Elmer FEM

open source multiphysical simulation software

ElmerGUI Manual

Mikko Lyly and Saeki Takayuki

April 6, 2023

ElmerGUI Manual

About this document

The ElmerGUI Manual is part of the documentation of Elmer finite element software. Elmer may be used also without the graphical user interface but for new users ElmerGUI often provides the easiest path to Elmer.

The present manual corresponds to Elmer software version 9.0.

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

Copyright information

External contributions to the tutorials are welcome.

Contents

Chapter 1

Introduction

ElmerGUI is a graphical user interface for the Elmer software suite [1]. The program is capable of importing finite element mesh files in various formats, generating finite element partitioning for various geometry input files, setting up PDE-systems to solve, and exporting model data and results for ElmerSolver.

ElmerGUI can also automatically call Paraview. Previously also ElmerPost and internal VTK based postprocessors were once supported but these are gradually becoming obsolete.

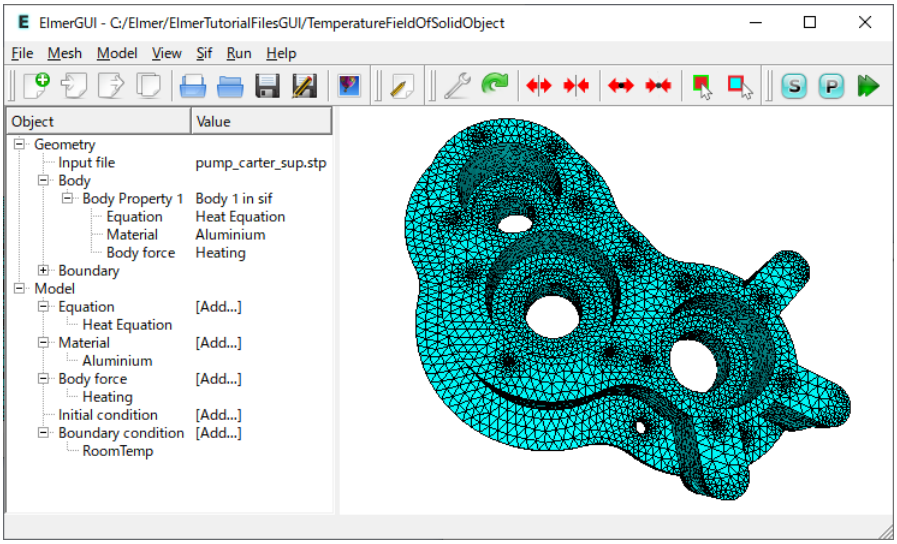

Figure 1.1: Main window of ElmerGUI.

The menus of ElmerGUI are programmable and it should be relatively easy to strip and customize the interface for proprietary applications. An example of customizing the menus is provided in appendix A.

ElmerGUI is also the interface to the parallel solver, `ElmerSolver_mpi`. The GUI hides from the user many operations that are normally performed from command line with various external tools related to domain decomposition, launching the parallel processes. This makes it possible to use ElmerSolver with multi-core processors, even on interactive desktop environments.

ElmerGUI relies on the Qt cross platform framework of Qt Company[4], and it uses the Qwt library by Josef Wilgen and Uwe Rathman[5] to plot scientific data. The CAD import features are implemented by the OCE library from Open CASCADE Community Edition developers[3] and Netgen[2] as finite element mesh generators.

Chapter 2

Installation from source

The source code of ElmerGUI is available from the Git repository hosted at GitHub. The GPL licensed source code may be downloaded by executing the command

```
git clone git://www.github.com/ElmerCSC/elmerfem
```

or

```
git clone https://www.github.com/ElmerCSC/elmerfem
```

This will retrieve the current development version of the whole Elmer-suite.

2.1 Linux

Before starting to compile, please make sure that you have the development package of Qt installed on your system (i.e., libraries, headers, and program development tools). Qt version 4.3 or newer is recommended. You may also wish to install Qwt 6, VTK version 8.2, and OCE 0.18, for additional functionality.

CMake can be used to generate the makefiles for compilation. The compilation of ElmerGUI is activated in the first place by setting the corresponding CMake variable as `-DWITH_ELMERGUI:BOOL=TRUE`. Other logical variables that affect the compilation of the program and that can similarly be set with `-D` are `WITH_QWT`, `WITH_VTK`, `WITH_OCC`, `WITH_PYTHONQT`, `WITH_MATC` and `WITH_PARAVIEW`. `WITH_QT5` is also required when compiling with Qt5 instead of Qt4.

Once the build process has finished, it suffices to set up the environment variable `ELMERGUI_HOME` and add it to `PATH`:

```
$ export ELMERGUI_HOME=/usr/local/bin
$ export PATH=$PATH:$ELMERGUI_HOME
```

The program is launched by the command

```
$ ElmerGUI
```

Chapter 3

Input files

3.1 Geometry input files and mesh generation

ElmerGUI is capable of importing finite element mesh files and generating two or three dimensional finite element partitioning for bounded domains with piecewise linear boundaries. It is possible to use one of the following mesh generators:

- ElmerGrid (built-in)
- Tetgen (optional)
- Netgen (built-in)

The default import filter and mesh generator is ElmerGrid. Tetgen is an optional module, which may or may not be available depending on the installation (installation and compilation instructions can be found from Elmer's source tree in `trunk/misc`)

An import filter or a mesh generator is selected automatically by ElmerGUI when a geometry input file is opened:

File $\rightarrow$ *Open...*

The selection is based on the input file suffix according to Table 3.1. If two or more generators are capable of handing the same format, then the user defined "preferred generator" will be used. The preferred generator is defined in

Mesh $\rightarrow$ *Configure...*

Once the input file has been opened, it is possible to modify the mesh parameters and remesh the geometry for better accuracy or computational efficiency. The mesh parameters can be found from $Mesh \rightarrow Configure...$. The control string for Tetgen has been discussed and explained in detail in Tetgen's user guide [6].

The mesh generator is reactivated from the Mesh menu by choosing

Mesh $\rightarrow$ *Remesh*

In case of problems, the meshing thread may be terminated by executing

Mesh $\rightarrow$ *Terminate meshing*

3.2 Elmer mesh files

An Elmer mesh consists of the following four text files (detailed description of the file format can be found from Appendix B):

Table 3.1: Input files and capabilities of the mesh generators.

Suffix	ElmerGrid	Tetgen	Netgen
.FDNEUT	yes	no	no
.grd	yes	no	no
.msh	yes	no	no
.mphtxt	yes	no	no
.off	no	yes	no
.ply	no	yes	no
.poly	no	yes	no
.smesh	no	yes	no
.stl	no	yes	yes
.unv	no	yes	no
.in2d	no	no	yes

```
mesh.header
mesh.nodes
mesh.elements
mesh.boundary
```

Elmer mesh files may be loaded and/or saved by opening the mesh directory from the File menu:

File → Load mesh...

and/or

File → Save as...

3.3 Project files

An ElmerGUI project consists of a project directory containing Elmer mesh files and an xml-formatted document `egproject.xml` describing the current state and settings. These files will be generated or updated when you save your project by choosing

File → Save project

This will save your project to the project directory you already specified. (If no project directory is specified yet, a window will appear to specify the project directory.) If you want to save your project in different directory, you can do that by choosing

File → Save project as...

Projects you already saved may be loaded by choosing

File → Load project...

or you can simply select one from

File → Recent projects

Note: Current version of ElmerGUI has a menu to start a new project

File → New project...

and by clicking the menu, a window will appear to specify project directory for the new project. You can also specify Elmer mesh directory or geometry input file you want to load for the project. Extra equation definition files can be also specified in the same window. This menu is just for convenience and previous style of workflow (i.e. open geometry input file, set conditions, then save project) still works.

The contents of a typical project directory are the following:

```
case.sif
egproject.xml
ELMERSOLVER_STARTINFO
mesh.boundary
mesh.elements
mesh.header
mesh.nodes
```

Chapter 4

Model definitions

4.1 Setup menu

The general setup menu can be found from

Model → *Setup...*

This menu defines the basic variables for the “Header”, “Simulation”, and “Constants” blocks for a solver input file. The contents of these blocks have been discussed in detail in the SolverManual of Elmer [1].

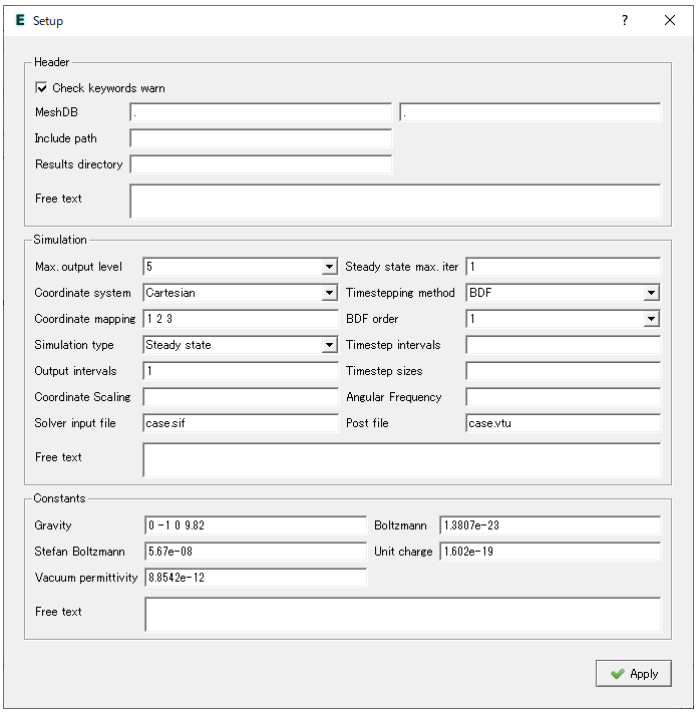

Figure 4.1: Setup Window.

4.2 Equation menu

The first “dynamical menu” constructed from the ElmerGUI definition files (see Appendix A) is

Model → Equation

This menu defines the PDE-system to be solved as well as the numerical methods and parameters used in the solution. It will be used to generate the "Solver" blocks in a solver input file.

A PDE-system (a.k.a "Equation") is defined in a Equation Window which shows up by choosing

Model → Equation → Add...

or clicking [Add...] label next to "Equation" item in Object Browser as shown in Figure 4.2.

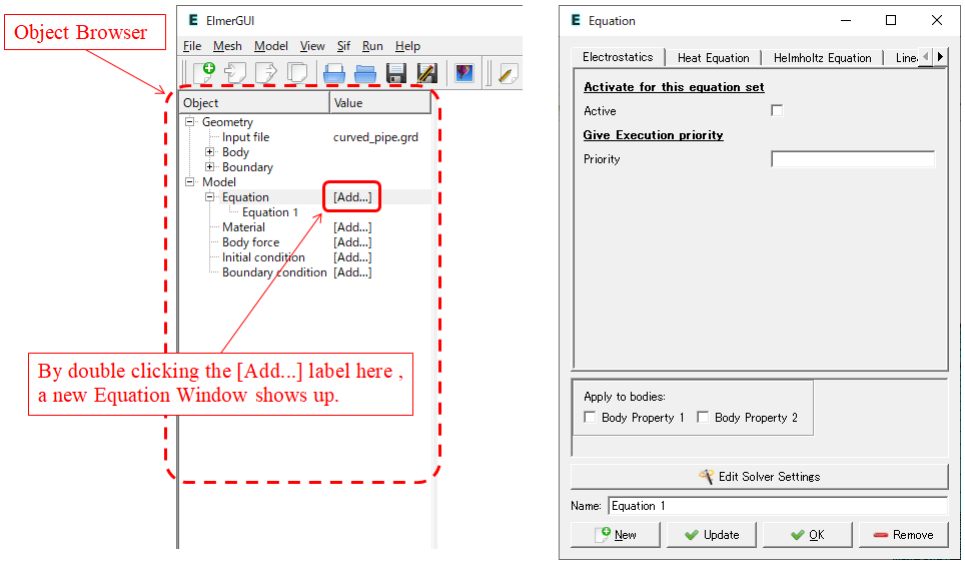

Figure 4.2: Adding a new Equation.

Once the PDE-system has been defined by activating the individual equations, the numerical methods and parameters can be selected and tuned in Solver Settings Window (Figure 4.3) which appears by pressing the "Edit Solver Settings" button. The name of the PDE-system is defined in the line edit box with label "Name". After pressing the OK-button, the equation remains visible and editable under the Model menu. It is also possible to show and edit the Equation Window by double clicking the name of the equation under "Equation" item in Object Browser.

If some of checkboxes in "Apply to bodies" in the Equation Window are checked, this equation is applied to these bodies. Another way to apply a equation to a body without using Equation Window is by holding down the SHIFT-key while double clicking one of its surfaces (or simply double clicking the surface while "Set body properties" button in the toolbar is pressed down). As shown in Figure 4.4, a Body Property Editor will then appear, listing all possible attributes that can be apply to the selection. The Body Property Editor will show up also by double clicking the body name under "Body" item in the Object Browser.

4.3 Material menu

The next menu is related to material and model parameters:

Model → Material

This menu will be used to generate the "Material" blocks in a solver input file.
In order to define a material parameter set and apply it to bodies, choose

Model → Material → Add...

or clicking [Add...] label next to "Material" item in Object Browser. This will open a Material Window (Figure 4.5).

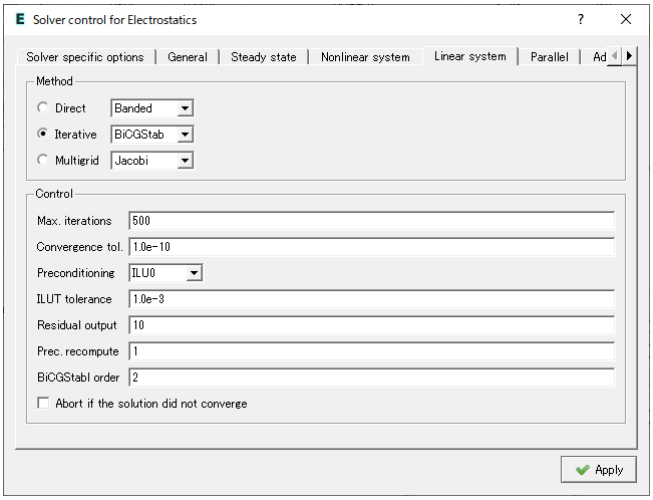

Figure 4.3: Solver Settings Window.

Again, it is possible to apply the material to a body by holding down the SHIFT-key while double clicking one of its surfaces (or simply double clicking the surface while "Set body properties" button in the toolbar is pressed down). A Body Property Editor will then appear, listing all possible attributes that can be apply to the selection. The Body Property Editor will show up also by double clicking the body name under "Body" item in the Object Browser.

Note: The value of density should always be defined in the "General" tab. This field should never be left undefined.

Note: If you set focus in a line edit box of a dynamical menu and press Enter, a small text edit dialog will pop up. This allows the input of more complicated expressions than just constants. As an example, go to *Model* → *Material* and choose *Add...* Place the cursor in the "Heat conductivity" line edit box of "Heat equation" and press Enter. You can then define the heat conductivity as a function of temperature as a piecewise linear function. An example is show in Figure 4.6. In this case, the heat conductivity gets value 10 if the temperature is less than 273 degrees. It then rises from 10 to 20 between 273 and 373 degrees, and remains constant 20 above 373 degrees.

If the user presses SHIFT and F1, a tooltip for the active widget will be displayed as shown in Figure 4.7.

4.4 Body force menu

The next menu in the list is

Model → *Body force*

This menu is used to construct the "Body force" blocks in a solver input file.
Again, choose

Model → *Body force* → *Add...*

to define a set of body forces and apply it to the bodies. It is also possible to click [Add...] label next to "Body force" item in Object Browser. This will open a Body Force Window (Figure 4.8).

4.5 Initial condition menu

The last menu related to body properties is

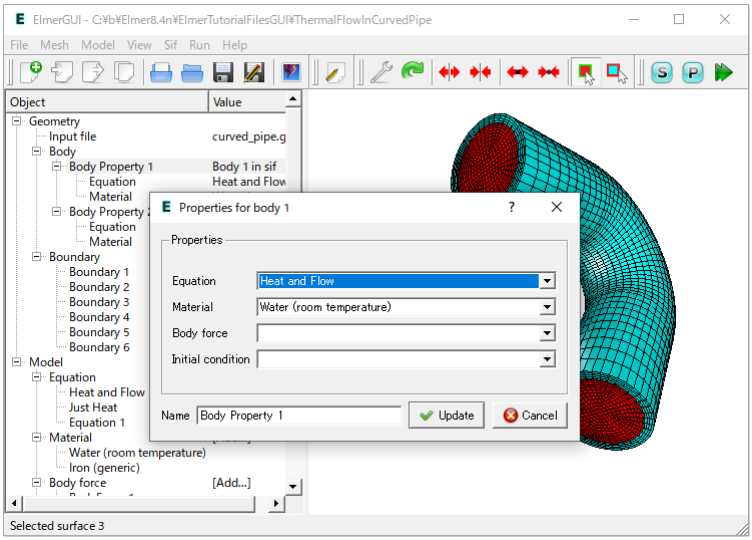

Figure 4.4: Body Property Editor is activated by holding down the SHIFT key while double clicking a surface.

Model → *Initial condition*

Once again, choose

Model → *Initial condition* → *Add...*

to define a set of initial conditions and apply it to the bodies. It is also possible to click [Add...] label next to "Initial condition" item in Object Browser. This will open an Initial Condition Window (Figure 4.9).

This menu is used to construct the "Initial condition" blocks in a solver input file.

4.6 Boundary condition menu

Finally, there is a menu entry for setting up the boundary conditions:

Model → *Boundary condition*

Choose

Model → *Boundary condition* → *Add...*

to define a set of boundary conditions and apply them to boundaries. It is also possible to click [Add...] label next to "Boundary condition" item in Object Browser. This will open a Boundary Condition Window (Figure 4.10).

It is possible to apply a boundary condition to a boundary by holding down the `Alt` or `AltGr`-key while double clicking a surface or edge. (or simply double clicking the surface or edge while "Set boundary properties" button in the toolbar is pressed down). As shown in Figure 4.11, a Boundary Property Editor will appear, listing all possible conditions that can be applied to the selection. The Boundary Property Editor will show up also by double clicking the boundary name under "Boundary" item in the Object Browser.

Choose a condition from the combo box and finally press `Ok`.

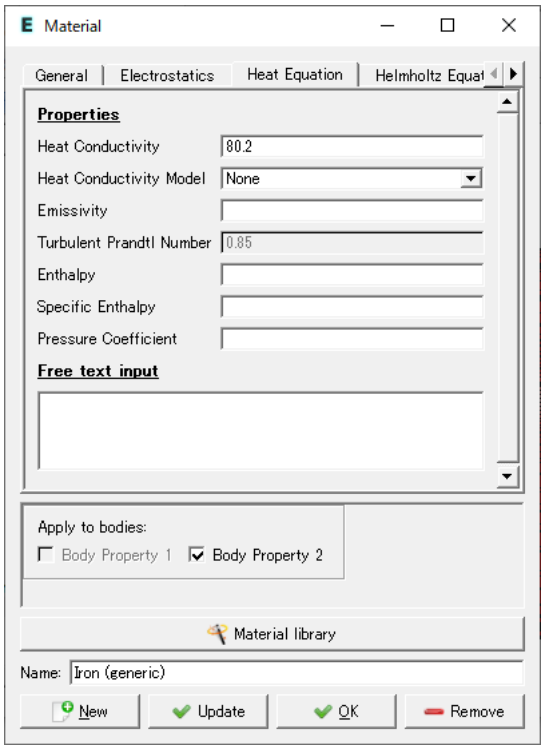

Figure 4.5: Material Window.

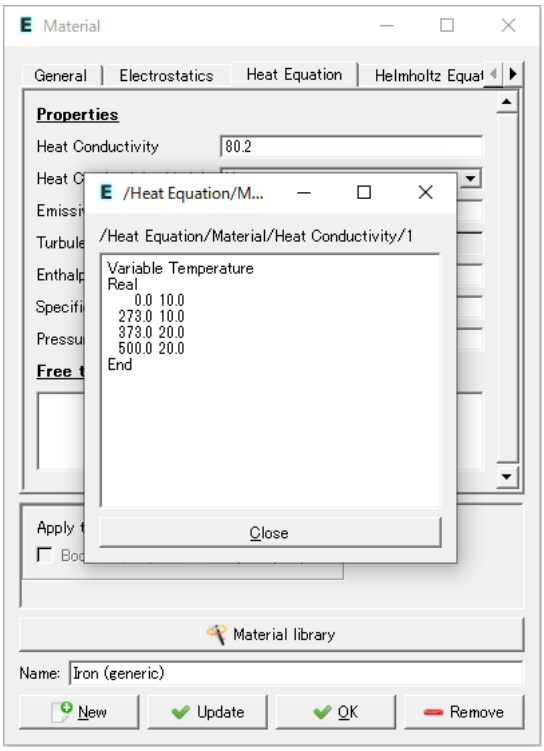

Figure 4.6: Text edit extension of a line edit box is activated by pressing Enter.

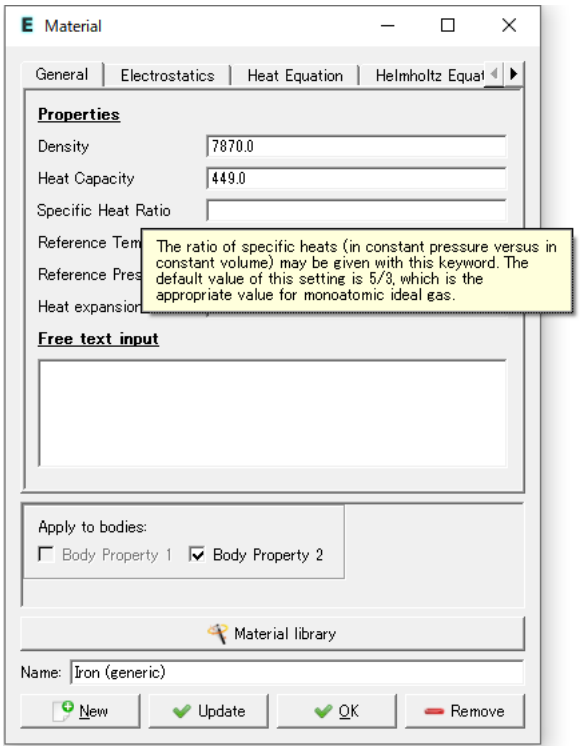

Figure 4.7: Tooltips are shown by holding down the SHIFT and F1 keys.

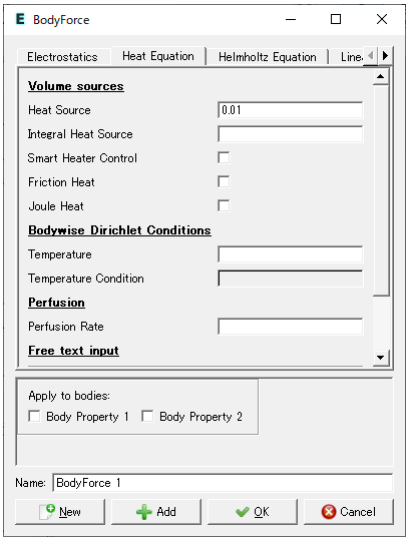

Figure 4.8: Body Force Window.

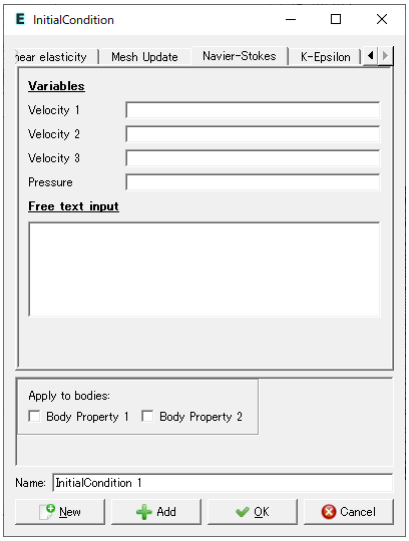

Figure 4.9: Initial Condition Window.

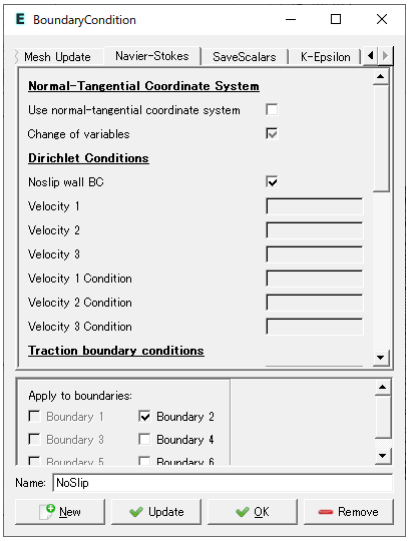

Figure 4.10: Boundary Condition Window.

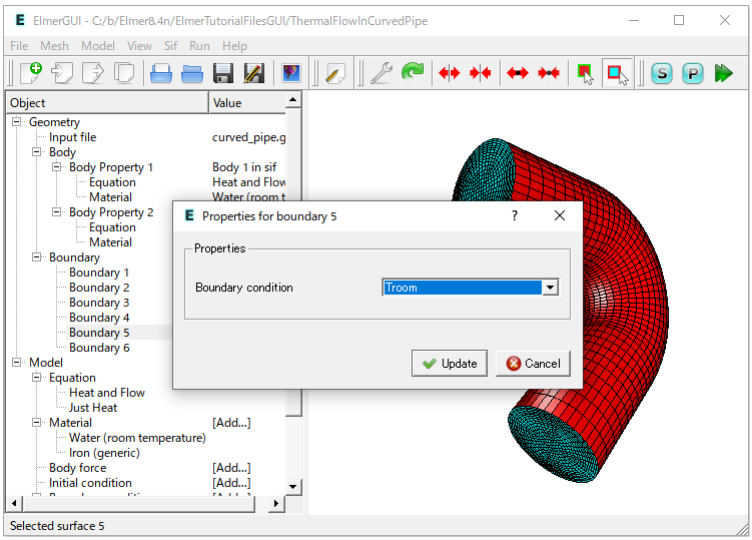

Figure 4.11: Boundary Property Editor activated by holding down the `AltGr` key while double clicking a surface.

Chapter 5

Utility functions

5.1 Boundary division and unification

Some of the input file formats listed in Table 3.1 are not perhaps so well suited for FE-calculations, even though widely used. The .stl format (stereo lithography format), for example, is by definition unable to distinguish between different boundary parts with different attributes. Moreover, the format approximates the boundary by disconnected triangles that do not fulfill the usual FE-compatibility conditions.

In order to deal with formats like .stl, ElmerGUI provides a minimal set of tools for boundary division and unification. The division is based on "sharp edge detection". An edge between two boundary elements is considered sharp, if the angle between the normals exceeds a certain value (20 degrees by default). The sharp edges are then used as a mortar to divide the surface into parts. The user may perform a sharp edge detection and boundary division from the Mesh menu by choosing

Mesh → Divide surface...

In 2D the corresponding operation is

Mesh → Divide edge...

The resulting parts are enumerated starting from the first free index.

Sometimes, the above process produces far too many distinct parts, which eventually need to be (re)unified. This can be done by selecting a group of surfaces by holding down the CTRL-key while double clicking the surfaces and choosing

Mesh → Unify surface...

The same operation in 2D is

Mesh → Unify edge...

The result will inherit the smallest index from the selected group.
The sharp edges that do not belong to a closed loop may be removed by

Mesh → Clean up

This operation has no effect on the boundary division, but sometimes it makes the result look better.

5.2 Saving pictures

The model drawn on the display area may be scanned into a 24-bit RGB image and saved in several picture file formats:

File → Save picture as...

The function supports .bmp, .jpg, .png, .pbm, .pgm, and .ppm file extensions.

5.3 View menu

The View menu provides several utility functions for controlling the visual behavior of ElmerGUI. The function names should be more or less self explanatory.

Chapter 6

Solver input files

The contents of the Model menu are passed to the solver in the form of a solver input file. A solver input file is generated by choosing

Sif → *Generate*

The contents of the file are editable in Solver Input File Window which shows up by choosing

Sif → *Edit...*

As shown in Figure 6.1, two types of syntax highlighting are available in Solver Input File Window from the menu on it:

Preference → *Syntax highlighting*

Font can be also changed by

Preference → *Font*

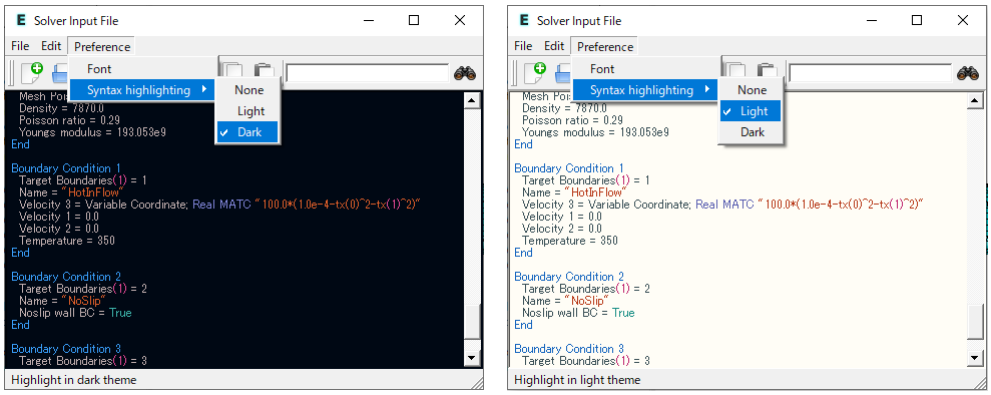

Figure 6.1: Syntax highlighting in Solver Input File Window

The new sif file needs to be saved before it becomes active. The recommended method is

File → *Save project*

In this way, also the current mesh and project files get saved in the same directory, avoiding possible inconsistencies later on.

Note: The previous versions of ElmerGUI automatically generated (i.e. overwrote) solver input file when saving or loading the project or mesh and it was inconvenient when manually modifying solver input file. Current version of ElmerGUI does not generate solver input file automatically when saving/loading project

or mesh.

Chapter 7

Solution and post processing

7.1 Running the solver

Once the solver input file has been generated and the project has been saved, it is possible to actually solve the problem:

Run → Start solver

This will launch either a single process for ElmerSolver (scalar solution) or multiple MPI-processes for ElmerSolver_mpi (parallel solution) depending on the definitions in

Run → Parallel settings...

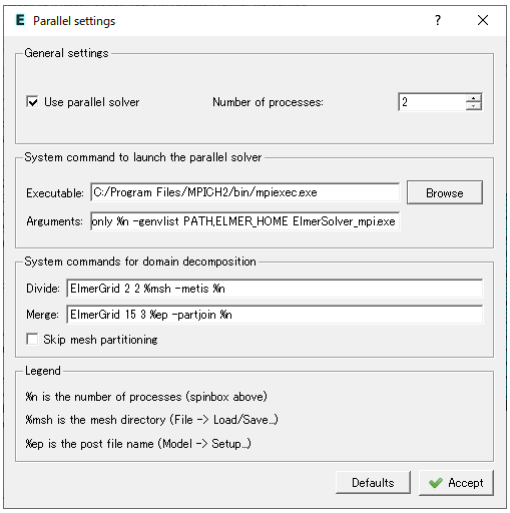

Figure 7.1: Parallel Settings Window.

As shown in Figure 7.1, the parallel menu has three group boxes. Usually, the user is supposed to touch only the "General settings" group and select the number of processes to execute. The two remaining groups deal with system commands to launch MPI-processes and external tools for domain decomposition. The parallel menu is greyed out if ElmerSolver_mpi is not present at start-up.

When the solver is running, there is a log window and a convergence monitor from which the iteration may be followed. In case of divergence or other troubles, the solver may be terminated by choosing

Run → Kill solver

Once the solver has started, Solver Log Window will appears to show the log. Solver Log Window also has menus for syntax highlighting and font selection like Solver Input File Window.

The solver will finally write a result file for postprocessor in the project directory. The name of the result file is defined in

Model → Setup...

It may be tiresome to generate solver input file, save the project then run solver every time you make a slight modification in conditions. “Generate, save and run” button (a button with green doubled triangle) in the toolbar enables these three actions by one clicking of the button. This will be helpful to quickly run the case modified via GUI. A similar button - “Save and run” button (a button with single green triangle) is also in Solver Input File Window. This button saves the solver input file and the project then runs the solver. This will be helpful to quickly run the case manually modified via Solver Input File Window. These buttons are shown in Figure 7.2.

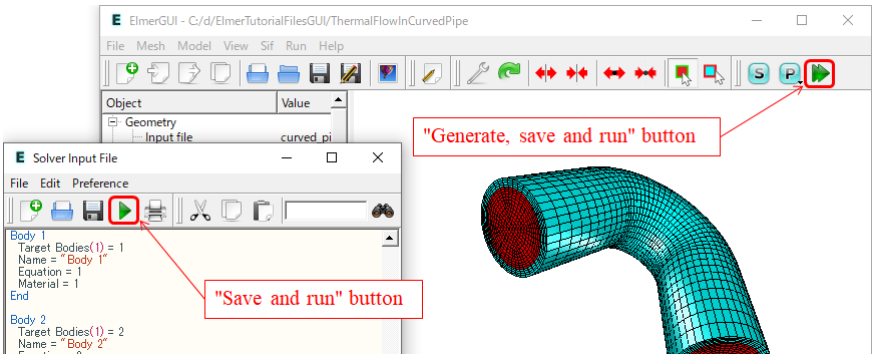

Figure 7.2: Buttons for quick run.

7.2 Post processing

If the path settings are set properly, ElmerGUI can call Paraview directly from

Run → Start ParaView

Note that you may always open Paraview also independently from ElmerGUI.

ElmerGUI still also includes calling possibility of the obsolete ElmerPost visualization tool. It will with time be eliminated also from the GUI.

The first alternative is activated from

Run → Start ElmerPost

This will launch ElmerPost, which will read in the result file and displays a contour plot representing the solution. If the results were produced by the parallel solver, the domain decomposition used in the calculations will be shown. Because ElmerPost can handle only “.ep” file, you should specify the Post file in Simulation section of Setup Window with extension of “.ep” such like “case.ep” while it is “case.vtu” by default.

The second post processor is based on the Visualization Toolkit, VTK. It is activated from

Run → ElmerVTK

A new window will then pop up, providing methods for drawing surfaces, contours, vectors, and stream lines.

In the toolbar of ElmerGUI main window, there is a button for launching postprocessor. The postprocessor to be launched can be selected by long pressing the button as shown in Figure 7.3.

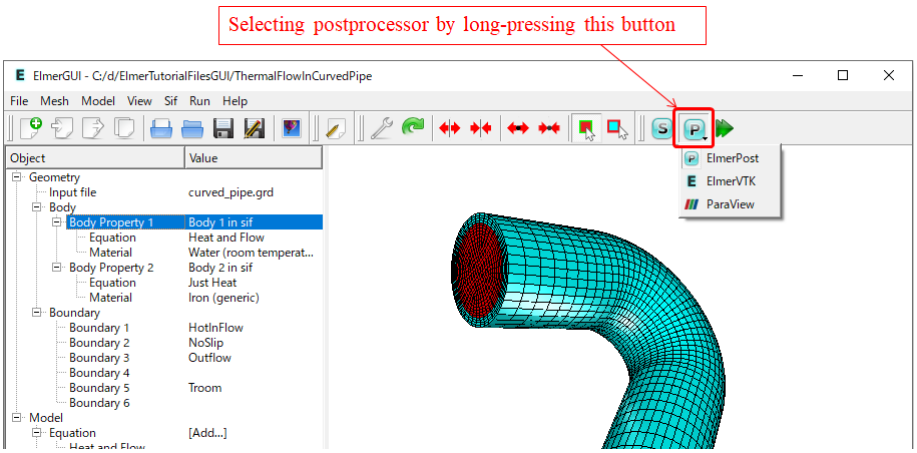

Figure 7.3: Selecting postprocessor in toolbar.

Bibliography

[1] Elmer web pages: https://www.csc.fi/elmer.

[2] Netgen/ngsolve web pages: https://ngsolve.org.

[3] Oce github pages: https://github.com/tpaviot/oce.

[4] Qt web pages: https://www.qt.io/jp.

[5] Qwt web pages: https://qwt.sourceforge.io.

[6] Tetgen web pages: http://wias-berlin.de/software/tetgen.

Appendix A

ElmerGUI initialization file

The initialization file for ElmerGUI is located in `ELMERGUI_HOME/edf`. It is called `egini.xml`:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE egini>
<egini version="1.0">

  Show splash screen at startup:
  <splashscreen> 1 </splashscreen>

  Show system tray icon:
  <systrayicon> 1 </systrayicon>

  Show system tray messages:
  <systraymessages> 1 </systraymessages>

  System tray message duration in milliseconds:
  <systraymsgduration> 3000 </systraymsgduration>

  Check the presence of external components:
  <checkexternalcomponents> 0 </checkexternalcomponents>

  Hide toolbars:
  <hidetoolbars> 0 </hidetoolbars>

  Plot convergence view:
  <showconvergence> 1 </showconvergence>

  Draw background image:
  <bgimage> 1 </bgimage>

  Background image file:
  <bgimagefile> :/images/bgimage.png </bgimagefile>

  Align background image to the bottom right corner of the screen:
  <bgimagealignright> 0 </bgimagealignright>

  Stretch background image to fit the display area (overrides align):
  <bgimagestretch> 1 </bgimagestretch>

  Maximum number of solvers / equation:
  <max_solvers> 10 </max_solvers>

  Maximum number of equations:
  <max_equations> 10 </max_equations>

  Maximum number of materials:
  <max_materials> 10 </max_materials>

  Maximum number of bodyforces:
  <max_bodyforces> 10 </max_bodyforces>
```

```
  Maximum number of initial conditions:
  <max_initialconditions> 10 </max_initialconditions>

  Maximum number of bodies:
  <max_bodies> 100 </max_bodies>

  Maximum number of bcs:
  <max_bcs> 500 </max_bcs>

  Maximum number of boundaries:
  <max_boundaries> 500 </max_boundaries>

</egini>
```

You may change the default behavior of ElmerGUI by editing this file. For example, to turn off the splash screen at start up, change the value of the tag `<splashscreen>` from 1 to 0. To change the background image, enter a picture file name in the `<bgimagefile>` tag. You might also want to increase the default values for solvers, equations, etc., in case of very complex models.

Appendix B

ElmerGUI material database

The file `ELMERGUI_HOME/edf/egmaterials.xml` defines the material database for ElmerGUI. The format of this file is the following:

```
<!DOCTYPE egmaterials>
<materiallibrary>

  <material name="Air (room temperature)" >
    <parameter name="Density" >1.205</parameter>
    <parameter name="Heat conductivity" >0.0257</parameter>
    <parameter name="Heat capacity" >1005.0</parameter>
    <parameter name="Heat expansion coeff." >3.43e-3</parameter>
    <parameter name="Viscosity" >1.983e-5</parameter>
    <parameter name="Turbulent Prandtl number" >0.713</parameter>
    <parameter name="Sound speed" >343.0</parameter>
  </material>

  <material name="Water (room temperature)" >
    <parameter name="Density" >998.3</parameter>
    <parameter name="Heat conductivity" >0.58</parameter>
    <parameter name="Heat capacity" >4183.0</parameter>
    <parameter name="Heat expansion coeff." >0.207e-3</parameter>
    <parameter name="Viscosity" >1.002e-3</parameter>
    <parameter name="Turbulent Prandtl number" >7.01</parameter>
    <parameter name="Sound speed" >1497.0</parameter>
  </material>
  ...
</materiallibrary>
```

The values of the parameters may be either constant, or functions of time, temperature, etc. A temperature dependent parameter may be defined e.g. as

```
<parameter name="A" >Variable Temperature; Real; 2 3; 4 5; End</parameter>
```

In this case, $A(2) = 3$ and $A(4) = 5$. Values between the points are interpolated linearly, and extrapolated in the tangent direction outside the domain. The number of points defining the interpolant may be arbitrary.

Appendix C

ElmerGUI definition files

The directory `ELMERGUI_HOME` contains a subdirectory called "edf". This is the place where all ElmerGUI definition files (ed-files) reside. The definition files are XML-formatted text files which define the contents and appearance of the Model menu.

The ed-files are loaded iteratively from the edf-directory once and for all when ElmerGUI starts. Later, it is possible to view and edit their contents by choosing

File → Definitions...

An ed-file has the following structure:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE edf>
<edf version="1.0">
   [PDE block]
   [PDE block]
   ...
   [PDE block]
</edf>
```

The structure of a [PDE block] is the following:

```
<PDE Name="My equation">
   <Name>
      My equation
   </Name>
   ...
   <Equation>
      [Widget block]
   </Equation>
   ...
   <Material>
      [Widget block]
   </Material>
   ...
   <BodyForce>
      [Widget block]
   <BodyForce>
   ...
   <InitialCondition>
      [Widget block]
   </InitialCondition>
   ...
   <BoundaryCondition>
      [Widget block]
   </BoundaryCondition>
</PDE>
```

Note that the name of the PDE is defined redundantly in two occurrences.

The basic structure of a [Widget block] is the following:

```
<Parameter Widget="Label">
   <Name> My label </Name>
</Parameter>
...
<Parameter Widget="Edit">
   <Name> My edit box </Name>
   <Type> Integer </Type>
   <Whatis> Meaning of my edit box </Whatis>
</Parameter>
...
<Parameter Widget="CheckBox">
   <Name> My check box </Name>
   <Type> Logical </Type>
   <Whatis> Meaning of my check box </Whatis>
</Parameter>
...
<Parameter Widget="Combo">
   <Name> My combo box </Name>
   <Type> String </Type>
   <Item> <Name> My 1st item </Name> </Item>
   <Item> <Name> My 2nd item </Name> </Item>
   <Item> <Name> My 3rd item </Name> </Item>
   <Whatis> Meaning of my combo box </Whatis>
</Parameter>
```

There are four types of widgets available:

- Label (informative text)
- CheckBox (switches)
- ComboBox (selection from list)
- LineEdit (generic variables)

Each widget must be given a name and a variable type: logical, integer, real, or string. It is also a good practice to equip the widgets with tooltips explaining their purpose and meaning as clearly as possible.

Below is a working example of a minimal ElmerGUI definition file. It will add "My equation" to the equation tabs in the Model menu, see Figure C.1. The file is called "sample.edf" and it should be placed in `ELMERGUI_HOME/edf`.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE edf>
<edf version="1.0">
  <PDE Name="My equation">
    <Name> My equation </Name>
    <Equation>
      <Parameter Widget="Label">
        <Name> My label </Name>
      </Parameter>
      <Parameter Widget="Edit">
        <Name> My edit box </Name>
        <Type> Integer </Type>
        <Whatis> Meaning of my edit box </Whatis>
      </Parameter>
      <Parameter Widget="CheckBox">
        <Name> My check box </Name>
        <Type> Logical </Type>
        <Whatis> Meaning of my check box </Whatis>
      </Parameter>
      <Parameter Widget="Combo">
        <Name> My combo box </Name>
        <Type> String </Type>
        <Item> <Name> My 1st item </Name> </Item>
        <Item> <Name> My 2nd item </Name> </Item>
        <Item> <Name> My 3rd item </Name> </Item>
        <Whatis> Meaning of my combo box </Whatis>
```

```
      </Parameter>
    </Equation>
  </PDE>
</edf>
```

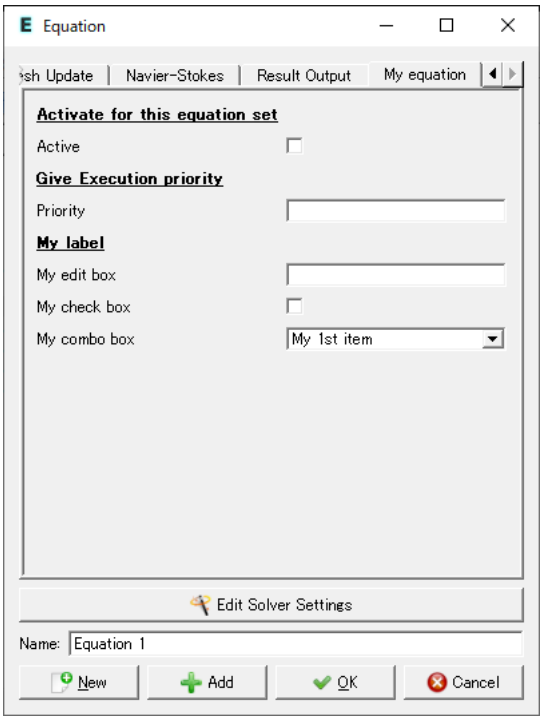

Figure C.1: Equation tab in Model menu produced by the sample ed-file.

More sophisticated examples with different tags and attributes can be found from the XML-files in `ELMERGUI_HOME/edf`.

Appendix D

Elmer mesh files

mesh.header

```
nodes elements boundary-elements
types
type1 elements1
type2 elements2
...
typeN elementsN
```

mesh.nodes

```
node1 tag1 x1 y1 z1
node2 tag2 x2 y2 z2
...
nodeN tagN xN yN zN
```

mesh.elements

```
element1 body1 type1 n11 ... n1M
element2 body2 type2 n21 ... n2M
...
elementN bodyN typeN nN1 ... nNM
```

mesh.boundary

```
element1 boundary1 parent11 parent12 n11 ... n1M
element2 boundary2 parent21 parent22 n21 ... n2M
...
elementN boundaryN parentN1 parentN2 nN1 ... nNM
```

Appendix E

Adding menu entries to ElmerGUI

As ElmerGUI is based on Qt4, it should be relatively easy to customize the menus and dialog windows. A new menu item, for example, is added as follows.

First, we declare the menu action and a private slot in `src/mainwindow.h`:

```
private slots:
    ...
    void mySlot();
    ...

private:
    ...
    QAction *myAct;
    ...
```

Then, in `src/mainwindow.cpp`, we actually create the action, connect an appropriate signal from the action to the slot, and add the action in a menu:

```
void MainWindow::createActions()
{
    ...
    myAct = new QAction(tr("*** My menu entry ***"), this);
    connect(myAct, SIGNAL(triggered()), this, SLOT(mySlot()));
    ...
}
```

and

```
void MainWindow::createMenus()
{
    ...
    meshMenu->addSeparator();
    meshMenu->addAction(myAct);
    ...
}
```

It finally remains to define the slot to which the triggering signal is connected. All processing related to the action should be done here:

```
void MainWindow::mySlot()
{
    cout << "Here we go!" << endl;
}
```

Appendix F

ElmerGUI mesh structure

The finite element mesh generated by ElmerGUI is of class `mesh_t` (declared in `src/meshtype.h`). The mesh is private to the class `GLWidget` (declared in `src/glwidget.h`), which is responsible of drawing and rendering the model.

F.1 GLWidget

The class `GLWidget` provides the following public methods for accessing the mesh:

`mesh_t* GLWidget::getMesh()`

Get the active mesh.

`void GLWidget::newMesh()`

Allocate space for a new mesh.

`void GLWidget::deleteMesh()`

Delete the current mesh.

`bool GLWidget::hasMesh()`

Returns true if there is a mesh. Otherwise returns false.

`void GLWidget::setMesh(mesh_t* myMesh)`

Set active mesh to myMesh.

The mesh can be accessed in `MainWindow` for example as follows (see previous section for more details):

```
void MainWindow::mySlot()
{
   if(!glWidget->hasMesh()) return;
   mesh_t* mesh = glWidget->getMesh();
   cout << "Nodes: " << mesh->getNodes() << endl;
   cout << "Edges: " << mesh->getEdges() << endl;
   cout << "Trias: " << mesh->getSurfaces() << endl;
   cout << "Tetras: " << mesh->getElements() << endl;
}
```

F.2 mesh_t

The class `mesh_t` provides the following public methods for accessing and manipulating mesh data:

`bool mesh_t::isUndefined()`

Returns true if the mesh is undefined. Otherwise returns false.

`void mesh_t::clear()`

Clears the current mesh.

`bool mesh_t::load(char* dir)`

Loads Elmer mesh files from directory dir. Returns false if loading failed. Otherwise returns true.

`bool mesh_t::save(char* dir)`

Saves the mesh in Elmer format in directory dir. Returns false if saving failed. Otherwise returns true.

`double* mesh_t::boundingBox()`

Returns bounding box for the current mesh (xmin, xmax, ymin, ymax, zmin, zmax, xmid, ymid, zmid, size).

`void mesh_t::setCdim(int cdim)`

Set coordinate dimension to cdim.

`int mesh_t::getCdim()`

Get coordinate dimension for the current mesh.

`void mesh_t::setDim(int dim)`

Set mesh dimension to dim.

`int mesh_t::getDim()`

Get mesh dimension.

`void mesh_t::setNodes(int n)`

Set the number of nodes to n.

`int mesh_t::getNodes()`

Get the number of nodes.

`void mesh_t::setPoints(int n)`

Set the number of point elements to n.

`int mesh_t::getPoints()`

Get the number of point elements.

`void mesh_t::setEdges(int n)`

Set the number of edge elements to n.

`int mesh_t::getEdges()`

Get the number of edge elements.

`void mesh_t::setSurfaces(int n)`

Set the number of surface elements to n.

`int mesh_t::getSurfaces()`

Get the number of surface elements.

`void mesh_t::setElements(int n)`

Set the number of volume elements to n.

`int mesh_t::getElements()`

Get the number of volume elements.

`node_t* mesh_t::getNode(int n)`

Get node n.

`void mesh_t::setNodeArray(node_t* nodeArray)`

Set node array point to nodeArray. Useful, if the user wants to take care of memory allocation by him/her self.

`void mesh_t::newNodeArray(int n)`

Allocate memory for n nodes.

`void mesh:t::deleteNodeArray()`

Delete current node array.

`point_t* mesh_t::getPoint(int n)`

Get point element n.

`void mesh_t::setPointArray(point_t* pointArray)`

Set point element array point to pointArray. Useful, if the user wants to take care of memory allocation by him/her self.

`void mesh_t::newPointArray(int n)`

Allocate memory for n point elements.

`void mesh_t::deletePointArray()`

Delete current point element array.

`edge_t* mesh_t::getEdge(int n)`

Get edge element n.

`void mesh_t::setEdgeArray(edge_t* edgeArray)`

Set edge element array point to edgeArray. Useful, if the user wants to take care of memory allocation by him/her self.

`void mesh_t::newEdgeArray(int n)`

Allocate memory for n edge elements.

`void mesh_t::deleteEdgeArray()`

Delete current edge element array.

`surface_t* mesh_t::getSurface(int n)`

Get surface element n.

```
void mesh_t::setSurfaceArray(surface_t* surfaceArray)
```

Set surface element array point to surfaceArray. Useful, if the user wants to take care of memory allocation by him/her self.

```
void mesh_t::newSurfaceArray(int n);
```

Allocate memory for n surface elements.

```
void mesh_t::deleteSurfaceArray()
```

Delete surface element array.

```
element_t* mesh_t::getElement(int n)
```

Get volume element n.

```
void mesh_t::setElementArray(element_t* elementArray)
```

Set volume element array point to elementArray. Useful, if the user wants to take care of memory allocation by him/her self.

```
void mesh_t::newElementArray(int n)
```

Allocate memory for n volume elements.

```
void mesh_t::deleteElementArray()
```

Delete current volume element array.

F.3 node_t

The class `node_t` has been declared in `src/meshtypes.h`. It provides the following public methods for accessing node data:

```
void node_t::setX(int n, double x)
```

Set component n of the position vector to x.

```
double node_t::getX(int n)
```

Get component n of the position vector.

```
void node_t::setXvec(double* v)
```

Set the position vector to v.

```
double* node_t::getXvec()
```

Get the position vector.

```
void node_t::setIndex(int n)
```

Set the index of the node to n.

```
int node_t::getIndex()
```

Get the index of the node.

F.4 Base element class element_t

The class `element_t` provides the following methods for accessing element data:

`void element_t::setNature(int n)`

Set element nature to n (either PDE_UNKNOWN, PDE_BOUNDARY, or PDE_BULK).

`int element_t::getNature()`

Get the element nature.

`void element_t::setCode(int n)`

Set element code to n (202 = two noded line, 303 = three noded triangle, ...)

`int element_t::getCode()`

Get the element code.

`void element_t::setNodes(int n)`

Set the number of nodes to n.

`int element_t::getNodes()`

Get the number of nodes.

`void element_t::setIndex(int n)`

Set element index to n.

`int element_t::getIndex()`

Get the element index.

`void element_t::setSelected(int n)`

Set the selection state (1=selected, 0=unselected).

`int element_t::getSelected()`

Returns 1 if element is selected. Otherwise returns 0.

`int element_t::getNodeIndex(int n)`

Get the index of node n.

`void element_t::setNodeIndex(int m, int n)`

Set the index of node m to n.

`int* element_t::getNodeIndexes()`

Get the indexes of all nodes.

`void element_t::newNodeIndexes(int n)`

Allocate space for n node indexes.

`void element_t::deleteNodeIndexes()`

Delete all node indexes.

F.5 Point element class point_t

The class `point_t` inherits all public members from class `element_t`. In addition to this, it provides the following methods for accessing and manipulating point element data:

```
void setSharp(bool b);
```

Mark the point element “sharp” (b=true) or not (b=false).

```
bool isSharp();
```

Returns true if the point element is “sharp”. Otherwise returns false.

```
void setEdges(int n);
```

Set the number of edges elements connected to the point to n.

```
int getEdges();
```

Get the number of edge elements connected to the point.

```
void setEdgeIndex(int m, int n);
```

Set the index of m’th edge element to n.

```
int getEdgeIndex(int n);
```

Get the index of n’th connected edge element.

```
void newEdgeIndexes(int n);
```

Allocate space for n edge element indexes.

```
void deleteEdgeIndexes();
```

Delete all edge element indexes.

F.6 Edge element class edge_t

The class `edge_t` inherits all public methods from `element_t`. It also provides the following methods for accessing and manipulating edge element data:

```
void edge_t::setSharp(bool b)
```

Mark the edge sharp (b=true) or not (b=false).

```
bool edge_t::isSharp()
```

Returns true if the edge is sharp.

```
void edge_t::setPoints(int n)
```

Set the number of point elements connected to the edge to n.

```
int edge_t::getPoints()
```

Get the number of point elements connected to the edge.

```
void edge_t::setPointIndex(int m, int n)
```

Set the index of point element m to n.

```
int edge_t::getPointIndex(int n)
```

Get the index of point element n.

`void edge_t::newPointIndexes(int n)`

Allocate space for n point element indexes.

`void edge_t::deletePointIndexes()`

Delete all point element indexes.

`void edge_t::setSurfaces(int n)`

Set the number of surface elements connected to the edge to n.

`int edge_t::getSurfaces()`

Get the number of surface elements connected to the edge.

`void edge_t::setSurfaceIndex(int m, int n)`

Set the index of surface element m to n.

`int edge_t::getSurfaceIndex(int n)`

Get the index of m'th surface element connected to the edge.

`void edge_t::newSurfaceIndexes(int n)`

Allocate space for n surface element indexes.

`void edge_t::deleteSurfaceIndexes()`

Delete all surface element indexes.

F.7 Surface element class surface_t

Finally, the class `surface_t` provides the following public methods for accessing and manipulating surface element data, besides of those inherited from the base element class `element_t`:

`void surface_t::setEdges(int n)`

Set the number of edge elements connected to the surface to n.

`int surface_t::getEdges()`

Get the number of edge elements connected to the surface element.

`void surface_t::setEdgeIndex(int m, int n)`

Set the index of m'th edge element to n.

`int surface_t::getEdgeIndex(int n)`

Get the index of n'th edge element connected to the surface element.

`void surface_t::newEdgeIndexes(int n)`

Allocate space for n edge element indexes.

`void surface_t::deleteEdgeIndexes()`

Delete all edge element indexes.

`void surface_t::setElements(int n)`

Set the number of volume elements connected to the surface element to n.

```
int surface_t::getElements()
```

Get the number of volume elements connected to the surface element.

```
void surface_t::setElementIndex(int m, int n)
```

Set the index of m'th volume element to n.

```
int surface_t::getElementIndex(int n)
```

Get the index of n'th volume element connected to the surface.

```
void surface_t::newElementIndexes(int n)
```

Allocate space for n volume element indexes.

```
void surface_t::deleteElementIndexes()
```

Delete all volume element indexes.

```
void surface_t::setNormalVec(double* v)
```

Set the normal vector to the surface element.

```
double* surface_t::getNormalVec()
```

Get the normal vector for the surface element.

```
double surface_t::getNormal(int n)
```

Get component n of the normal vector.

```
void surface_t::setNormal(int n, double x)
```

Set component n of the normal to x.

```
void surface_t::setVertexNormalVec(int n, double* v)
```

Set the normal vector for vertex n to v.

```
void surface_t::addVertexNormalVec(int m, double* v)
```

Add vector v to the normal in vertex n.

```
void surface_t::subVertexNormalVec(int m, double* v)
```

Subtract vector v from the normal in vertex n.

```
double* surface_t::getVertexNormalVec(int n)
```

Get the normal vector in vertex n.