



# ELMER

**Guide to FEM simulations**

Authors:

Manuel Carmona

José María Gómez

José Bosch

Manel López

Óscar Ruiz



November/2019

Barcelona, Spain.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License



<http://creativecommons.org/licenses/by-nc-nd/3.0>



## Table of contents

I.	Introduction to Elmer .....	5
II.	Elmer GUI.....	6
III.	Elmer commands .....	10
III.1.	Header section.....	11
III.2.	Constants section .....	12
III.3.	Simulation section.....	12
III.4.	Body section .....	13
III.5.	Material section.....	14
III.6.	Body Force section .....	14
III.7.	Initial Condition section.....	14
III.8.	Boundary Condition section .....	14
III.9.	Equation section.....	15
III.10.	Solver section.....	15
III.11.	Examples.....	16
i)	Flow through a circular tube .....	16
IV.	Solvers.....	19
IV.1.	SaveScalars .....	19
IV.2.	SaveLine .....	20
IV.3.	Fluidic Force .....	21
IV.4.	Particle Dynamics .....	22
V.	Examples.....	25
V.1.	Obstructed artery (axi-symmetric).....	25
VI.	ElmerPost.....	26
VI.1.	Graphics window .....	26
VI.2.	Commands window .....	27
VI.3.	Commands line .....	32
VII.	Types of simulations .....	35
VII.1.	Transient .....	35
VII.2.	Coupled simulations .....	36
VII.3.	Axisymmetric models-simulations .....	36
VIII.	UDFs and Solver Code.....	37



VIII.1. Defining a solver.....	38
IX. ParaView post-processor .....	44
IX.1. Elmer-ParaView connection.....	44
IX.2. General ParaView concept.....	44
IX.3. ParaView GUI.....	45
IX.4. Filters .....	46
IX.4.1. Calculator filter .....	47
IX.4.2. Glyph filter .....	47
IX.4.3. Stream Tracer filter.....	48
IX.4.4. Contour filter.....	49
IX.4.5. Warp by Vector filter .....	50
IX.4.6. Warp by Scalar filter .....	51
IX.4.7. Clip filter.....	51
IX.4.8. Reflect filter .....	52
IX.4.9. Rotational Extrusion filter.....	52



# I. Introduction to Elmer

Elmer [1] is a combination of different software aimed at the simulation of multiphysics problems using the Finite Element Method (FEM). Three of these software are: ElmerGUI, ElmerSolver, ElmerPost. Elmer is an open source software, released under the GNU General Public License (GPL).

Elmer can be used in two different ways (or combining both):

- ⦿ By using its Graphical User Interface (GUI). (A command text file can be generated after a GUI session).
- ⦿ By using a command text file.

Elmer has not good capabilities for geometry generation and meshing. Therefore, as a general procedure, the geometry and mesh should be imported into Elmer. It accepts different geometry and mesh formats. Among them, it accepts the GMSH mesh format (although sometimes we have to generate old formats to be able to import it, using for example the option '*-format msh2*').



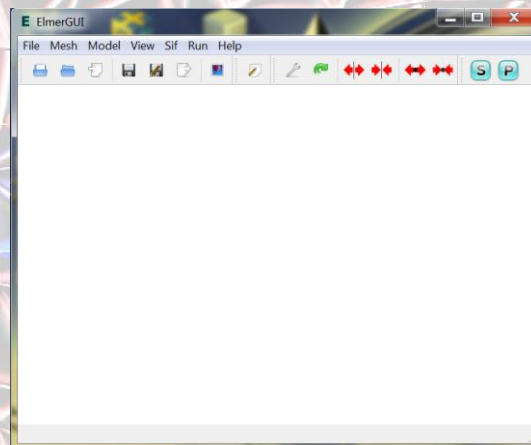
## II. Elmer GUI

Most of the things that can be done in Elmer can be accessed by its GUI. Nevertheless, there are some specific things that will have to be done by manipulating its input script (explained in later chapters).

This chapter is going to explain the most common options used to define the boundaries and loads, and the definition of the simulation parameters (type of simulation and its options).

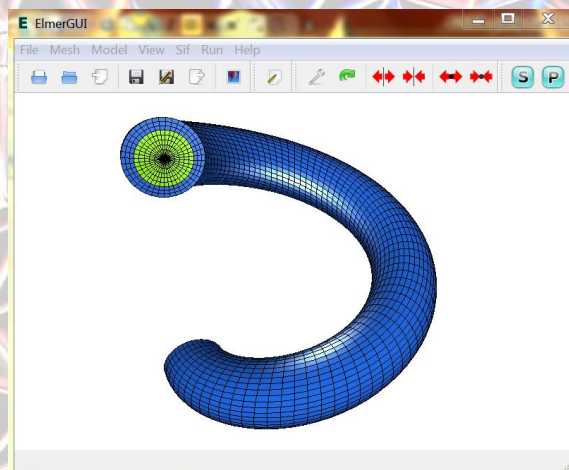
We will assume that the geometry and the mesh of the problem has been generated by an external software. Indeed, it will be assumed that Gmsh<sup>2</sup> was used for this purpose.

Before running ElmerGUI, the best thing is to create a directory, where we will create the Elmer project and put there the mesh file from Gmsh. We will import this mesh in Elmer, although once imported this file will never be used again by Elmer. Now, we can run ElmerGUI. It will appear the GUI, like shown in *Figure 1*.



**Figure 1.** Elmer GUI.

First, we will import the mesh file, with *File* → *Open*. You should be able to view the mesh that was generated with Gmsh:

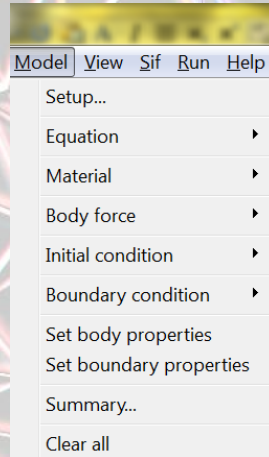


**Figure 2.** Importing mesh.



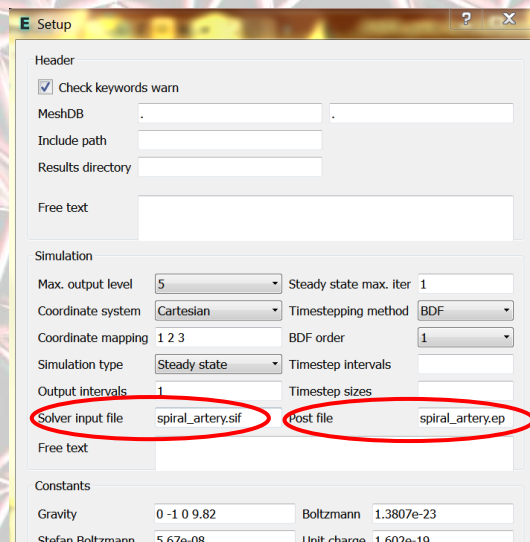
By using the mouse, we can change the view of the model: Left (rotation), center (pan), wheel (zoom). You can reset the view parameters in *View* → *Model reset view*.

In order to define all the boundaries and loads of the model, we will have to go through the option of the *Model* section of the menu, as shown in *Figure 3*.



**Figure 3. Model submenu.**

First of all, we will go to *Model* → *Setup*. There are several parameters that we will have to set, but most of them we will see them at the end of this section. By now we will just change the names of the files: '*Solver input file*' and '*Post file*':



**Figure 4. Setup.**

The parameters in the simulation options are:

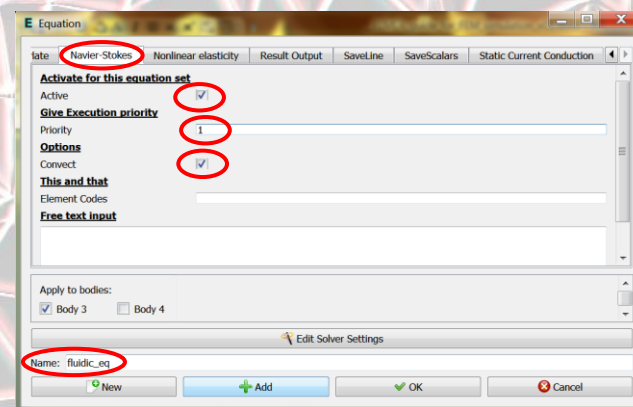
- ⊙ *Max. output level*: It refers to the quantity of information provided by Elmer during simulation. A high level means more information.
- ⊙ *Coordinate system*: Here we can also select axi-symmetric axis.



- ⊙ *Coordinate mapping*: To change the relationship between the axis and the associated numbers.
- ⊙ *Simulation type*: For selecting steady-state or transient simulations.
- ⊙ *Steady state max. iter*: Maximum number of iterations for obtaining convergence for every steady state or time point. It should be bigger than 1 for nonlinear and bi-directional coupled problems.
- ⊙ *Timestepping method*: Here we can choose which timestepping method to use in transient simulations.
- ⊙ *BDF order (Backward Difference Formula)*: We can choose the order of the BDF method (approximation of the derivatives).
- ⊙ *Timestep intervals*: Number of timesteps to be run.
- ⊙ *Timestep sizes*: The value of the time increments in transient simulation.
- ⊙ *Output intervals*: Frequency of the simulated timesteps to be saved to the results file.

Maybe now it is a good moment to save the project, and we will repeat this action at the end of the process. For this, just go to *File* → *Save Project* and select the directory we already created previously.

The next step consists in defining which equations we want to solve in every part (body) of our model. We have to go to *Model* → *Equation* → *Add*. It appears a window with different tabs corresponding each one to a specific equation of a specific field (fluidic, thermal, etc.). We have to activate each equation we want to solve, specifying to which body applies these activated equations. More than one equation can be solved in the same body.

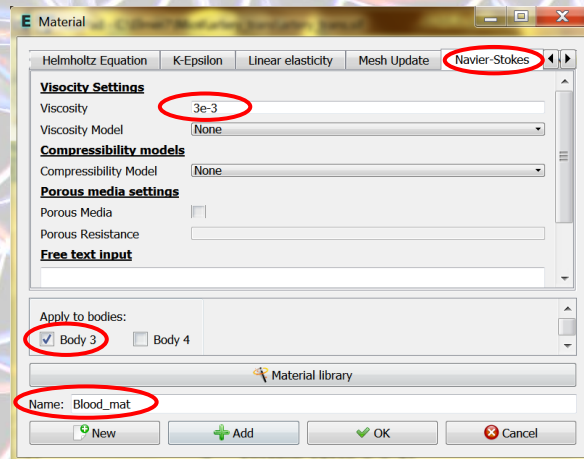


**Figure 5. Equations setup.**

In this case, we have shown the application of the fluidic equation to Body 3 (the fluidic volume). We activate the application for Body 3, set an appropriate name for this equation setup and change the solver parameters that apply in each case (for example, we have set the priority number to 1). We have to apply this also to the Body 4, but with another equation (like 'NonLinear elasticity', for example).

Now we can set the material properties of our bodies: blood and membrane. We have to go to *Model* → *Material* → *Add*. In general, we have to set the corresponding properties in the General tab and in those corresponding to the activated equations tabs.

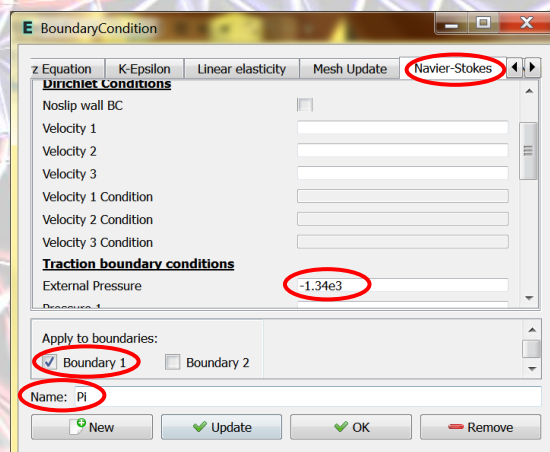




**Figure 6. Materials setup.**

Next two steps are the definition of body forces and initial conditions. In our case, we are not going to set them.

And now we set the boundary conditions. We have to go to *Model* → *Boundary* → *Add*. If we have some doubts on the number of the surfaces, we can double-click to show at the bottom of the window which surface number has been selected.



**Figure 7. Boundaries setup.**

Here we have to apply the boundaries in all the appropriate tabs, depending on the ones that were selected for the equations for each body. Clicking on 'Return' inside a box will pop up a window to enter more than a line (for example, for introducing a table or an equation).

Once we have defined all these steps, we can generate the SIF file (Elmer script file) (states for Solver Input File) with *Sif* → *Generate* and save again the project (*Save Project*).

And finally, we could already start running the solution with *Run* → *Start solver*.

If we have to modify the SIF file manually, we will have to avoid generating the SIF file again and even save the project (because it also generates the SIF file again). When you open the project, try also to have the Sif file open with an editor to avoid overriding it.



### III. Elmer commands

The Elmer commands file (named Solver Input File (SIF)) has a defined structure for defining the different aspects related to the simulation of a defined mesh. Therefore, these commands define aspects like:

- ⊙ Boundary conditions.
- ⊙ Loads.
- ⊙ Solvers.
- ⊙ Simulation parameters.

This file can be generated from zero, modified from another already existing file or create it graphically with ElmerGUI after specifying all needed options. We will make an introduction to ElmerGUI in the next chapter. In this chapter we will show some of the commands that we will find in this file.

The command file has a default extension SIF (Solver Input File). This file is organized in sections. Each section starts with the name of the section (header), followed by the commands applied to this section and it is ended by an *End* command. The different headers can be:

- ⊙ *Header*: Generally just used for indicating the location of the mesh files.
- ⊙ *Constants*: For defining constants.
- ⊙ *Simulation*: To provide parameters related to simulation, like the type of simulation, name of output file, etc.
- ⊙ *Body n*: Provides the material, body forces, equation, solver, boundary conditions and initial conditions used for simulating the body number *n*. This *n* number identifies the body in the mesh file.
- ⊙ *Material n*: For defining material properties.
- ⊙ *Body Force n*: For defining body forces.
- ⊙ *Initial Condition n*: For defining initial conditions.
- ⊙ *Boundary Condition n*: For defining a boundary condition.
- ⊙ *Equation n*: For defining an equation set to be solved.
- ⊙ *Solver n*: For defining a solver to be used.

All headers containing '*n*' can be found multiple times. For distinguishing each one, we have to provide a different integer number (*n*).

There are only a few things that can be put outside a section:

- ⊙ *Line Comments*: They must start with a '*'*' symbol.
- ⊙ *MATC expressions*: MATC allows to define mathematical expressions in Elmer. They are also used to define parameters/variables that will be used afterwards in sections. In general, MATC expressions start with a '*\$*' symbol.
- ⊙ *Check Keywords "Warn"*: This command activates outputting warning messages.
- ⊙ *echo on* and *echo off*.



Many commands in Elmer consist of assigning a value to an Elmer keyword, by using the equal sign ('='). For the assigned value, we can specify the type of variable (Real, Integer, Logical, String or File) before the value is given. The value can also be a vector (example: *temps(3)= 300 320 340*) or a 2D array (example: *cond(2,3) = Real 3.0 2.0 1.0 \*  
*4.0 3.0 2.0*).

(The slash is not needed in last versions of Elmer)

Each value of the vector can be obtained with round brackets: *temps(2)*.

In some occasions, parameters can depend on variables like time, position, temperature, or on a variable that we want to solve for. These dependencies can be defined by a table or by a MATC expression. An example using a table for defining a thermal conductivity as function of temperature is the following:

```
Conductivity = Variable Temperature
Real
    273  95.2
    273 1000
    300 1020
    400 1000
End
```

We could do a similar thing by using a MATC expression:

```
Conductivity = Variable Temperature
MATC "k0*(1+alpha*tx)"
```

'tx' is used for the independent variable (in this case the temperature). We could have more than one variable. In that case, we use the vector notation (example: *tx(1)* or *tx(2)*)

Examples of variables that can be used are: *Time, Temperature, Pressure, Displacement 1, Coordinate 1, Electric Current 1, Magnetic Field 1*, etc.

Other language aspects:

- ⦿ '::': Two semicolons are used for separating two instructions in the same line.
- ⦿ *RUN*: Instruction for executing the FEM solution.

The mesh is divided in parts called bodies. For GMSH, these bodies are related to the physical groups created during the mesh generation.

Let's see more specific commands for each section.

### III.1. Header section

In this section we will define the location and names of the different files related with Elmer. We can just put a command for declaring the location of the mesh files:



```
Header  
Mesh DB "directory" "meshfilename"  
End
```

Another example:

```
Header  
CHECK KEYWORDS Warn  
Mesh DB ". ". "  
Include Path ""  
Results Directory ""  
End
```

We can define more things in this section, but we will not use it. Just for mentioning them, we can indicate how many of the other sections exist in this file, we can also indicate another directory for placing the results (*Results Directory "directory"*) and include other paths if needed (*Include Path "directory"*). We can also place here the command *CHECK KEYWORDS Warn*.

### III.2. Constants section

In this section we define constants if we need it. The specific model that we want to solve can require the definition of some constants. This is explained in the Models Manual of Elmer for each specific simulation field (fluidic, thermal, etc).

```
Constants  
Gravity(4) = 0 -1 0 9.82  
Stefan Boltzmann = 5.67e-08  
Permittivity of Vacuum = 8.8542e-12  
Boltzmann Constant = 1.3807e-23  
Unit Charge = 1.602e-19  
End
```

### III.3. Simulation section

In this section we provide some parameters defining general aspects of the simulation procedure itself (independently of the specific field: thermal, mechanical, etc.) that we want to perform. For example, if the simulation is transient or stationary, coordinate system, time steps, etc. More specifically, we can define:

- ⦿ *Simulation Type*: With the keywords *Transient* or *Steady State*.



- ⊙ *Coordinate Mapping*: It is a vector of numbers providing the relation between the coordinates in the mesh file and the coordinates in the simulation. If they are the same, we will use 1, 2 and 3 for a 3-dimensional case.
- ⊙ *Coordinate System*: With a text keyword for defining the type of coordinate system (*Cartesian 1D*, *Cartesian 2D*, *Cartesian 3D*, *Polar 2D*, *Polar 3D*, *Cylindric*, *Cylindric Symmetric*, *Axi Symmetric*).
- ⊙ *Timestepping Method*: String with five possible options: *BDF*, *Newmark*, *Implicit Euler*, *Explicit Euler* and *Crank-Nicolson*.
- ⊙ *Timestep Intervals*: Vector of integers defining the number of intervals (or substeps) inside every time step.
- ⊙ *Timestep Sizes*: Vector providing the size in the time units of every time step.
- ⊙ *Output File*: Name of the results output file (.dat).
- ⊙ *Output Intervals*: Vector of integers providing the frequency of the obtained results that will be saved to the output file.
- ⊙ *Post File*: Name of the results file that is understood by Elmer Post. (.ep)
- ⊙ *Steady State Max Iterations*: Maximum number of iterations of every time calculation in order to get a converged solution.

#### *Simulation*

```

Max Output Level = 5
Coordinate System = Cartesian
Coordinate Mapping(3) = 1 2 3
Simulation Type = Steady state
Steady State Max Iterations = 1
Output Intervals = 1
Timestepping Method = BDF
BDF Order = 1
Solver Input File = tube.sif
Post File = tube.ep
End
  
```

### III.4. Body section

It is used to define, for each body of the mesh file (or created with Elmer), which other sections apply for it. We have to specify which *Equation*, *Material*, *Body Force* and *Initial Condition* sections applies.

#### *Body 1*

```
Target Bodies(1) = 1
```



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

### III.5. Material section

Here we define the properties of the material. The properties that we have to define depend on the kind of simulation that we want to perform, like for example thermal or fluidic. These properties have to be checked on the Elmer Models Manual.

```
Material 1  
Name = "Fluid"  
Viscosity = 1.0  
Density = 1e3  
End
```

### III.6. Body Force section

In this section we specified the loads applied. The loads that we have to specify depend on the kind of simulation. We have to obtain this information from the Elmer Models Manual.

### III.7. Initial Condition section

Similarly to body forces, we have to check the Elmer Models Manual to obtain the initial conditions that can be applied to our specific simulation.

### III.8. Boundary Condition section

In this section, we define first the boundary where it will be applied (assigning their number to the vector *Target Boundaries*). And afterwards, we define the boundaries conditions that apply to them. The possible options have to be found in the Elmer Models Manual.

```
Boundary Condition 1  
Target Boundaries(1) = 2  
Name = "Output P"
```



```

Pressure 1 = 0
Pressure 3 = 0
Pressure 2 = 1
End
  
```

It can be mentioned that some Dirichlet boundary conditions come together with the option of “Condition” (for example, “Temperature Condition” or “Displacement 1 Condition”). The boundary condition will be applied to the model only if this “Condition” is positive.

### III.9. Equation section

Each equation is related to a specific physical model (thermal, fluidic, etc). In this section we have to indicate which equation(s) will apply to a body. It can be one, or more than one. We refer to each equation by using the number(s) of the respective solver sections. We associate these numbers to the *Active Solvers* vector.

```

Equation 1
Name = "Fluidic equation"
NS Convect = False
Active Solvers(1) = 1
End
  
```

### III.10. Solver section

Here we specify one physical model to be solved and some options related to this physical model and which method will be used to solve it (solver). The name of each equation and the different options can be obtained in the Elmer Models Manual. The options for solvers and their options can be found in the Solvers Manual.

General options in this section are:

- ⊙ *Variable = Variable\_name*: For defining a variable with name *Variable\_name*. A vector can also be defined with names of subcomponents. Example: *Variable = vp[Vel:3, P:1]* (defines a variable called *vp* with four components, the first three are called *Vel* and the fourth component is called *P*).  
If we only need to specify a variable with three components: *Variable = -dofs 3 vp*.
- ⊙ We can define when to execute a solver. By default, if we have more than one solver, they are executed in the order that they are defined. With the command *Exec Solver order* we can change this *order* is a string that can have values of: *never*, *always*, *before timestep*, *after timestep*, *before all*, *after all*, *before saving* and *after saving*.



### *Solver 1*

*Equation = Navier-Stokes*

*Procedure = "FlowSolve" "FlowSolver"*

*Variable = Flow Solution[Velocity:3 Pressure:1]*

*Exec Solver = Always*

*Stabilize = True*

*Bubbles = False*

*Lumped Mass Matrix = False*

*Optimize Bandwidth = True*

*Steady State Convergence Tolerance = 1.0e-5*

*Nonlinear System Convergence Tolerance = 1.0e-7*

*Nonlinear System Max Iterations = 20*

*Nonlinear System Newton After Iterations = 3*

*Nonlinear System Newton After Tolerance = 1.0e-3*

*Nonlinear System Relaxation Factor = 1*

*Linear System Solver = Iterative*

*Linear System Iterative Method = BiCGStab*

*Linear System Max Iterations = 500*

*Linear System Convergence Tolerance = 1.0e-10*

*Linear System Preconditioning = ILU0*

*Linear System ILUT Tolerance = 1.0e-3*

*Linear System Abort Not Converged = False*

*Linear System Residual Output = 1*

*Linear System Precondition Recompute = 1*

*End*

## III.11. Examples

### i) Flow through a circular tube

#### *Header*

*CHECK KEYWORDS Warn*

*Mesh DB ". " ". "*

*Include Path ""*

*Results Directory ""*

*End*

#### *Simulation*

*Max Output Level = 5*

*Coordinate System = Cartesian*



```
Coordinate Mapping(3) = 1 2 3
Simulation Type = Steady state
Steady State Max Iterations = 1
Output Intervals = 1
Timestepping Method = BDF
BDF Order = 1
Solver Input File = tube.sif
Post File = tube.ep
End

Constants
Gravity(4) = 0 -1 0 9.82
Stefan Boltzmann = 5.67e-08
Permittivity of Vacuum = 8.8542e-12
Boltzmann Constant = 1.3807e-23
Unit Charge = 1.602e-19
End

Body 1
Target Bodies(1) = 1
Name = "Body 1"
Equation = 1
Material = 1
End

Solver 1
Equation = Navier-Stokes
Procedure = "FlowSolve" "FlowSolver"
Variable = Flow Solution[Velocity:3 Pressure:1]
Exec Solver = Always
Stabilize = True
Bubbles = False
Lumped Mass Matrix = False
Optimize Bandwidth = True
Steady State Convergence Tolerance = 1.0e-5
Nonlinear System Convergence Tolerance = 1.0e-7
Nonlinear System Max Iterations = 20
Nonlinear System Newton After Iterations = 3
Nonlinear System Newton After Tolerance = 1.0e-3
Nonlinear System Relaxation Factor = 1
Linear System Solver = Iterative
Linear System Iterative Method = BiCGStab
Linear System Max Iterations = 500
Linear System Convergence Tolerance = 1.0e-10
Linear System Preconditioning = ILU0
```



*Linear System ILUT Tolerance = 1.0e-3*  
*Linear System Abort Not Converged = False*  
*Linear System Residual Output = 1*  
*Linear System Precondition Recompute = 1*  
*End*

*Equation 1*  
*Name = "Fluidic equation"*  
*NS Convect = False*  
*Active Solvers(1) = 1*  
*End*

*Material 1*  
*Name = "Fluido"*  
*Viscosity = 1.0*  
*Density = 1e3*  
*End*

*Boundary Condition 1*  
*Target Boundaries(1) = 2*  
*Name = "Output P"*  
*External Pressure = 0*  
*End*

*Boundary Condition 2*  
*Target Boundaries(1) = 3*  
*Name = "wall"*  
*Noslip wall BC = True*  
*End*

*Boundary Condition 3*  
*Target Boundaries(1) = 1*  
*Name = "Input vz"*  
*Velocity 3 = 1.0e-3*  
*End*



## IV. Solvers

### IV.1. SaveScalars

```
[ Equation = SaveScalars
  Procedure = "SaveData" "SaveScalars"]
```

It is used for two purposes: saving results to a file and calculating derived quantities (those that depend on the results variables). Data is saved in ASCII format.

The main options for this solver are:

- ⊙ Name of the file to be written: *Filename = filename.*
- ⊙ Specify the variables to be saved: *Variable i = namevar.*
- ⊙ Operator to be applied to variable *i*: *Operator i = op.*
- ⊙ Specify a factor applied to the operator *i*: *Coefficient i = coef.*
- ⊙ Restrict the *n* nodes (*points list*) to be saved: *Save Points(n)= points list.*

If we need to perform a calculation (like fluxes) over a boundary, we have to activate the option of *Save Scalars* in the respective Boundary section (*Save Scalars = True*).

The different operators that we can use are: *max, min, max abs, min abs, mean, variance, deviation; volume, int mean, int variance; boundary sum, boundary dofs, boundary mean, boundary max, boundary min, boundary max abs, boundary min abs, area, boundary int, boundary int mean; diffusive energy, convective energy, potential energy; diffusive flux, convective flux, boundary int, boundary int mean, area; dofs, norm, nonlinear change, steady state, nonlin iter, nonlin converged, coupled converged, bounding box, partitions.*

The same file, but adding the extension *.names* is created indicating the meaning of each column of the saved data file.

This would be an example for saving all degrees of freedom at node 4:

```
Solver 4
  Exec Solver = After Timestep
  Equation = SaveScalars
  Procedure = "SaveData" "SaveScalars"
  Filename = "ss_d01.dat"
  Save Points(1) = 4
End
```

```
Equation 3
  Name = "Save scalar values"
  Active Solvers(1) = 4
End
```



This another example is for obtaining the volume flow at a boundary:

```
Solver 4  
  Exec Solver = After Timestep  
  Equation = SaveScalars  
  Procedure = "SaveData" "SaveScalars"  
  Filename = "ss_d02.dat"  
  Moving Mesh = logical True  
  Variable 1 = Velocity 2  
  Operator 1 = boundary int  
End
```

```
Boundary Condition 2  
  Target Boundaries(1) = 2  
  Name = "Po"  
  Mesh Update 2 = 0  
  External Pressure = 0  
  Save Scalars = True  
End
```

## IV.2. SaveLine

```
[ Equation = SaveLine  
  Procedure = "SaveData" "SaveLine" ]
```

It is used for saving result values along a line to a file, in ASCII format.

The main options for this solver are:

- ⊙ Name of the file to be written: *Filename = filename.*
- ⊙ Specify the variables to be saved: *Variable i = namevar.*
- ⊙ Definition of the lines: *Polyline Coordinates(n,dim) = pointscoordinates.*  
*n* is the number of points of the lines and *dim* the spatial dimensions (1, 2 or 3). *n* must be even, as each line is defined by two points.

We can also indicate to do this calculation for already defined boundaries. In that case, we have to activate the option of *Save Line* in the respective Boundary section (*Save Line = True*).

As in the case of *SaveScalars*, an additional file with extension *.names* is created.



This example shows how to save the velocity profile on a fluidic channel as function of the position:

```
Solver 5
Exec Solver = After Simulation
Equation = SaveLine
Procedure = "SaveData" "SaveLine"
Filename = "sl_d01.dat"
Variable 1 = Coordinate 1
Variable 2 = Velocity 2
Polyline Coordinates(2,2) = 0.0 0.0 0.5e-3 0
End
```

### IV.3. Fluidic Force

```
[ Equation = Fluidic Force
Procedure = "FluidicForce" "ForceCompute"]
```

It is used to compute the fluidic forces applied by fluids to solid boundaries. We can obtain two forces: the normal force and the tangential (shear) force. Additionally, we can also ask for saving the shear stresses in a file.

The main options for this solver are:

- ⦿ Calculate also the viscous forces: *Calculate Viscous Force = True*.
- ⦿ Calculate also shear stresses: *Shear Stress Output = True*.

For those boundaries where we want to compute these forces, we have to specify it in the corresponding Boundary section activating the variable *Calculate Fluidic Force (Calculate Fluidic Force = True)*.

A file for the shear stresses is created in case it is activated. By default, its name is "shearstress.dat".

This example shows how to define this solver to get these forces:

```
Solver 6
Exec Solver = After Simulation
Equation = Fluidic Force
Procedure = "FluidicForce" "ForceCompute"
Calculate Viscous Force = True
Shear Stress Output = True
End
```



```

Boundary Condition 6
Target Boundaries(1) = 7
Name = "FSI"
Fsi BC = True
Mesh Update 1 = Equals Displacement 1
Mesh Update 2 = Equals Displacement 2
Noslip wall BC = True
Calculate Fluidic Force = True
End
  
```

#### IV.4. Particle Dynamics

```

[ Equation = Particle Dynamics
  Procedure = "ParticleDynamics" "ParticleDynamics" ]
  
```

It is used to compute the trajectories of particles taking into account their dynamics. This means that it is taken into account their inertia, and also it is possible to take into account the interaction between particles (collision and contact models). An application example could be the trajectory of particles within a fluid (without explicitly modeling the particles in the fluid). The forces that are taken into account are: gravity, electrostatic, viscous force ( $-b \cdot (\vec{v} - \vec{v}_0)$ ) and buoyancy.

The main options for this solver are:

Particles: The number of particles and their spatial distribution can be indicated in several different ways.

- ⊙ Number of particles to be considered: *Number of Particles = nparticles.*
- ⊙ Initial spatial distribution of particles: *Coordinate Initialization Method = method.*  
Possible methods are: *nodal ordered, elemental ordered, sphere random, box random* and *box random cubic*. For box options, we have to indicate *Min Initial Coordinate i = imin* and *Max Initial Coordinate i = imax*. For the sphere option we have to specify *Particle Cell Radius = rad.*
- ⊙ Specify initial velocity of particles: *Initial Velocity(n,dim) = velocities.*  
Random velocity distributions can be indicated with different methods: *Velocity Initialization Method = method* (where we can choose *thermal random, even random* and *constant random*). We have also to provide a random amplitude: *Initial Velocity Amplitude = amp.*
- ⊙ To indicate to reinitialize the position of the particles at each call, use: *Reinitialize Particles = True.*
- ⊙ Eliminate particles on the walls (stuck): *Delete Wall Particles = True.*

Timestepping for calculation of trajectories:



- ⊙ We can set up: *Timestep Size, Max Timestep Size, Min Timestep Size, Timestep Distance, Timestep Courant Number, Max Characteristic Speed, Max Timestep Intervals, Max Cumulative Time, and Simulation Timestep Sizes.*

Particle interaction parameters:

- ⊙ Activate particle collisions and contact: *Particle Particle Collision = True, Particle Particle Contact = True.*
- ⊙ Activate particle box collisions and contact: *Box Particle Collision = True, Box Particle Contact = True.*

Physical properties particle interaction parameters:

- ⊙ Related to particle: *Particle Mass, Particle Radius, Particle Gravity = True, Particle Lift=True, Particle Damping, Particle Drag Coefficient, Particle Bounciness, Particle Spring, Particle Charge, Particle Decay Distance.*
- ⊙ Related to wall-particle interaction: *Wall Particle Radius, Wall Particle Spring, Wall Particle Bounciness, Particle Distance = True.*
- ⊙ At boundaries: *Wall Particle Collision = True, Particle Reflect = True.*

As an example, we have just obtained the bouncing of particles on the floor, with a free fall. The domain is simply a box domain:

```

$tsv=0.001
Simulation
  Max Output Level = 5
  Coordinate System = Cartesian
  Coordinate Mapping(3) = 1 2 3
  Simulation Type = transient
  Output Intervals = 1
  Timestepping Method = BDF
  BDF Order = 1
  Timestep intervals = 200
  Timestep Sizes = $tsv
  Solver Input File = pd_tc01.sif
  Post File = pd_tc01.ep
End

Constants
  Gravity(4) = 0 -1 0 9.82
End

Solver 1
  Equation = ParticleDynamics
  Procedure = "ParticleDynamics" "ParticleDynamics"
  Number of Particles = Integer $ 10
  Coordinate Initialization Method = String "box random"
  
```



```
Min Initial Coordinate 1 = Real 0
Max Initial Coordinate 1 = Real 0.1
Min Initial Coordinate 2 = Real 0
Max Initial Coordinate 2 = Real 0.05
Simulation Timestep Sizes = Logical True
Timestep Size = Real $tsv
Time Order = Integer 2
Particle Gravity = Logical True
Particle Mass = Real 0.1
Particle Radius = Real 0.01
Box Particle Periodic = Logical False
Particle Accurate At Face = Logical True
Particle Accurate Always = Logical True
Particle To Field Reset = Logical True
Statistical Info = Logical True
Particle Info = Logical True
Box Particle Collision = Logical True
Wall Particle Spring = Real 100000
Wall Particle Bounciness = Real 1
Vtu Format = Logical True
End

Solver 2
Equation = String "ResultOutput"
Procedure = File "ResultOutputSolve" "ResultOutputSolver"
Output File Name = File "kinetic"
Output Format = String "vtu"
Show Variable = Logical True
End

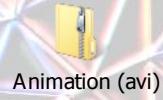
Solver 3
Equation = String "ParticleOutput"
Procedure = File "SaveGridData" "ParticleOutputSolver"
Filename Prefix = String "particles"
Output Format = String "vtu"
End

Equation 1
Navier-Stokes = FALSE
Active Solvers(3) = 1 2 3
End
```



## V. Examples

### v.1. Obstructed artery (axi-symmetric)

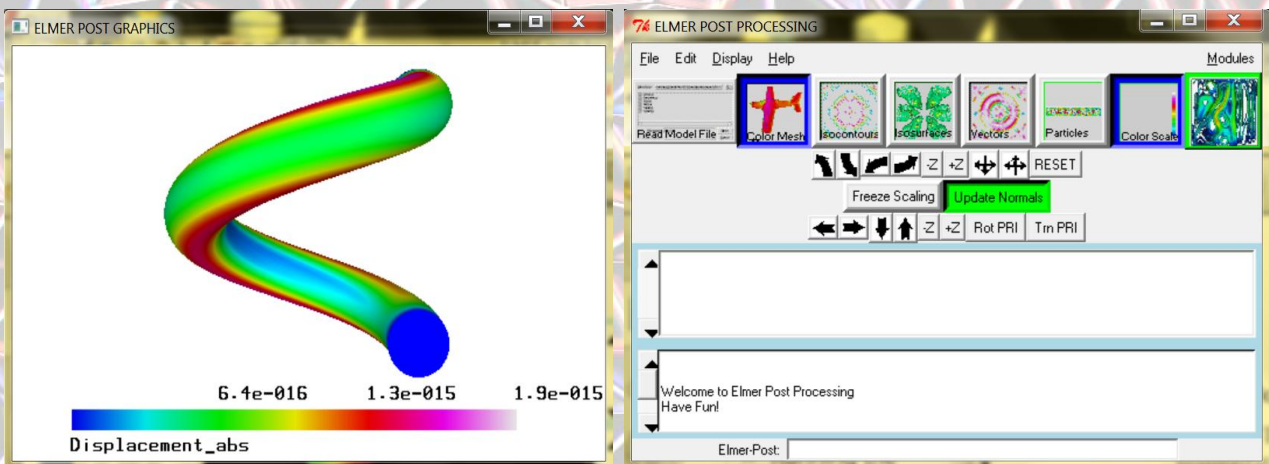


**Figure 8.** Particles (red) flowing through the obstructed artery.



## VI. ElmerPost

Although Elmer comes with two post-processors (ElmerPost and VTK), we will just keep viewing results with ElmerPost. We can run ElmerPost through the ElmerGUI (*Run* → *Start postprocessor*) or directly through its icon. In the first case, the current results file will be read directly, while in the second case we will have to read the results file (.ep). After running the postprocessor, we will get two windows: the graphics window and the commands window (*Figure 9*). Here we are just going to explain some key aspects about using ElmerPost. Additionally, we can see results in other post-processing software, like ParaView [3].

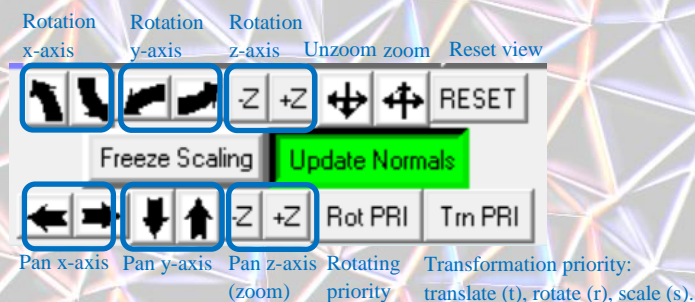


*Figure 9.* Post-processor windows.

### VI.1. Graphics window

In this window, we can do the typical mouse actions that can be expected: rotation (right), zoom (left & right), Pan (left).

In the commands window, we have some buttons controlling the view (Graphics commands), shown in *Figure 10*.



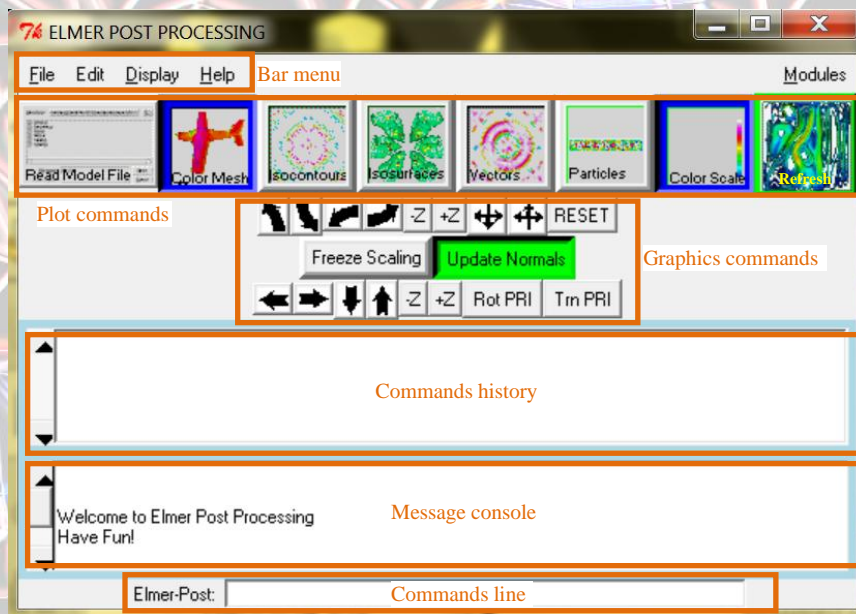
*Figure 10.* Graphics commands.



By default, when something is plotted in the graphics window, the view is rescaled in order to fit it in the window (this is referred as 'Update Normals'). With the 'Freeze Scaling' option, we can avoid changing the scale, keeping the last used scaling factor.

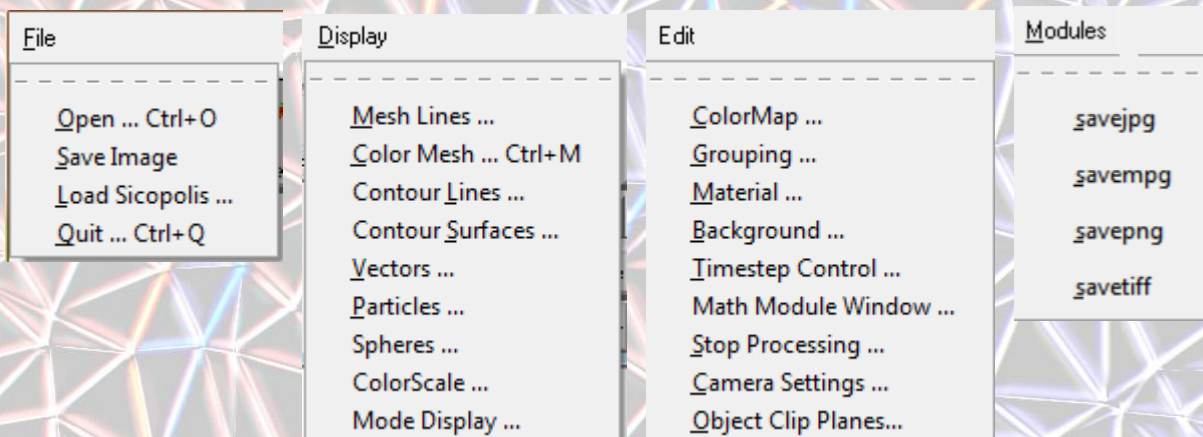
## VI.2. Commands window

We have several regions on this commands window, as shown in *Figure 11*.



*Figure 11. Commands window.*

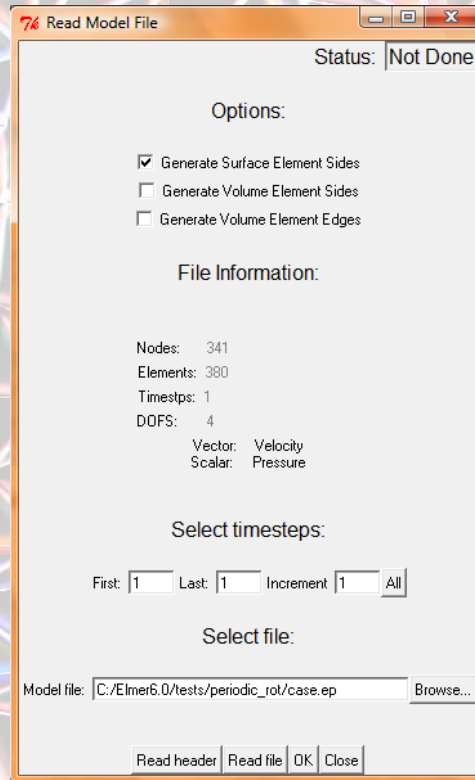
In the bar menu, we can access most of the ElmerPost functionalities and configuration parameters. The different submenus are shown in *Figure 12*.



*Figure 12. Bar submenus.*

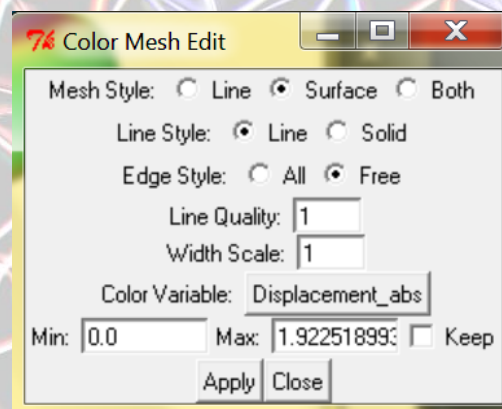


In the plot commands buttons, we have mainly different ways to visualize the obtained results from simulation. Nevertheless, the first button corresponds to the reading of the results file (generally with extension ep, although other formats are also accepted). The options for reading the file are shown in *Figure 13*.



**Figure 13. Read model.**

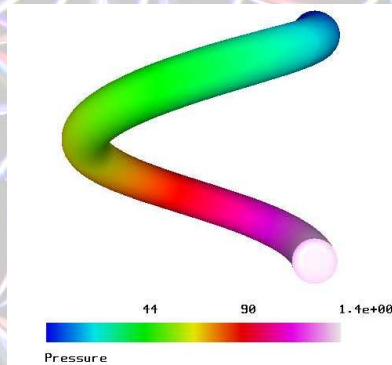
The first way to visualize results is called 'Color Mesh'. Basically, it paints the mesh with colors depending on the values of the results obtained from a variable. In *Figure 14*, it is shown the different options. In 'Color Variable', we can choose which variable to use for painting.



**Figure 14. Color mesh.**

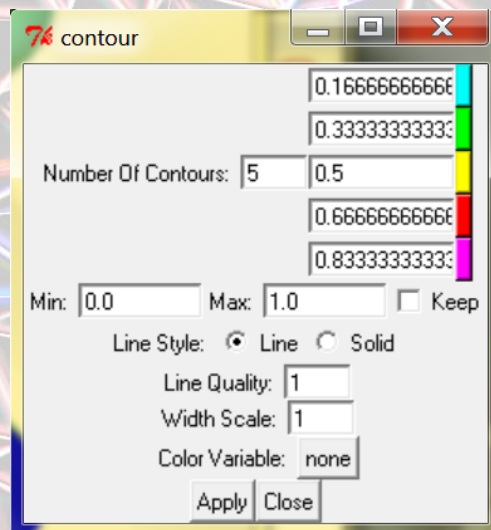
An example is shown in *Figure 15*.



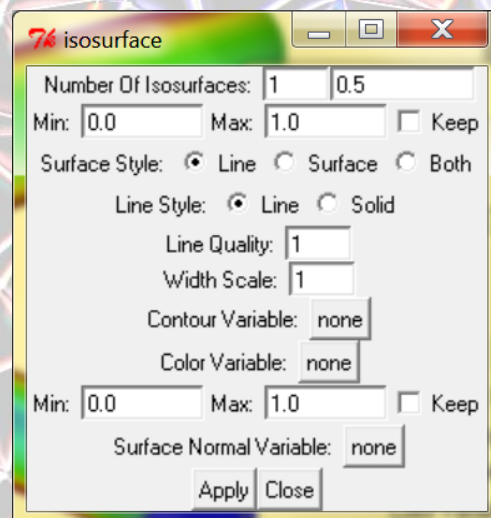


**Figure 15. Color mesh example (pressure distribution).**

With isocontours, we can observe lines having the same value of a certain variable. With isosurfaces, we can obtain surfaces having the same variable value. Isosurfaces can be used for obtained cross-sections of the model, defining the contour variable as position and another variable for 'Color Variable'.



**Figure 16. Isocontours.**



**Figure 17. Isosurface.**

An example for isosurfaces can be seen in *Figure 18*.



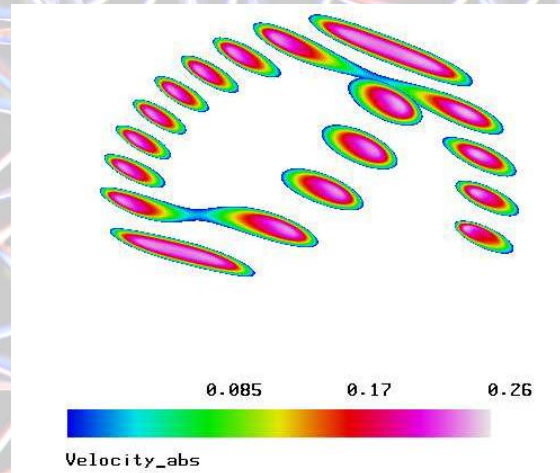


Figure 18. Isosurface example (absolute velocity).

Many times, we will want to draw vector magnitudes as vectors. This can be done with the 'Vectors' button.

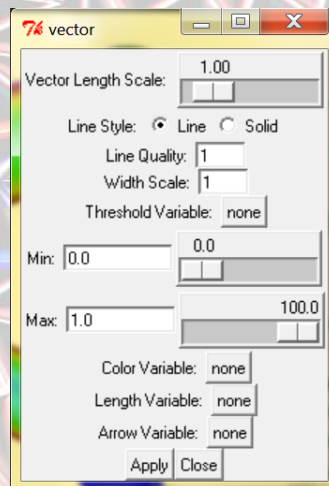


Figure 19. Vectors.

An example is shown in Figure 20, showing the velocity vectors.

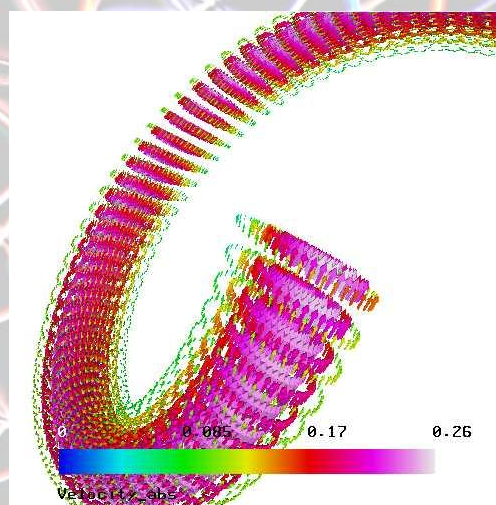


Figure 20. Vectors example (velocity).



The particles button allows us to obtain the trajectories of particles within a fluid. For this purpose we have to set a variable ('*Particle Variable*') containing the initial positions of the particles to be considered. This variable is a matrix, where the second index is referred to the particle number (starting from 0). The first index is for the different parameters that have to be provided for each particle. This parameters are: x, y, z, initial color guess (for example, 0), guess of the initial element number containing the particle (for example 0).



**Figure 21. Particles.**

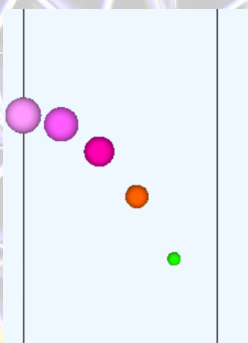
As an example, we have used a simple 2D axisymmetric fluidic problem. We have defined the '*Particle Variable*' named as *part* in the following way:

```

math nps=5;
math rad=5e-4;
do i 0 (nps-1) {
  math posx=$i*rad/nps;
  math part(0,$i)=posx;
  math part(1,$i)=0;
  math part(2,$i)=0;
  math part(3,$i)=0;
  math part(4,$i)=0;
}

```

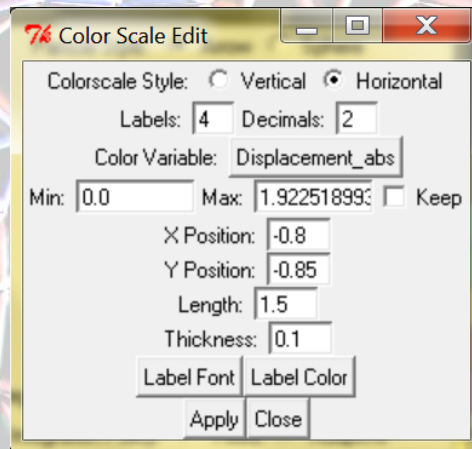
Click '*Apply*' and then '*Advance*' repeatedly in order to obtain the particle positions as time goes on. One intermediate result is shown in *Figure 22*.



**Figure 22. Particles position after some time, starting all at the input.**



The color scale, provided with the colored bar in the graphics window, can be setup with the 'Color Scale' button.



**Figure 23. Color scale.**

The graphics commands are used to change orientation, pan, zoom, etc. We can also fix the scale of the plotted results (with the 'Freeze Scaling' button).

We have a commands line. We can introduce commands in the TCL/TK language and also in the MATC language. MATC is used for managing mathematical calculations using matrices (generally filled with the results data). In this last case (MATC) we have to start the command with '*math*'. We are just going to use it for creating some plots. A more detailed explanation is given in a next section.

Additionally, we have a console where the commands history is plotted and another console for messages.

### VI.3. Commands line

As mentioned previously, we have a commands line where we can add instructions in TCL/TK language and also in MATC language. These commands can be entered manually, or included in a file, executing it with the command '*source file*'. The first language (TCL/TK) is an "external" language, quite used in some other software. Detailed information can be found in books and through internet. The second language (MATC) is a quite specific language for Elmer, which is used for performing calculations with matrices. Elmer documentation comes with a MATC tutorial. We are just going to comment some of the most important aspects of these two languages and apply it to some examples.

In the Time Step Control Window, we can refer to the current time step with the variable  $t$ . And the vector position for a given current time step can be obtained with  $\text{time}(t)$ .



Before starting looking at these two languages, we have to know that, when results are read by ElmerPost, the results are saved in matrices. The names of the variables can be seen in the "Read Model" window. As an example, for a fluidic simulation we will have the variables *Velocity* and *Pressure*. Obviously, these matrices contain the velocity and pressure results respectively. They have as many rows as components (*Velocity* has three rows, each corresponding to the velocities in directions x, y and z respectively, while *Pressure* only have one row). And they have as many columns as nodes we have in the model. In a transient simulation, the second index corresponds to the time point (we can obtain it with the function *time(time\_step)*). Depending on the type of simulation (for example, for transient simulations) there can be more indexes. The relationship between the nodes and the column index is given by another matrix called *nodes*. This matrix contains the position of the nodes of our model. It has three rows corresponding to the x, y and z positions. And it has as many columns as nodes in our model. If we modify the nodes matrix, we will change the position of the nodes, and so it will be shown in the graphics windows. This can be useful sometimes for a more convenient visualization of the model/results, as we will see later.

With the *display* command, we can refresh the graphics window.

Let's see some basics of the TCL/TK programming language:

- ⊙ Comments: #
- ⊙ Command syntax: *command arg1 arg2 arg3 ...*
- ⊙ Variables:
  - ✱ Not declared.
  - ✱ Assign a value: *set var val*.
  - ✱ In order to obtain the value of a variable, we have to use *\$var*.
  - ✱ Command *info* to obtain information about variables (among other things).
  - ✱ Text strings can be grouped with quotes (") or rounded brackets ({}).
  - ✱ All variables are strings. If not, they have to be indicated explicitly with the command *expr*.
- ⊙ Expressions can be grouped with square brackets ([]).
- ⊙ Mathematical functions: *abs(x)*, *acos(x)*, *asn(x)*, *atan(x)*, *atan2(y,x)*, *ceil(x)*, *cos(x)*, *cosh(x)*, *double(x)*, *exp(x)*, *floor(x)*, *fmod(x,y)*, *hypot(x,y)*, *int(x)*, *log(x)*, *log10(x)*, *pow(x,y)*, *rand()*, *round(x)*, *sin(x)*, *sinh(x)*, *sqrt(x)*, *srand(arg)*, *tan(x)*, *tanh(x)*.
- ⊙ Conditional: *if condition then action*.
- ⊙ For loop: *for {initialization} {condition} {increment} instruction*.
- ⊙ Foreach loop: *foreach variable list instruction*.
- ⊙ While loop: *while condition instruction*.

MATC is a library used for evaluating mathematical expressions with matrices. These expressions can be used in the SIF file, also can be evaluated during simulation, and it can also be used in ElmerPost. When a MATC expression is going to be used in ElmerPost, we have to include the word *math* at the beginning of the expression. Some basic aspects of MATC are:

- ⊙ Two types of variables: matrices and strings.
- ⊙ Ranges can be specified: Examples: 0:5, 5:0.



- ⊙ Conditional if: *if (expr) expr; else expr;*
- ⊙ For loop: *for (i=vector) expr;*
- ⊙ While loop: *while(expr) expr;*
- ⊙ Function:

```
function name(arg1,arg2,...)
! Optional function description (seen with help("name"))
import var1    #import global variable
export var2    #convert a local variable to global
expr;
_name = value  #Returned value
```

- ⊙ Operators: ', @, ~, ^, \*, #, /, +, -, ==, <, <=, >, >=, :, &, |, ?, %, =.
- ⊙ Functions: `funcdel(name)`, `funclist(name)`, `sprintf(fmt[,vec])`, `scanf(str,fmt)`, `matcvt(matrix, type)`, `cvtmat(special,type)`, `eval(str)`, **source(name)**, `help` or `help('symbol')`, `fread(fp,n)`, `fscanf(fp,fmt)`, `fgets(fp)`, `fwrite(fp,buf,n)`, `fprintf(fp,fmt[,vec])`, `fputs(fp,str)`, `fopen(name, mode)`, `freopen(fp,name,mode)`, `fclose(fp)`, `save(name,a[,ascii_flag])`, `load(name)`, `min(matrix)`, `max(matrix)`, `sum(matrix)`, `trace(matrix)`, `det(matrix)`, `inv(matrix)`, `tril(x)`, `triu(x)`, `eig(matrix)`, `jacob(a,b,eps)`, `lud(matrix)`, `hesse(matrix)`, `eye(n)`, `zeros(n,m)`, `ones(n, m)`, `rand(n,m)`, `diag(matrix)` or `diag(vector)`, **vector(start,end,inc)**, **size(matrix)**, `resize(matrix,n,m)`, where(a), `exists(name)`, **who**, `format(precision)`.
- ⊙ Mathematical functions: `abs(x)`, `acos(x)`, `asin(x)`, `atan(x)`, `ceil(x)`, `cos(x)`, `cosh(x)`, `exp(x)`, `floor(x)`, `ln(x)`, `log(x)`, `pow(x,y)`, `sin(x)`, `sinh(x)`, `sqrt(x)`, `tan(x)`, `tanh(x)`.



## VII. Types of simulations

### VII.1. Transient

In Elmer we can specify how many points in time we want to simulation what is the time increment between these points in time. Each point in time is a time step (*TimeStep* variable, taking values from 1 until the number of time steps) and the time increment is the time size. The number of time steps is defined with the '*Timestep Intervals*' option. For defining the time increment between timesteps we use the '*Timestep Sizes*' option. Both options are vectors, allowing us to define different "time sections" during the simulation. As an example, if we want to make a transient simulation that is lasting 10s in total but with two different time sections: the first one lasting 1s second (solved every 0.1s) and the second one arriving up to 10s (solved every 1s) we could set:

```
Timestep Sizes(2) = 0.1 1
Timestep Intervals(2) = 10 9
```

We can set a non-uniform time distribution by using the command '*Timestep Size*' with *TimeStep* as variable (this variable takes the values from 1 up to the number of intervals, as mentioned before). Example:

```
Timestep Size = Variable TimeStep
Real MATC "t0*1.05^(tx-1)"
```

*tx* in the MATC expressions refers to the variable *TimeStep*.

In the previous expression, the first time point simulated is at  $t_0$ . The second at  $t_0+t_0 \cdot 1.05$ . The third at  $t_0+t_0 \cdot 1.05+t_0 \cdot 1.05 \cdot 1.05$ . And so on. If we want to start the simulation at a given  $t_0$ , and starting from this time point a logarithmic scale, we could use a function in the SIF file in order to provide the time increments to '*Timestep Size*'. For example, we can include at the beginning of the SIF file (outside of any section) the following code:

```
$ti=1e-4
$tf=1e-1
$nps=60
$a=ti
$b=(tf/ti)^(1.0/(nps-1))

$ function calctsize(cts) import ti,tf,nps,a,b {\
  if (cts==1) {\
    _calctsize=ti;\
  }\
  else {\
    _calctsize=a*(b^(cts-1))*(1-1/b);\
  }\
}
```



```
}
```

And then we can specify in the Simulation section:

```
Timestep Intervals = $nps  
Timestep Size = Variable TimeStep  
Real MATC "calctsize(tx)"
```

For setting parameters depending on time, we have to create tables, with the variable *Time*.

```
Body Force 1  
Stress Bodyforce 2 = Variable Time  
Real  
0.0 -1.0  
4.0 -1.0  
4.01 0.0  
5.0 0.0  
End  
End
```

Initial conditions are generally needed for transient simulation. Nevertheless, usually these conditions are taken zero.

## VII.2. Coupled simulations

Coupled simulations refer to the problems where more than one field interacting each other has to be solved in a common domain or in the boundary between two domains. They are specified in solvers and boundary conditions.

We have to mention the difference in one-directional coupling and bi-directional coupling. In case of a directional coupling, we will have to set (in Setup) the "*Steady state max. iter*" to a sufficiently high number to be sure that the whole coupling problem converges.

## VII.3. Axisymmetric models-simulations

In Elmer, the axis of rotation is always the y axis. We have to take it into account for building the model. Also, we have to add the boundary conditions associated with the rotation axis.



## VIII. UDFs and Solver Code

Elmer allows to incorporate compiled functions as simple functions and as new solvers. This code have to be implemented in Fortran 90, compiled alone and incorporated to Elmer as an executable of a shared object. To incorporate these functions in Elmer, we will have to write the following line in the Sif file:

```
Procedure "filename" "procedure"
```

The shared object file (filename) can have different functions (procedures). With this instruction, we select the specific function of the file.

Most of the functions provided by Elmer are located in the *DefUtils* module. Moreover, the main data structure is an element, and not a node. The type *Element\_t* contains the element information. The data in each section of the Sif file is accessed through a pointer of the type *ValueList\_t*. The functions providing these pointers have a name starting with *Get*, followed by a name related to the section. For example, *GetSimulation()*. Similarly, we can also get the information for a specific element, by using, for example, *GetMaterial(Element, Found)*. *Found* (optional) will be set to TRUE if the requested information was obtained successfully. If *Element* is not provided, it is used the 'current' element. For getting constants, we use functions like *val=GetInteger(List, Name, Found)*.

If we want to obtain values at specific mesh locations, we have to use the function *val(:) = GetReal(List, Name, Found, Element)*. It returns a real array with the values for all nodes of the indicated element.

The real variable *Time* can be used in the Sif file.

The information from within Elmer solver can be obtained through the type *Solver\_t*. Different functions are: *GetElementNOFNodes(Element)*, *GetActiveElement(ElementNumber)*, *GetBoundaryElement(ElementNumber)*, *GetElementNodes(ElementNodes, Element, Solver)* (*ElementNodes* is a pointer of type *Nodes\_t*; *ElementNodes%x(i)* is the x coordinate of the indicated node by *i*), *U = Element % Type % NodeU(i)* (local coordinates) (similarly for V and W), *Normal = NormalVector(BoundaryElement, Nodes, U, V, CheckDirection)*.

For obtaining the nodal values of field variables we use the functions *GetScalarLocalSolution(vals, name, Element, Solver)*, *GetVectorLocalSolution(vals, name, Element, Solver)*.

For obtaining the values, as before, but for the complete mesh, we have to use the function *VariableGet(Solver % Mesh % Variables, 'Variable')*. It returns a pointer of the *Variable\_t* type.

In Elmer, the time is obtained from a pointer of type *Time\_t*, and considering it as a variable. Therefore, we can get it with the command *Tv => VariableGet(Solver % Mesh % Variables, 'Time')* and *Tmp = Tv % Values(1)*.

We can show messages by using three functions: *Info('FunctionName', 'The displayed message', level=levelnumber)*, *Warn('FunctionName', 'The displayed warning')*, *Fatal('FunctionName', 'The*



*displayed error message*'). For defining the message to be displayed, we can use the function *WRITE*.

We can write user defined functions for obtaining nodal values. The format is the following:

```
FUNCTION name_func ( model, n, var ) RESULT(result)
```

```
END FUNCTION name_func
```

*n* is the node, *var* are input variables and *result* is the returned value of the function.

## VIII.1. Defining a solver

First of all, we have to distinguish between the solver code itself and the solver interface. Evidently, the most important part for a solver is the code development. But you can also build an interface in Elmer in order to introduce the different parameters of the solver. This interface is not compulsory (all solver information can be given in the Sif file), but it is desirable.

For defining a solver, first of all we have to obtain the linearized partial differential equations (PDEs) of our problem. The general expression for these equations is given by:

$$M_{ij} \cdot \frac{\partial u_j}{\partial t} + K_{ij} \cdot u_j = F_i$$

where  $u_j$  is the degree of freedom (DOF) of our problem in the  $j$  direction (for example, fluid velocity along  $x$  direction),  $M_{ij}$  is the mass matrix,  $K_{ij}$  is the stiffness matrix and  $F_i$  is the force vector in the  $i$  direction. Basically, we have to provide the different matrices and vectors of this equation.

The solver subroutine has to include the following sections:

- ⊙ Initialization.
- ⊙ Start non-linear iterations.
- ⊙ Matrix assembly for domain (or bulk) elements (through element loop).
- ⊙ Matrix assembly for von Neumann and Newton conditions at boundary element (boundary elements loop).
- ⊙ Specify Dirichlet boundary conditions.
- ⊙ Solve the problem.
- ⊙ Check relative change of norms < Nonlinear Tolerance or nonlinear max. iterations within the nonlinear loop.

You can find many examples of the already implemented solvers in the Elmer source code, in the directory *elmerfem-devel\fem\src\modules*. Here, we are just going to provide a rough view of the code structure. It is important to divide the different tasks in subroutines.



We are just going to explain a simple solver, provided by Elmer, mostly for educational purposes. It is the advection-diffusion-reaction solver, with file name 'ModelPDE.F90'. First, we provide the code, and then we will explain the different sections. As it is stated in the file, it can also be used as a starting point to generate more complex solvers. The code is the following:

```
!-----
!> A prototype solver for advection-diffusion-reaction equation,
!> This equation is generic and intended for education purposes
!> but may also serve as a starting point for more complex solvers.
!-----
SUBROUTINE AdvDiffSolver( Model,Solver,dt,TransientSimulation )
```

Solver subroutine. dt: timestep size,  
TransientSimulation: logical value.

```
USE DefUtils
IMPLICIT NONE
!-----
TYPE(Solver_t) :: Solver
TYPE(Model_t) :: Model
REAL(KIND=dp) :: dt
LOGICAL :: TransientSimulation
```

The first step in the subroutine is to declare  
some needed constants and variables and  
their types, and also the libraries to be used.

```
! Local variables
!-----
TYPE(Element_t),POINTER :: Element
REAL(KIND=dp) :: Norm
INTEGER :: n, nb, nd, t, active
INTEGER :: iter, maxiter
LOGICAL :: Found
!-----
```

```
maxiter = ListGetInteger( GetSolverParams(),&
  'Nonlinear System Max Iterations',Found,minv=1)
IF(.NOT. Found ) maxiter = 1
```

Start nonlinear iterations.

```
! Nonlinear iteration loop:
!-----
DO iter=1,maxiter
```

```
! System assembly:
!-----
```

```
!Initialize the system and do the assembly:
CALL DefaultInitialize()
Active = GetNOFActive()
DO t=1,Active
  Element => GetActiveElement(t)
  n = GetElementNOFNodes()
  nd = GetElementNOFDOfs()
  nb = GetElementNOFBDOFs()
  CALL LocalMatrix(Element, n, nd+nb)
END DO
```

Matrix assembly for domain elements.

```
CALL DefaultFinishBulkAssembly()
```

```
Active = GetNOFBoundaryElements()
DO t=1,Active
  Element => GetBoundaryElement(t)
  IF(ActiveBoundaryElement()) THEN
    n = GetElementNOFNodes()
    nd = GetElementNOFDOfs()
    nb = GetElementNOFBDOFs()
    CALL LocalMatrixBC(Element, n, nd+nb)
  END IF
END DO
```

Matrix assembly for von Neumann and  
Newton conditions at boundary element.

```
CALL DefaultFinishBoundaryAssembly()
CALL DefaultFinishAssembly()
```



```
CALL DefaultDirichletBCs()
```

Specify Dirichlet boundary conditions.

```
! And finally, solve:
```

```
Norm = DefaultSolve()
```

Solve the problem.

```
IF( Solver % Variable % NonlinConverged == 1 ) EXIT
```

Check relative change of norms < Nonlinear Tolerance or nonlinear max. iterations within the nonlinear loop.

```
END DO
```

```
CONTAINS
```

```
! Assembly of the matrix entries arising from the bulk elements
```

```
!-----  
SUBROUTINE LocalMatrix( Element, n, nd )
```

```
!-----  
INTEGER :: n, nd  
TYPE(Element_t), POINTER :: Element  
!-----  
REAL(KIND=dp) :: diff_coeff(n), conv_coeff(n), react_coeff(n), &  
time_coeff(n), D,C,R, rho, Velo(3,n), a(3), Weight  
REAL(KIND=dp) :: Basis(nd), dBasisdx(nd,3), DetJ, LoadAtIP  
REAL(KIND=dp) :: MASS(nd,nd), STIFF(nd,nd), FORCE(nd), LOAD(n)  
LOGICAL :: Stat, Found  
INTEGER :: i, t, p, q, dim  
TYPE(GaussIntegrationPoints_t) :: IP  
TYPE(ValueList_t), POINTER :: BodyForce, Material  
TYPE(Nodes_t) :: Nodes  
SAVE Nodes  
!-----
```

```
dim = CoordinateSystemDimension()
```

```
CALL GetElementNodes( Nodes )
```

```
MASS = 0._dp
```

```
STIFF = 0._dp
```

```
FORCE = 0._dp
```

```
LOAD = 0._dp
```

```
BodyForce => GetBodyForce()
```

```
IF ( ASSOCIATED(BodyForce) ) &  
Load(1:n) = GetReal( BodyForce, 'field source', Found )
```

```
Material => GetMaterial()
```

```
diff_coeff(1:n) = GetReal(Material, 'diffusion coefficient', Found)
```

```
react_coeff(1:n) = GetReal(Material, 'reaction coefficient', Found)
```

```
conv_coeff(1:n) = GetReal(Material, 'convection coefficient', Found)
```

```
time_coeff(1:n) = GetReal(Material, 'time derivative coefficient', Found)
```

```
Velo = 0._dp
```

```
DO i=1,dim
```

```
Velo(i,1:n) = GetReal(Material, &  
'convection velocity' // TRIM(I2S(i)), Found)
```

```
END DO
```

```
! Numerical integration:
```

```
!-----
```

```
IP = GaussPoints( Element )
```

```
DO t=1, IP % n
```

```
! Basis function values & derivatives at the integration point:
```

```
!-----
```

```
stat = ElementInfo( Element, Nodes, IP % U(t), IP % V(t), &  
IP % W(t), detJ, Basis, dBasisdx )
```

```
! The source term at the integration point:
```

```
!-----
```

```
LoadAtIP = SUM( Basis(1:n) * LOAD(1:n) )
```



```

rho = SUM(Basis(1:n)*time_coeff(1:n))
a = MATMUL(Velo(:,1:n),Basis(1:n))
D = SUM(Basis(1:n)*diff_coeff(1:n))
C = SUM(Basis(1:n)*conv_coeff(1:n))
R = SUM(Basis(1:n)*react_coeff(1:n))

Weight = IP % s(t) * DetJ

! diffusion term (D*grad(u),grad(v)):
! -----
STIFF(1:nd,1:nd) = STIFF(1:nd,1:nd) + Weight * &
  D * MATMUL( dBasisdx, TRANSPPOSE( dBasisdx ) )

DO p=1,nd
DO q=1,nd
! advection term (C*grad(u),v)
! -----
STIFF(p,q) = STIFF(p,q) + Weight * &
  C * SUM(a(1:dim)*dBasisdx(q,1:dim)) * Basis(p)

! reaction term (R*u,v)
! -----
STIFF(p,q) = STIFF(p,q) + Weight * R*Basis(q) * Basis(p)

! time derivative (rho*du/dt,v):
! -----
MASS(p,q) = MASS(p,q) + Weight * rho * Basis(q) * Basis(p)
END DO
END DO

FORCE(1:nd) = FORCE(1:nd) + Weight * LoadAtIP * Basis(1:nd)
END DO

IF(TransientSimulation) CALL Default1stOrderTime(MASS,STIFF,FORCE)
CALL LCondensate( nd-nb, nb, STIFF, FORCE )
CALL DefaultUpdateEquations(STIFF,FORCE)
! -----
END SUBROUTINE LocalMatrix
! -----

! Assembly of the matrix entries arising from the Neumann and Robin conditions
! -----
SUBROUTINE LocalMatrixBC( Element, n, nd )
! -----
INTEGER :: n, nd
TYPE(Element_t), POINTER :: Element
! -----
REAL(KIND=dp) :: Flux(n), Coeff(n), Ext_t(n), F,C,Ext, Weight
REAL(KIND=dp) :: Basis(nd),dBasisdx(nd,3),DetJ,LoadAtIP
REAL(KIND=dp) :: STIFF(nd,nd), FORCE(nd), LOAD(n)
LOGICAL :: Stat,Found
INTEGER :: i,t,p,q,dim
TYPE(GaussIntegrationPoints_t) :: IP

TYPE(ValueList_t), POINTER :: BC

TYPE(Nodes_t) :: Nodes
SAVE Nodes
! -----
BC => GetBC()
IF (.NOT.ASSOCIATED(BC) ) RETURN

dim = CoordinateSystemDimension()

CALL GetElementNodes( Nodes )

```



```

STIFF = 0._dp
FORCE = 0._dp
LOAD = 0._dp

Flux(1:n) = GetReal( BC,'field flux', Found )
Coeff(1:n) = GetReal( BC,'robin coefficient', Found )
Ext_t(1:n) = GetReal( BC,'external field', Found )

! Numerical integration:
!-----
IP = GaussPoints( Element )
DO t=1,IP % n
! Basis function values & derivatives at the integration point:
!-----
stat = ElementInfo( Element, Nodes, IP % U(t), IP % V(t), &
  IP % W(t), detJ, Basis, dBasisdx )

Weight = IP % s(t) * DetJ

! Evaluate terms at the integration point:
!-----

! Given flux:
!-----
F = SUM(Basis(1:n)*flux(1:n))

! Robin condition (C*(u-u_0)):
!-----
C = SUM(Basis(1:n)*coeff(1:n))
Ext = SUM(Basis(1:n)*ext_t(1:n))

DO p=1,nd
DO q=1,nd
  STIFF(p,q) = STIFF(p,q) + Weight * C * Basis(q) * Basis(p)
END DO
END DO

FORCE(1:nd) = FORCE(1:nd) + Weight * (F + C*Ext) * Basis(1:nd)
END DO
CALL DefaultUpdateEquations(STIFF,FORCE)
!-----
END SUBROUTINE LocalMatrixBC
!-----

! Perform static condensation in case bubble dofs are present
!-----
SUBROUTINE LCondensate( N, Nb, K, F )
!-----
USE LinearAlgebra
INTEGER :: N, Nb
REAL(KIND=dp) :: K(:,,:),F(:,),Kbb(Nb,Nb), &
  Kbl(Nb,N), Klb(N,Nb), Fb(Nb)

INTEGER :: m, i, j, l, p, Ldofs(N), Bdofs(Nb)

IF ( Nb <= 0 ) RETURN

Ldofs = (/ (i, i=1,n) /)
Bdofs = (/ (i, i=n+1,n+nb) /)

Kbb = K(Bdofs,Bdofs)
Kbl = K(Bdofs,Ldofs)
Klb = K(Ldofs,Bdofs)
Fb = F(Bdofs)

CALL InvertMatrix( Kbb,nb )

```



```

F(1:n) = F(1:n) - MATMUL( Klb, MATMUL( Kbb, Fb ) )
K(1:n,1:n) = K(1:n,1:n) - MATMUL( Klb, MATMUL( Kbb, Kbl ) )
!-----
END SUBROUTINE LCondensate
!-----

!-----
END SUBROUTINE AdvDiffSolver
!-----

```

The format of the solver interface xml file is:

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE edf>
<edf version="1.0" >
  <PDE Name=" AdvDiff" >
    <Name>AdvDiff</Name>

    <BodyForce>
      <Parameter Widget="Label" > <Name> Properties </Name> </Parameter>
      <Parameter Widget="Edit" >
        <Name> Source </Name>
        <Type> String </Type>
        <Whatis> Give the source term. </Whatis>
      </Parameter>
    </BodyForce>

    <Solver>
      <Parameter Widget="Edit" >
        <Name> Procedure </Name>
        <DefaultValue> "AdvDiff" "AdvDiffSolver" </DefaultValue>
      </Parameter>
      <Parameter Widget="Edit">
        <Name> Variable </Name>
        <DefaultValue> Potential</DefaultValue>
      </Parameter>
    </Solver>

    <BoundaryCondition>
      <Parameter Widget="Label" > <Name> Dirichlet conditions </Name> </Parameter>
      <Parameter Widget="Edit">
        <Name> Potential </Name>
        <Whatis> Give potential value for this boundary. </Whatis>
      </Parameter>
    </BoundaryCondition>
  </PDE>
</edf>

```



## IX. ParaView post-processor

ParaView [3] is a good alternative as post-processor for results obtained from Elmer. This post-processor was originally developed for visualization of big amounts of data and this is the case for many simulation results obtained in FEM, and especially those related to the biomedical field.

This chapter is just intended to provide some basic concepts and procedures for obtaining common visualization results from simulation result files obtained from Elmer. If more detailed information is needed, you can access to the ParaView documentation or tutorials [4]. Although we are just going to use the GUI, it allows to use Python scripting for almost any functionality.

### IX.1. Elmer-ParaView connection

ParaView accepts some file formats containing the data to be visualized. One of them is *vtu* (VTK unstructured data) format. Elmer allows the generation of this format for the results file just stating as the output file (*Post File*) a name with an extension *.vtu*. VTK refers to *Visualization Toolkit*. Basically, it makes use of the data-flow model. You can think of it like a block diagram, where you start from data (vectors from simulation results) and each block are functions (called filters) that modify this data for our purposes (generally calculation and visualization).

### IX.2. General ParaView concept


With ParaView we will have to manage two aspects: data operations (*filter* concept) and data visualization. Both aspects are related because usually we perform some data operations with the goal of representing graphically the results of these calculations.

Regarding data management, basically ParaView reads data from a file (in our case, simulation results) and it generates a vector for each data (temperature, velocity in x direction, etc.). Each vector is one of the results in the result file for all nodes. It can be considered a result variable. For example, a vector of temperature is a vector with all the temperatures at all nodes. We can make calculations from these vectors. Each calculation is like a formula (or procedure) applied to the data. In ParaView this formula or procedure is called a filter. Each filter has inputs (where we provide the vectors to be used in the calculations) and outputs were the result is obtained. Again, we can apply another filter to this previous output, and so on. Of course, we can generate different *parallel* calculations starting from the same vector. Each filter will have different parameters to be fixed at our specific calculation, although they have default values.

Regarding data visualization, ParaView provides different types of views:

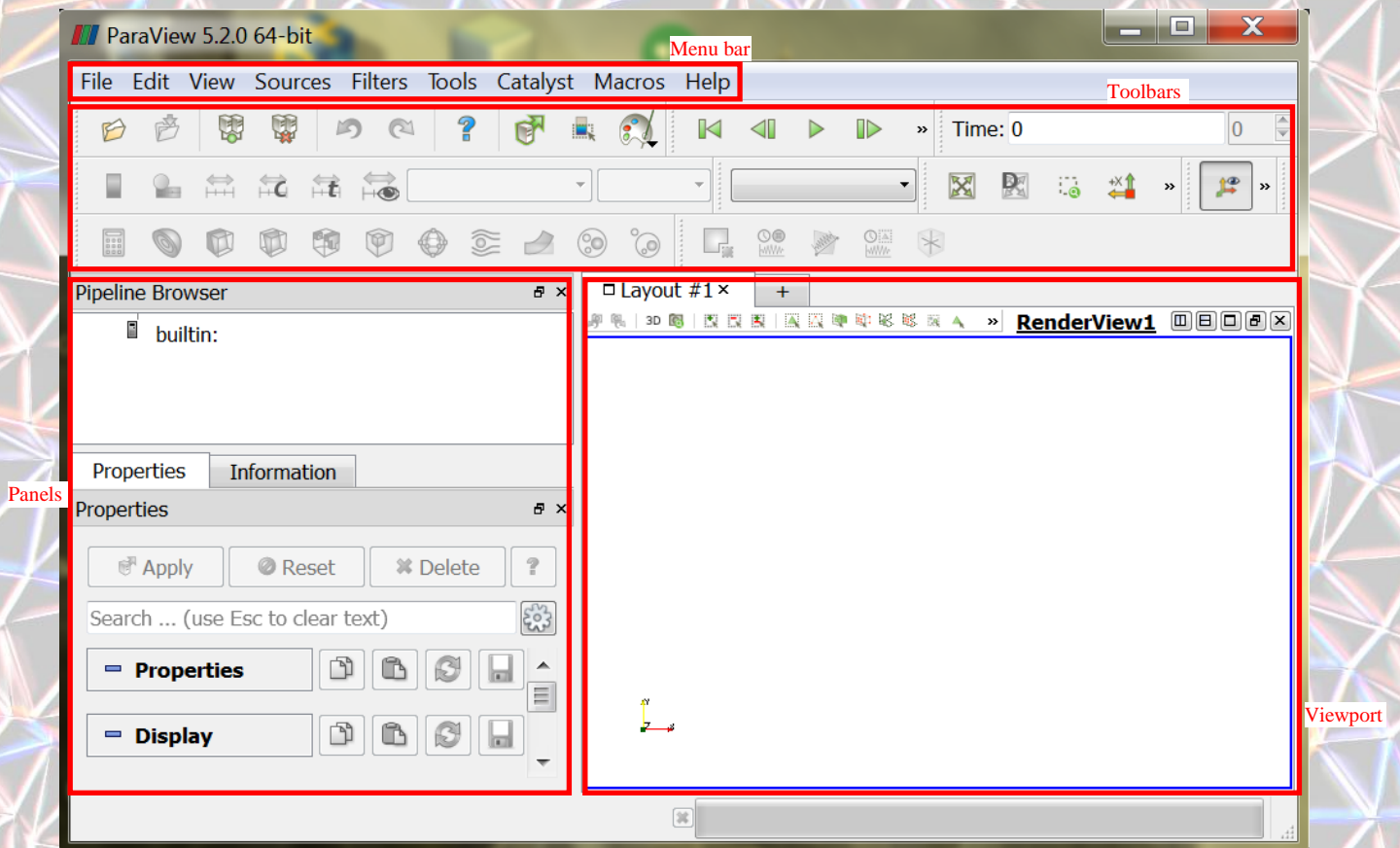
- ⦿ Rendering views: For visualizing geometries graphically.
- ⦿ Chart views: Graphs and plots for plotting non-geometric results.



-  Comparative views: It is the same than the both previous views but specifically for visualizing the effects of parameter changes.

### IX.3. ParaView GUI

The GUI of ParaView is the following one:



The GUI has four basic parts: menu, toolbars, panels and the Viewport.

The Viewport is the window where we will observe the geometry, results and different types of graphs. There are different types of ways to show this data. They are called *views*.

The panels are where we will access the data tree (*'pipeline browser'*) and where we will set parameters.

Regarding the menu, we will just go through some of the options that we will use during this course. With *File* → *Open* we can read the results file. With *Load State* and *Save State* we can save and read a session of ParaView. And the *Filters* option contains all the filters that we can apply to the dataset. We will learn to use only some of them.



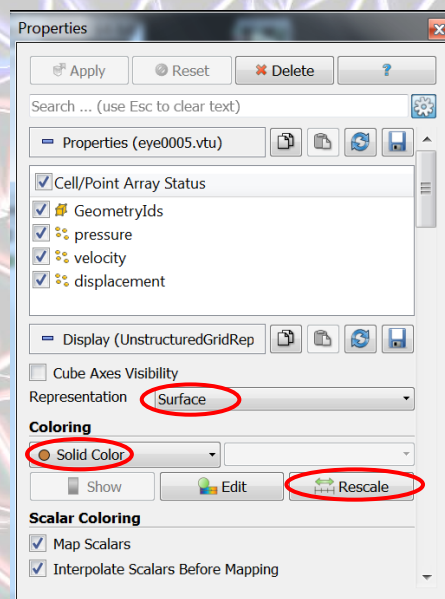
## IX.4. Filters

We are going to apply filters to a specific simulation result obtained from Elmer. It was intended for obtaining the results of the fluid flow of the aqueous humor and the mechanical effect on the eye. Some of the values used in this simulation may not be realistic.

In order to read the results into ParaView, we just have to do *File* → *Open* and select the corresponding vtu file.

Apart from the usual menus and toolbars, after reading the results we will see all the bodies in the *RenderView* window, but without representing any result. On the left there are two important windows:

- ② The 'Pipeline Browser' is where we will have all the data and filters applied, in a tree arrangement. In our case, we have only the imported data *eye0005.vtu* (we can change the name easily just double-clicking on it).
- ② The Properties window (Figure 24) is where we can set the different options related to this data or filter, including visualization options. We can also select the data that we want to visualize in the *Coloring* section.




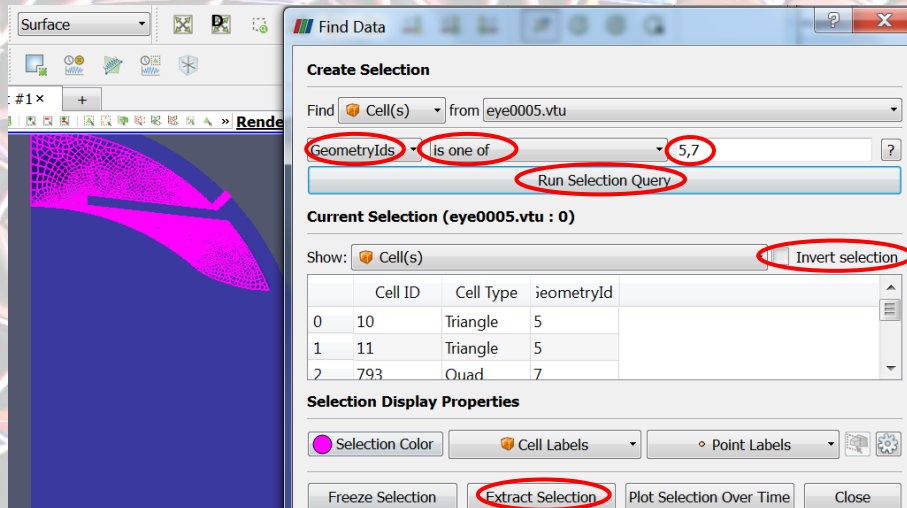
**Figure 24. Properties window.**

At the top of Figure 24, we can see the different data available (*GeometryIds* (body numbers), *pressure*, *velocity* and *displacement*) of our data. In the *Coloring* section we can specify which variable we want to represent at the graphics window (*RenderView* by default). We have also the *Rescale* button for adjusting colors (of the color map) to the selected data. In *Representation*, we can show the geometry in some different forms (if you want to see also the mesh, just select '*Surface With Edges*').



You can test by yourself other options in the *Properties* windows to see the effect. For example, the *Opacity* parameter can be sometimes useful.

The first thing that we can do is to obtain data just for some parts of our model. For example, we can separate the fluid from the mechanical part. Or we could just separate the different bodies (cornea, iris, etc.). For this purpose we can use the *Edit* → *Find Data* (  ) option. *Figure 25* shows the selection of the fluid bodies (*Geometry Ids* 5 and 7). We must click on '*Run Selection Query*' and then on '*Extract Selection*'. You can see the selection on the graphics window, and then you will have another element in the '*Pipeline Brower*' (I have called it '*fluid*').



*Figure 25. Find Data window.*


We could do the same for the rest of bodies, but it is easier to check the option '*Invert selection*', and click again '*Extract Selection*'. Don't forget to click on '*Apply*' (this is usually required for many changes to take effect) in the *Properties* window. In order to make the selection disappear, just set a value of 0 and press '*Run Selection Query*'.

At the left of every data in the *Pipeline Brower* it appears an eye, which can be open or closed. This is for allowing the visualization (or not) of this data in the graphics window. You can have several data open for viewing at the same time.

#### IX.4.1. Calculator filter

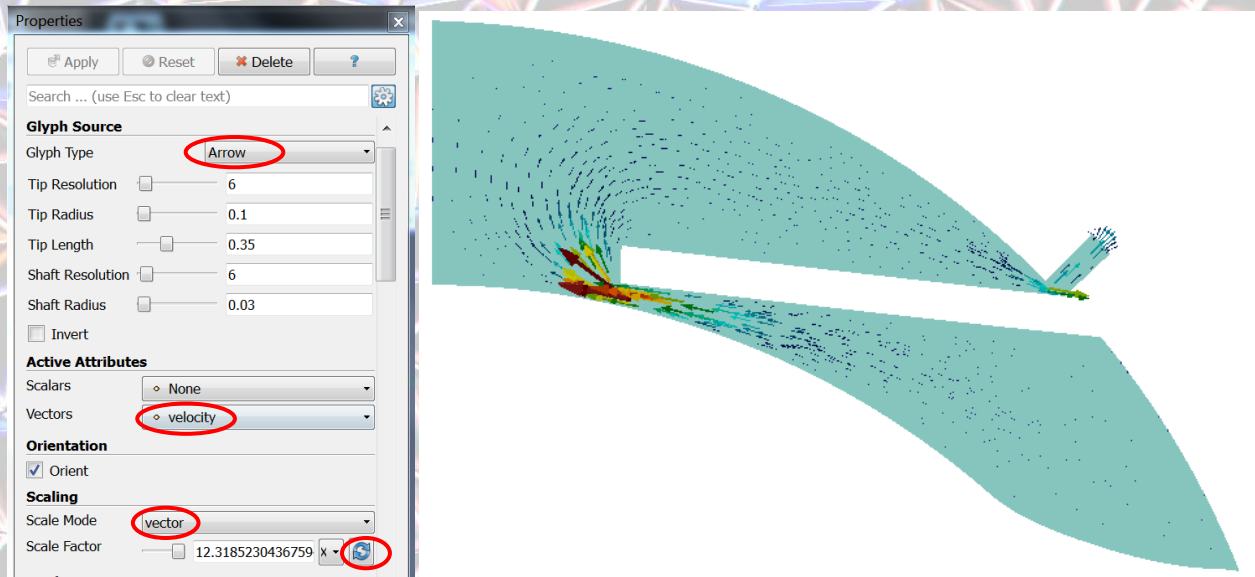
The *Calculator* filter allows us to generate new variables (derived variables) from the originally imported variables or those obtained from other filters.

#### IX.4.2. Glyph filter

Plotting vectors is a quite usual operation when reviewing results, like for velocity, displacements, electric fields, etc. This can be achieved with ParaView applying a *Glyph* filter (  ) (*Filters* → *Common* → *Glyph*). In our example, we can select first the '*fluid*' data and then we click on the



*Glyph* filter function (the filter is always applied to the selected data in the *Pipeline Browser*). The typical options in the *Properties* windows are shown in *Figure 26*, as well as the result.



**Figure 26.** *Properties* window for the *Glyph* filter for obtaining vectors.

#### IX.4.3. Stream Tracer filter

It is also typical to show the streamlines. Streamlines represent the trajectory that particles without mass would follow in the fluid flow. For obtaining streamlines, ParaView have the *Stream Tracer* filter (👁️). First select the 'fluid' data and then click on this filter. In '*Vectors*' select *velocity*. In '*Maximum Streamline Length*' it is better to choose a big enough value in order to avoid cutted lines. In '*Seed Type*' choose '*High Resolution Line*'. It will appear a line between two points. We can place these two points using the mouse (first select the appropriate view direction in the graphics windows in order to move these points just on this plane). I have placed the line near the entrance of the fluid. In '*Resolution*' we can choose the number of lines to be plotted (I have chosen 20). Finally, select also which color variable to use for plotting the lines.



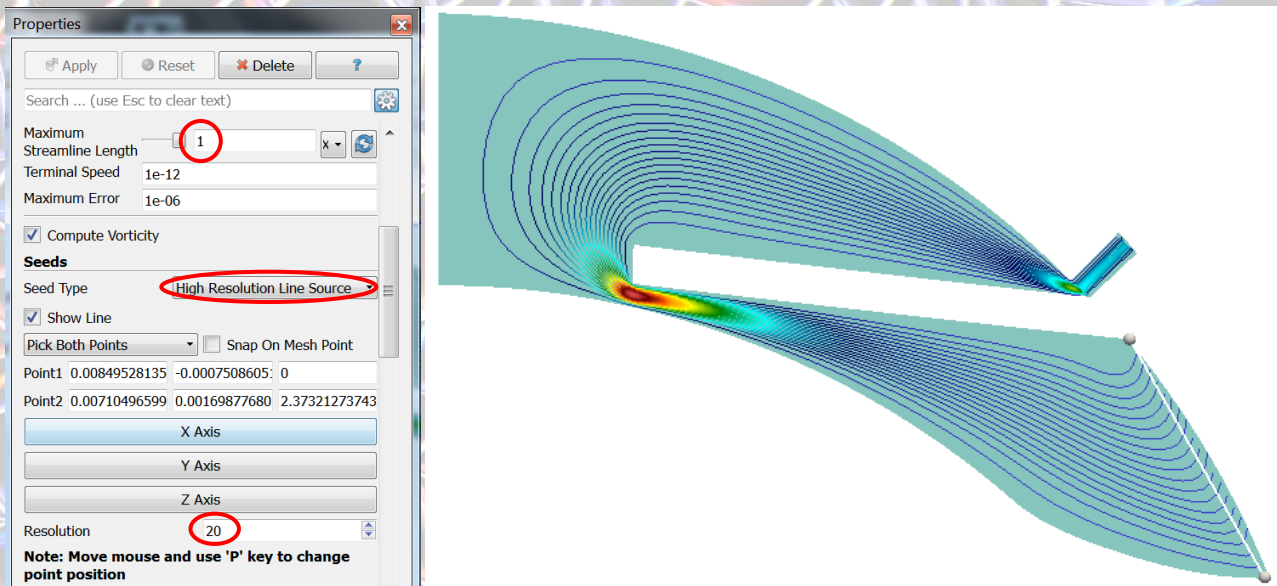


Figure 27. Stream Tracer filter options and result on the fluid data.

#### IX.4.4. Contour filter

Manytimes it is useful to observe surfaces where the value of a variable remains constant (isosurfaces). The *Contour* filter allows to generate these isosurfaces (in case of a 3D problem; isolines in case of a 2D problem). We have to select the variable and the value (or values) of isosurfaces that we want to generate.

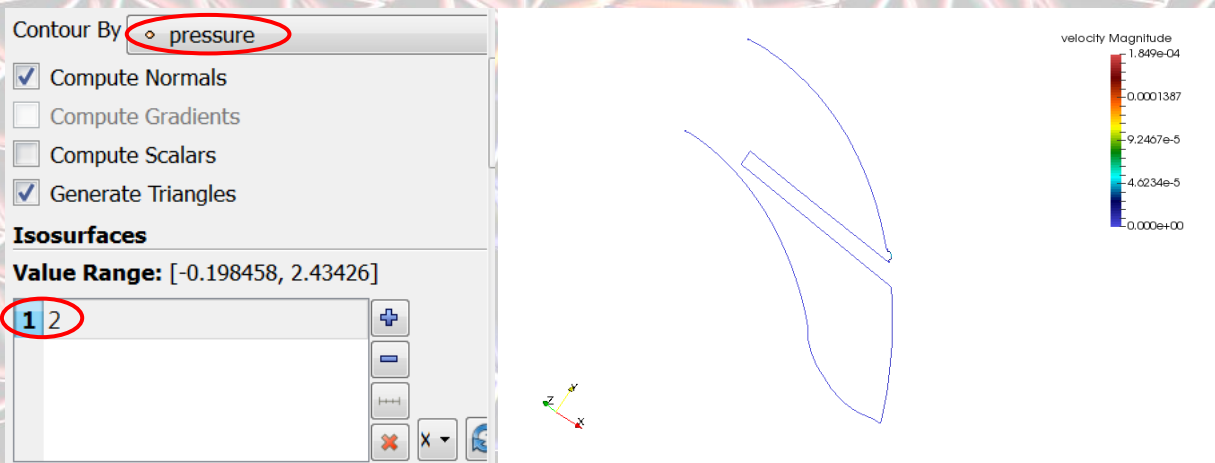
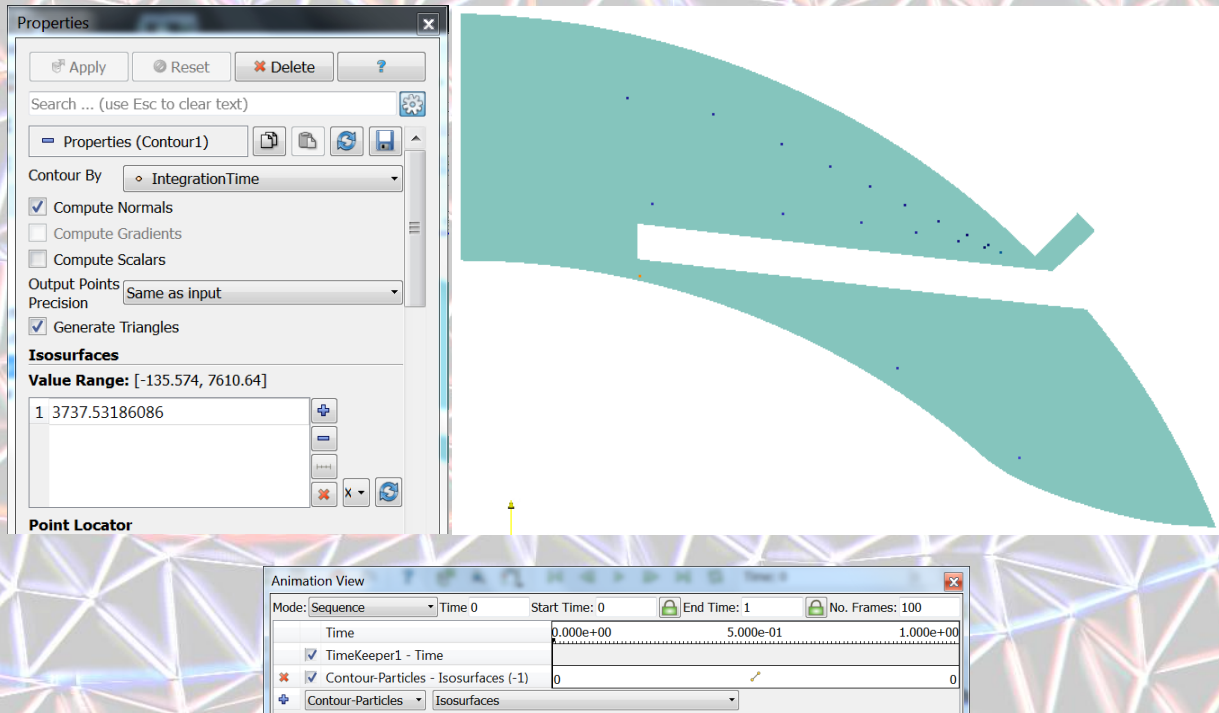


Figure 28. Contour filter options and result on the fluid data (pressure line in this case).

Starting from the streamlines of the previous filter, we can also animate particles to see how they would flow along this channel. For this purpose, we can use the *Contour* filter ( ) applied to the streamlines. This filter provides isolines or isocontour; these are lines or surfaces where the chosen variable is constant. In our case we will choose as variable 'IntegrationTime' in 'Contour By'.




Therefore, the lines that we will obtain are lines that have the same time. Applied to the streamlines, we will obtain "pieces" of streamlines. ParaView will calculate these lines for different time values. For obtaining this animation, let's view the Animation window (in *View* → *Animation View*). In this window we have to include the animation of this filter by clicking on the + sign in the Contour filter (with 'Isosurfaces' option) in this window. Increase also the number of frames. Just click on the play button to reproduce the animation.



**Figure 29.** Contour filter options, Animation view and result of animation at a specific time.

We can also create a video file by selecting *File* → *Save Animation*. You can select the range of frames to use in the animation.

#### IX.4.5. Warp by Vector filter

For mechanical parts, for example, a common operation is to scale the deformation (as usually they are too small to be clearly seen with naked eye). This operation can be done with the 'Warp by Vector' filter (). It simply applies a factor to the deformations. We just have to select which vector to use to calculate the scaled deformations (generally '*displacement*' for a mechanical problem), set a *Scale Factor* and choose a variable to be plotted.



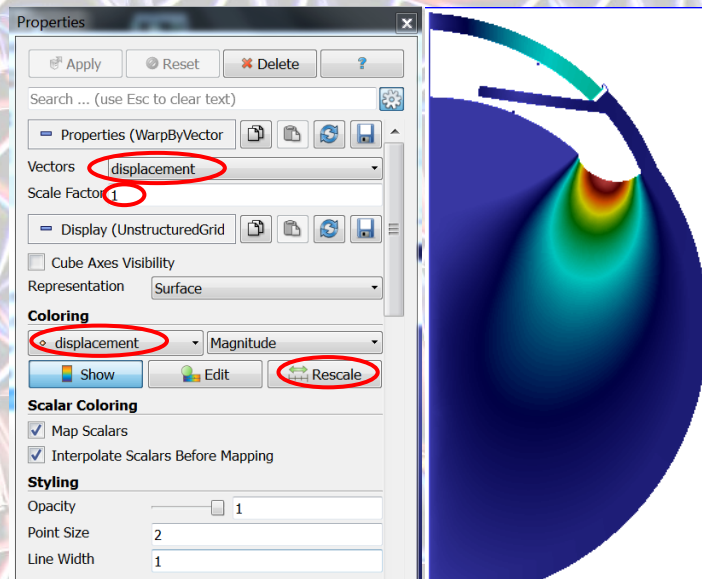


Figure 30. Warp by Vector filter options and result.

#### IX.4.6. Warp by Scalar filter

Some variables are not vectors and cannot be represented by the previous filter (*Warp by Vector*). But in ParaView we have available the *Warp by Scalar* filter (it can be found in *Filters* → *Alphabetical* → *Warp by Scalar*). In this case, as the variable has no direction (because it is not a vector), you have to specify in which direction you have to ‘deform’ the entities to show the variations of the variable. As an example, the pressure in a fluid has no direction (it is a scalar), but we can tell this filter to show the pressure variations by deforming the surfaces in the out-of-plane direction (z-direction in our case). At the same time, you can also select which variable to plot on this deformed surface (it has been selected also *Pressure*).

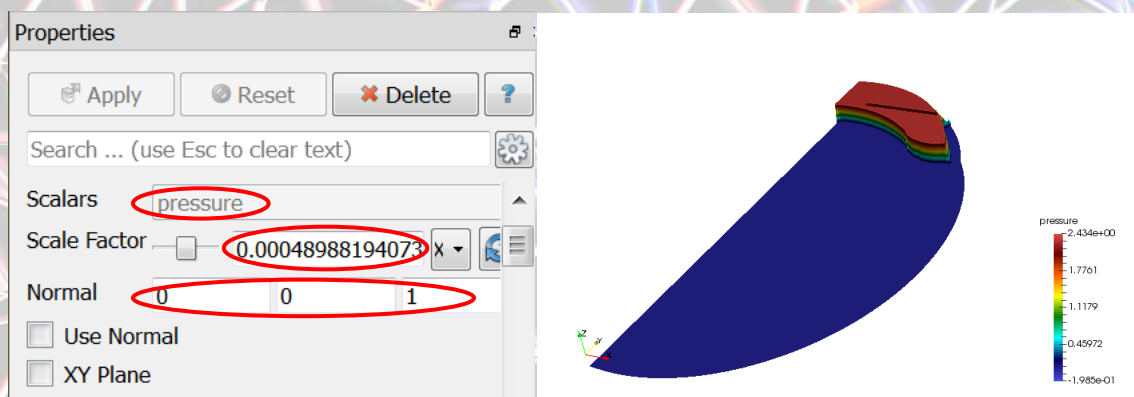


Figure 31. Warp by Scalar filter options and result.

#### IX.4.7. Clip filter

The *Clip* filter allows to observe a section of our model. It is especially useful in 3D problems.



#### IX.4.8. Reflect filter

In case where we have used reflect symmetries, we can generate the complete model by reflecting our reduced model with the *Reflect* filter.

#### IX.4.9. Rotational Extrusion filter

With the *Rotational Extrusion* filter, we can rotate our model (generally surfaces) around the z-axis. It is especially useful in axi-symmetric models, for obtaining the full model. Nevertheless, this filter is somehow tricky and it does not work for any surface without previous arrangements.



Acronyms:

2D: Two-dimensional

3D: Three-dimensional

CAD: Computer Aided Design

DOF: Degree of Freedom

FEM: Finite Element Method

GPL: General Public License

PDE: Partial Differential Equation

---

[1] <http://www.csc.fi/english/pages/elmer>.

[2] M. Carmona, J.M. Gómez, J. Bosch, M. López, "GMSH - Guide for mesh generation v02",  
Universitat de Barcelona, 2014.  
<http://hdl.handle.net/2445/63888>.

[3] <http://www.paraview.org>.

[4] <http://www.paraview.org/paraview-guide>.