Ostero: a didactic finite element code for solid mechanics

Date		Author	
November 30, 2016		Matias Rivero	
Document type	Pages	Email	
Techical Report	33	matias.rivero@bsc.es	

Abstract

Ostero intends to be a didactic finite element code for the numerical simulation of solid deformable bodies. It was developed as part of a PhD thesis currently done at Barcelona Supercomputing Center (BSC), to use it as a test framework for several mechanical models and problems. It was a very useful tool to perform proof-of-concept evaluations of contact mechanics algorithms before implementing them in the BSC code called Alya, which is a bigger, powerful but more complex code. As a general fact, Ostero was designed in order to first: provide the user a clear idea of the structure that usually a finite element code for linear and non-linear mechanics has and second: to allow the user to use Ostero as a workbench for personal tests. So far Ostero has only support for 2D problems, using triangles or quadrilateral elements and is limited to quasi-static analysis. Ostero allows to solve geometrically linear and non-linear problems, using elastic material models. For the non-linear case, Kirchhoff and hyperelastic (neo-Hookean) material models are available. Ostero was also designed to interact seamlessly with open source meshing and post-processing tool. To this aim, Ostero reads Gmsh mesh files and writes Vtk files, which can be easily visualized using the ParaView post-processing tool.

Revision history

Rev.	Date	Author	
v0.1	July 18, 2016	Matias Rivero	First issue - Some examples still missing
v0.2	July 18, 2016	Matias Rivero	Added *complete* example and *two materials* example
v0.3	July 19, 2016	Matias Rivero	Added *point_bcs* example and *square_quads* example
v0.4	July 20, 2016	Matias Rivero	Added *DCB* example
v0.5	Nov. 30, 2016	Matias Rivero	Corrected some typos in *square* example input file

Contents

1 Description

Ostero is a finite element code that solves the equilibrium equation which governs the mechanics of a deformable body subjected to external forces, Dirichlet and/or Neumann boundary conditions. In other words, Ostero allows to determine the response of a deformable solid body to an applied external load or displacement. Ostero allows to solve a mechanical problem considering geometric linearity or non-linearity. For the specific case of the geometrical non-linear model, Ostero uses an implicit scheme and the update of tangent matrix is performed at each time step. When solving using a geometrically linear setting, the equations of equilibrium are formulated in the undeformed state, and are not updated with the deformation. In some engineering problems, as the deformations are considered small and the deviation from the original geometry is not perceptible, the use of a geometrically linear setting is a very good approximation to the non-linear model. The mathematical complexity generated by a more realistic theory and the associated increment of the computation time doesn't compensate the small error introduce by ignoring the update of deformations in the equilibrium equations. But, in engineering there are also a number of problems where the deformation (large strains and/or large rotations) can not be ignored. In those cases a geometrically non-linear model should be used in Ostero to account for the large deformations. It's part of the engineering criteria to choose which model, linear or non-linear, could be use.

Ostero intends to be a didactic code, and its main objective is to allow the user to understand the very basic structure of a linear and non-linear mechanics finite element code. Also intends to provide a framework for beta testing of different models such as elastic material models, contact, plasticity, fracture, etc.

Ostero is based on the solidz module of the Alya code, which is a multi–physics parallel simulation code developed at Barcelona Supercomputing Center (BSC). So far it can solve 2D quasi-static problems using triangles or quadrilateral elements. The non-linear module include the isolinear (or Kirchhoff) material model and several formulations of the hyperelastic neo-Hookean material model. As many engineering applications involve small strains but large rotations, the effect of large deformation are primarily due to rotations. In these cases the response of the material may then be modeled by a simple extension of the linear elastic laws with small modifications (see [1]), such as the isolinear material model. For those elastic materials subjected to large strains for which the work is independent of the load path (no dissipation of energy), the hyperelastic material model is the most suitable choice.

Ostero was designed to read mesh files generated using the open source meshing tool Gmsh, with no need of aditional conversions to adapt the mesh file format to the input requested by Ostero. Ostero writes outputs in Vtk ASCII format, which can be easily postprocessed using the open source post-processing tool ParaView.

Ostero was written in Python and Fortran to exploit the main advantages and properties of each programming language. The parsing of the input parameters, boundary conditions and other options is done in the main program, coded in Python. The main program also includes the main execution loop, which call the subroutines that performs the elementary matrix calculations and the assembly operations from the Fortran module. This external Fortran module is imported in the main program as an external library.

2 How to get Ostero

Ostero is hosted in Bitbucket. Bitbucket is a web-based hosting service for projects that use Mercurial or Git revision control systems. Bitbucket is similar to GitHub (which primarily uses Git), but the main difference is that Bitbucket allows free private repositories, while in GitHub only

public repositories are available for free users. Besides being hosted in Bitbucket server, Ostero is under Git revision control, which provides a perfect framework for collaborative development. Git is a free and open source distributed version control system, which allows to manage changes in the code in a very efficient way. Is an essential tool for collaborative projects, but also very useful for individual programmers. Git takes a peer-to-peer approach to version control, as opposed to the client-server approach of centralized systems, as SVN. Rather than a single, central repository on which clients synchronize, in Git each peer's working copy is the complete repository which includes the complete history information of the codebase.

There are two main ways to get ostero:

1. The easiest alternative is to download the source code from the main page of Ostero https://bitbucket.org/matrivero/ostero, in *Downloads* section. The source code is available to download in compressed zip form, so just uncompress the downloaded file using the unzip tool or any other zip decompressor tool.

2. If you want to get a copy of Ostero with Git support (i.e. to keep track of your own changes or to be able to update Ostero without losing your modifications) you have to clone the repository to your home system. To do this, first you need to check if Git is installed on your system by executing the following command in a bash terminal:

```
1  :~$ git --version
2  git version 1.7.10.4
```

If Git is not installed on your system and you are using a Debian or Ubuntu Linux distribution (or any Linux distribution that has support for the aptitude package manager), you can install Git by doing:

```
1  :~$ sudo apt-get install git-all
```

If on the contrary, Git is installed on your system, run the following command to clone the repository in your computer:

```
1  :~$ git clone https://bitbucket.org/matrivero/ostero.git
2  Cloning into 'ostero'...
3  remote: Counting objects: 852, done.
4  remote: Compressing objects: 100% (790/790), done.
5  remote: Total 852 (delta 271), reused 231 (delta 61)
6  Receiving objects: 100% (852/852), 35.84 MiB | 5.80 MiB/s, done.
7  Resolving deltas: 100% (271/271), done.
```

You can check that Ostero was correctly cloned by doing:

```
1   :~$ cd ostero
2   :~/ostero$ ls
3   drwxr-xr-x  7 user user   4096 Jun 18 18:38 alya_benchmark
4   drwxr-xr-x 13 user user   4096 Jun 22 10:54 examples
5   -rw-r--r--  1 user user  29073 Jun 20 14:53 external.f90
6   -rwxr-xr-x  1 user user  22932 Jul 12 15:43 finite_strain.py
7   -rw-r--r--  1 user user  34602 Jun 18 18:38 LICENCE.txt
8   -rw-r--r--  1 user user   3812 Jun 18 18:38 README.md
9   -rw-r--r--  1 user user    297 Jun 20 20:28 TODO
10  -rw-r--r--  1 user user   5487 Jun 18 18:38 writeout.py
```

3 First steps

In order to execute Ostero you need a Python interpreter for 2.X version, Numpy and the f2py package, which is a Fortran to Python interface generaton. Since 2007, f2py is part of Numpy.

First, if you are not sure if your system fulfill the requirements to run Ostero, you can check if Python, Numpy and f2py are installed in your system (and which versions you are using).

The latest version of most Linux distribution come with Python 2.7 out of the box. To see which version of Python you have installed, simply execute the following command in a bash terminal:

```
:~$ python --version
Python 2.7.3
```

To check if Numpy and f2py is installed in your system, simply execute:

```
:~$ python -c "import numpy; print(numpy.__version__)"
1.6.2
:~$ f2py -v
2
```

To install Numpy, in case is not installed in your system, you can do it using the aptitude packet manager, getting Numpy from repositories:

```
:~$ sudo apt-get install python-numpy
```

Once you have checked the system requirements, to start using Ostero the first step is to compile the external Fortran library by doing:

```
:~/ostero$ f2py -c -m external external.f90
```

If the external library was generated ok (check for external.so in the installation root directory), you are ready to execute Ostero. Let's start with an example. First, change the directory to the example folder i.e.:

```
:~/ostero$ cd examples/square
```

Then, execute the example by doing:

```
:~/ostero/examples/square$ ../../finite_strain.py input_file.dat boundary_file.dat
ISOLINEAL MATERIAL MODEL / NONLINEAR FORMULATION / PLANE STRESS APPROXIMATION

Solving time step 1 ...
Newton-Raphson iteration: 1
Displacement increment error: 1.0
Newton-Raphson iteration: 2
Displacement increment error: 0.000556623737174
Newton-Raphson iteration: 3
Displacement increment error: 2.96846401262e-10

Solving time step 2 ...
Newton-Raphson iteration: 1
Displacement increment error: 0.500137516494
Newton-Raphson iteration: 2
Displacement increment error: 0.00027907811949
Newton-Raphson iteration: 3
Displacement increment error: 1.4971842011e-10

Solving time step 3 ...
Newton-Raphson iteration: 1
```

```
22  Displacement increment error: 0.333578181727
23  ...
```

The execution should finish successfully and one output file should be created for each time step. The output format is Vtk, which can be postprocessed using the open-source application ParaView.

The main program of Ostero is *finite_strain.py*. This main program recieves two additional arguments which need to be specified by the user. The first argument is the input file and the second is the boundaries file. The name of these files can be user-defined, but the order in which they are called must be respected. The first argument must be the input file while the second must be the boundaries file.

In the input file you must specify:

- The case name. Is just a user-specified name for the case under consideration (i.e. 2DBeam). If nothing is specified the default will be **NONAME**. The output files will be titled using this name. (keyword *case_name*)

- The mesh path. Is the relative or absolute path to the mesh file. If the mesh file is located in the execution folder, is enough to specify the name of the mesh. The mesh must be in Gmsh format. (keyword *mesh_path*)

- The geometrical treatment. In other words, if the calculation will be performed taking into account the finite strain theory or the linear theory. The accepted keyords for this option are **LINEAR** or **NONLINEAR**. (keyword *geometrical_treatment*)

- The material constitutive model for the non-linear treatment. Is a mandatory option if the **NONLINEAR** geometrical treatment was selected. For the **LINEAR** geometrical treatment this keyword is ignored. (keyword *constitutive_model*, options **ISOL** for isolinear (Kirchhoff) model or **BELY**, **ZIEN** or **LAUR** for neo-Hookean hyperelastic model as described in Belytschko's [1], Zienkiewicz's [2] and Laursen's [3] books respectively)

- The material 2D approximation for the **ISOL** option in nonlinear geometrical treatment or for the geometrical linear model. If the **LINEAR** or **NONLINEAR** and **ISOL** options are selected, the user must specify if the 2D approximation will be plane stress or plane strain. (keyword *submodel*, options **PLANE_STRESS** or **PLANE_STRAIN**)

- The size of each time step. Is a unique number that will define the length of each simulation time step. All time steps will be equal. (keyword *time_step_size*)

- The total number of time steps. Is a unique numer which defines the total number of time steps. The multiplication between the total number of time steps and the size of each time step will give the total simulation time. (keyword *total_steps*)

Any new line introduced in the input file, at any location, that doesn't start with any of the keywords described before will be interpreted as a general comment.

In the boundaries file you must specify:

- The volume definition and its mechanical properties. The order of parameters is: the name of the physical entity which defines the surface to which the material properties will be applied; the Young modulus; the Poisson modulus and the denisty. As Ostero solves in quasi-static mode and using an implicit scheme, the density value is used only if Gravity is activated, but for parsing reasons, this value is always needed. (keyword *$VolumeDefinition*)

- The definition of Dirichlet boundary conditions. The order of parameters is: the name of the physical entity which defines the line, group of points or individual point to which the Dirichlet boundary condition will be applied; pair of tags which indicate if the condition is active for x_1 and/or x_2 direction (1 active, 0 unactive); the value of displacement for x_1 and x_2 and the start and end time steps, which defines when the boundary condition will be active. (keyword *$BoundaryConditionsDisplacement*)

- The definition of Neumann boundary conditions. The order of parameters is: the name of the physical entity which defines the line or lines (Neumann boundary) to which the boundary condition will be applied and the pressure value, which will be applied in the normal direction to the Neumann boundary. Negative values mean compressive pressure while positive values represent an imposed traction pressure. (keyword *$BoundaryConditionsPressure*)

- The gravity effect (if present). The order of parameters is: the magnitude of gravity acceleration and its direction in x_1-x_2 plane. (keyword *$Gravity*)

To add comments in the boundaries file, they must be in a new line and start with the special character "#" or "!".

4 Examples

To demostrate how Ostero works, in the next chapter we will analyze several examples which show all the functionalities included in the code. Also this examples intends to show the user how to define the physical problem and create the input files.

4.1 ostero/examples/square

In this example we are going to model the non-linear deformation of a square geometry composed of an isolinear (Kirchhoff) material, which is subjected to a completely fixed Dirichlet boundary condition on its bottom and a compressive Neumann boundary condition on its top. In this first example we will start from the very beginning of any finite elements analysis, that is the generation of the triangular mesh. The mesh generation procedure is the same for every mesh used in these examples, so the methodology explained in this section can be easily applied to generate the meshes of the remaining examples (despite there is no need to do that because their are included in the distribution) or the meshes for other 2D analysis using Ostero.

As mentioned before, Ostero reads meshes generated with the open source meshing tool Gmsh, which can be downloaded in binary form here. Using Gmsh we will generate a square as the union of 4 lines of size 5 meters (to be consistent with the units of measure, we will express all the physical quantities in the International System of Units).

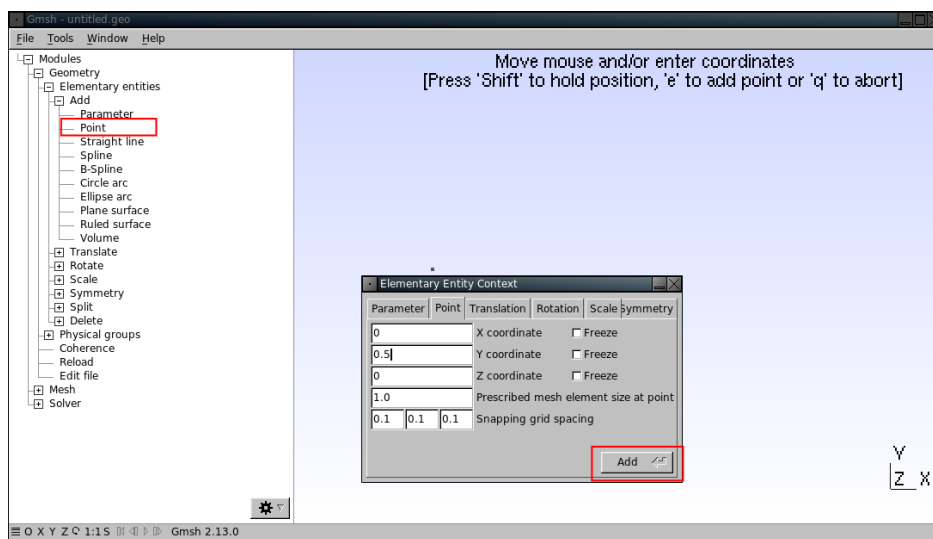Once Gmsh is on your system, simply execute the program by writing:

```
:~/gmsh-2.13.0-Linux/bin$ ./gmsh
```
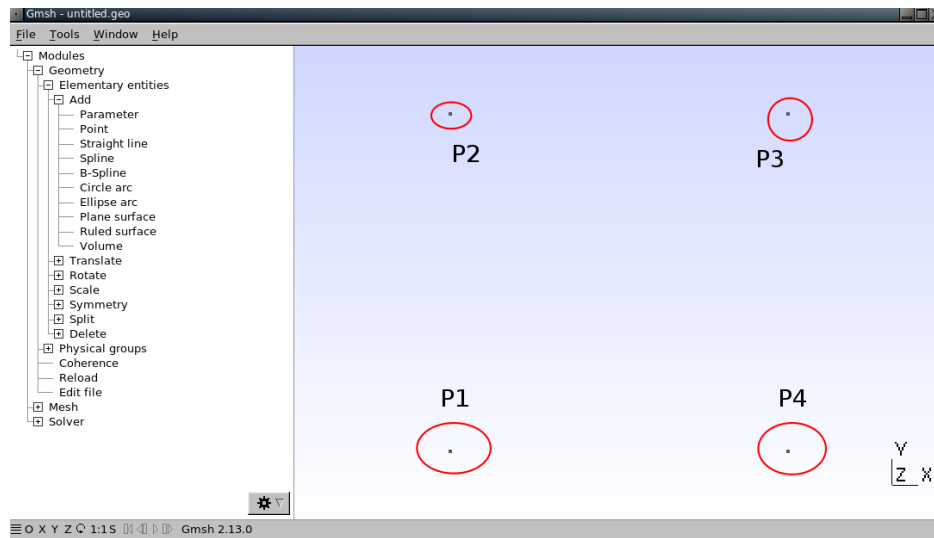
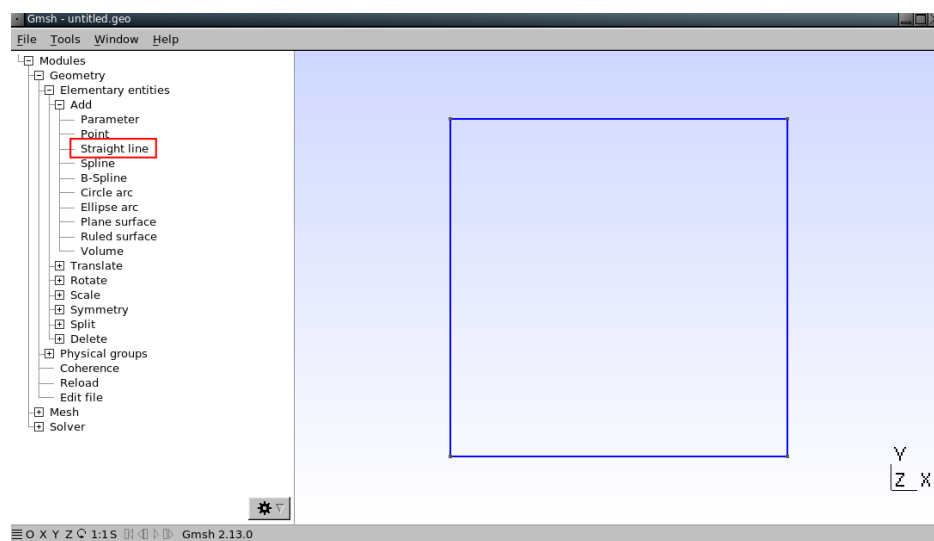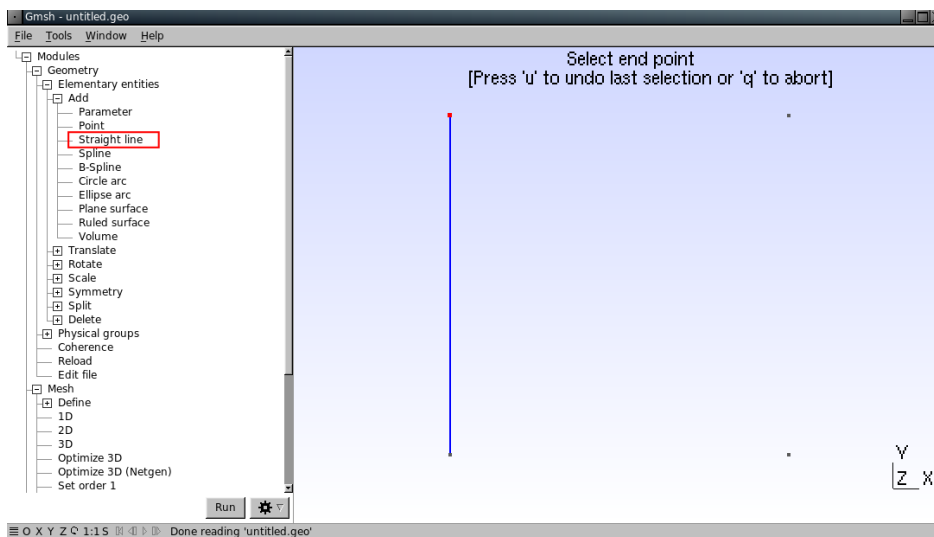and the graphical user interface (GUI) of Gmsh should appear:
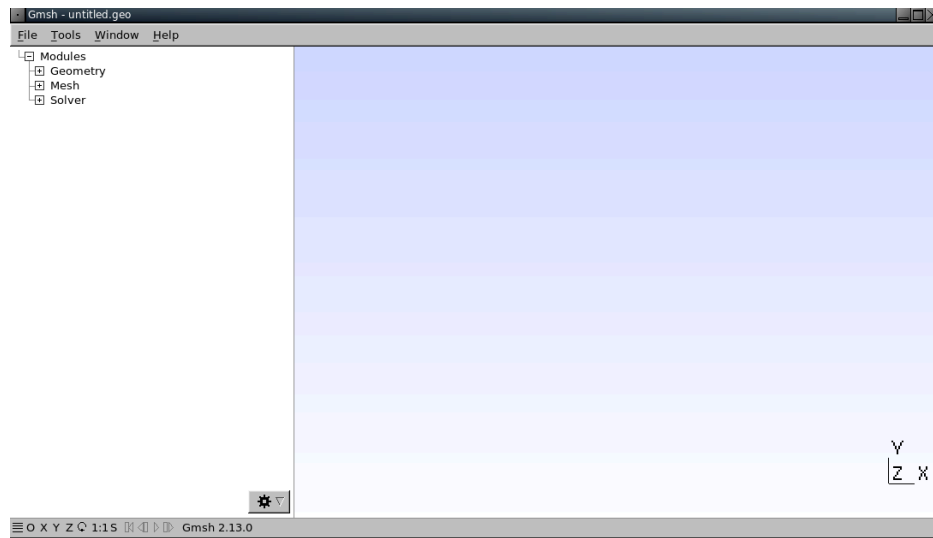
The first step is to add 4 points, which will be the 4 vertex of the square. Its locations are:

- Point 1: (0.0 ; 0.0)

- Point 2: (0.0 ; 5.0)

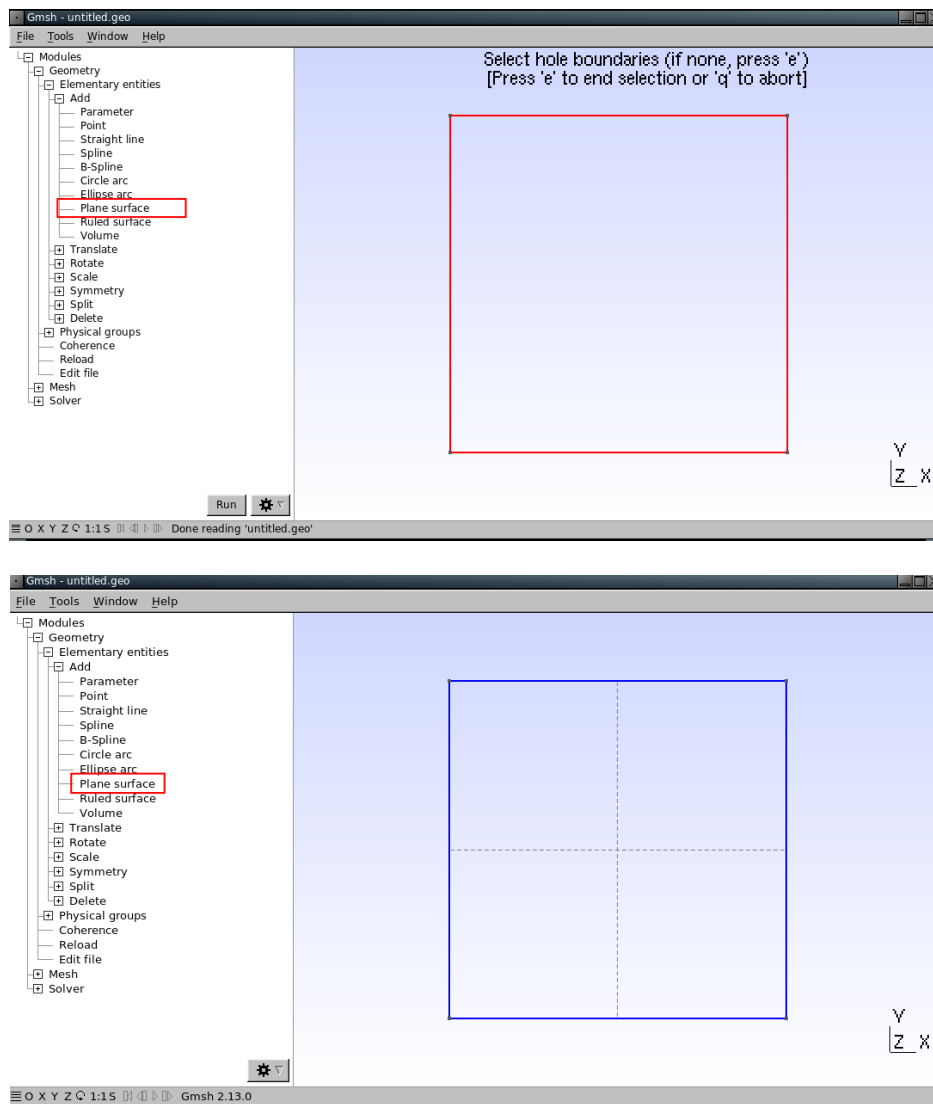- Point 3: (5.0 ; 5.0)

- Point 4: (5.0 ; 0.0)

Now, we are ready to conect the points with straight lines to create the wireframe:

Once we have created the wireframe, it is time to construct the 2D plane surface. To do this we must select the wireframe that will be the boundary of the domain:

Up to now we have created the geometry. Now we need to add tags that will tell Ostero which will be the simulation domain where the material will be defined, and which will be the boundaries were the different boundary conditions will be applied. As in this example we are interested on appling boundary condition on the bottom and top part of the geometry, we will apply tags to those regions. In Gmsh these tags are added by means of the Physical Groups option, which allows to define tags for points, lines, surfaces or volumes. In this case we will add two different physical groups for each line and one physical group for the hole surface:
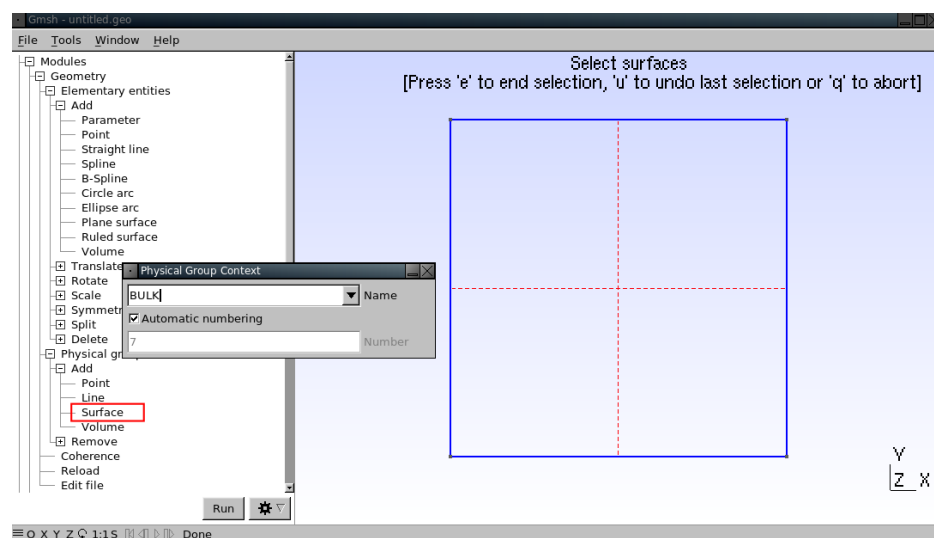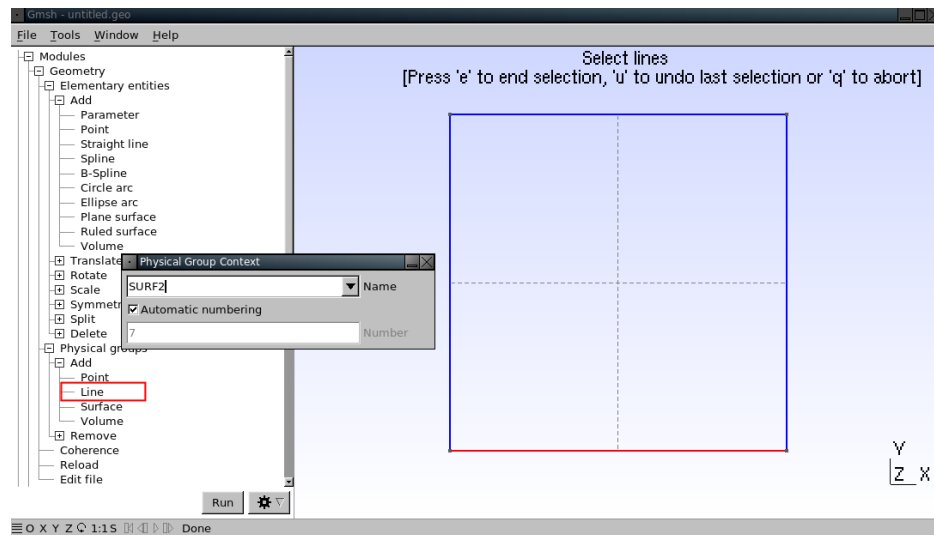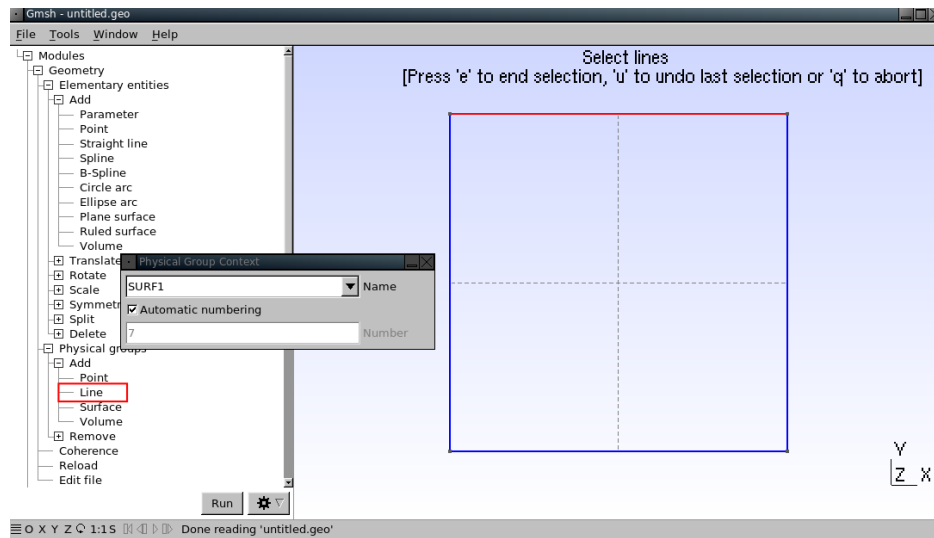
Once you have reached this point, you can close the Gmsh GUI. By default Gmsh saves all the commands executed in the GUI in a file called *untitled.geo*, located in the execution path of Gmsh. This file can be renamed and opened later from inside the Gmsh GUI for future uses.

If everything went fine, the untitled.geo (or_any_name_the_user_want.geo) file will look as the following:

```
1   Point(1) = {0, 0, 0, 1.0};
2   Point(2) = {0, 5, 0, 1.0};
3   Point(3) = {5, 5, 0, 1.0};
4   Point(4) = {5, 0, 0, 1.0};
5   Line(1) = {1, 2};
6   Line(2) = {2, 3};
7   Line(3) = {3, 4};
8   Line(4) = {4, 1};
9   Line Loop(5) = {1, 2, 3, 4};
10  Plane Surface(6) = {5};
11  Physical Line("SURF1") = {2};
12  Physical Line("SURF2") = {4};
13  Physical Surface("BULK") = {6};
```

If in your .geo version the name of the physical entities are missing, just add them by hand.

The next and final step is to generate the mesh. To do this we load the untitled.geo file in Gmsh by doing:

```
1   :~/gmsh-2.13.0-Linux/bin$ ./gmsh untitled.geo
```

and set the element size factor to define the size of the mesh to 0.26.
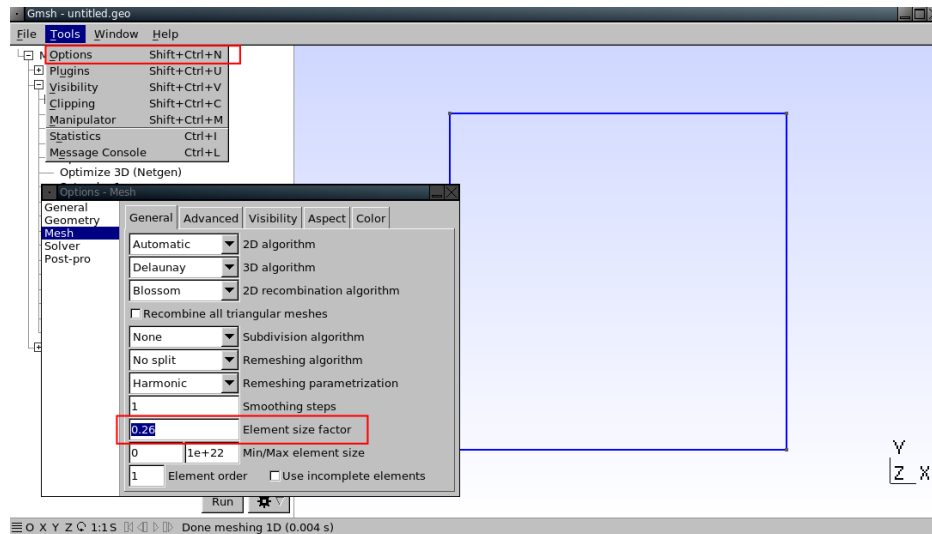
Optionally, we can visualize the created physical entities to check that they are ok:
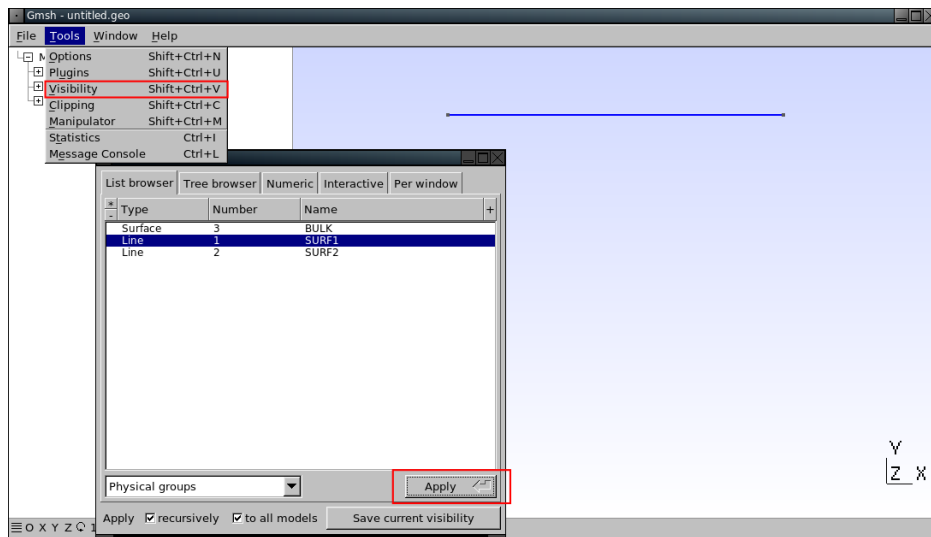

Finally, we generate and save the mesh:
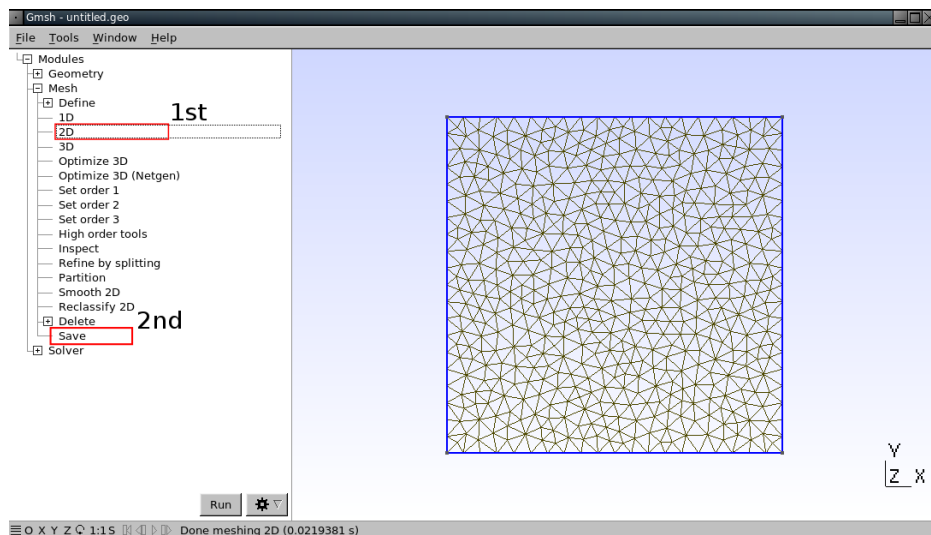
The mesh will be written in a file with extension .msh and name equal to the one assigned to the .geo file. In our case, as we didn't change the name to the .geo, the mesh will be written as untitled.msh. We will rename the mesh to square.msh and check if it was correctly saved:

```
:~/gmsh-2.13.0-Linux/bin$ mv untitled.msh square.msh
:~/gmsh-2.13.0-Linux/bin$ ./gmsh square.msh
```
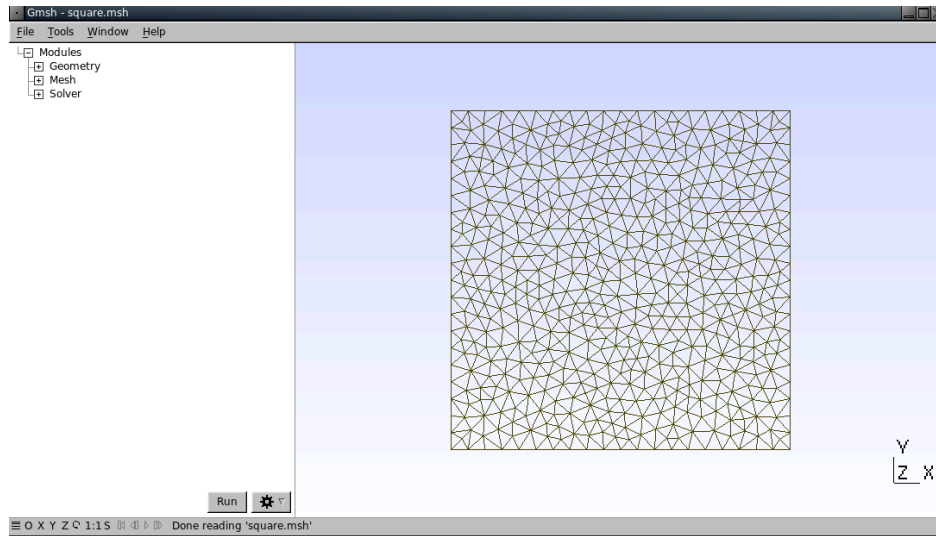
Now that we have generated the mesh, it is time to set up the input and boundaries file for the finite element analysis. Let's start from scratch creating a new folder called *square2* inside the examples folder of Ostero, and coping the recently created mesh to that location:

```
:~/ostero/examples$ mkdir square2
:~/ostero/examples$ cd square2
:~/ostero/examples/square2$ cp /gmsh-2.13.0-Linux/bin/square.msh .
```

Next step is to set the input file. As we said before, we would like to use the square.msh mesh. Also we want to solve considering geometrical non-linearities, using the isolinear material model and supposing we are in a plane stress assumption. We would like to run a simulation analysis of 0.05 seconds of total time, in 5 time steps of 0.01 seconds each one. With all this in mind, let's create the input file and name it *input.dat*, which should have the following content:

```
case_name SQUARE
mesh_path square.msh
geometrical_treatment NONLINEAR
constitutive_model ISOL
sub_model PLANE_STRESS
time_step_size 0.01
total_steps 5
```

Now it is time to specify the material parameters and boundary conditions for the analysis. First, we want to apply the material properties to the hole domain, which was named *BULK* during the mesh generation. To this material we will assign a Young modulus of 6.896E7 N/m^2, a Poisson modulus of 0.32 and a density equal to 1 kg/m^3. Then we want to impose a Neumann boundary condition over the top surface, which was named *SURF1*, equal to -100000.0 Pa (is negative because is a compressive pressure). Finally, we want to fix the bottom part of the body (*SURF2*) in both directions x_1 and x_2, during the total simulation. To fix, we impose a null displacement (Dirichlet boundary condition). The boundaries file, which will be named as boundaries.dat, should look like the following:

```
$VolumeDefinition
BULK 6.896E7 0.32 1
$BoundaryConditionsDisplacement
SURF2 1 1 0.0 0.0 1 5
$BoundaryConditionsPressure
SURF1 -100000.0
```

15

The final step is to run the simulation by executing Ostero with the input file and boundaries file as command arguments:

```
 1  :~/ostero/examples/square2$ ../../finite_strain input.dat boundaries.dat
 2  ISOLINEAL MATERIAL MODEL / NONLINEAR FORMULATION / PLANE STRESS APPROXIMATION
 3
 4  Solving time step 1 ...
 5  Newton-Raphson iteration: 1
 6  Displacement increment error: 1.0
 7  Newton-Raphson iteration: 2
 8  Displacement increment error: 0.000439386037081
 9  Newton-Raphson iteration: 3
10  Displacement increment error: 9.02152198392e-11
11
12  Solving time step 2 ...
13  Newton-Raphson iteration: 1
14  Displacement increment error: 0.500109778173
15  Newton-Raphson iteration: 2
16  Displacement increment error: 0.000220112630595
17  Newton-Raphson iteration: 3
18  Displacement increment error: 4.53296713812e-11
19
20  Solving time step 3 ...
21  Newton-Raphson iteration: 1
22  Displacement increment error: 0.333528700297
23  Newton-Raphson iteration: 2
24  Displacement increment error: 0.000147022482619
25  Newton-Raphson iteration: 3
26  Displacement increment error: 3.03647367894e-11
27
28  Solving time step 4 ...
29  Newton-Raphson iteration: 1
30  Displacement increment error: 0.250247531434
31  Newton-Raphson iteration: 2
32  Displacement increment error: 0.000110478153067
33  Newton-Raphson iteration: 3
34  Displacement increment error: 2.28900053569e-11
35
36  Solving time step 5 ...
37  Newton-Raphson iteration: 1
38  Displacement increment error: 0.200281948004
39  Newton-Raphson iteration: 2
40  Displacement increment error: 8.85521539567e-05
41
42  EXECUTION FINISHED SUCCESSFULLY!
```

For each time step, inlcuding the initial mesh, Ostero writes the output in a Vtk file. Once the simulation is finished, we can check that the output was generated ok by listing all the contents of the case directory:

```
 1  :~/ostero/examples/square2$ ls
 2  -rw-r--r-- 1 user user Jul 15 18:53 boundaries.dat
 3  -rw-r--r-- 1 user user Jul 16 18:50 input.dat
 4  -rw-r--r-- 1 user user Jul 16 14:51 square.msh
 5  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.0.vtk
 6  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.1.vtk
 7  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.2.vtk
 8  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.3.vtk
 9  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.4.vtk
10  -rw-r--r-- 1 user user Jul 16 18:50 SQUARE_NONLINEAR_ISOLIN_PLANE_STRESS_out.5.vtk
```

Vtk files can be opened with the open source postprocessing tool ParaView, which can be downloaded from here. ParaView can be downloaded as a binary package, so there is no need of compilation from sources. If ParaView is not in your system, simply download, uncompress if necessary, and execute the binary file.
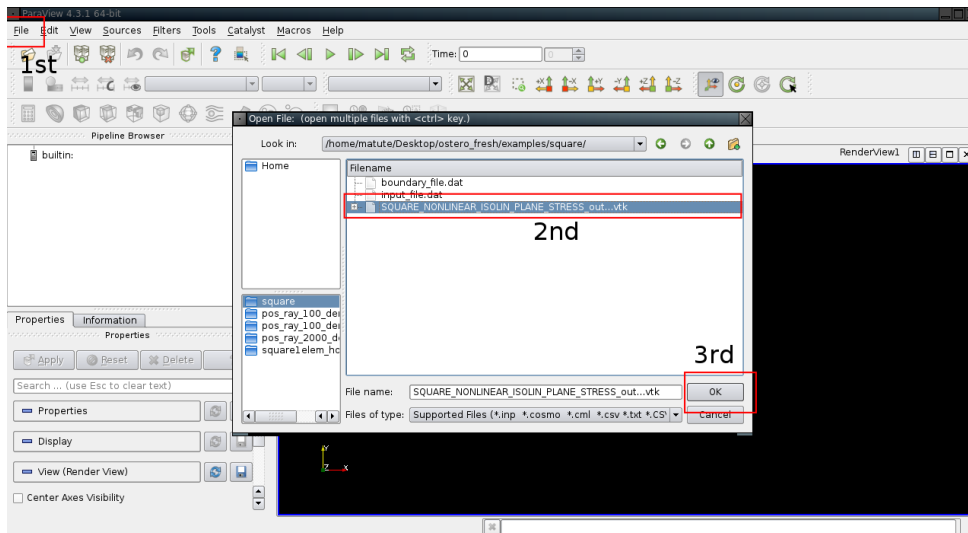
To launch ParaView, change directory to the bin folder and simply execute:
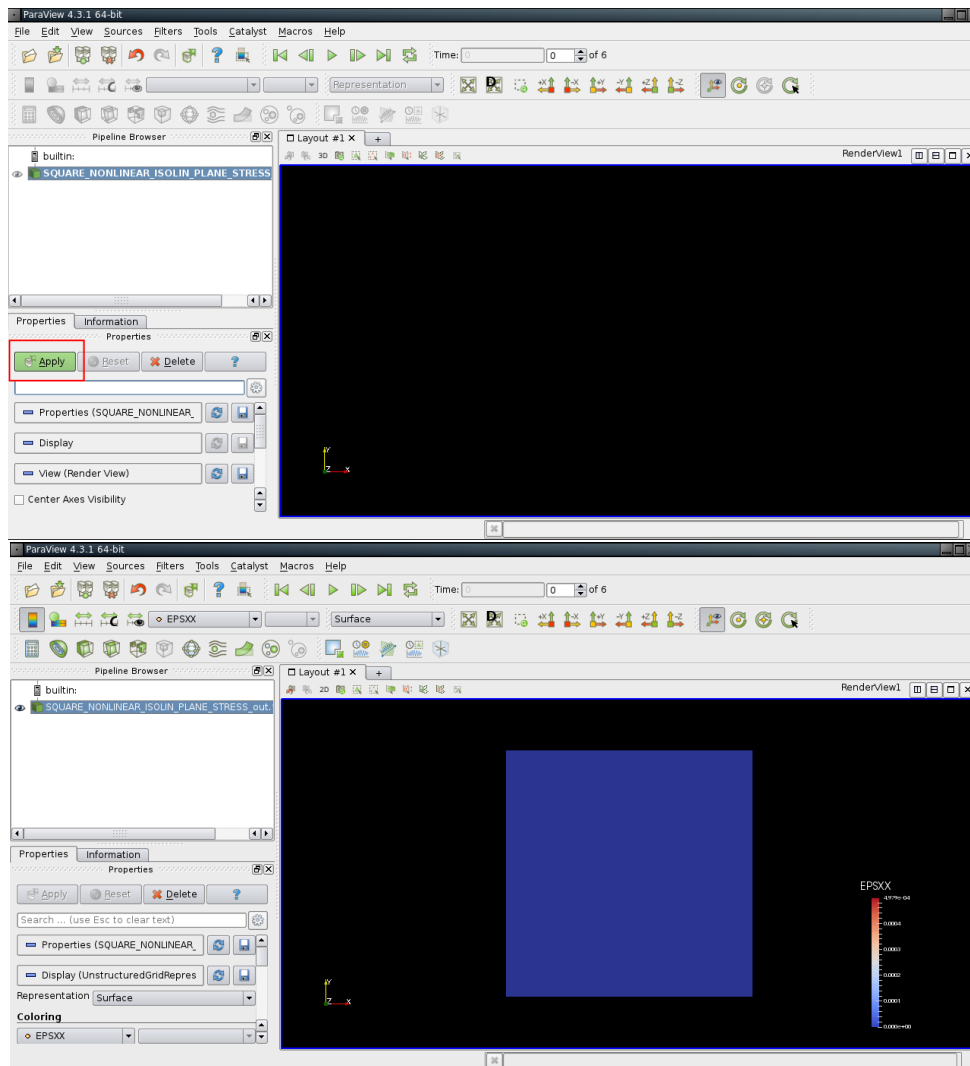
```
:~/ParaView-4.3.1-Linux-64bit/bin$ ./paraview
```

The main screen of ParaView should appear:
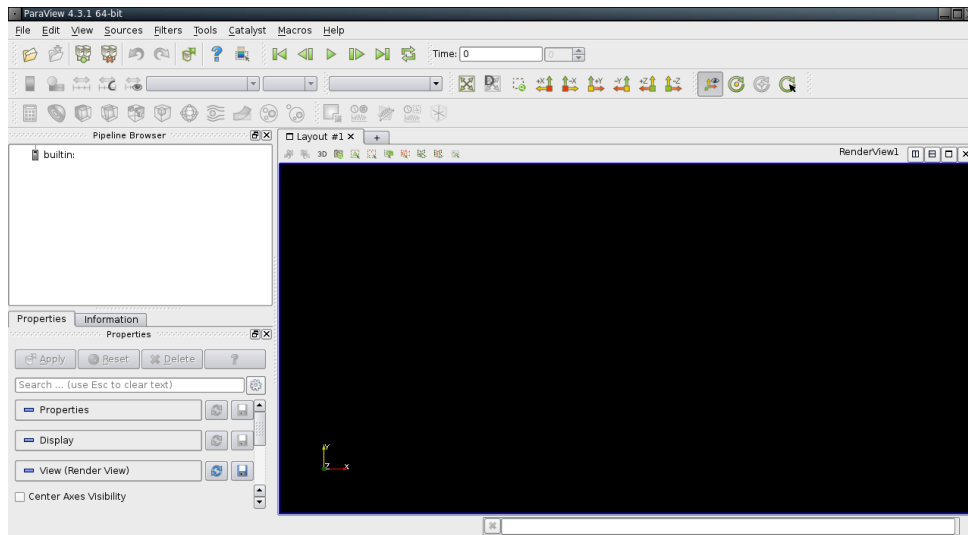
To open a set of results, go to File -> Open, navigate to the results path and select the hole bunch of Vtk files generated by Ostero, as shown in the following picture:

Once the results were loaded, simply press the apply button to visualize them on screen:

To see the mesh deformation, first select and then apply the *Wrap by Vector* filter:
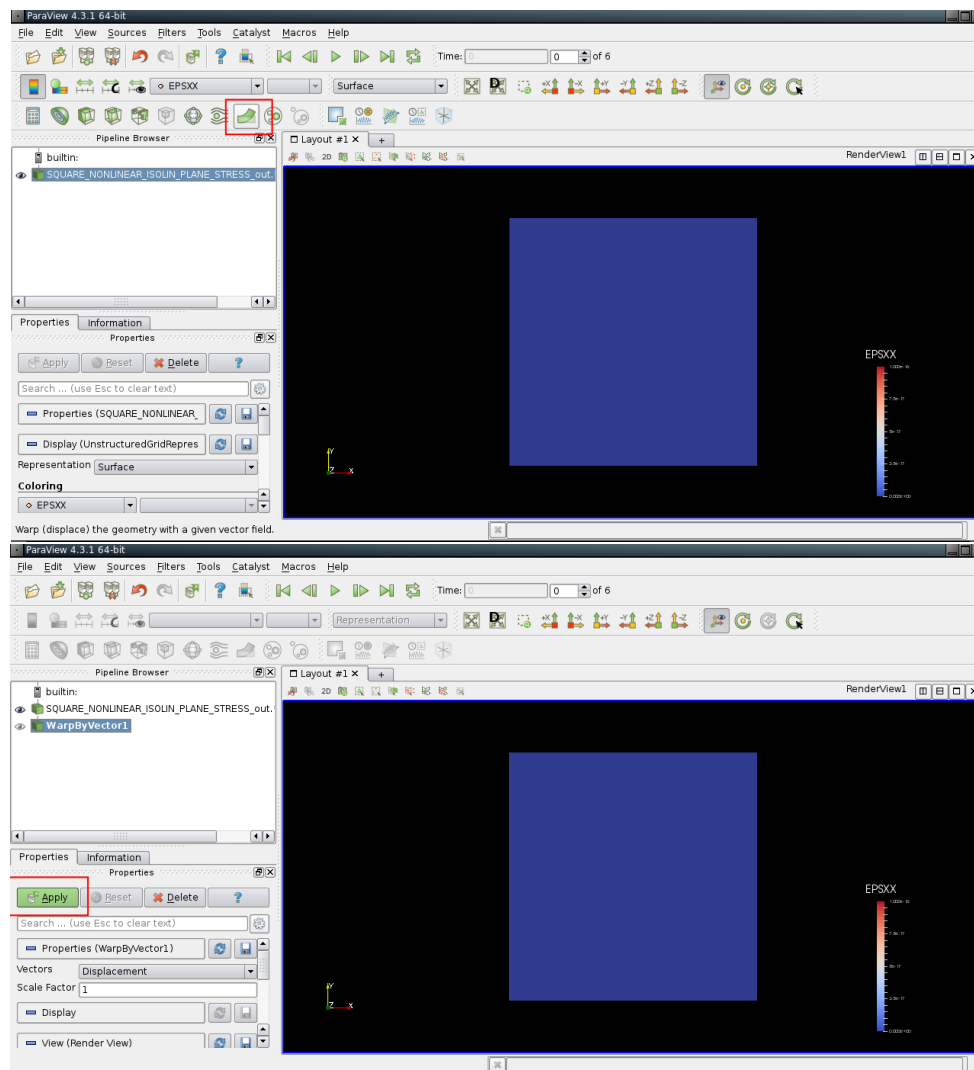
Then apply a scale factor of 100 to magnify the results and advance to the last time step:
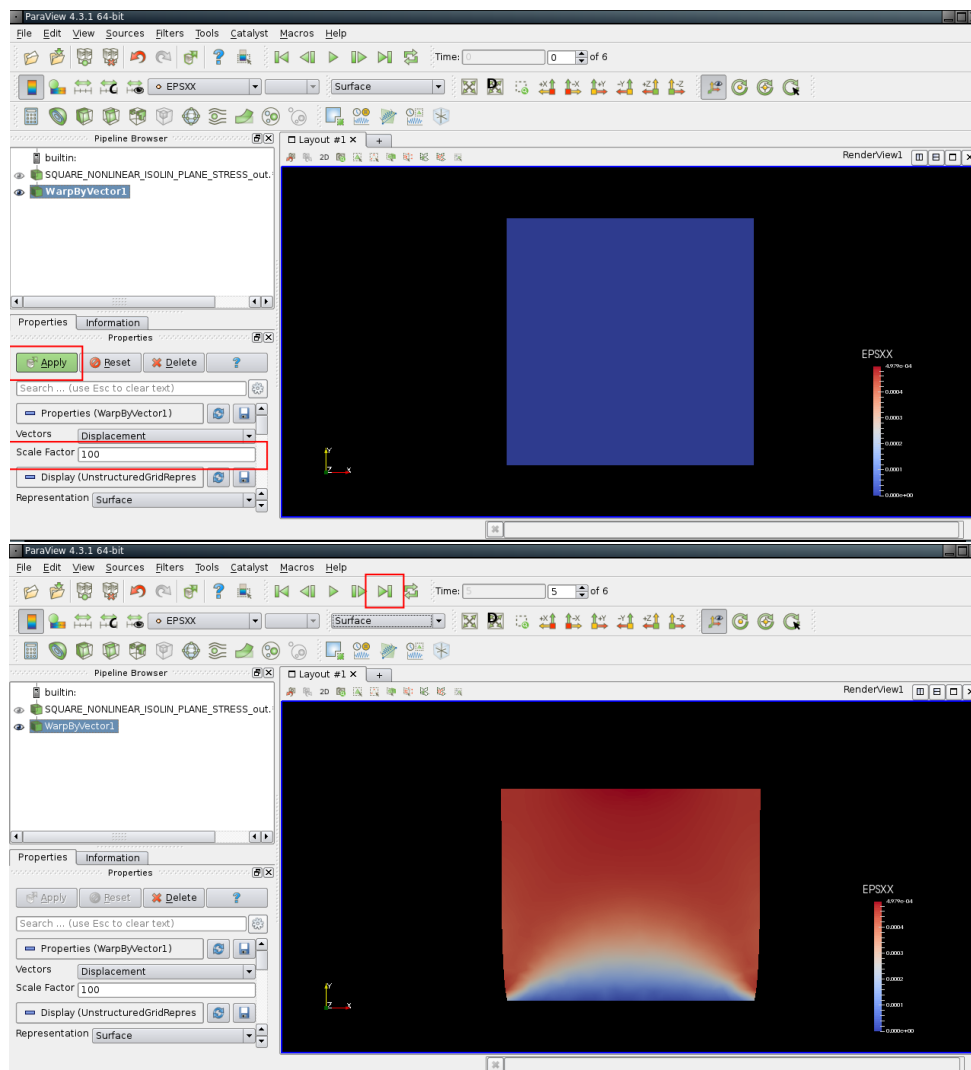
Also is possible to change the variable to postprocess and the geometry features that will be displayed on screen:
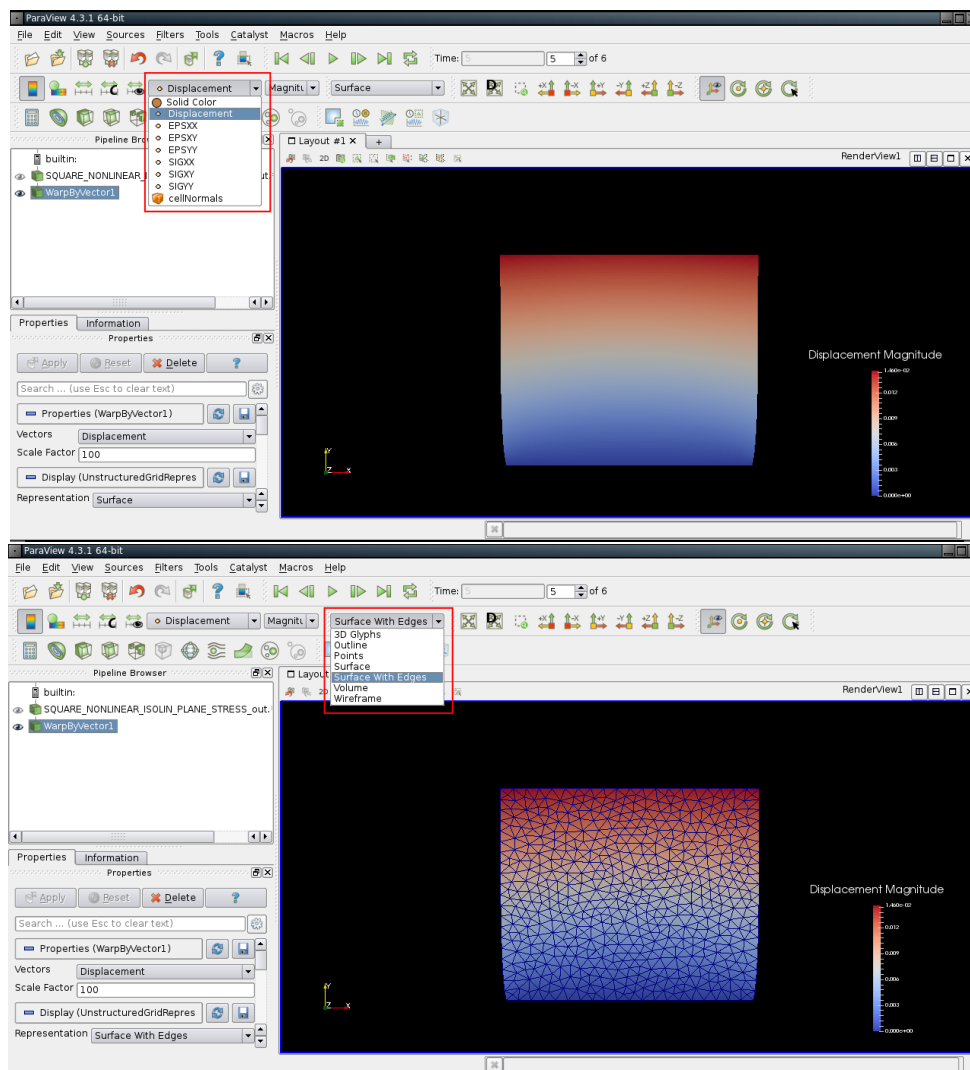
You can also save a screenshot of a given time step of the simulation by doing File -> Save Screenshot:
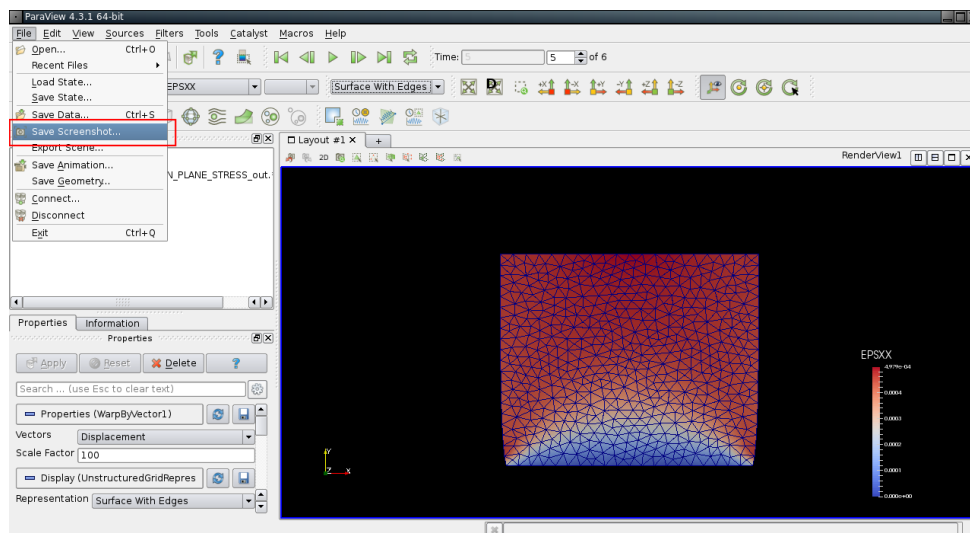
4.2   ostero/examples/complete

In this example we are going to solve a non-linear problem using a Kirchhoff material under plane stress assumption. We will simulate 5 time steps of 0.01 seconds each. The input file is defined as following:

```
1  case_name COMPLETE
2  mesh_path complete.msh
3  geometrical_treatment NONLINEAR
4  constitutive_model ISOL
5  sub_model PLANE_STRESS
6  time_step_size 0.01
7  total_steps 5
```

The domain, tagged as *BULK*, has a material with Young modulus equal to 6.896E7 N/m^2, a Poisson modulus of 0.32 and a density equal to 10 kg/m^3. Two kind of boundary conditions will be applied: Dirichlet boundary conditions over *SURF1* and *SURF2* and Neumann over *SURF3*:
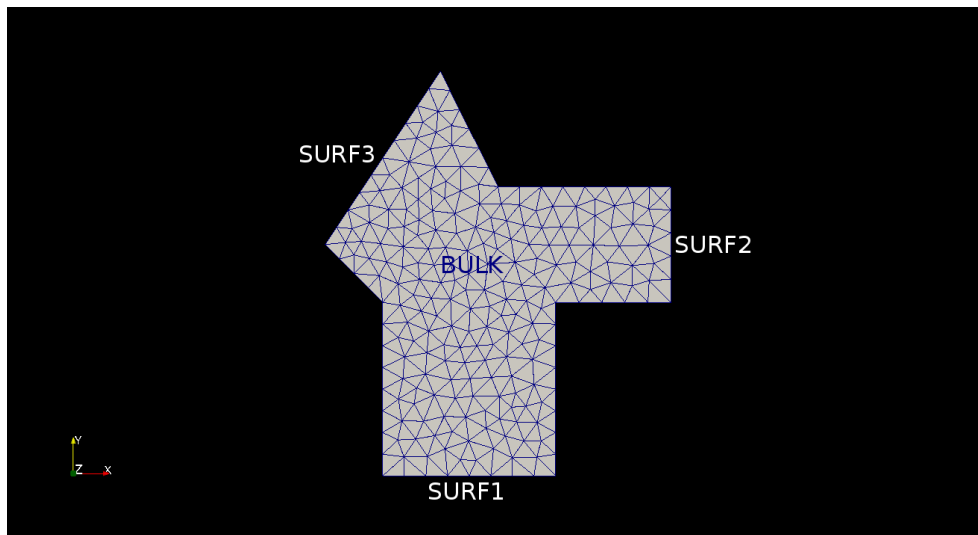
We will fix boundary *SURF1*, both in x_1 and x_2 direction. To boundary *SURF2* we will apply an incremental displacement of 0.04 meters per time step. As we are going to simulate 5 time steps, the total displacement will be of 0.2 meters. Over *SURF3* we will apply a compressive pressure equal to 100000 Pa. Given the previous conditions, the boundaries file will be:

```
1  $VolumeDefinition
2  BULK 6.896E7 0.32 10
3  $BoundaryConditionsDisplacement
4  SURF1 1 1 0.0 0.0 1 5
5  SURF2 1 1 0.2 0.0 1 5
6  $BoundaryConditionsPressure
7  SURF3 -100000.0
```

Given the mesh, the input and boundaries file we can run Ostero and postprocess the results to see the geometry deformation under the imposed boundary conditions:
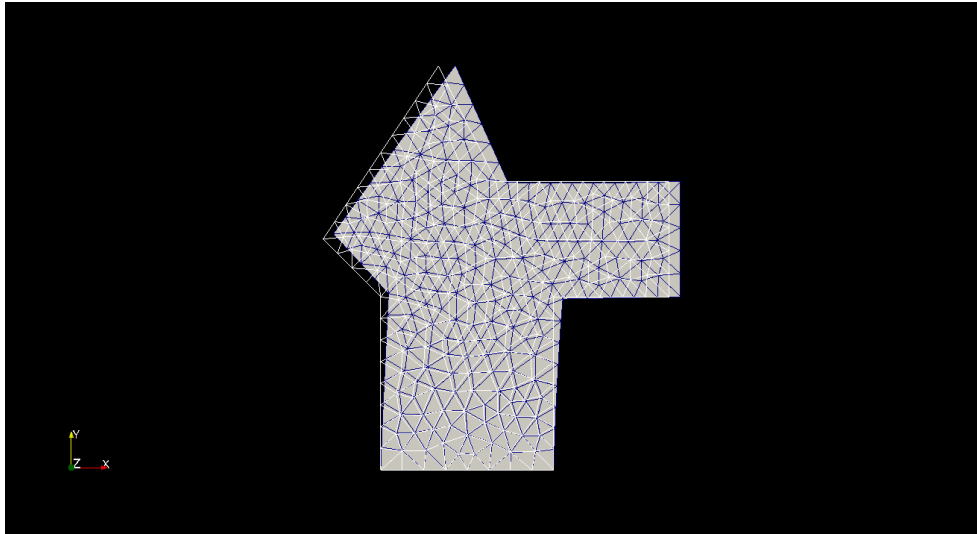
Also we can analyse the stresses distribution σ_{xx} and σ_{yy} in the simulation domain:
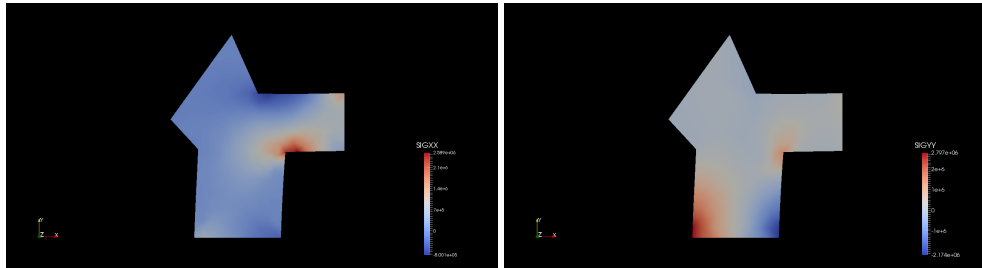


4.3 ostero/examples/two_materials

This example consist of two square bodies, one in top of the other. The particularity of this example is that the two bodies are composed of different materials. We are going to analyse this case using the linear theory under plane stress assumption, during 5 time steps of 0.01 seconds each. The input file results:

```
case_name TWO_MATERIALS
mesh_path two_materials.msh
geometrical_treatment LINEAR
sub_model PLANE_STRESS
time_step_size 0.01
total_steps 5
```

The top body, tagged *TRISTOP*, has a material with Young modulus equal to 6.896E6 N/m^2, a Poisson modulus of 0.32 and a density equal to 0.1 kg/m^3. The bottom body, tagged *TRISBOTTOM*, has a material with Young modulus equal to 6.896E7 N/m^2, a Poisson modulus of 0.32 and a density equal to 1 kg/m^3. The system is under gravitational force along the x_2 direction (gravity acceleration equal to 9.8 m/s^2), the bottom is assumed as fixed in x_1 and x_2 direction during the complete simulation and on the top a compressive pressure equal to 100000 Pa is applied.
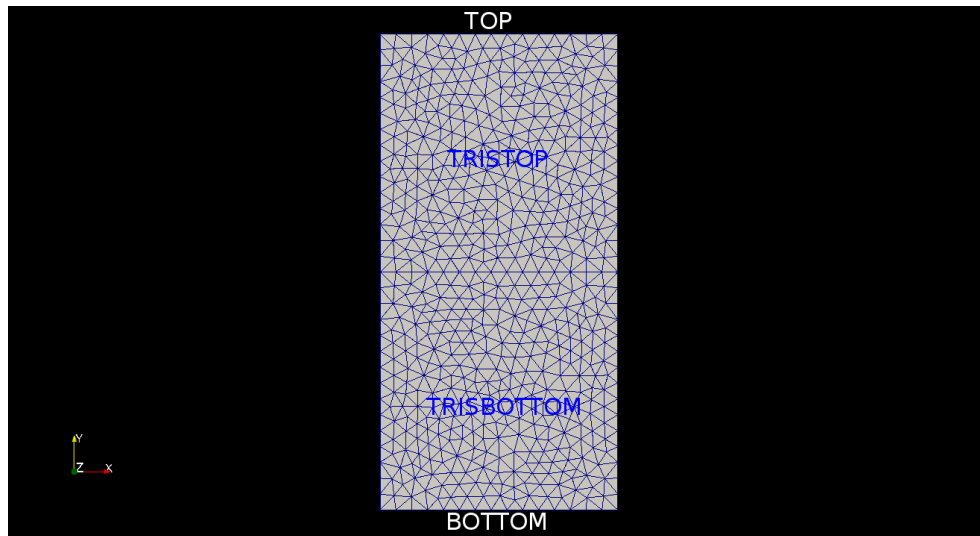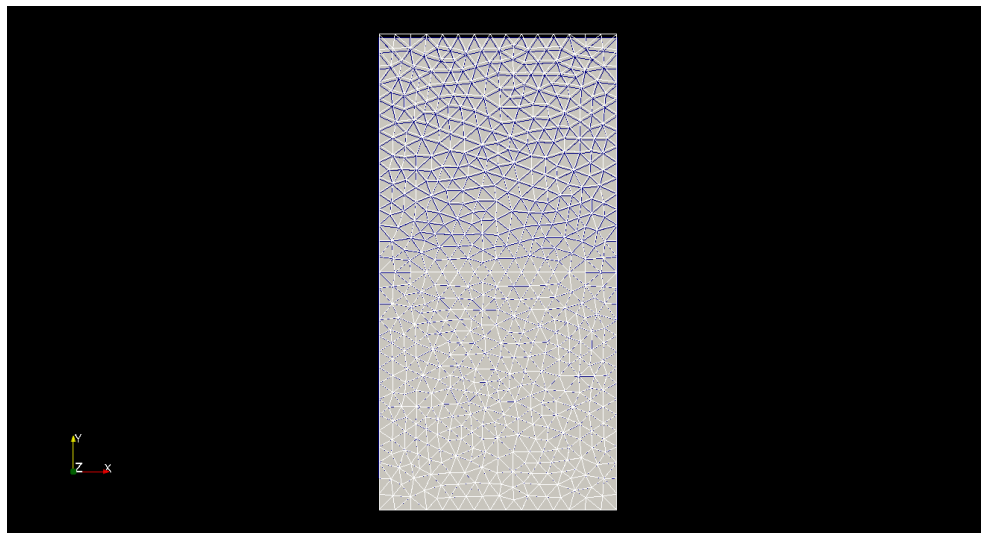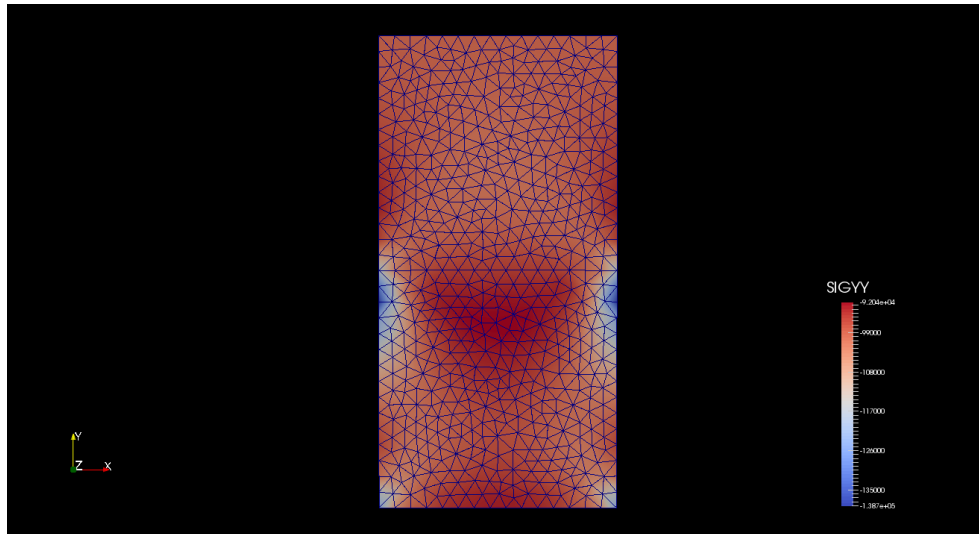
The boundaries file is then as follow:

```
1   $VolumeDefinition
2   TRISTOP 6.896E6 0.32 0.1
3   TRISBOTTOM 6.896E7 0.32 1
4   $BoundaryConditionsDisplacement
5   BOTTOM 1 1 0.0 0.0 1 5
6   $BoundaryConditionsPressure
7   TOP -100000.0
8   $Gravity
9   9.8 0 1
```

Now we can run the case and postprocess the results to visualize the mesh deformation and the stresses distribution on x_2 direction (σ_{yy}):

Also we can postprocess the strain distribution on x_2 direction (ϵ_{yy}). We can observe that, as expected, the strains are one order of magnitude bigger in the softer body, which is the one tagged as *TRISTOP*:
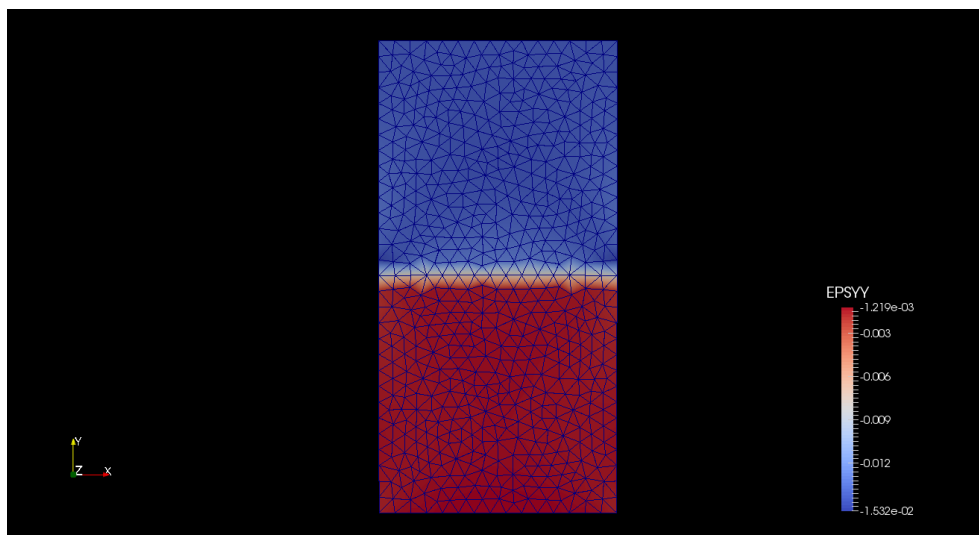
4.4 ostero/examples/point_bcs

This example shows the usage of Dirichlet boundary conditions over nodes instead of boundaries. This is a transparent procedure for Ostero, as the boundaries file doesn't change for one case or the other. The difference exists during the mesh generation, which is slightly different from what we have explained in the *square* example. Instead of tagging lines as physical entities, we must tag the point or group of points where we want to impose the Dirichlet boundary condition.

We will create a triangular mesh over a square geometry. We will define the hole domain as a physical surface named *BULK*, the four lines of the boundary as *TOP*, *BOTTOM* and *LATER-ALS*, and the four vertices as physical points named as *ARRIBAIZQ*, *ARRIBADER*, *ABAJOIZQ* and *ABAJODER*:

25
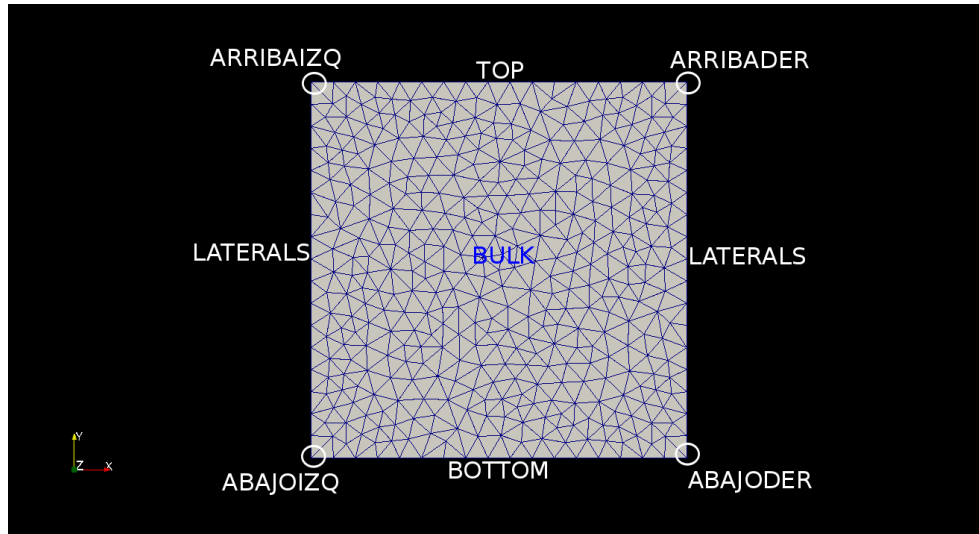
Mesh definition:

```
1  Point(1) = {0, 0, 0, 1.0};
2  Point(2) = {1, 0, 0, 1.0};
3  Point(3) = {1, 1, 0, 1.0};
4  Point(4) = {0, 1, 0, 1.0};
5  Line(1) = {1, 2};
6  Line(2) = {2, 3};
7  Line(3) = {3, 4};
8  Line(4) = {4, 1};
9  Line Loop(5) = {1, 2, 3, 4};
10 Plane Surface(6) = {5};
11
12 Physical Point("ARRIBAIZQ") = {4};
13 Physical Point("ARRIBADER") = {3};
14 Physical Point("ABAJOIZQ") = {1};
15 Physical Point("ABAJODER") = {2};
16 Physical Line("TOP") = {3};
17 Physical Line("BOTTOM") = {1};
18 Physical Line("LATERALS") = {2, 4};
19 Physical Surface("BULK") = {6};
```

We plan to solve a non-linear case using an hyperelastic neo-Hookean material model using the Belytschko's formulation. We will set 5 time steps of 1 second size each. The input file is:

```
1  case_name POINTS
2  mesh_path point.msh
3  geometrical_treatment NONLINEAR
4  constitutive_model BELY
5  time_step_size 1
6  total_steps 5
```

Regarding the boundary definitions and material properties, we will set the following values: the material Young modulus is 6.896E7 N/m^2, Poisson modulus is 0.32 and density is 1 kg/m^3. We will apply a compressive pressure of 500000 Pa. We will fix the *ABAJOIZQ* node in both directions and the *ABAJODER* node only in x_2 direction during the total simulation. The boundaries file is:
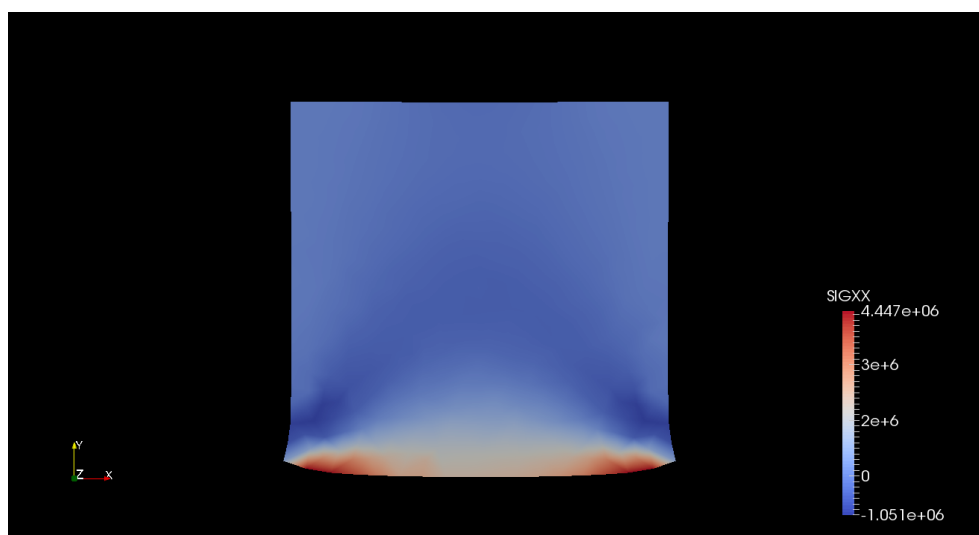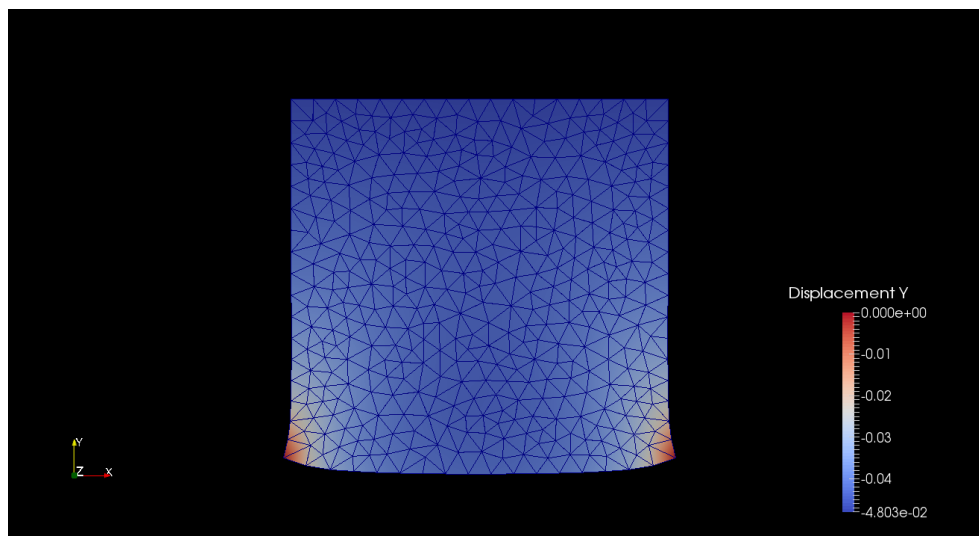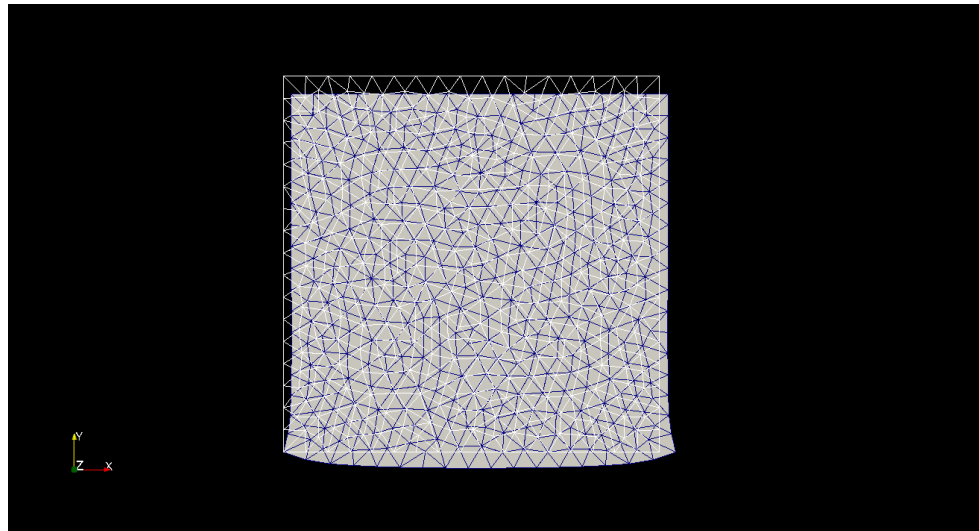
```
1  $VolumeDefinition
2  BULK 6.896E7 0.32 1
3  $BoundaryConditionsDisplacement
4  ABAJOIZQ 1 1 0.0 0.0 1 5
5  ABAJODER 0 1 0.0 0.0 1 5
```

```
6   $BoundaryConditionsPressure
7   TOP  -1000000.0
```

We run the case and postprocess the results to visualize the mesh deformation, displacements on x_2 direction and stress σ_{xx} distribution:

4.5 ostero/examples/square_quads

This example is to show that Ostero can deal with quadrilateral elements. The recognition of the element type is done automatically by Ostero while reading the mesh file, so there is no need to specify the mesh type in the input file or in the boundaries file. Due to this feature, Ostero can also manage mixed meshes, having both triangles and quadrilateral type elements.

We will analyse a non-linear case using a Kirchhoff material model under plane stress assumption in 5 time steps of 1 second each. We will fix the *BOTTOM* boundary and apply a compressive pressure over *TOP*:
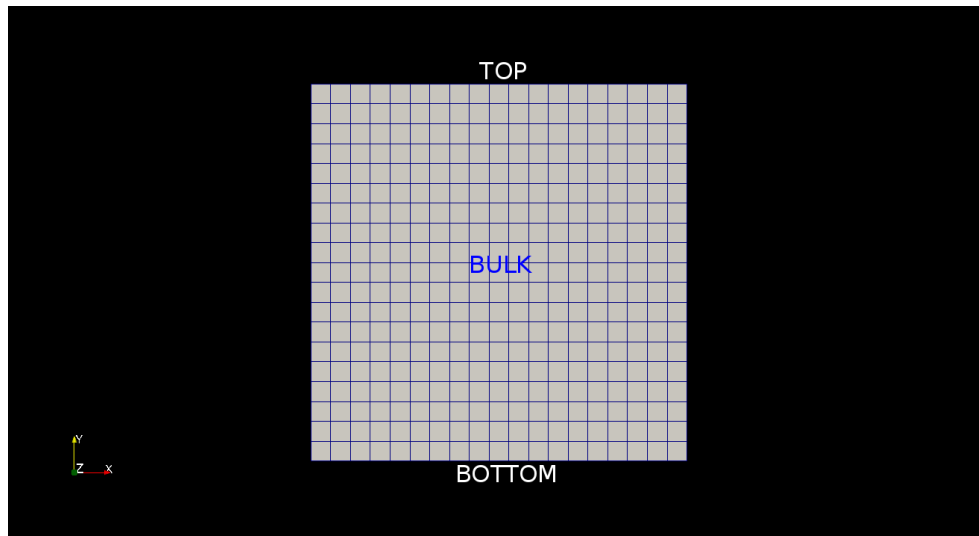
The mesh definition (.geo file) is:

```
1   Point(1) = {0, 0, 0, 1.0};
2   Point(2) = {1, 0, 0, 1.0};
3   Point(3) = {1, 1, 0, 1.0};
4   Point(4) = {0, 1, 0, 1.0};
5   Line(1) = {1, 2};
6   Line(2) = {2, 3};
7   Line(3) = {3, 4};
8   Line(4) = {4, 1};
9   Transfinite Line{1,3} = 20;
10  Transfinite Line{2,4} = 20;
11  Line Loop(5) = {1, 2, 3, 4};
12  Plane Surface(6) = {5};
13  Transfinite Surface{6};
14  Recombine Surface {6};
15
16  Physical Line("TOP") = {3};
17  Physical Line("BOTTOM") = {1};
18  Physical Line("LATERALS") = {2, 4};
19  Physical Surface("BULK") = {6};
```

The input and boundaries files are:

```
1   case_name SQUARE_QUADS
2   mesh_path square_quads.msh
3   geometrical_treatment NONLINEAR
4   constitutive_model ISOL
5   sub_model PLANE_STRESS
6   time_step_size 0.01
7   total_steps 5
```
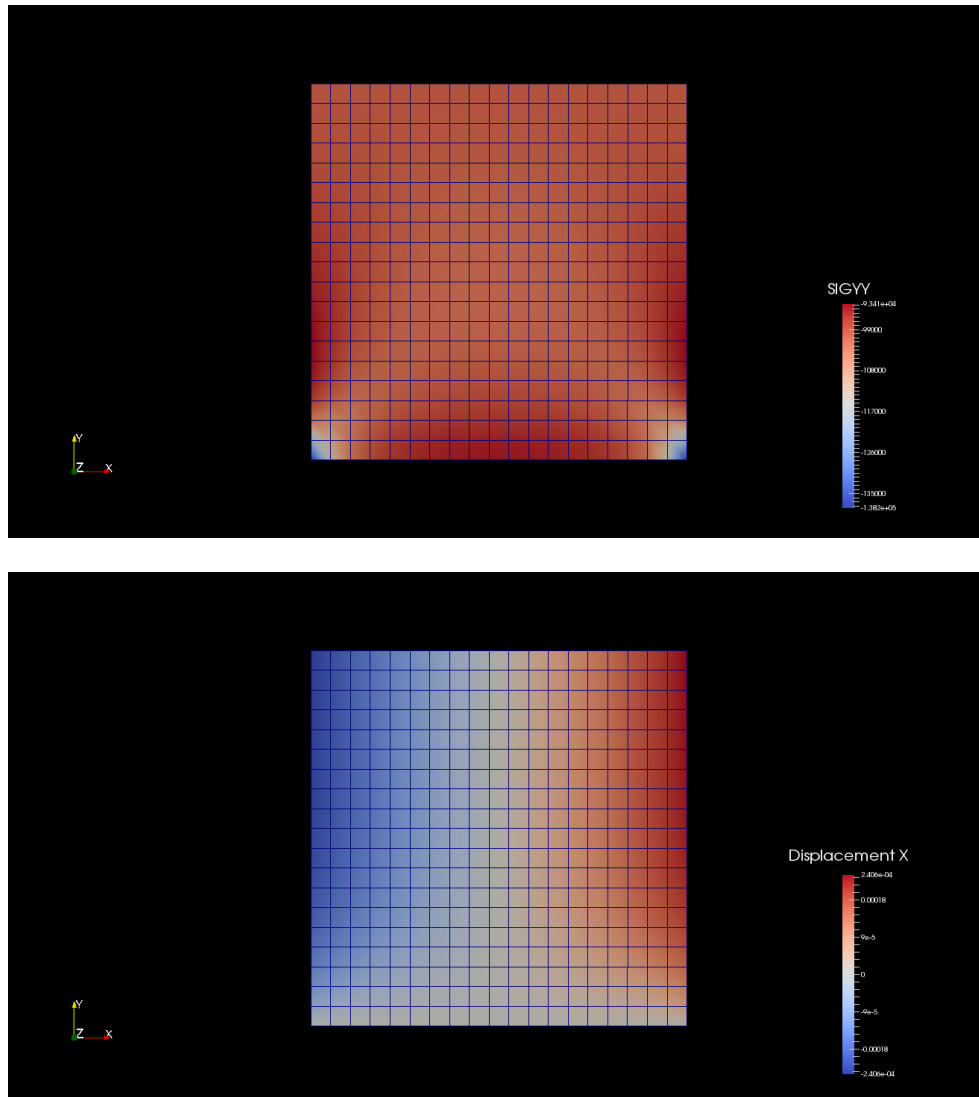
28

```
1  $VolumeDefinition
2  BULK 6.896E7 0.32 1
3  $BoundaryConditionsDisplacement
4  BOTTOM 1 1 0.0 0.0 1 5
5  $BoundaryConditionsPressure
6  TOP -100000.0
```

Running the case and postprocessing the results we obtain for the stress distribution in x_2 (σ_{yy}) and displacement in x_1 coordinate the following:

4.6   ostero/examples/DCB

This last example is to demostrate the capability that Ostero has to impose non-constant Dirichlet boundary conditions. To show this feature we are going to simulate a simplified and slightly modified *double cantilever beam* (DCB) example. The difference would be that insted of appling concentrated forces we are going to apply imposed displacements on corresponding nodes.

The vertical displacements would be applied in nodes *ARRIBAIZQ* and *ABAJOIZQ*, defined as point physical entities. The boundary *LATRIGHT* would remain fixed in both directions during the complete simulation.
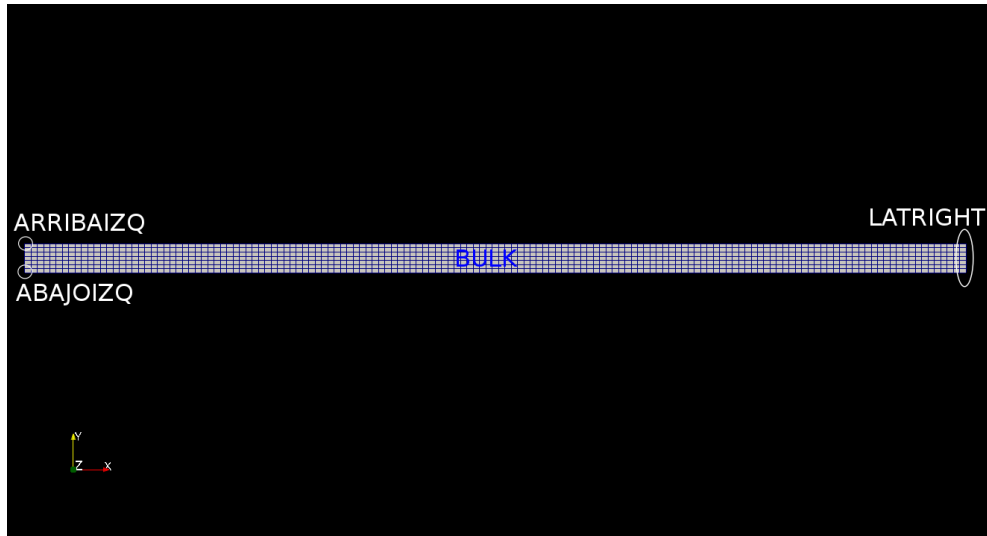
We will solve assuming finite strains and considering an hyperelastic neo-Hookean material as formulated by Zienkiewicz. We will consider 1 second per time step, and 7 time steps for the total simulation.

```
1  case_name DCB
2  mesh_path dcb.msh
3  geometrical_treatment NONLINEAR
4  constitutive_model ZIEN
5  time_step_size 1
6  total_steps 7
```

To the simulation domain *BULK* we associate a material with the following paramenters: Young modulus equal to 6.896E7 N/m^2, Poisson modulus equal to 0.32 and a density of 1 kg/m^3. We will fix the *LATRIGHT* boundary in both directions and to the *ARRIBAIZQ* and *ABAJOIZQ* nodes we will apply the following displacements on x_2 direction (x_1 direction will be free to move):
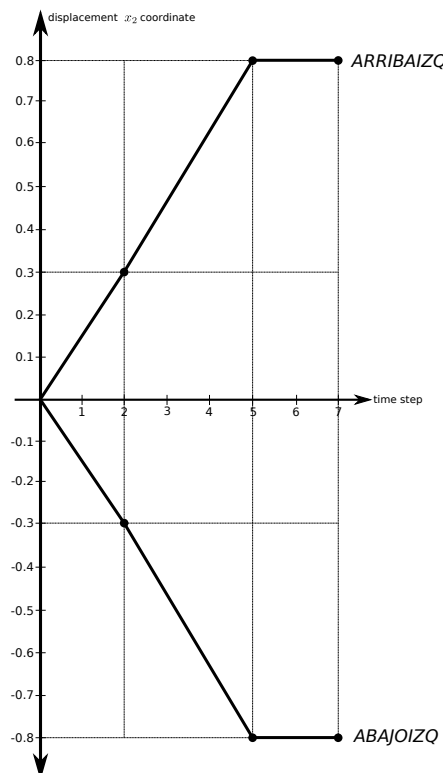
Given the desired Dirichlet boundary conditions, the boundaries file results in (please note how the boundary conditons in *ARRIBAIZQ* and *ABAJOIZQ* are imposed):

```
$VolumeDefinition
BULK 6.896E7 0.32 1
$BoundaryConditionsDisplacement
LATRIGHT 1 1 0.0 0.0 1 7
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ARRIBAIZQ 0 1 0.0 0.3 1 2
ARRIBAIZQ 0 1 0.0 0.8 3 5
ARRIBAIZQ 0 1 0.0 0.8 6 7
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ABAJOIZQ 0 1 0.0 -0.3 1 2
ABAJOIZQ 0 1 0.0 -0.8 3 5
ABAJOIZQ 0 1 0.0 -0.8 6 7
```
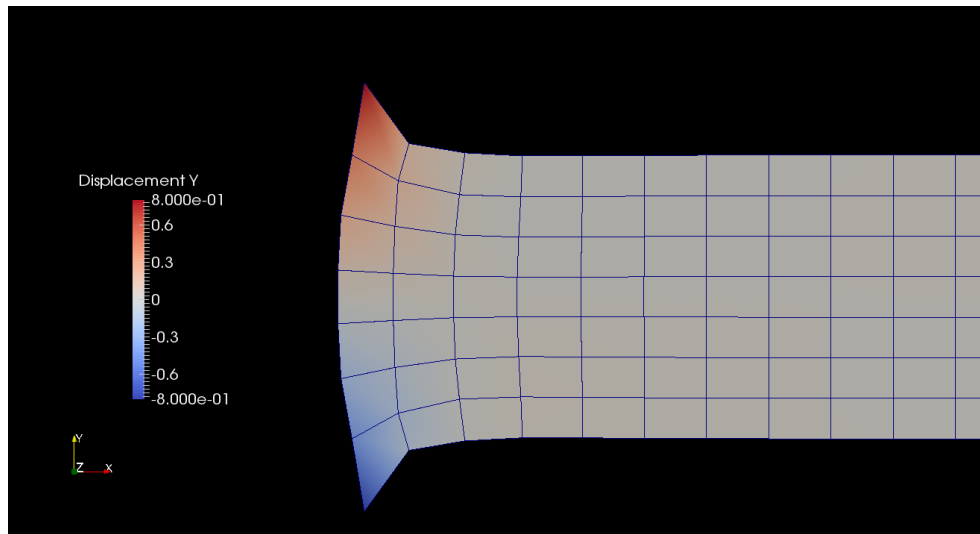
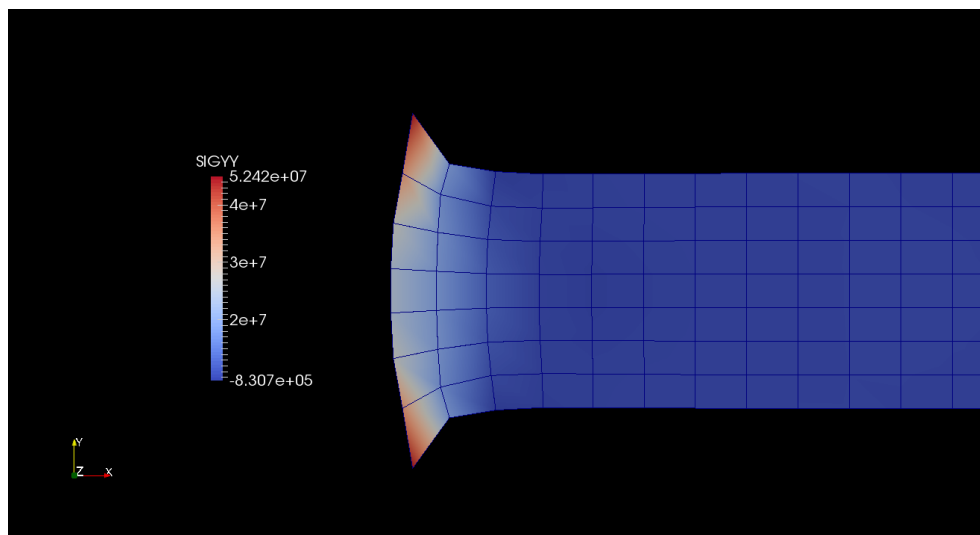Once Ostero executes, we postprocess the results. Firstly, we analyse the mesh deformation:
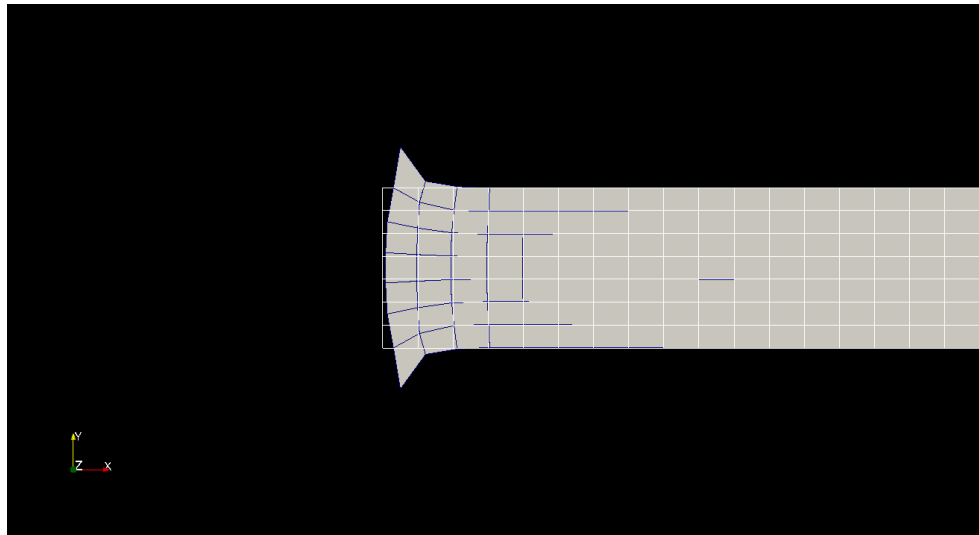
Then, we show the displacements in a zoomed area centered on the left end of the beam:
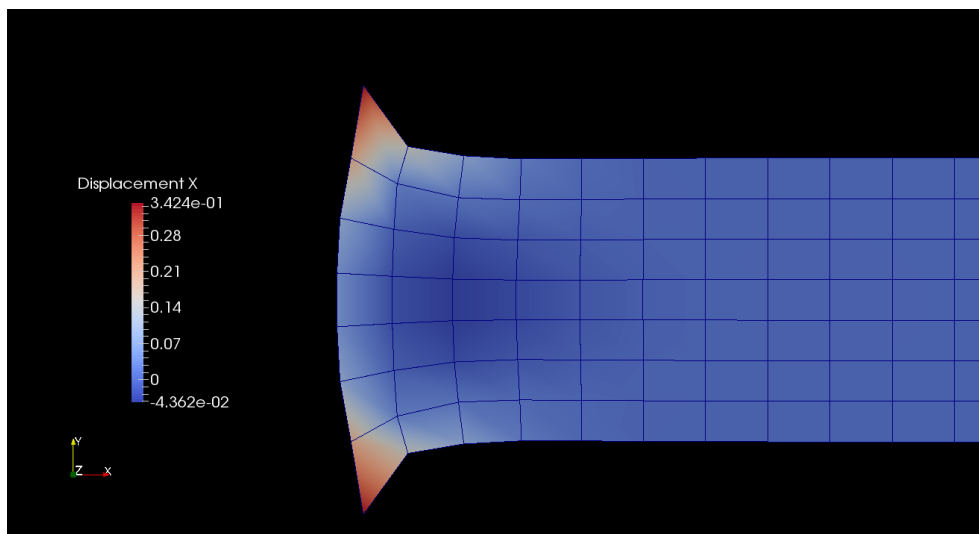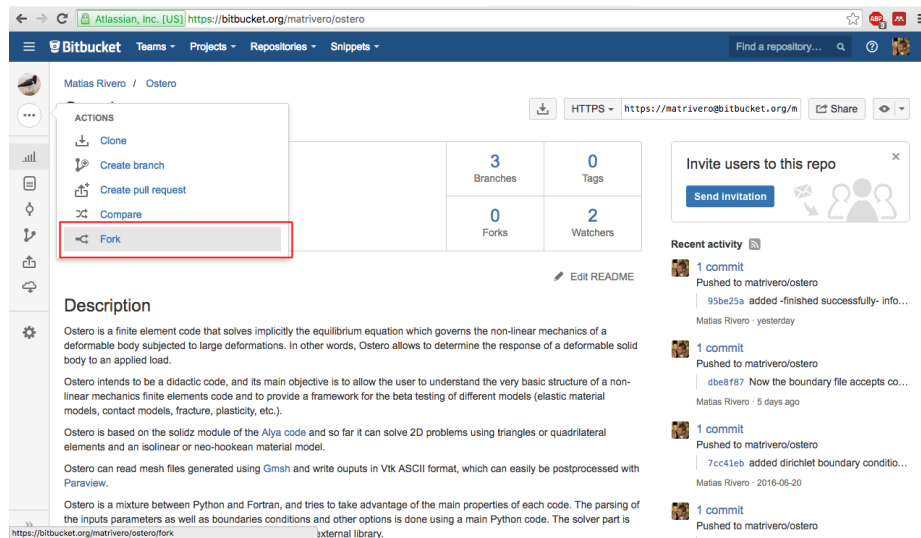
Finally, we display the stress distribution on x_2 direction (σ_{yy}):

5 How to collaborate with Ostero

You will have to create a Bitbucket account and have Git installed on your system.

First, sign in to your Bitbucket account, navigate to the Ostero's home page and fork the repository, by clicking in the Fork button:

This will create a server-side copy of Ostero's repository.

Once you have forked Ostero, clone your forked repository to have a working copy of the project on your local machine. Then create a new branch and start editing the code. Once you have finished your implementations and commits, push the new feature or modification you have implemented to your Bitbucket repository (the forked one). Finally, create a *Pull request* through your Bitbucket account. This will create a request to merge your new feature located in your forked repository to the main codebase of Ostero.

The pull request you sent will be reviewed by the main developer of Ostero and he/she will approve the request, reject it or comment on the request asking for fixes.

To see more in detail how to collaborate with Ostero or any other open source project, see this excellent example from this Git tutorial:

https://www.atlassian.com/git/tutorials/making-a-pull-request/example

References

[1]   Ted Belytschko, Wing Kam Liu, and Brian Moran. *Nonlinear Finite Elements for Continua and Structures*. 1st. Wiley, 2006.

[2]   Olgierd C. Zienkiewicz and Robert Leroy Taylor. *The Finite Element Method: Solid mechanics*. Butterworth-Heinemann, 2000.

[3]   Tod A. Laursen. *Computational Contact and Impact Mechanics*. 1st. Springer, 2003.