ElmerGUI manual v. 0.2

Mikko Lyly

December 6, 2008

Contents

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 partitionings for models with piecewise linear boundaries, setting up PDE-systems to solve, and exporting model data and results for ElmerSolver and ElmerPost to visualize. There is also an internal postprocessor, which can be used as an alternative to ElmerPost, to draw color surfaces, contours, vector fields, and visualize time dependent data.

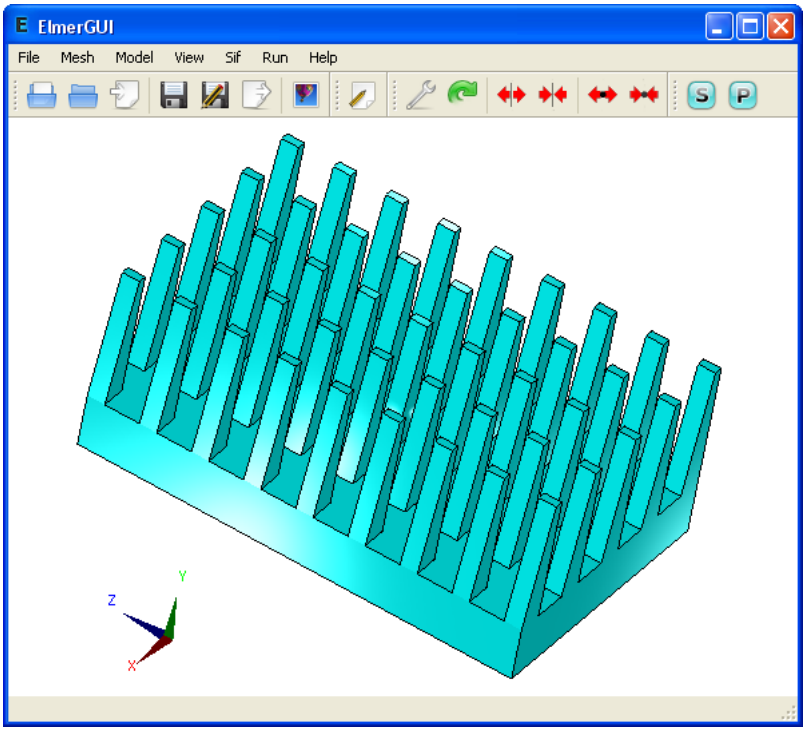

Figure 1: Main window of ElmerGUI.

One of the main features of ElmerGUI is the interface to the parallel solver, `ElmerSolver_mpi`. The GUI hides from the user a number of operations that are normally performed from command line with various external tools related to domain decomposition, launching the paralle processes, and merging the results. This makes it possible to use ElmerSolver with multicore processors even on interactive desktop environments.

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

ElmerGUI relies on the Qt4 cross platform framework from Trolltech [6], and it uses the Qwt5 libray by Josef Wilgen and Uwe Rathman[7]. The internal postprocessor is based on the VTK library of Ken Martin, Will Schroeder, and Bill Lorensen [10]. The CAD import features are implemented by linking against the OCC library fromOpenCASCADE S.A.S. [5]. The program is also capable of using Tetgen [8] and Netgen [4] as external finite element mesh generators.

2 Installation from source

The source code of ElmerGUI is available from the subversion repository of SourceForge.Net. The GPL licenced source may be downloaded by executing the following command with a SVN client program(on Windows the Tortoise SVN client is recommended):

```
svn co https://elmerfem.svn.sourceforge.net/svnroot/elmerfem/trunk trunk
```

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

2.1 Linux

Bofore staring to compile, please make sure that you have the development package of Qt 4 installed on your system (i.e., libraries, headers, and program development tools). Qt version 4.2 or newer is recommended. You may also wish to install Qwt 5, VTK version 5, and OpenCASCADE 6.3, for additional functionality.

The program is compiled by executing the following sequence of commands in a terminal window:

```
$ cd elmerfem/trunk/ElmerGUI
$ qmake
$ make
```

It is possible that the project file "ElmerGUI.pro" needs to be edited depending on how and where the external libraries have been installed. The lines that need attention can be found from the beginning of the file.

Once the build process has finished, it suffices to set up the environment variable `ELMERGUI_HOME`, add it to `PATH`, and copy the file "ElmerGUI" and the directory "edf" in this location:

```
$ export ELMERGUI_HOME=$ELMER_HOME/bin
$ export PATH=$PATH:$ELMERGUI_HOME
```

```
$ cp -r ./ElmerGUI ./edf $ELMERGUI_HOME
$ chmod -R 755 $ELMERGUI_HOME/edf
```

Later, it is possible to upgrade ElmerGUI by typing the following commands within the build directory:

```
$ svn update
$ make
```

and by copying the new executable as well as the updated edf directory in `ELMERGUI_HOME`.

2.2 Windows

On 32-bit Windows systems, it is possible to use precompiled binary packages, which are distributed through the file release system of SourceForge.NET [2]. The binary packages may depend on external components, which must be installed prior to downloading Elmer. The list of possible dependencies can be found from the "Release notes" of the file release.

It is also possible to compile ElmerGUI from source. This can be done either with Visual Studio 2008 C++ Express Edition [9], or by using the MinGW compiler suite [3].

The compilation instructions for MinGW are more or less the same as for Linux. The only exception is that then OCC should probably be excluded from compilation, and the cad import functionality will be unavailable. The advantage of using MinGW, on the other hand, is that it is then possible to use Qt's precompiled binary packages [6], which otherwise have to be built from source. MinGW might also be the natural choice for users that prefer Unix-like operating environments.

The easiest way to compile ElmerGUI with VC++ is to invoke "Visual Studio 2008 Command Prompt" and execute the following sequence commands within ElmerGUI's source directory:

```
> qmake
> nmake
```

Again, it is necessary to verify that the paths to all external components have been correctly defined in "ElmerGUI.pro". The build process is performed in release mode, producing the executable "ElmerGUI.exe".

The final task is to introduce the environment variable `ELMERGUI_HOME`, add it to path, and copy the executable "ElmerGUI.exe" as well as the directory "edf" in `ELMERGUI_HOME`.

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 partitionings for bounded domains with piecewise linear boundaries. It is possible to use one of the following mesh generators:

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

The default import filter and mesh generator is ElmerGrid. Tetgen and Netgen are optional modules, 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 from the File menu:

File → *Open...*

The selection is based on the input file suffix according to Table 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 → *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 the same place as the preferred generator. The control string for Tetgen has been discussed and explained in detail in [8].

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

Table 1. Input files and capabilities of the mesh generators.

The mesh generator is reactivated from the Mesh menu by choosing

Mesh → Remesh

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

Mesh → 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):

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

These files have to reside in the same mesh directory.

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. Projects may be loaded and/or saved from the File menu as

File → Load project...

and/or

File → Save project...

When an ElmerGUI project is loaded, a new solver input file will be generated and saved in the project directory using the sif-name defined in

Model → Setup...

If there is an old solver input file with the same name, it will be overwritten.

The contents of a typical project directory is the following:

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

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].

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 by choosing

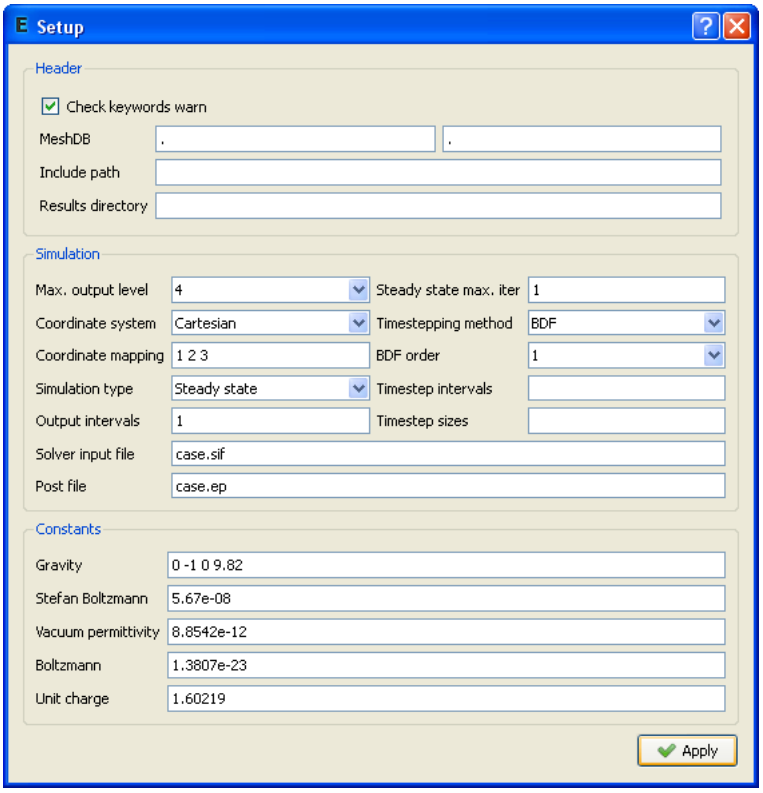

Figure 2: Setup menu.

Model → *Equation* → *Add...*

Go through the tabs and check "Active" all individual equations that constitute your system. The numerical methods and parameters can be selected and tuned by pressing the "Edit Solver Settings" button. Finally, name the PDE-system in the "Name" line edit box, and apply the system to appropriate bodies. Once the PDE-system has been defined, press the Ok-button. The equation remains visible and editable under the Model menu.

It is also possible to attach an equation to a body by holding down the SHIFT-key while double clicking one of its surfaces. A pop up menu will then appear, listing all possible attributes that can be attached to the selection. Choose the attributes and press Ok.

4.3 Material menu

The next menu is related to material and model parameters:

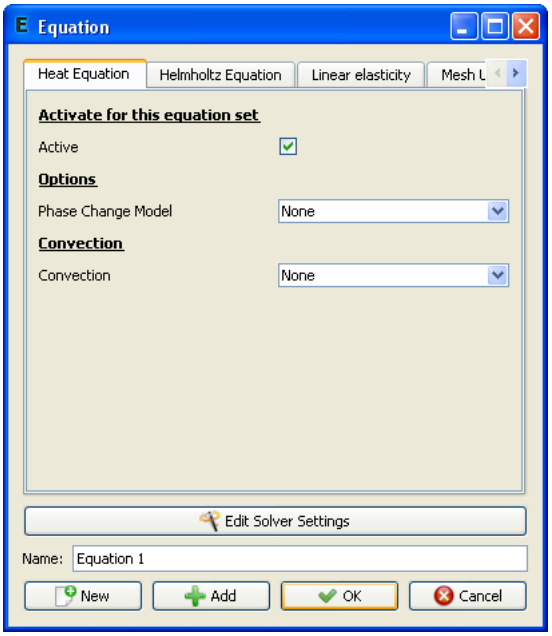

Figure 3: Equation menu.

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 attach it to bodies, choose

Model → *Material* → *Add...*

Again, it is possible to attach the material to a body by holding down the SHIFT-key while double clicking one of its boundaries.

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

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 N. 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.

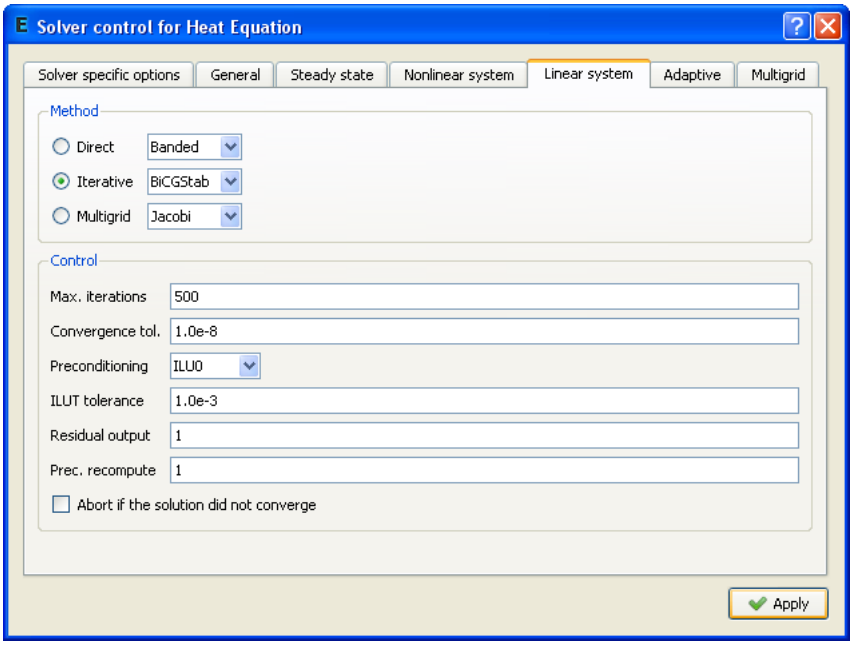

Figure 4: Solver settings menu.

If the user presses SHIFT and F1, a tooltip for the active widget will be displayed.

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 attach it to the bodies.

4.5 Initial condition menu

The last menu related to body properties is

Model → *Initial condition*

Once again, choose

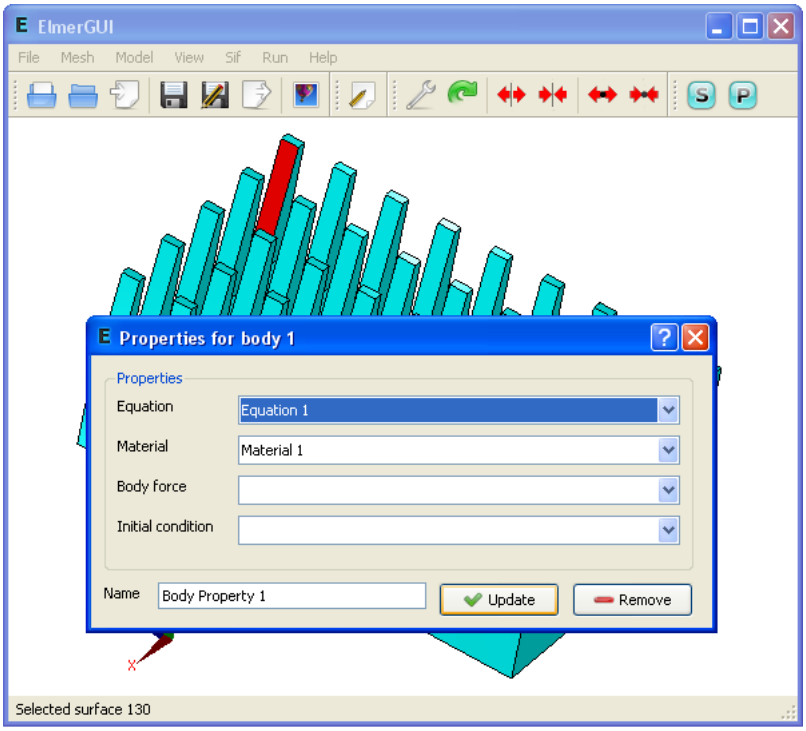

Figure 5: Body property editor is activated by holding down the SHIFT key while double clicking a surface.

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

to define a set of initial conditions and attach it to the bodies.

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 attach them to boundaries.

It is possible to attach a boundary condition to a boundary by holding down the ALT or ALTGR-key while double clicking a surface or edge. A pop up menu will appear, listing all possible conditions that can be attached to

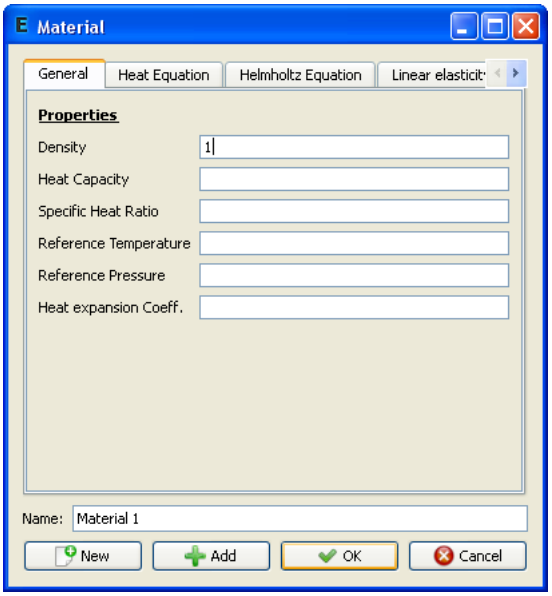

Figure 6: Material menu.

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

5 Utility functions

5.1 Boundary division and unification

Some of the input file formats listed in Table 1 are not perhaps best suited for FE-calculations, eventhough widely used. The .stl format (stereo litography 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 normal FE-compatibility conditions.

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

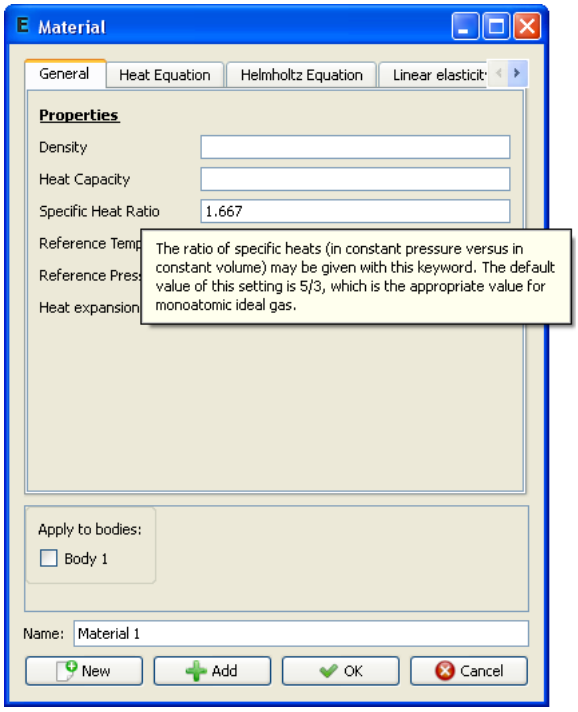

Figure 7: Tooltips are shown by holding down the SHIFT and F1 keys.

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.

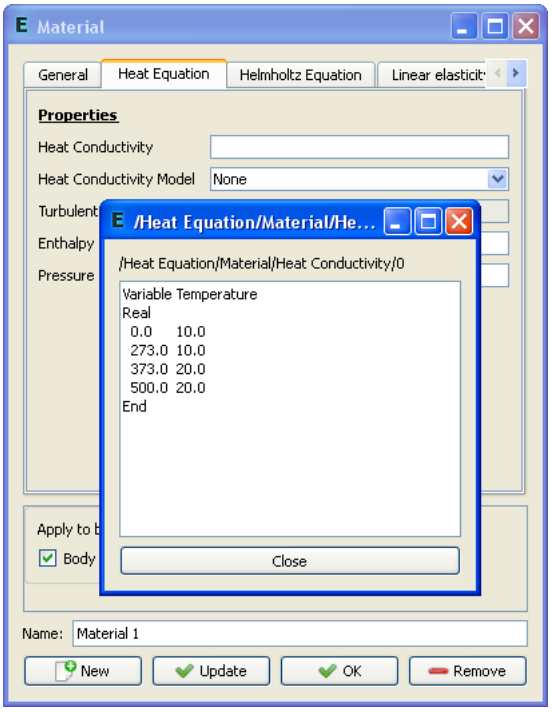

Figure 8: Text edit extension of a line edit box is activated by pressing Enter.

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 behaviour of ElmerGUI. The function names should be more or less self explanatory.

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

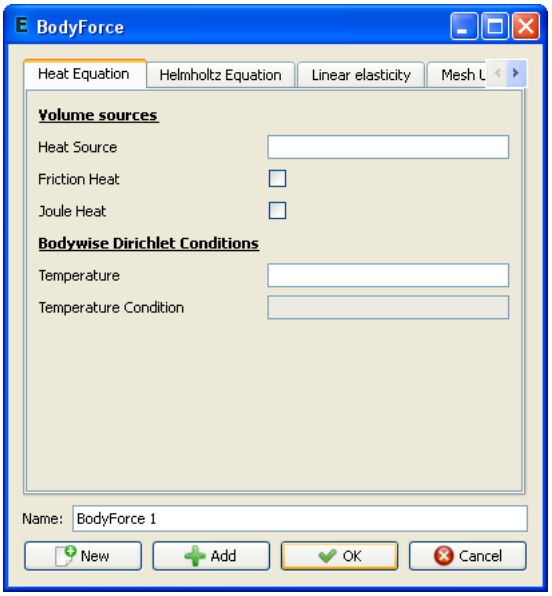

Figure 9: Body force menu.

Sif → *Generate*

The contents of the file are editable:

Sif → *Edit...*

The new sif file needs to 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.

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 actualy 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

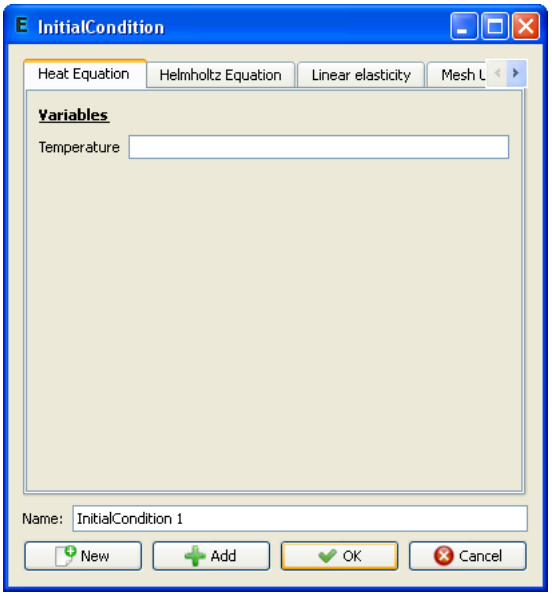

Figure 10: Initial condition menu.

the definitions in

Run → Parallel settings...

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

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

Model → Setup...

7.2 Post preocessing I (ElmerPost)

ElmerGUI provides two different post processors for drawing and displaying results.

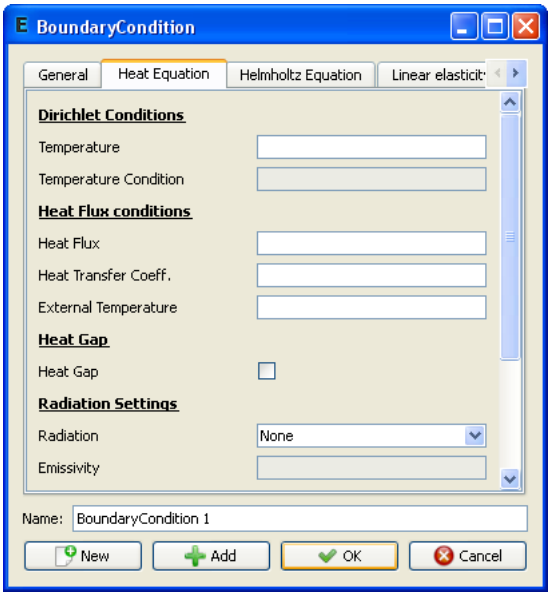

Figure 11: Boundary condition menu.

The first alternative is activated from

Run → Start postprocessor

This will launch ElmerPost, which will read in the results and displays a contour plot representing the solution. If the results were prodoced by the parallel solver, the domain decomposition used in the calculations will be shown.

7.3 Post preocessing II (VTK)

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

Run → Postprocessor (VTK)...

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

7.3.1 Python interface

If ElmerGUI has been compiled with PyhthonQt-support, there is a Python console available for scripting. The console is located under the Edit-menu:

Edit → PythonQt console...

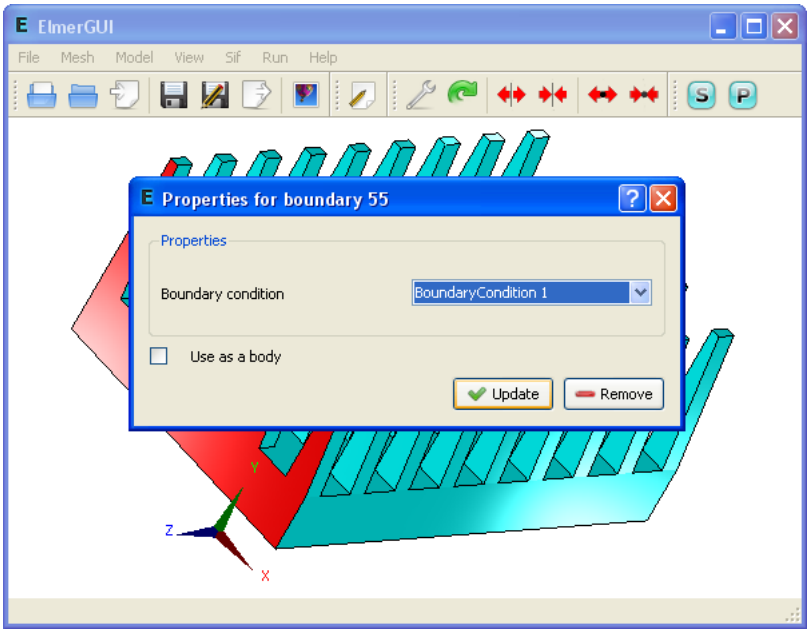

Figure 12: Boundary property editor activated by holding down the ALTGR key while double clicking a surface.

The console provides access to the "public slots" of the following C++ classes:

```
egp             // elmergui post processor
matc            // matc language interpreter
preferences     // controls for preferences
surfaces        // controls for surface plots
vectors         // controls for vector fields
isoContours     // controls for iso contours
isoSurfaces     // controls for iso surfaces
streamLines     // controls for stream lines
colorBar        // cotrols for the color bar
timeStep        // controls for transient results
```

Each of these classes provides a number of useful methods for data and display manipulation. To fix ideas, the following sequence of commands will read in the result file and draws the "Temperature" field as a surface plot. More examples can be found in section N.N.

```
py> egp.ReadPostFile("case.ep")
py> egp.SetSurfaces(1)
py> surfaces.SetFieldName("Temperature")
py> egp.Render()
```

The methods are the following (not in order):

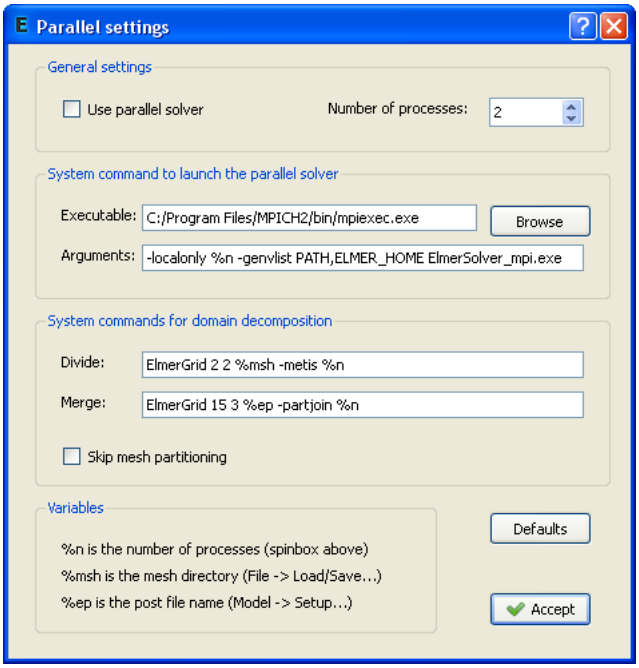

Figure 13: Parallel settings dialog.

```
egp:
  void SetPostFileStart(int);                 // start time
  void SetPostFileEnd(int);                   // end time
  bool ReadPostFile(QString);                 // read post file
  void Redraw();                              // regenarate all actors
  void Render();                              // update display
  void SetSurfaces(bool);                     // surface plot mode
  void SetVectors(bool);                      // vector plot mode
  void SetIsoContours(bool);                  // iso contour mode
  void SetIsoSurfaces(bool);                  // iso surface mode
  void SetStreamLines(bool);                  // stream line mode
  void SetColorBar(bool);                     // show/hide colorbar
  void SetMeshPoints(bool);                   // show/hide nodes
  void SetMeshEdges(bool);                    // show/hide edges
  void SetFeatureEdges(bool);                 // show/hide feature edges
  void SetAxes(bool);                         // show/hide coord. axes
  bool GetClipAll();                          // clipping on?
  void SetClipAll(bool);                      // set clippping mode
  void SetClipPlaneOx(double);                // clip plane origin
  void SetClipPlaneOy(double);                // clip plane origin
  void SetClipPlaneOz(double);                // clip plane origin
  void SetClipPlaneNx(double);                // clip plane normal
  void SetClipPlaneNy(double);                // clip plane normal
```

```
  void SetClipPlaneNz(double);                 // clip plane normal
  double GetCameraDistance();                  // get camera distance
  void SetCameraDistance(double);              // set camera distance
  double GetCameraPositionX();                 // get camera position
  double GetCameraPositionY();                 // get camera position
  double GetCameraPositionZ();                 // get camera position
  void SetCameraPositionX(double);             // set camera position
  void SetCameraPositionY(double);             // set camera position
  void SetCameraPositionZ(double);             // set camera position
  double GetCameraFocalPointX();               // get focal point
  double GetCameraFocalPointY();               // get focal point
  double GetCameraFocalPointZ();               // get focal point
  void SetCameraFocalPointX(double);           // set focal point
  void SetCameraFocalPointY(double);           // set focal point
  void SetCameraFocalPointZ(double);           // set focal point
  void CameraDolly(double);                    // camera dolly
  void CameraRoll(double);                     // camera roll
  void CameraAzimuth(double);                  // camera azimuth
  void CameraYaw(double);                      // camera yaw
  void CameraElevation(double);                // camera elevation
  void CameraPitch(double);                    // camera pitch
  void CameraZoom(double);                     // camera zoom
  void SetInitialCameraPosition();             // reset camera position
  double GetLength();                          // get model length
  int NofNodes();                              // get number of nodes
  bool MatcCmd(QString);                       // evaluate matc command
  void domatcSlot();                           // flush matc console
  bool SavePngFile(QString);                   // save png image
  void RotateX(double);                        // rotate model
  void RotateY(double);                        // rotate model
  void RotateZ(double);                        // rotate model
  void SetOrientation(double, double, double); // set orientation
  void SetPositionX(double);                   // set position
  void SetPositionY(double);                   // set position
  void SetPositionZ(double);                   // set position
  void SetPosition(double, double, double);    // set position (abs)
  void AddPosition(double, double, double);    // set position (inc)
  void SetOrigin(double, double, double);      // set origin
  void SetScaleX(double);                      // set display scale
  void SetScaleY(double);                      // set display scale
  void SetScaleZ(double);                      // set display scale
  void SetScale(double, double, double);       // set scale
  void Reset();                                // reset display

matc:
  bool SetCommand(QString);                    // set command

preferences:
  void SetSurfaceMeshForPoints(bool);          // draw nodes on surface
```

```
  void SetVolumeMeshForPoints(bool);          // draw nodes in volume
  void SetPointSize(int);                     // node size
  void SetPointQuality(int);                  // node quality
  void SetClipPlaneForPoints(bool);           // clip nodes
  void SetSurfaceMeshForEdges(bool);          // draw edges on surface
  void SetVolumeMeshForEdges(bool);           // draw edges in volume
  void SetTubeFilterForEdges(bool);           // draw edges as tubes
  void SetClipPlaneForEdges(bool);            // clip edges
  void SetLineWidthForEdges(int);             // edge line width
  void SetTubeQualityForEdges(int);           // edge tube quality
  void SetTubeRadiusForEdges(int);            // edge tube radius
  void SetSurfaceMeshForFeatureEdges(bool);   // feature edges for surf
  void SetVolumeMeshForFeatureEdges(bool);    // feature edges for vol
  void SetTubeFilterForFeatureEdges(bool);    // feature edges as tubes
  void SetClipPlaneForFeatureEdges(bool);     // clip feature edges
  void SetDrawBoundaryEdges(bool);            // boundary edges
  int GetFeatureAngle();                      // get feature angle
  void SetFeatureAngle(int);                  // set feature angle
  void SetLineWidthForFeatureEdges(int);      // feature line with
  void SetTubeQualityForFeatureEdges(int);    // feature tube quality
  void SetTubeRadiusForFeatureEdges(int);     // feature tube radius
  void SetClipPlaneOx(double);                // clip plane origin
  void SetClipPlaneOy(double);                // clip plane origin
  void SetClipPlaneOz(double);                // clip plane origin
  void SetClipPlaneNx(double);                // clip plane normal
  void SetClipPlaneNy(double);                // clip plane normal
  void SetClipPlaneNz(double);                // clip plane normal

surfaces:
  QString GetFieldName();                     // get field name
  bool SetFieldName(QString);                 // set field name
  void SetMinVal(double);                     // minimum
  void SetMaxVal(double);                     // maximum
  void SetKeepLimits(bool);                   // keep limits
  void SetComputeNormals(bool);               // shade model
  void SetFeatureAngle(int);                  // set feature angle
  void SetOpacity(int);                       // set opacity
  void SetClipPlane(bool);                    // set clipping

vectors:
  QString GetFieldName();                     // get field name
  QString GetColorName();                     // get color name
  QString GetThresholdName();                 // get threshold name
  bool SetFieldName(QString);                 // set field name
  bool SetColorName(QString);                 // set color name
  bool SetThresholdName(QString);             // set threshold name
  void SetLength(int);                        // set arrow length
  void SetQuality(int);                       // set arrow quality
  void SetEveryNth(int);                      // reduce arrows
```

```
  void SetClipPlane(bool);                    // set clipping
  void SetComputeNormals(bool);               // smooth arrows
  void SetRandomMode(bool);                   // reduce randomly
  void SetScaleByMagnitude(bool);             // scaling
  void SetMinColorVal(double);                // color min
  void SetMaxColorVal(double);                // color max
  void SetKeepColorLimits(bool);              // keep color limits
  void SetMinThresholdVal(double);            // treshold min
  void SetMaxThresholdVal(double);            // threshold max
  void SetUseThreshold(bool);                 // use threshold
  void SetKeepThresholdLimits(bool);          // keep threshold limits

isoContours:
  QString GetFieldName();                     // get field name
  QString GetColorName();                     // get color field name
  bool SetFieldName(QString);                 // set field name
  bool SetColorName(QString);                 // set color field name
  void SetMinFieldVal(double);                // minimum
  void SetMaxFieldVal(double);                // maximum
  void SetContours(int);                      // number of contours
  void SetKeepFieldLimits(bool);              // keep limits
  void SetMinColorVal(double);                // color min
  void SetMaxColorVal(double);                // color max
  void SetKeepColorLimits(bool);              // keep color limits
  void SetUseTubeFilter(bool);                // draw with tubes
  void SetUseClipPlane(bool);                 // clip
  void SetLineWidth(int);                     // line width
  void SetTubeQuality(int);                   // tube quality
  void SetTubeRadius(int);                    // tube radius

isoSurfaces:
  QString GetFieldName();                     // get field name
  QString GetColorName();                     // get color name
  bool SetFieldName(QString);                 // set field name
  bool SetColorName(QString);                 // set color name
  void SetMinFieldVal(double);                // minimum
  void SetMaxFieldVal(double);                // maximum
  void SetContours(int);                      // number of contours
  void SetKeepFieldLimits(bool);              // keep limits
  void SetMinColorVal(double);                // color min
  void SetMaxColorVal(double);                // color max
  void SetKeepColorLimits(bool);              // keep color limits
  void SetComputeNormals(bool);               // shade model
  void SetUseClipPlane(bool);                 // clip
  void SetFeatureAngle(int);                  // set feature angle
  void SetOpacity(int);                       // set opacity

streamLines:
  QString GetFieldName();                     // get field name
```

```
  QString GetColorName();                    // get color field name
  bool SetFieldName(QString);                // set field name
  bool SetColorName(QString);                // set color field name
  void SetMaxTime(double);                   // max time
  void SetStepLength(double);                // time step length
  void SetThreads(int);                      // number of threads
  void SetIntegStepLength(double);           // integ. step length
  void SetUseSurfaceMesh(bool);              // use surface mesh
  void SetUseVolumeMesh(bool);               // use volume mesh
  void SetIntegrateForwards(bool);           // integrate forwards
  void SetIntegrateBackwards(bool);          // integrate backwards
  void SetMinColorVal(double);               // color minimum
  void SetMaxColorVal(double);               // color maximum
  void SetKeepColorLimits(bool);             // keep color limits
  void SetDrawLines(bool);                   // draw lines
  void SetDrawRibbons(bool);                 // draw ribbons
  void SetLineWidth(int);                    // line width
  void SetRibbonWidth(int);                  // ribbon width
  void SetSphereSource(bool);                // sphere source
  void SetLineSource(bool);                  // line source
  void SetPointSource(bool);                 // point source
  void SetSphereSourceX(double);             // sphere origin
  void SetSphereSourceY(double);             // sphere origin
  void SetSphereSourceZ(double);             // sphere origin
  void SetSphereSourceRadius(double);        // sphere radius
  void SetSphereSourcePoints(int);           // nof points in sphere
  void SetLineSourceStartX(double);          // line start
  void SetLineSourceStartY(double);          // line start
  void SetLineSourceStartZ(double);          // line start
  void SetLineSourceEndX(double);            // line end
  void SetLineSourceEndY(double);            // line end
  void SetLineSourceEndZ(double);            // line end
  void SetLineSourcePoints(int);             // nof point on line

colorBar:
  bool SetFieldName(QString);                // set field name
  void SetLayoutHorizontal(bool);            // draw horizontal
  void SetLayoutVertical(bool);              // draw vertical
  void SetAnnotateFieldName(bool);           // draw field name
  void SetLabels(int);                       // nof labels
  void SetLineWidth(double);                 // line width
  void SetLength(double);                    // set length

timeStep:
  void SetCurrent(int);                      // current time step
  void SetStart(int);                        // first step
  void SetStop(int);                         // last step
  void SetIncrement(int);                    // increment
  void SetMatcCmd(QString);                  // matc cmd
```

```
  void SetRegenerateBeforeDrawing(bool);       // regenerate actors
  void SetSaveFrames(bool);                    // save picture files
  void SetSaveDirectory(QString);              // set directory
  void Loop();                                 // start/stop loop
  bool IsLooping();                            // is looping?
  void DrawCurrent();                          // draw current frame
```

References

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

[2] Elmerfem in sourceforge: http://sourceforge.net/projects/elmerfem.

[3] Mingw home: http://www.mingw.org/.

[4] Netgen web pages: http://www.hpfem.jku.at/netgen/.

[5] Occ web pages: http://www.opencascade.org/.

[6] Qt web pages: http://trolltech.com/products/qt/.

[7] Qwt web pages: http://qwt.sourceforge.net/.

[8] Tetgen web pages: http://tetgen.berlios.de/.

[9] Visual studio 2008 express edition: http://www.microsoft.com/express/.

[10] Vtk web pages: http://www.vtk.org/.

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 millisecons:
<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 behaviour of ElmerGUI by editing this file. For example, to turn off the splash screen at start up, change the value of the tag `<splshscreen>` 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.

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 he interpolant may be arbitrary.

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 occurances.

The basic stucture 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 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 N. 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>
```

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

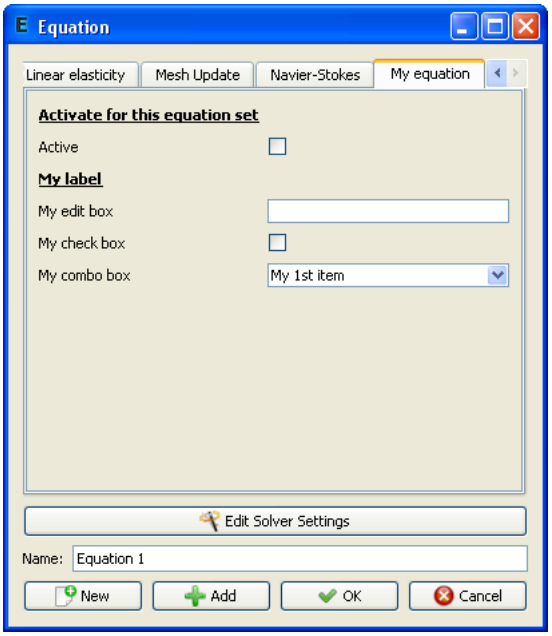

Figure 14: Equation tab in Model menu produced by the sample ed-file.

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
```

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;
}
```

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 elemet 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 inde 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.