution. Defining stress analysis this way saves computational time. It is possible to iterate all the three solvers until convergence by commenting the `Exec Solver` line.

```
Solver 3
  Exec Solver = After All
  Equation = Stress Analysis
  Variable = Displacement
  Variable DOFs = 3
  Linear System Solver = Direct
  Linear System Direct Method = Banded
  Nonlinear System Max Iterations = 1
  Nonlinear System Convergence Tolerance = 1.0e-6
  Nonlinear System Newton After Iterations = 3
  Nonlinear System Newton After Tolerance = 1.0e-12
  Nonlinear System Relaxation Factor = 1.0
  Steady State Convergence Tolerance = 1.0e-6
End
```

The material of the structure has a temperature dependent electric conductivity. This, as well as other material parameters, is defined in the material block. Note that a MEMS unit system is used.

```
Material 1
```

```
  Electric Conductivity = Variable Temperature
      Real
        298.0   4.3478e10
        498.0   1.2043e10
        698.0   5.1781e9
        898.0   2.7582e9
        1098.0  1.6684e9
        1298.0  1.0981e9
        1683.0  1.0
        2000.0  1.0
      End

  Density = 2.3e-15
  Heat Conductivity = 32.0e6
  Youngs Modulus = 169.0e3
  Poisson Ratio = 0.22
  Heat Expansion Coefficient = 2.9e-6
  Reference Temperature = 298.0
End
```

Finally, the initial condition, thermal heat load for stress analysis, and the boundary conditions are defined.

```
Initial Condition 1
   Temperature = 298.0
End

Body Force 1
  Heat Source = Equals Joule Heating
End

Boundary Condition 1
  Target Boundaries = 1
  Potential = 0
  Temperature = 298
  Displacement 1 = 0.0
  Displacement 2 = 0.0
  Displacement 3 = 0.0
End

Boundary Condition 2
  Target Boundaries = 2
  Potential = 7
  Temperature = 298
  Displacement 1 = 0.0
  Displacement 2 = 0.0
  Displacement 3 = 0.0
End
```

Results

The problem converges after 27 steady state iterations on the tolerance limits defined above. The calculation takes about 180 cpu seconds of which 40 cpus is spent in solving the stress analysis equation. The calculations were performed on a Compaq Alpha Server with a 1 GHz central processor.

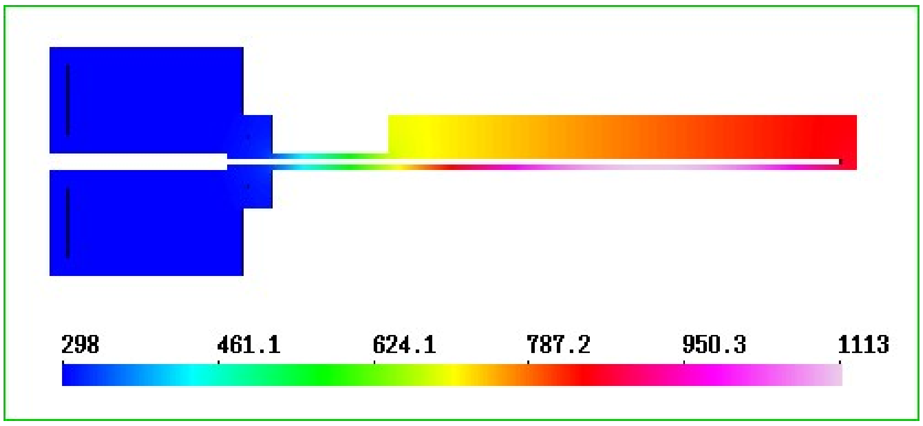

Figure 5.2: Temperature distribution.

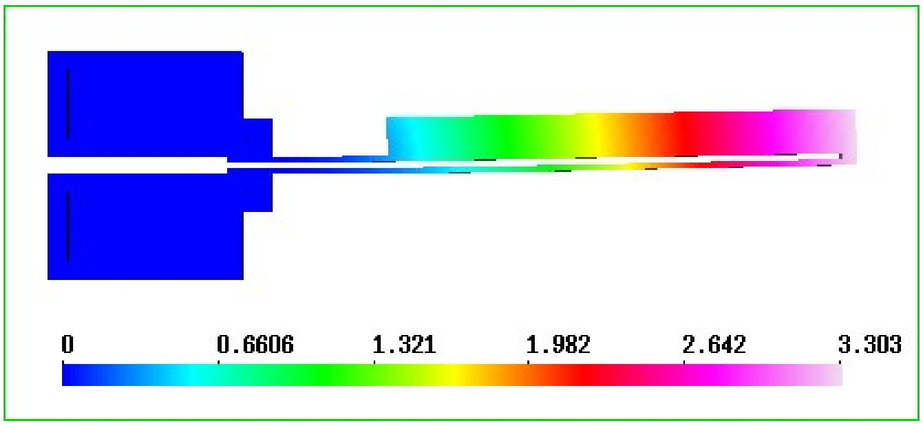

Figure 5.3: The displacement of the actuator.

Result for temperature distribution and the displacement are shown in Figs 5.2 and 5.3. The temperature rises unrealistically high in this example because all heat transfer mechanisms out of the structure are neglected. Presumably at least the heat radiation is of major importance in this case. For displacement, the results show a movement of about 3.3 micrometers for the actuator tip.

Tutorial 6

Axisymmetric coating process

Director: CoatingProcess
Solvers: FlowSolve, FreeSurfaceReduced
Tools: ElmerGrid, editor
Dimensions: 2D, Steady-state

Case definition

The optical fibers are quite fragile and must therefore be coated with a layer of polymer before they are stored. This means that the coating process must be done with the same speed as the drawing of optical fibers. When the diameter of the fiber is only 125 μm this sets high demands for the coating geometry since it must provide even coating at high draw speeds. In Elmer a tailored free surface boundary condition allows an efficient solution of this particular problem.

Solution procedure

The mesh is done with ElmerGrid in the directory coat by the command

```
ElmerGrid 1 2 coat.grd
```

Therefore the header reads

```
Header
  Mesh DB "." "coat"
End
```

The geometry is axisymmetric and the problem is solved in steady state. Typically around 10 iterations is needed to solve the problem but to be on the safe side 30 is set as the maximum.

```
Simulation
  Coordinate System = Axi Symmetric
  Simulation Type = Steady State
  Steady State Max Iterations = 30
  Output Intervals = 1
  Output File = "coat.result"
  Post File = "coat.vtu"
End
```

In this case there is only one body which comprises of the polymer floating between the coating cup and the optical fiber.

```
Body 1
  Equation = 1
  Material = 1
End
```

The presented solution used four different solvers. The Navier-Stokes solver is required to solve the flow field for the polymer.

```
Solver 1
  Equation = Navier-Stokes
  Stabilize = True
  Internal Move Boundary = Logical False
  Nonlinear System Max Iterations = 5
  Nonlinear System Convergence Tolerance = 1.0e-7
  Nonlinear System Newton After Iterations = 2
  Nonlinear System Newton After Tolerance = 1.0e-2
  Nonlinear System Relaxation Factor = 0.7
  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Preconditioning = ILU1
  Linear System Max Iterations = 100
  Linear System Convergence Tolerance = 1.0e-10
  Steady State Convergence Tolerance = 1.0e-7
End
```

A tailored free surface solver is used to find the position of the free surface with a given flow field. The variable being solved is the displacement of the free surface. Relaxation is used to avoid over-shooting during the iteration. This solver does not solve any matrix equations. Instead it solves the radius from the mass conservation constraint for each node on the free surface separately. There is a possibility to do the mapping also within the solver using a 1D scheme but this is disabled by setting the `Perform Mapping` to be `False`.

```
Solver 2
  Equation = "Free Surface Reduced"
  Procedure = "FreeSurfaceReduced" "FreeSurfaceReduced"
  Variable = Dx
  Variable DOFs = 1
  Nonlinear System Relaxation Factor = 0.7
  Nonlinear System Convergence Tolerance = 1.0e-3
  Steady State Convergence Tolerance = 1.0e-3
  Perform Mapping = Logical False
End
```

The mesh update solver is required to map the computational mesh so that it corresponds to the altered geometry. Here the displacements of the free surface have already been computed and this solver solves the displacements inside the domain. Note that solvers 1, 2 and 3 are coupled and therefore the system must be solved iteratively

```
Solver 3
  Equation = Mesh Update
  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGSTAB
  Linear System Preconditioning = ILU
  Linear System Convergence Tolerance = 1.0e-12
  Linear System Max Iterations = 200
  Linear System Symmetric = True
  Steady State Convergence Tolerance = 1.0e-4
End
```

In the end, an additional solver is used to compute the forces acting on the fiber. This does not affect the results.

```
Solver 4
  Equation = Fluidic Force
  Procedure = "FluidicForce" "ForceCompute"
  Calculate Viscous Force = Logical True
End
```

Additionally there are two solvers for saving the results in a form that is more useful than plain pictures. The `SaveScalars` saves the scalar values, such as the diameter and force values, and the `SaveLine` saves the free surface.

```
Solver 5
  Equation = SaveScalars
  Procedure = "SaveData" "SaveScalars"
  Filename = "scalars.dat"
End

Solver 6
  Equation = SaveLine
  Procedure = "SaveData" "SaveLine"
  Filename = "kurvi.dat"
End
```

The equation includes only the solvers that need a permutation vector pointing out the active nodes. Therefore the save utilities do not need to belong to the set of active solvers.

```
Equation 1
  Active Solvers(4) = 1 2 3 4
End
```

The material parameters are those of the polymer. Additionally elasticity parameters are needed because the solver that updates the mesh is actually a linear elasticity solver.

```
Material 1
  Density = 1.0
  Viscosity = 1.0
  Poisson Ratio = 0.3
  Youngs Modulus = 1.0
End
```

Five different boundary conditions are needed. The origin is a symmetry axis and therefore the radial velocity is set to zero. The axial velocity is the draw velocity.

```
Boundary Condition 1
  Name = "Symmetry"
  Target Boundaries = 1
  Velocity 2 = -10.0      ! The draw velocity
  Velocity 1 = 0.0
  Compute Fluidic Force = Logical True
  Mesh Update 1 = 0.0
End
```

The free surface has a condition stating that the reduced order free surface solver should be solved for that. Additionally the free surface is a boundary condition for the mesh update, and a line to be saved.

```
Boundary Condition 2
  Name = "Free"
  Target Boundaries = 2
  Mesh Update 1 = Equals Dx
  Mesh Update 2 = 0.0
  Free Surface Reduced = Logical True
  Save Line = Logical True
End
```

At the outlet the radial velocity should vanish and the axial coordinate should be fixed.

```
Boundary Condition 3
  Name = "Outlet"
  Target Boundaries = 3
  Velocity 1 = 0.0
  Mesh Update 2 = 0.0
End
```

At the inlet it is assumed that there is no radial velocity and that the pressure acting on the surface is zero.

```
Boundary Condition 4
  Name = "Inlet"
  Target Boundaries = 4
  Velocity 1 = 0.0
  Pressure = 0.0
  Mesh Update 2 = 0.0
End
```

Finally, no-slip conditions are set for the boundaries with the walls of the coater.

```
Boundary Condition 5
  Name = "No-slip"
  Target Boundaries = 5
  Velocity 1 = 0.0
  Velocity 2 = 0.0
  Mesh Update 1 = 0.0
  Mesh Update 2 = 0.0
End
```

Results

In the given case solution is obtained after 13 iterations. The solution gives the final radius, the forces, and the profile of the free surface. To visualize the true free surface you may visualize the last timestep using Paraview, or some other software capable of reading VTU files.

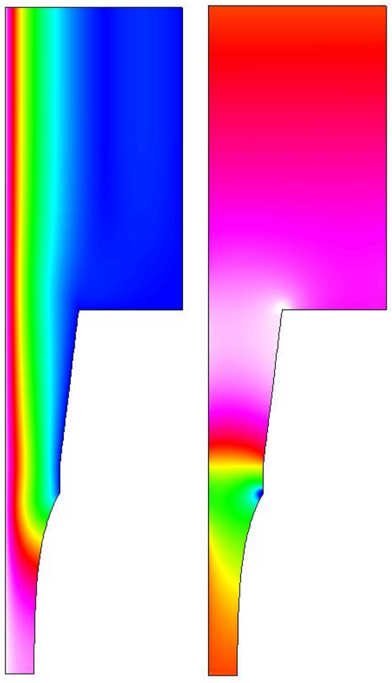
Figure 6.1: The velocity and pressure fields in a simple coating geometry. The solution utilizes the reduced dimensional free surface solver.

Tutorial 7

Blood ejection from a ventricle into aorta

Test: ArteryOutlet
Directory: ArteryFlow
Solvers: FlowSolve,ElasticSolve, OutletCompute
Tools: Editor, Fortran 90 compiler, ElmerGrid
Dimensions: 2D, Transient

Case description

This tutorial is about simulating blood ejection in to the elastic human aorta. The idea is to mimic left ventricle contraction and resulting pulse propagation in an elastic conduit. In the simulation about 0.8 deciliters of blood is ejected to a 50 cm long elastic aorta during a time period of 400 ms. In order to get the outlet of the model behave physiologically more realistic way, a one dimensional model is coupled with the higher order model.

Solution procedure

First we generate the mesh of 366 eight-node quadrilaterals elements with the command

```
ElmerGrid 1 2 contra
```

Next we generate one dimensional mesh to the outlet of the 2D model. The program `AddOneDim` is posed to be run in the mesh directory `contra`. The length, the number of the elements, and the coordinate direction of the 1D section will be asked.

In the simulation block the timestep is set equal to 1 ms and total simulation time equal to 600 ms. The geometry consists of five bodies of which the first three are for the fluid volume. Body number 1 os the contracting volume. Body 2 is a short rigid channel between the body 1 and the elastic artery. Artificial compressibility method is used for the fluid volume (body 3) which is in contact with the elastic wall (body 4). One dimensional model is the body 5. Material settings for those are following:

```
! Bodies 1 and 2 (blood)
Material 1
  Density = 1000
  Viscosity = 3.5e-3
  Youngs Modulus = 1
  Poisson Ratio = 0.3
End
```

```
! Body 3 (blood)
Material 2
  Density = 1000
  Viscosity = 3.5e-3
  Youngs Modulus = 1
  Poisson Ratio = 0.3
  Compressibility Model = Artificial Compressible
  Artificial Compressibility  = 3.3E-5
End

! Body 4 (elastic wall)
Material 3
  Density = 1010
  Youngs Modulus = 3.0e5
  Poisson Ratio = 0.45
End

! One dimensional model
Material 4
   Density = 1010.0
   Artery Wall Youngs Modulus = Real 3.0e5
   Artery Radius = Real 0.0135
   Artery Wall Thickness = Real 0.002
   Artery Poisson Ratio  = Real 0.45
End
```

Notice that the radius of the one dimensional model (`Artery Radius`) is to the midplane of the wall (inner radius + half of the wall thickness). The overall FSI iteration scheme is started by one dimensional solver (`OutletCompute`, see the solver manual), after that Navier-Stokes, elasticity and mesh update solvers are run. Steady state convergence tolerance is set equal to 1.0E-4 for each of the solvers. The nonlinearities of each of the solvers are computed within the FSI scheme loop, that is, the flag `Nonlinear System Max Iterations` is set equal to 1. Artificial compressibility coefficient is computed by the equation $c = (1 - \nu^2)[D/(E\,h)]$, where ν is the Poisson ratio of the artery wall, D, E and h are the inner diameter, Young's modulus and the thickness of the artery, respectively.

The only driving force of the system, the wall motion of the contracting fluid domain is given by the fortran function `Motion`, see the figure 7.1. The boundary condition setting is

```
! Moving boundary
Boundary Condition 1
  Target Boundaries = 1
  Velocity 1 = 0
  Velocity 2 = Equals Mesh Velocity 2
  Mesh Update 1 = Real 0
  Mesh Update 2 = Variable Time
       Real Procedure "./Motion" "Motion"
End
```

At the outlet, the pressure boundary condition is given by the function `OutletPres` and the corresponding radial displacement of the end wall of the outlet is given by the function `OutletdX`

```
! Outlet pressure of the 2D model
Boundary Condition 2
  Target Boundaries = 2
  Flux Integrate = Logical True
  Flow Force BC = True
  Pressure 2 = Variable Time
```

```
      Real Procedure "./ArteryOutlet" "OutletPres"
  Mesh Update 2 = Real 0
End

! Radial displacement of the end wall at the outlet of 2D model
Boundary Condition 9
  Target Boundaries = 9
  Displacement 1 = Variable Time
      Real Procedure "ArteryOutlet" "OutletdX"
  Displacement 2 = 0
End
```

FSI interface boundary is described as following

```
! FSI interface boundary
Boundary Condition 11
  Target Boundaries = 11
  Velocity 1 = Equals Mesh Velocity 1
  Velocity 2 = Equals Mesh Velocity 2
  Mesh Update 1 = Equals Displacement 1
  Mesh Update 2 = Equals Displacement 2
  Force BC = Logical True
End
```

Finally, the coupling of the 1D model with the 2D is done at the inlet boundary as

```
Boundary Condition 16
  Target Boundaries = 16
  Fluid Coupling With Boundary = Integer 2
  Structure Coupling With Boundary = Integer 9
End
```

Results

The contraction is curve seen in the figure 7.1 and the velocity fields at different time levels are presented in the figure 7.2. Postprocessing instructions are given in the file `PostProcessingInstr.txt`.

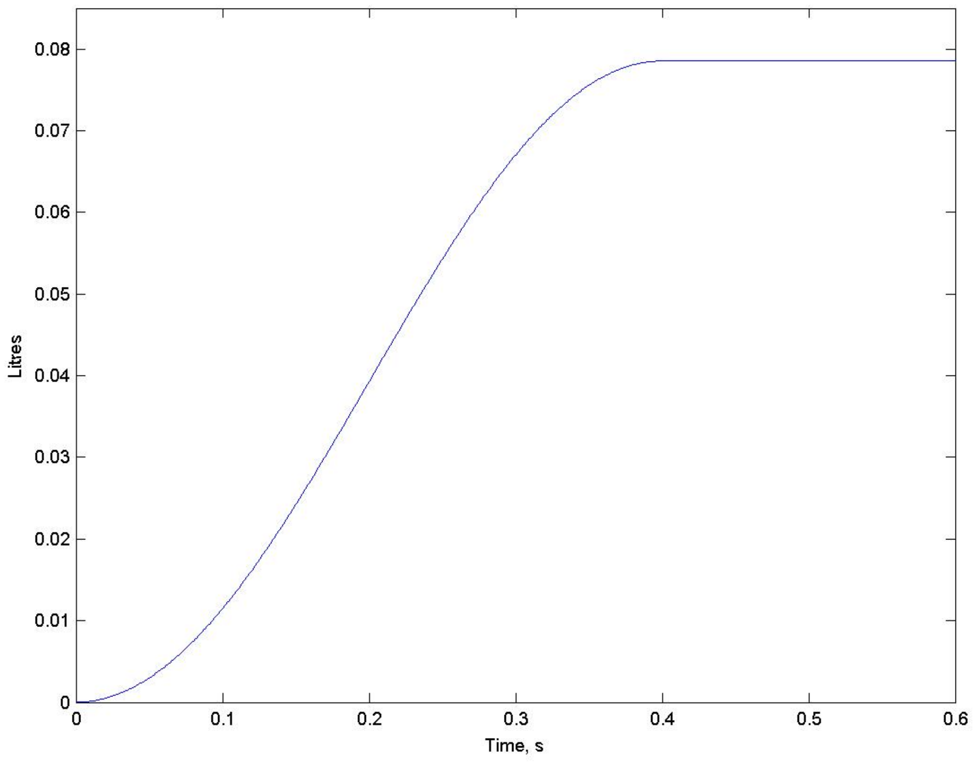

Figure 7.1: Contraction curve generated by the function `Motion`.

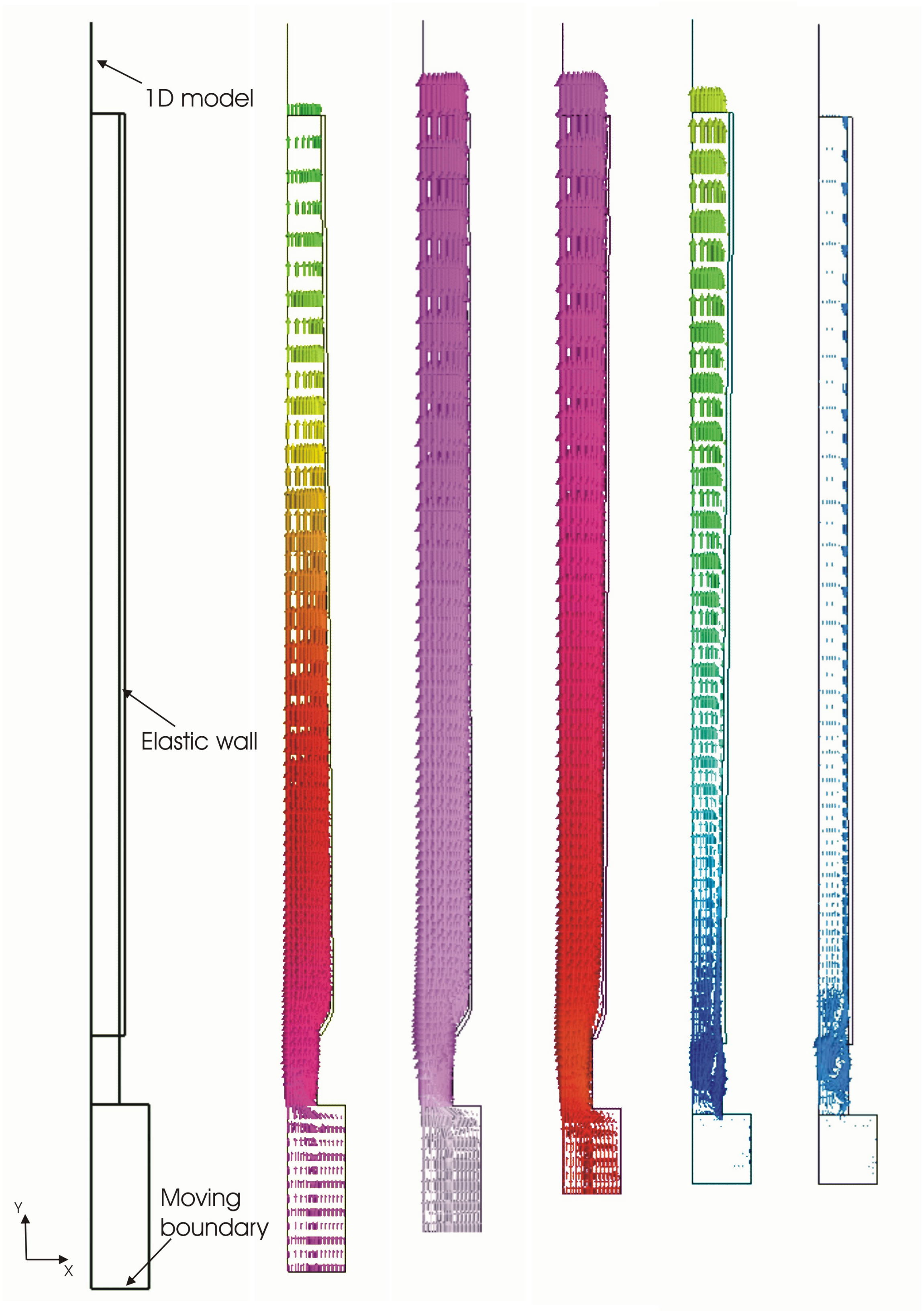

Figure 7.2: The geometry of the model and velocity fields at 5 time steps, 100, 200, 300, 400 and 500 ms. The displacements of the wall are magnified by factor of 10.

Tutorial 8

Temperature distribution with BEM

Directory: PoissonBEM
Solvers: PoissonBEMSolver
Tools: ElmerGrid, editor
Dimensions: 2D

Case definition

This tutorial uses boundary element method (BEM) to solve Poisson equation. Even though Elmer is primarily a finite element software the are limited support also for BEM computation. One should however note that Elmer does not include any multilevel strategies essential for a good performance. For more details about BEM check the Elmer Models Manual. The simulation setting is described in Figure 8.1. A heater with constant heat flux is placed inside a box and the walls of the box are in fixed temperature. We are interested in the temperature distribution in the medium around the heater (Ω) and on the surface of the heater (Γ_1). We also want to know the heat flux through the walls of the box (Γ_2).

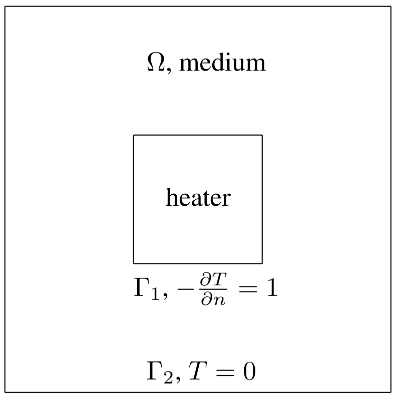

Figure 8.1: Simulation setting

Solution Procedure

First we create a mesh with ElmerGrid. The mesh is defined in `heater.grd` and it is created with command

```
ElmerGrid 1 2 heater
```

The solver input file `PoissonBEM.sif` starts with the definition of the mesh directory.

```
Header
  Mesh DB "." "heater"
End
```

The simulation uses 2D Cartesian geometry, searches a steady state and since there is no coupled solvers only one iteration is needed. Numerical results for restart are written to file `BEM_Temperature.result` and file for Paraview visualization is `BEM_Temperature.vtu`.

```
Simulation
  Coordinate System =  Cartesian 2D
  Coordinate Mapping(3) = 1 2 3

  Simulation Type = Steady
  Steady State Max Iterations = 1

  Output Intervals = 1
  Post File = "BEM_Temperature.vtu"
  Output File = "BEM_Temperature.result"
End
```

There is just one body, the medium around the heater, and it uses equation 1.

```
Body 1
  Name = "medium"
  Equation = 1
End
```

In equation block we say that we use the solver named `PoissonBEM`.

```
Equation 1
  PoissonBEM = Logical True
End
```

In solver block the `Equation` keyword must match the one in equation block. We also need to define the procedure, name the variable (`Temperature`) and tell the degrees of freedom of the variable. Keyword `Optimize Bandwidth` must be set to false with BEM solver. Since we were interested in the flux, we must now export it to the results. The lines beginning `Exported` must be exactly as below. Keywords beginning `Linear System` can be used except that the preconditioning cannot be ILU.

```
Solver 1
  Equation = PoissonBEM
  Procedure = "PoissonBEM" "PoissonBEMSolver"
  Variable = Temperature
  Variable DOFs = 1

  Optimize Bandwidth = False

  Exported Variable 1 = String Flux
  Exported Variable 1 DOFs = 1

  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Preconditioning = Jacobi
  Linear System Max Iterations = 100
  Linear System Convergence Tolerance = 1.0e-8

  Steady State Convergence Tolerance = 1.0e-6
End
```

Finally we give the boundary conditions for the heater surface and for the walls of the box. The keyword `Body Id` tells the reference body of this boundary. Here it is 1. The keyword `Normal Target Body` tells the direction of the outer normal. Value -1 means the side where there are no volume elements. We didn't mesh the inside of the heater and so we can use value -1 in both cases. The heat flux from heater to medium is 1 and the walls of the box are set to zero temperature. The keyword `Temperature` matches the name of the variable in solver block.

```
Boundary Condition 1
  Name = "heater_surface"
  Target Boundaries = 1

  Body Id = 1
  Normal Target Body = Integer -1
  Flux = Real 1
End

Boundary Condition 2
  Name = "box_walls"
  Target Boundaries = 2

  Body Id = 1
  Normal Target Body = Integer -1
  Temperature = 0
End
```

Results

Problem is solved with command `Solver`. The results are here viewed with `ElmerPost`. In Figure 8.2 is the temperature distribution.

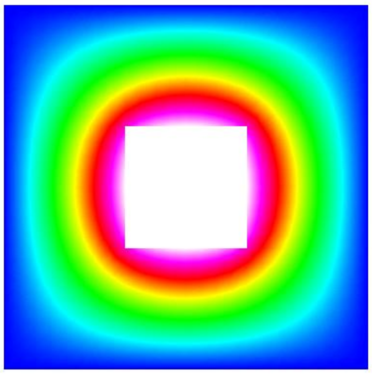

Figure 8.2: The temperature distribution.

Tutorial 9

Adding user defined equation solver

Directory: Temperature1D
Solvers: PoissonSolver
Tools: Editor, Fortran 90 compiler, ElmerGrid
Dimensions: 1D, Steady-state

Problem description

This tutorial is about creating the code for a simple poisson equation solver. The solver is applied to 1d case with internal source term and fixed boundaries.

Mathematically the problem we solve is

$$\left\{ \begin{array}{rcll} -\Delta\Phi & = & f & \text{in } \Omega \\ \Phi & = & 0 & \text{on } \Gamma \end{array} \right. \tag{9.1}$$

Although this example is in 1d the same solver code also applies to 2D and 3D problems.

Solution procedure

Own codes solving some specific equation may be added dynamically to Elmer software. Here we create a very simple equation solver code. The final code may be found in the tutorial directory as well as the files for running the example. The solution may be attempted as follows:

- Copy all the files from tutorial directory to current directory
- Setup Elmer
- Give the following commands:

```
elmerf90 -o Poisson Poisson.f90
ElmerGrid 1 2 1dheat
ElmerSolver
ElmerPost
```

The solver code

The example Fortran code may be found in the tutorial files under the name Poisson.f90. The example run is defined in 1dheat.sif. Only a rough guideline is given here of both of the files, refer to the files themselves for more details.

All the equation solvers in Elmer have the following common interface

```
SUBROUTINE PoissonSolver( Model, Solver, dt, TransientSimulation )
  USE SolverUtils

  TYPE(Model) :: Model
  TYPE(Solver_t), POINTER :: Solver
  REAL(KIND=dp) :: dt
  LOGICAL :: TransientSimulation

    ...
END SUBROUTINE PoissonSolver
```

The argument Model contains pointers to the whole definition of the Elmer run. The argument Solver contains parameters specific to our equation solver. The argument dt and TransientSimulation are the current timestep size, and a flag if this run is steady or transient. These don't concern us this time.

When starting the ElmerSolver looks the solver input (.sif) file for a Solver section with keyword "Procedure". This should contain reference to the compiled code

```
Procedure = "Poisson" "PoissonSolver"
```

where the first string in the right hand side is the file name of the compiled code, and second argument is the name of the subroutine to look for in the given file.

In the Solver section one also gives the name of the field variable (here Poisson) and the DOFs/node (here 1).

The basic duty of the equation solver is to solve one or more field variables inside the time progressing- or steady state iteration-loop of ElmerSolver. Here we use FEM to discretize the Poisson equation and finally solve the equation by calling ElmerSolver utility SolveSystem.

The solution progresses the following way:

- Get the space for variables and temporaries from ElmerSolver and compiler. The matrix structure and space for solution and RHS vector have already been allocated for you before you enter the equation solver.

  The matrix is of type Matrix_t and may be obtained from the arguments as

  ```
  TYPE(Matrix_t), POINTER :: StiffMatrix
  StiffMatrix => Solver % Matrix
  ```

  Usually one doesn't need to know the internal storage scheme or the fields of the Matrix type, but one just passes this pointer further to ElmerSolver utility routines.

  Similarly, the force vector may be accessed as follows:

  ```
  REAL(KIND=dp), POINTER :: ForceVector(:)
  ForceVector => StiffMatrix % RHS
  ```

  The solution vector is obtainable similarly

  ```
  TYPE(Variable_t), POINTER :: Solution
  Solution => Solver % Variable
  ```

  The Variable_t structure contains the following fields

  – DOFs: the number of degrees of freedom for one node. This value is for information only and should'nt be modified.

  – Perm: an integer array that is nonzero for nodes that belong to computational volume for this equation. The entry $Perm(i)$ holds the index of the global matrix row (for 1 DOF) for nodal point i. This array should'nt be modified by the equation solver.

  - Values: Space for the solution vector values. Note that the values are ordered the same way as the matrix rows, .i.e. the value of Potential at node n is stored at

    ```
    val = Solution % Values( Solution % Perm(n) )
    ```

- Initialize the global system to zero. Calling the utility routing

  ```
  CALL InitializeToZero( StiffMatrix, ForceVector )
  ```

  is usually enough.

- Go trough the elements for which this equation is to be solved, get the elemental matrices and vectors and add them to the global system:

  ```
  DO i=1,Solver % NumberOfActiveElements
     CurrentElement => Solver % Mesh % Elements( Solver % ActiveElements(i) )
        ...
     CALL LocalMatrix( ... )
     CALL UpdateGlobalEquations( ... )
  END DO
  CALL FinishAssembly( ... )
  ```

  Here the LocalMatrix is your own subroutine computing elemental matrices and vectors. In the example code LocalMatrix uses three routines from ElmerSolver utilities. The function

  ```
    dim = CoordinateSystemDimension()
  ```

  returns the dimension of the current coordinate system, i.e. the return value is 1, 2 or 3 depending on the input file setting of keyword "Coordinate System". The function GaussPoints returns structure containing the integration point local coordinates and weights

  ```
    TYPE(GaussIntegrationPoints_t) :: IntegStuff
    IntegStuff = GaussPoints( Element )
  ```

  The fields of the type GaussIntegrationPoints_t are

  ```
  INTEGER :: n
  REAL(KIND=dp) :: u(:), v(:), w(:), s(:)
  ```

  the integer value n is the number of points selected. The arrays u,v and w are the local coordinates of the points, and the array s contains the weights of the points. One may call the GaussPoints-routine with second argument,

  ```
    IntegStuff = GaussPoints( Element, n )
  ```

  if the default number of integration points for given element is not suitable.

  Inside the integration loop the function ElementInfo is called:

  ```
     TYPE(Element_t), POINTER :: Element
     TYPE(Nodes_t) :: Nodes
     REAL(KIND=dp) :: U,V,W,detJ, Basis(n), dBasisdx(n,3), ddBasisddx(n,3,3)

     stat = ElementInfo( Element, Nodes, U, V, W, detJ,  &
          Basis, dBasisdx, ddBasisddx, .FALSE. )
  ```

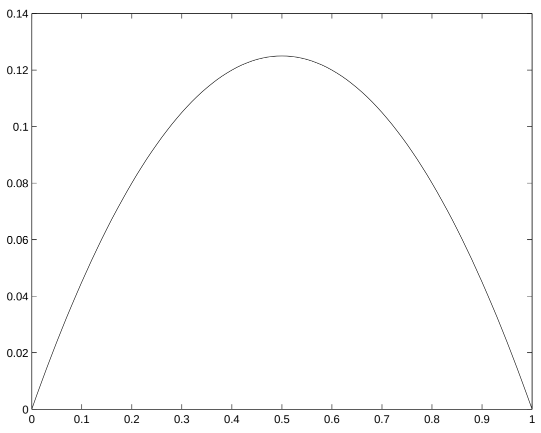

Figure 9.1: Solution of the Poisson Equation.

This routine returns determinant of the element jacobian (detJ), basis function values (Basis(n)), basis function global derivative values (dBasisdx(n,3)), basis function second derivative values (ddBasisddx(n,3,3)). The second derivatives are only computed if the next logical flag is set to true. All the values are computed at the point U,V,W inside element defined by structures Element and Nodes.

Refer to the code for more details.

- Set boundary conditions. Here only dirichlet boundary conditions are used. These may be set by using the utility routine SetDirichletBoundaries.
- Solve the system by calling utility routine SolveSystem.

Results

In the elmerpost file there is a variable called Potential which contains the solution of this simple example. See figure 9.1

Tutorial 10

Volume flow boundary condition

Directory: FlowLinearRestriction
Solvers: FlowSolve, SolveWithLinearRestriction
Tools: Editor, Fortran 90 compiler, ElmerGrid
Dimensions: 2D, Transient

Case definition

This tutorial gives an example how to use SolveWithLinearRestriction. It also describes how to execute own functions before the original system is solved. In order to understand the case reader should be familiar with compressed row storage matrices and Elmer basics. This tutorial gives only the guidelines and reader is advised to read the files in order to get more through understanding.

We simulate the flow of incompressible fluid in a pipe. The pipe has a length of 5 and a width of 1. On the left end we want to describe a certain time dependent volume flow. In other words, we don't want to describe the velocity field here but we want the velocity field be such that it transports certain amount of volume in time interval. We could integrate the correct volume flow, but let's now approximate it to make the more important aspects more visible. Our approximation here is that the volume flow is proportional to average velocity on the edge i.e.

$$\frac{1}{N}\sum_{i=1}^{N} u_i = \frac{volume}{time} \tag{10.1}$$

Here u_i are the nodal velocities parallel to the pipe on the left edge and N is the number of nodes on the left edge. We want to set a nicely scaled sinusoidal volume flow on the edge, which leads to

$$\sum_{i=1}^{N} u_i = 10N\sin(2\Pi t) \tag{10.2}$$

This equation we can (easily) force with Lagrange multiplier.

Solution procedure

First we make a uniform mesh of 800 four-node quadrilaterals with command

```
ElmerGrid 1 2 mflow
```

Next we construct the solver input file. Header is simply

```
Header
  Mesh DB "." "mflow"
End
```

The simulation block is also very simple. Here we need to define the time stepping method and timescale.

```
Simulation
  Coordinate System = Cartesian 2D

  Simulation Type = Transient
  Steady State Max Iterations = 1

  Timestepping Method = BDF
  BDF Order = 1

  Timestep Sizes = 0.02
  Timestep Intervals = 100

  Output Intervals = 1

  Output File = "mflow.result"
  Post File = "mflow.ep"
End
```

The body, material and equation blocks are as usual. The material parameters, of course, have affect on the solution and interested reader is encouraged to modify these values and recalculate the solution.

```
Body 1
  Material = 1
  Equation = 1
End

Material 1
  Density = 3.0
  Viscosity = 0.1
End

Equation 1
  Navier-Stokes = TRUE
  Active Solvers(1) = 1
End
```

The solver block has the usual Navier-Stokes keywords and two keywords for volume flow boundary. The `Before Linsolve` keyword defines binary file and function that is called before the system is solved. This function we must write and compile and we will come to it shortly. The following keyword, `Export Lagrange Multiplier`, states that we are not interested in the value of the Lagrange multiplier and it is therefore not saved.

```
Solver 1
  Equation = Navier-Stokes
  Stabilize = True

  Before Linsolve = "./AddMassFlow" "AddMassFlow"
  Export Lagrange Multiplier = Logical FALSE

  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Preconditioning = ILU1
  Linear System Max Iterations = 500
  Linear System Scaling = False
  Linear System Convergence Tolerance = 1.0e-8
  Nonlinear System Max Iterations = 15
  Nonlinear System Convergence Tolerance = 1.0e-8
  Nonlinear System Newton After Tolerance = 1.0e-4
  Nonlinear System Newton After Iterations = 8
  Nonlinear System Relaxation Factor = 1.0

  Steady State Convergence Tolerance = 1.0e-7
End
```

In boundary conditions we state that both parallel and perpendicular velocities are zero on the pipe sides and on both edges the perpendicular velocity is zero. Here we also define the number tags for the boundaries. The tag 2 is assigned to boundary that has number 4 in grd-file, which is the left edge of the pipe. To this tag number 2 we shall refer in our AddMassFlow-function.

```
Boundary Condition 1
  Target Boundaries(2) = 1 3
  Velocity 1 = 0.0
  Velocity 2 = 0.0
End

Boundary Condition 2
  Target Boundaries = 4
  Velocity 2 = 0.0
End

Boundary Condition 3
  Target Boundaries = 2
  Velocity 2 = 0.0
End
```

AddMassFlow function

Here we shall only give some rough guidelines of the function, for more information check the code. This function creates the constraint matrix and RHS that forces the equation mentioned above. Then it calls SolveWithLinearRestriction to solve the system. The constraint matrix is actually only a row-vector and the RHS is only one value.

- The function parameters are defined in Elmer so you shouldn't change them.
- First we set a pointer to EMatrix-field of the given system matrix. If the pointed matrix is not yet allocated, calculate the number of nodes on the edge we want to define the volume flow. This gives us the number of non-zeros in our constraint matrix and we can allocate the matrix.
- Set the rows, cols and diag -fields of the matrix. This sets the non-zeros on their right places in the constraint matrix.
- Set all values of the constraint matrix to unity.
- Calculate the RHS-value. The current time was checked in the beginning of the function, so this is possible.
- Call SolveWithLinearRestriction
- Return 1 which tells the ElmerSolver that the system is already solved.

The function is the compiled with command

```
elmerf90 -o AddMassFlow AddMassFlow.f90
```

Here it is assumed that the source file name is AddMassFlow.f90.

Results

Just say `ElmerSolver` and you should get the solution in few minutes. The velocity perpendicular to the pipe is practically zero and the velocity parallel to the pipe is an example of Womersley velocity profile [1]. An interesting feature of this velocity profile is that on some time steps the fluid flows to both directions, see figure 10.1.

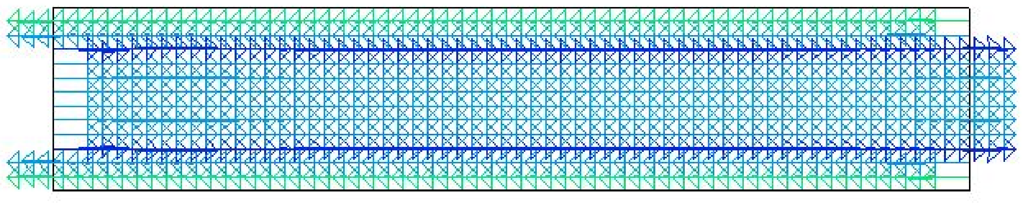

Figure 10.1: Solution of the velocity field. Note the flow to both directions.

[1]J.Physiol (1955) 127, 553-563

Tutorial 11

Streamlines

Directory: FlowStreamlines
Solvers: StreamSolver, FlowSolve
Tools: ElmerGrid, editor
Dimensions: 2D

Case definition

The case definition is the same as in the incompressible flow passing a step. The mathematical definition of the stream function ψ is

$$u = \frac{\partial \psi}{\partial y}, \quad v = -\frac{\partial \psi}{\partial x}. \tag{11.1}$$

where u, v are the velocity components in x, y geometry. For more info check Elmer Models Manual.

Solution Procedure

First we create a mesh with ElmerGrid. The mesh is defined in `step.grd` and it is created with command

```
ElmerGrid 1 2 step
```

You may need to compile the StreamSolver yourself. If the Elmer environment is successfully setup the compilation command should look like the following lines,

```
elmerf90 -o StreamSolver StreamSolver.f90
```

The solver input file `streamlines.sif` starts with the definition of the mesh directory.

```
Header
  Mesh DB "." "step"
End
```

The simulation uses 2D Cartesian geometry and searches a Steady State. There is no coupled solvers so only one iteration is needed. Numerical results are written to file `streamlines.result` and ElmerPost file is `streamlines.ep`.

```
Simulation
  Coordinate System =  Cartesian 2D
  Coordinate Mapping(3) = 1 2 3

  Simulation Type = Steady
  Steady State Max Iterations = 1
  Output Intervals = 1
  Post File = "streamlines.ep"
  Output File = "streamlines.result"
End
```

There is just one body and it uses equation 1 and is of material 1.

```
Body 1
  Equation = 1
  Material = 1
End
```

The equation block states that we use Solvers 1 and 2 to solve the problem and that we use Navier-Stokes equations.

```
Equation 1
  Active Solvers(2) = 1 2
  Navier-Stokes = True
End
```

In material block we define the density and the viscosity of the fluid.

```
Material 1
  Density = 1
  Viscosity = 0.01
End
```

Solver 1 is for the Navier-Stokes equations. Here we give the linear system solver [1] and convergence critérions for linear, nonlinear and steady state solution of the Navier-Stokes equations.

```
Solver 1
  Equation = "Navier-Stokes"
  Stabilize = True

  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Max Iterations = 500
  Linear System Convergence Tolerance = 1.0e-8
  Linear System Preconditioning = ILU1

  Nonlinear System Convergence Tolerance = 1.0e-6
  Nonlinear System Max Iterations = 15
  Nonlinear System Newton After Iterations = 8
  Nonlinear System Newton After Tolerance = 1.0e-4
  Nonlinear System Relaxation Factor = 1.0

  Steady State Convergence Tolerance = 1.0e-6
End
```

Then the solver for streamlines.

- Name of the equation. This may be what ever you like.
- Name of the binary file and the subroutine. If you compiled the StreamSolver yourself, then you may need to change this to `Procedure = "./StreamSolver" "StreamSolver"`.
- Name of the variable. This may be what ever you like.

[1] Biconjugate gradient method with incomplete LU preconditioning

- Stream function is scalar, so the degree of freedom is 1.

Next set of keywords is for the StreamSolver. More info on keywords is in the Elmer Models Manual.

- Name of the flow field variable. The name of the FlowSolves variable is FlowSolution.
- Global number of the offset node. 1 is always a safe choice.
- Shift the smallest value to zero.
- Scale the maximum value to 1.
- Use the normal stream function i.e. don't use Stokes stream function.

Then we define the linear system solver and convergence criterions.

```
Solver 2
  Equation = "StreamSolver"
  Procedure = "StreamSolver" "StreamSolver"
  Variable = "StreamFunction"
  Variable DOFs = 1

  Stream Function Velocity Variable = String "Flow Solution"
  Stream Function First Node = Integer 1
  Stream Function Shifting = Logical TRUE
  Stream Function Scaling = Logical TRUE
  Stokes Stream Function = Logical FALSE

  Linear System Solver = Iterative
  Linear System Iterative Method = BiCGStab
  Linear System Max Iterations = 500
  Linear System Convergence Tolerance = 1.0e-8
  Linear System Preconditioning = ILU1

  Steady State Convergence Tolerance = 1.0e-6
End
```

Finally we give the boundary conditions. The condition 1 is for the lower and upper side of the step (Γ_1,Γ_2,Γ_3,Γ_5 in case definition). Here both velocities are zero. The condition 2 is for the output edge (Γ_4). Here vertical velocity is zero. The condition 3 is for the input edge (Γ_6). Here horizontal velocity is 1 and vertical velocity is zero.

```
Boundary Condition 1
  Target Boundaries = 1
  Velocity 1 = 0
  Velocity 2 = 0
End

Boundary Condition 2
  Target Boundaries = 2
  Velocity 2 = 0
End

Boundary Condition 3
  Target Boundaries = 3
  Velocity 1 = 1
  Velocity 2 = 0
End
```

Results

Problem is solved with command `Solver`. The results are then viewed with ElmerPost. In figure 11.1 are some contour lines of the stream function. These are also flows streamlines. The contour values are manually selected to get a nice picture. Note the swirl after the step.

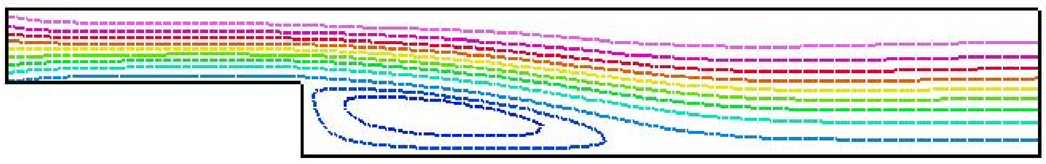

Figure 11.1: The streamlines of the flow.

Tutorial 12

Timoshenko beam model of a cantilever

Directory: TimoshenkoBeamCantilever
Solvers: BeamSolver3D
Tools: ElmerGrid, Python, Gmsh
Dimensions: 3D, Steady-state

Case definition

In this tutorial, the geometry of a basic cantilever beam is created with Gmsh via its Python interface and simulated with Timoshenko beam elements when the applied force is a constant pressure load. Different numbers of elements for the cantilever are used to demonstrate divergence from the analytical solution for low number of elements. The workflow will be implemented in python and it demonstrates shortly how multiple simulations with different parameters can be created from a pre-existing sif-file.

Requirements

Before you do this tutorial, you have to do the following things:

- install the required python packages (e.g. via pip install) found in requirements.txt. If you do not get the exact versions, that should not matter.
- replace in the file create_geometry.py the variable of the ElmerGrid path
- replace in the file total_workflow.py the variable of the ElmerSolver path

We expect the reader to have basic Python skills, but no advanced knowledge except knowing how to run a script and the ability to read the code/syntax.

Workflow

If you want to see the entire workflow, simply run python total_workflow.py. For running its individual parts, run the single subroutines:

- **create_geometry.py:** Create the geometry via Gmsh and convert it to an Elmer usable format via ElmerGrid. Optional: you can view the geometry via the Gmsh graphical user interface.
- **create_sif.py:** Take the existing constant_pressure_load.sif and modify it for different mesh resolutions for postprocessing files not to overwrite each other.
- **plot_results.py:** Collect simulations and display them.

Geometry Creation

We import the packages and functions that we need for the task. We need gmsh to create and mesh the geometry, numpy for some basic array functionality (although it could be done without it) and from the subprocess package of the basic libraries of Python the function run to use Elmergrid:

```
from subprocess import run
```

```
import numpy as np
```

```
import gmsh
```

Enter the path to Elmergrid. If it was added to the system path already, it looks like this

```
elmergrid = r"ElmerGrid"
```

We want to do the geometry creation repeatedly, so we define it as a function of the characteristic mesh length which determines the number of elements in the model and optionally allow the Gmsh graphical user interface (GUI) to be opened to have a final look at the geometry.

```
def create_geo(lc=1e-2, gui=False):
```

When starting to work on a Gmsh model, you first have to start Gmsh

```
gmsh.initialize()
```

and create the empty object

```
gmsh.model()
```

that you will later fill with nodes, lines, surfaces etc. that are then meshed. We now create the start and endpoint of our beam at the positions (0,0) and (1,0).

```
i = 0
for x,y in zip([0.,1.],[0.,0.]):
    i = i + 1
    gmsh.model.geo.addPoint(x, y, 0., lc, i)
```

Notice that every entity in Gmsh has a tag or index (here i). Indices start always from 1. Next we create a line to connect our nodes. This will later form our physical beam.

```
gmsh.model.geo.addLine(1, 2, 1)
```

Before geometric entities can be meshed or manipulated outside the standard Gmsh kernel, they must be synchronized with the Gmsh model creating/updating relevant internal data structures. Synchronizations can be called at any time but they are expensive, so minimize the number of synchronization points for large complicated geometries.

```
gmsh.model.geo.synchronize()
```

In Gmsh, entities are grouped together into Physical Groups to later assign material properties, boundary conditions etc. to these. Usually also only the physical groups are explicitly meshed. For each group the dimension of the objects which are to be grouped is first mentioned (0 points, 1 lines, 2 surfaces, 3 bodies), then a list of the tags/ids of the objects and finally a name. Here we define our beam and the point where we apply the boundary condition:

```
gmsh.model.addPhysicalGroup(1, [1],name = "beam")
```

```
gmsh.model.addPhysicalGroup(0, [1], name = "anchor")
```

We then generate a mesh where the dimensionality of the intended mesh must given. As we intend to create line/beam elements, it is 1.

```
gmsh.model.mesh.generate(1)
```

As we want to study mesh convergence, we need to know the number of elements. We iterate over all existing Physical Groups, find the ones that are lines by checking the group dimension to be one and count the number of elements tags.

```
for phys in gmsh.model.getEntities():
   if phys[0] == 1:
      eltyps, eltags, ndtags = gmsh.model.mesh.getElements(phys[0],
                                                          phys[1])
      nelements = eltags[0].shape[0]
```

We save the number of elements to disk in a simple csv file

```
np.savetxt("cantilever-0_nelem.csv".format(str(lc)), np.array([nelements]))
```

and save the mesh to the disk as well:

```
filename = "cantilever-0.msh".format(str(lc))
gmsh.write(filename)
```

Gmsh automatically infers the format of the file by its ending. Optionally one can start the graphical user interface of Gmsh to inspect the geometry before finishing:

```
if gui:
   gmsh.fltk.run()
```

We clear out the geometry

```
gmsh.clear()
```

and close Gmsh

```
gmsh.finalize()
```

The latter two steps must always be taken as otherwise the geometry is considered open and will linger in the background and may cause strange errors as tags of entities are still assigned and must not be reassigned as it will cause an error. We now need to convert the mesh in Gmsh format to an Elmer friendly one. We do this with ElmerGrid by calling the shell from python

```
run([elmergrid, "14", "2", filename], shell=True, check=True)
```

The check flag causes an error if the statement returns an error in the shell. If not present, this program would fail without raising an error. If you encounter problems with the latter statement, you have probably not entered the ElmerGrid path correctly. If for some reason this line does not work despite your best efforts, just comment it out

```
#run([elmergrid, "14", "2", filename], shell=True, check=True)
```

and run ElmerGrid by yourself

```
ElmerGrid 14 2 cantilever.msh
```

by adapting the file name accordingly. We now want to see whether this program works, so we enter at the bottom of the program this standard expression

```
if __name__ == '__main__':
```

that ensures, that this routine is only executed when you explicitly call this program file via

```
python create_geometry.py
```

and not if you call the function create_geo from another program. We now perform a simple visual check that everything runs fine

```
   lc = 1e0
   create_geo(lc, True)
```

which should result in a geometry with one beam element. Do not get confused as Gmsh also counts the nodes as elements, so its report will mention three. The GUI output should look like Fig. 12.1.

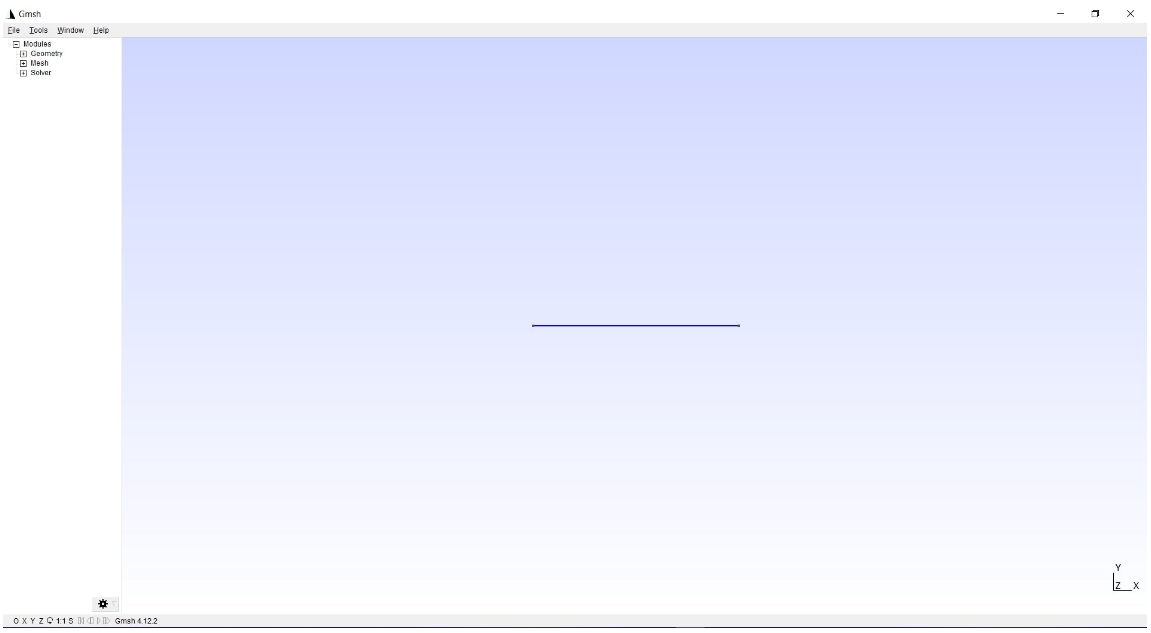

Figure 12.1: Output of the Gmsh GUI.

Solution Procedure

We tell Elmer to find the geometric information in the directory cantilever

```
Header
  Mesh DB "." "cantilever"
End
```

and specify the coordinate system, some general output options and the output file in which displacements will be stored:

```
Simulation
  Max Output Level = 5
  Coordinate System = Cartesian 3D
  Simulation Type = Steady
  Output Intervals = 1
  Steady State Max Iterations = 1
  Post File = "cantilever.vtu"
End
```

We will have later to change this automatically. Now we mention which equations, materials and body forces act on body 1

```
Body 1
  Equation = 1
  Material = 1
  Body Force = 1
End
```

which is our only geometric object here. The material data and the geometric data of the beam have to be entered. We make a simplification here, by assuming that we have a beam shape that does not depend on the orientation of the beam with regards to loading, in other words a cylindrical beam:

```
Material 1
```

```
 Youngs Modulus = Real 2.0e-1
 Shear Modulus = Real 1.0
 Second Moment of Area 2 = Real 1.0
 Second Moment of Area 3 = Real 1.0
 Cross Section Area = Real 1.0
 Torsional Constant = Real 1.0
 Density = 2700.0
End
```

The density is not needed here. We specify the constant pressure load as body force acting in y-direction

```
Body Force 1
  Body Force 1 = 0.0
  Body Force 2 = 1.0e-2
  Body Force 3 = 0.0
End
```

and set up the beam solver.

```
Equation 1 :: Active Solvers(1) = 1

Solver 1
  Equation = "Timoshenko Beam Equations"
  Procedure = "BeamSolver3D" "TimoshenkoSolver"

  Linear System Solver = "Direct"
End
```

We can choose the most simple direct solver here as our system is quite small. Solver 2 is just a way to save some scalar data in the file cantilever.dat of the free end to later compare it with the analytic solution. We will have to change the name of the file later as well.

```
Solver 2
  Equation = "Save Scalars"
  Exec Solver = After Timestep
  Procedure = "SaveData" "SaveScalars"
  Filename = cantilever.dat
  Variable 1 = U 1
  Variable 2 = U 2
  Variable 3 = U 3
  Variable 4 = Theta 1
  Variable 5 = Theta 2
  Variable 6 = Theta 3
  Save Points(1) = 2
End
```

We fix the first node in place

```
Boundary Condition 1
  Target Nodes(1) = 1
  U 1 = Real 0.0
  U 2 = Real 0.0
  U 3 = Real 0.0
  Theta 1 = Real 0.0
  Theta 2 = Real 0.0
  Theta 3 = Real 0.0
End
```

and are done with the simulation. No other boundary conditions need to be mentioned as everything else is free. Run the simulation by entering

```
ElmerSolver constant_pressure_load.sif
```

in the command line, and you should find in the directory cantilever a file called cantilever_t0001.vtu which you can display with ParaView (Fig. 12.2).

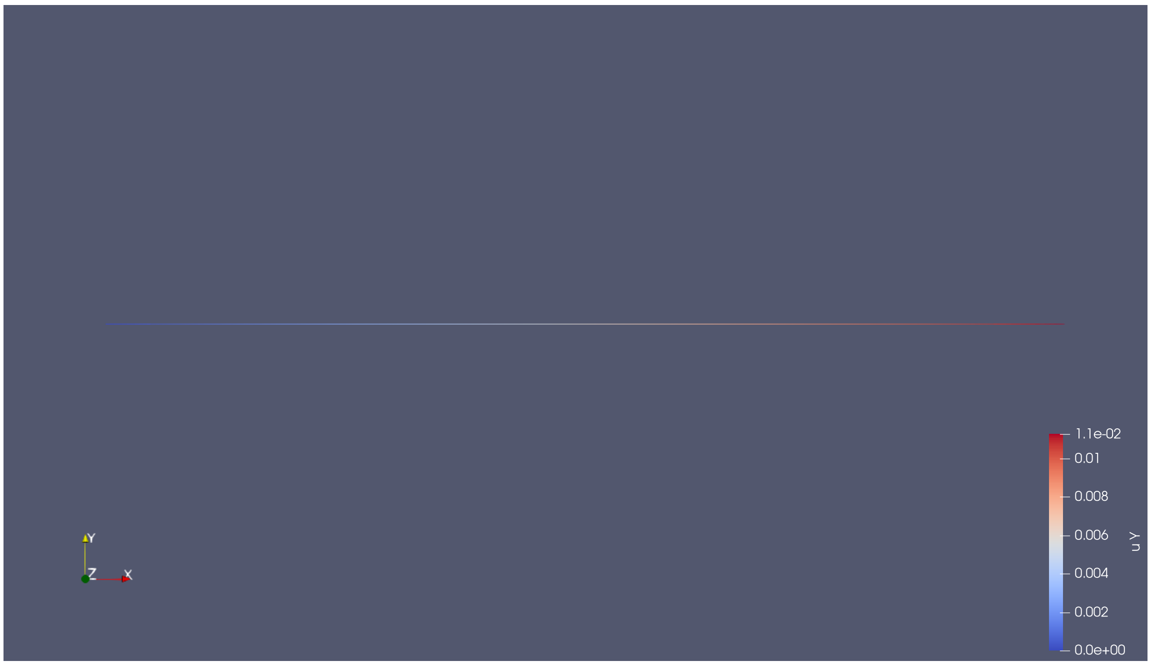

Figure 12.2: Output of ParaView.

Creating New sif-Files

We do not need to import any packages here, as everything can be done with basic Python functions. We want to update a specific sif file ad replicate it with by changing a few lines and leave 99 % of the file untouched. Therefore as input for our function we have that file's name, a list of lines that we want to exchange and the replacements of these specific lines. The characteristic mesh length scale is there to change the name of the file.

```
def write_new_sif(file,
                  search_strings,
                  replacements,
                  lc):
```

We read the already existing sif file and all its lines.

```
    with open(file, 'r') as f:
        lines = f.readlines()
```

We open a new file whose name is a modified version of the already existing sif file.

```
   with open(file.split(".")[0]+"-"+str(lc)+".sif", "w") as f:
```

and iterate over all its lines.

```
      for line in lines:
```

We check each line whether a string expression is contained in that line. If not, we copy the line. Otherwise we replace it by the corresponding entry in the replacement list

```
            flags = [string in line for string in search_strings]
            if any(flags):
               ind = [i for i,flag in enumerate(flags) if flag][0]
               f.write(replacements[ind].format(str(lc)))
            else:
               f.write(line)
    return
```

We quickly test our function with the sif file for the previously generate geometry.

```
if __name__ == '__main__':
    lc = 1e0
```

We exchange the lines that load the geometry, the output file for ParaView and the file containing the displacements of the free end with names changed appropriately for different characteristic mesh length scales.

```
    write_new_sif(file = "constant_pressure_load.sif",
                  search_strings = ['  Mesh DB "." "cantilever"',
                                    '  Post File = "cantilever.vtu"',
                                    '  Filename = cantilever.dat'],
                  replacements = ['  Mesh DB "." "cantilever-0"\ n',
                                  '  Post File = "cantilever-0.vtu"\ n',
                                  '  Filename = cantilever-0.dat\ n'],
                  lc = lc)
```

Check whether the lines have been changed accordingly and move on.

Plotting Results

We import numpy for some basic array functionality and matplotlib as our standard tool for creating plots in python.

```
import numpy as np
import matplotlib.pyplot as plt
```

We define our function for a collection of characteristic length scales and enable optional figure saving as we do not want to dump every figure on our disc.

```
def plot_displacements(lcs,
                       save_fig=False):
```

For convenience we select a single font size for things like axis labels, etc.

```
    font = 14
```

Set up two lists as collection bags for the number of elements and the displacements and start looping over the characteristic length scales

```
    nelems = []
    displ = []
    for lc in lcs:
```

We read the number of elements from our previously created csv file when making the geometry, directly put them into the list and do the same for the displacements created by Elmer during the solution process.

```
        nelems.append(np.loadtxt("cantilever-0_nelem.csv".format(str(lc))))
        displ.append(np.loadtxt("cantilever-0.dat".format(str(lc)))[[4,-1]])
```

After looping we merge the displacements into one array by stacking the list entries on top of each other to receive an array with a variable number of rows and two columns (as we only read two types of displacement). We then split the displacements in two arrays for readability reasons, namely the deflection $w$ and the rotation Θ:

```
    displ = np.vstack(displ)
    w = displ[:,0]
    theta = displ[:,1]
```

We create a figure with one row and two columns

```
    fig, axs = plt.subplots(1,2,figsize=(12,8))
```

and fill each plot with the corresponding element vs. displacement data plotted as lines (as an alternative one could use "scatter" instead of plot to have data points instead of lines).

```
    axs[0].plot(nelems,w)
    axs[1].plot(nelems,theta)
```

We now calculate the analytical solutions and plot them as horizontal red lines to which our simulations should converge with increasing number of elements

```
    A,I,G,L,E,f = 1,1,1,1,0.2,1e-2
    axs[0].axhline(y=L**4 * f * ( 1 + 4*E*I / (G*A*L**2) ) / (8*E*I),
                   color = "r", linestyle="--")
    axs[1].axhline(y=L**3 * f / (6*E*I),
                   color = "r", linestyle="--")
```

We transform the x-axis to a logarithmic scale as the number of elements may give a large interval and is nonzero

```
    axs[0].set_xscale("log")
    axs[1].set_xscale("log")
```

and adapt the limits of the y scale,

```
    axs[0].set_ylim(8e-3,1.2e-2)
    axs[1].set_ylim(8e-3,8.5e-3)
```

create axis labels

```
    axs[0].set_xlabel(r"elements",fontsize=font)
    axs[1].set_xlabel(r"elements",fontsize=font)
    axs[0].set_ylabel(r"deflection $w$",fontsize=font)
    axs[1].set_ylabel(r"rotation $\ theta$",fontsize=font)
```

and allow for the option to save our figure to the disk.

```
    if save_fig:
        plt.savefig("nr-elements-displacements.pdf", format="pdf",
                    bbox_inches="tight")
```

We now cause our figure to be shown on the screen as so far you should not have seen anything

```
    # show the plot as pop up window
    plt.show()
```

and mark the end of the function by return

```
    return
```

Strictly speaking the latter is unnecessary as nothing is returned, but helps with readability of the code. As our usual exercise we test our program and you should end up with a single plotted data point.

```
if __name__ == '__main__':
    plot_displacements(lcs = np.array([1e0]))
```

Total Workflow for Mesh Convergence Study

In this section we piece together all our previous programs. We need subprocess.run again for calling Elmer-solver and numpy for basic array routines.

```
from subprocess import run

from numpy import array
```

We import all our previously created functions

```
from create_geometry import create_geo
from create_sif import write_new_sif
from plot_results import plot_displacements
```

and add our path to ElmerSolver.

```
elmersolver = "ElmerSolver"
```

We now write the main routine of our solver, create some array with a collection of characteristic length scales that we would like to use and start iteration. (If lc is larger than 1, you will end up with just 1 element, so lc = 1 is the sensible upper bound here)

```
if __name__ == '__main__':
    lcs = array([1e0,1e-1,1e-2,1e-3])
    for lc in lcs:
```

We create the geometry and mesh it with Gmsh to convert it to an Elmer compatible file format

```
        create_geo(lc)
```

and create the corresponding sif file whose geometry input location and its result output files are changed to avoid overwriting previous results.

```
        write_new_sif(file = "constant_pressure_load.sif",
                      search_strings = ['  Mesh DB "." "cantilever"',
                                        '  Post File = "cantilever.vtu"',
                                        '  Filename = cantilever.dat'],
                      replacements = ['  Mesh DB "." "cantilever-0"\ n',
                                        '  Post File = "cantilever-0.vtu"\ n',
                                        '  Filename = cantilever-0.dat\ n'],
                      lc = lc)
```

We call Elmer solver to solve the system of equations

```
        run([elmersolver,
             "constant_pressure_load-0.sif".format(str(lc))],
            shell=True,
            check=True)
```

and continue to the next iteration or stop the iteration process. In the end we plot our accumulated results

```
    plot_displacements(lcs, True)
```

which should show a plot similar to Fig. 12.3.

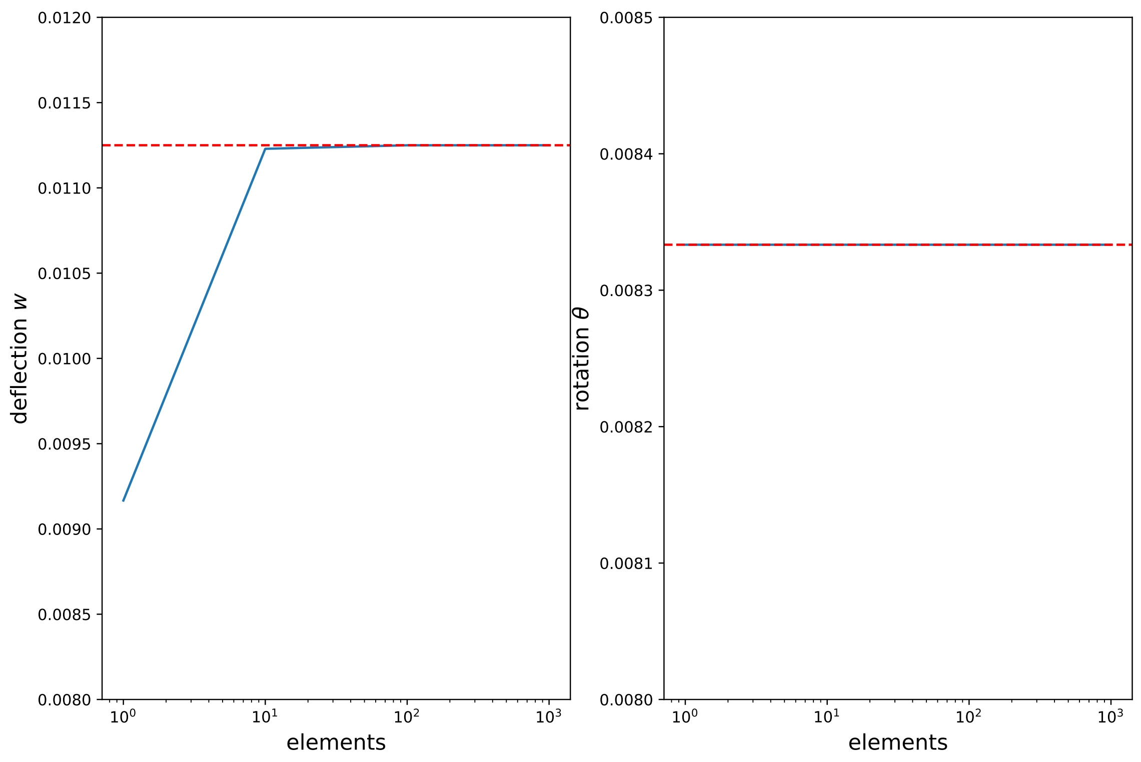

Figure 12.3: Collected results.

Concluding Remarks

So far this tutorial has presented a minor introduction to the use of the Timoshenko beam solver and applying Elmer in conjunction with Gmsh and also exemplified the usefulness of Python for the construction of workflows to study mesh convergence. For the inexperienced use, we would like to mention that mesh convergence is not usually done with the help of analytical solutions as they are only available in a few cases. Instead one typically analyses the development of the residuals and target outcomes of interest. It goes without saying that if the residuals do not decrease with increasing mesh refinement, you are not converging and you should check the physics and numerics of your problem.

Index