SolidsPy Documentation

Release 1.0.16

Juan Gómez & Nicolás Guarín-Zapata

Mar 03, 2023

Contents:

1 SolidsPy: 2D-Finite Element Analysis with Python **1**
- 1.1 Features . . . 2
- 1.2 Installation . . . 2
- 1.3 How to run a simple model . . . 2
- 1.4 License . . . 3
- 1.5 Citation . . . 3

2 SolidsPy's tutorials **5**
- 2.1 Start here: 2×2 square with axial load . . . 5
- 2.2 Creation of a simple geometry using Gmsh . . . 9
- 2.3 Geometry in Gmsh and solution with SolidsPy . . . 10

3 Usage **27**

4 Contributing **29**
- 4.1 Types of Contributions . . . 29
- 4.2 Contributor Guidelines . . . 30

5 Credits **33**
- 5.1 Principal developers . . . 33
- 5.2 Contributions . . . 33

6 SolidsPy reference **35**
- 6.1 solids_GUI: simple interface . . . 35
- 6.2 Assembly routines . . . 36
- 6.3 FEM routines . . . 38
- 6.4 Numeric integration routines . . . 42
- 6.5 Postprocessor subroutines . . . 43
- 6.6 Postprocessor subroutines . . . 45
- 6.7 Solver routines . . . 51
- 6.8 Element subroutines . . . 51

7 History **55**
- 7.1 1.0.15 (2018-05-09) . . . 55
- 7.2 1.0.14 (2018-05-08) . . . 55
- 7.3 1.0.13 (2018-05-07) . . . 55
- 7.4 1.0.12 (2018-04-16) . . . 55

7.5 1.0.0 (2017-07-17) 55

8 Roadmap **57**

9 Indices and tables **59**

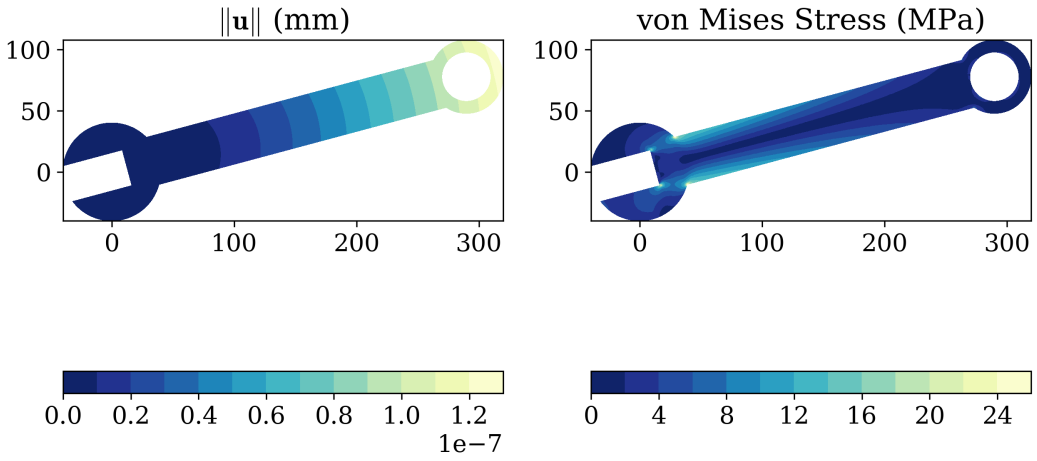
Bibliography **61**

Python Module Index **63**

Index **65**

CHAPTER 1

SolidsPy: 2D-Finite Element Analysis with Python



A simple finite element analysis code for 2D elasticity problems. The code uses as input simple-to-create text files defining a model in terms of nodal, element, material and load data.

- Documentation: http://solidspy.readthedocs.io
- GitHub: https://github.com/AppliedMechanics-EAFIT/SolidsPy
- PyPI: https://pypi.org/project/solidspy/
- Free and open source software: MIT license

1.1 Features

- It is based on an open-source environment.
- It is easy to use.
- The code allows to find displacement, strain and stress solutions for arbitrary two-dimensional domains discretized into finite elements and subject to point loads.
- The code is organized in independent modules for pre-processing, assembly and post-processing allowing the user to easily modify it or add features like new elements or analyses pipelines.
- It was created with academic purposes and is used to teach:
  - IC0285 Computational Modeling (Universidad EAFIT).
  - IC0602 Introduction to the Finite Element Methods (Universidad EAFIT).

1.2 Installation

The code is written in Python and it depends on `numpy`, `scipy` and `sympy`. It has been tested under Windows, Mac, Linux and Android.

To install *SolidsPy* open a terminal and type:

```
pip install solidspy
```

To specify through a GUI the folder where the input files are stored you will need to install easygui.

To easily generate the required SolidsPy text files out of a Gmsh model you will need meshio.

These two can be installed with:

```
pip install easygui
pip install meshio
```

1.3 How to run a simple model

For further explanation check the docs.

Let's suppose that we have a simple model represented by the following files (see tutorials/square example for further explanation).

- `nodes.txt`

```
0  0.00  0.00   0  -1
1  2.00  0.00   0  -1
2  2.00  2.00   0   0
3  0.00  2.00   0   0
4  1.00  0.00  -1  -1
5  2.00  1.00   0   0
6  1.00  2.00   0   0
7  0.00  1.00   0   0
8  1.00  1.00   0   0
```

- `eles.txt`

```
0   1   0   0   4   8   7
1   1   0   4   1   5   8
2   1   0   7   8   6   3
3   1   0   8   5   2   6
```

- mater.txt

```
1.0  0.3
```

- loads.txt

```
3  0.0  1.0
6  0.0  2.0
2  0.0  1.0
```

Run it in Python as follows:

```
import matplotlib.pyplot as plt  # load matplotlib
from solidspy import solids_GUI  # import our package
disp = solids_GUI()  # run the Finite Element Analysis
plt.show()    # plot contours
```

This would not work properly in Anaconda for Mac OS. In that case is suggested to use an IPython console to run the example.

1.4 License

This project is licensed under the MIT license. The documents are licensed under Creative Commons Attribution License.

1.5 Citation

To cite SolidsPy in publications use

> Juan Gómez, Nicolás Guarín-Zapata (2018). SolidsPy: 2D-Finite Element Analysis with Python, <https://github.com/AppliedMechanics-EAFIT/SolidsPy>.

A BibTeX entry for LaTeX users is

```
@software{solidspy,
 title = {SolidsPy: 2D-Finite Element Analysis with Python},
 author = {Gómez, Juan and Guarín-Zapata, Nicolás},
 year = 2018,
 keywords = {Python, Finite elements, Scientific computing, Computational mechanics},
 abstract = {SolidsPy is a simple finite element analysis code for
   2D elasticity problems. The code uses as input simple-to-create text
   files defining a model in terms of nodal, element, material and
   load data.},
 url = {https://github.com/AppliedMechanics-EAFIT/SolidsPy}
}
```

CHAPTER 2

SolidsPy's tutorials

Here we show through simple examples how to conduct analyses using SolidsPy.

First we describe the structure of the text files for the problem of small 2×2 square plate under axial loading. In the second part of the documents we describe the creation of a SolidsPy model with the aid of Gmsh. This is necessary when conducting analysis in large and complex geometries. In this case the Gmsh files need to be converted into text files using subroutines based upon meshio.

2.1 Start here: 2×2 square with axial load

Author Nicolás Guarín-Zapata

Date May, 2017

In this document we briefly describe the use of SolidsPy, through a simple example corresponding to a square plate under point loads.

2.1.1 Input files

The code requires the domain to be input in the form of text files containing the nodes, elements, loads and material information. These files must reside in the same directory and must have the names `eles.txt`, `nodes.txt`, `mater.txt` and `loads.txt`. Assume that we want to find the response of the 2×2 square under unitary vertical point loads shown in the following figure. Where one corner is located at (0,0) and the opposite one at (2,2).

The file `nodes.txt` is composed of the following fields:

- Column 0: Nodal identifier (integer).
- Column 1: x-coordinate (float).
- Column 2: y-coordinate (float).
- Column 3: Boundary condition flag along the x-direction (0 free, -1 restrained).
- Column 4: Boundary condition flag along the y-direction (0 free, -1 restrained).

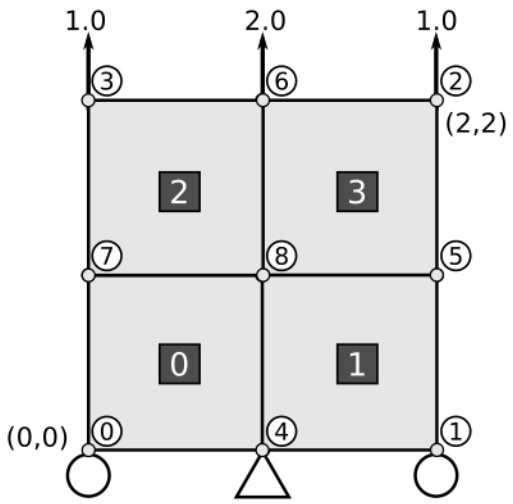


Fig. 1: 4-element solid under point loads.

The corresponding file has the following data

```
0  0.00  0.00   0  -1
1  2.00  0.00   0  -1
2  2.00  2.00   0   0
3  0.00  2.00   0   0
4  1.00  0.00  -1  -1
5  2.00  1.00   0   0
6  1.00  2.00   0   0
7  0.00  1.00   0   0
8  1.00  1.00   0   0
```

The file `eles.txt` contain the element information. Each line in the file defines the information for a single element and is composed of the following fields:

- Column 0: Element identifier (integer).
- Column 1: Element type (integer):
  - 1 for a 4-noded quadrilateral.
  - 2 for a 6-noded triangle.
  - 3 for a 3-noded triangle.
- Column 2: Material profile for the current element (integer).
- Column 3 to end: Element connectivity, this is a list of the nodes conforming each element. The nodes should be listed in counterclockwise orientation.

The corresponding file has the following data

```
0   1   0   0   4   8   7
1   1   0   4   1   5   8
2   1   0   7   8   6   3
3   1   0   8   5   2   6
```

The file `mater.txt` contain the material information. Each line in the file corresponds to a material profile to be assigned to the different elements in the elements file. In this example, there is one material profile. Each line in the file is composed of the following fields:

- Column 0: Young's modulus for the current profile (float).
- Column 1: Poisson's ratio for the current profile (float).

The corresponding file has the following data

```
1.0  0.3
```

The file `loads.txt` contains the point loads information. Each line in the file defines the load information for a single node and is composed of the following fields

- Column 0: Nodal identifier (integer).
- Column 1: Load magnitude for the current node along the x-direction (float).
- Column 2: Load magnitude for the current node along the y-direction (float).

The corresponding file has the following data

```
3  0.0  1.0
6  0.0  2.0
2  0.0  1.0
```

2.1.2 Executing the program

After installing the package, you can run the program in a Python terminal using

```
>>> from solidspy import solids_GUI
>>> solids_GUI()
```

In Linux and you can also run the program from the terminal using

```
$ python -m solidspy
```

If you have `easygui` installed a pop-up window will appear for you to select the folder with the input files

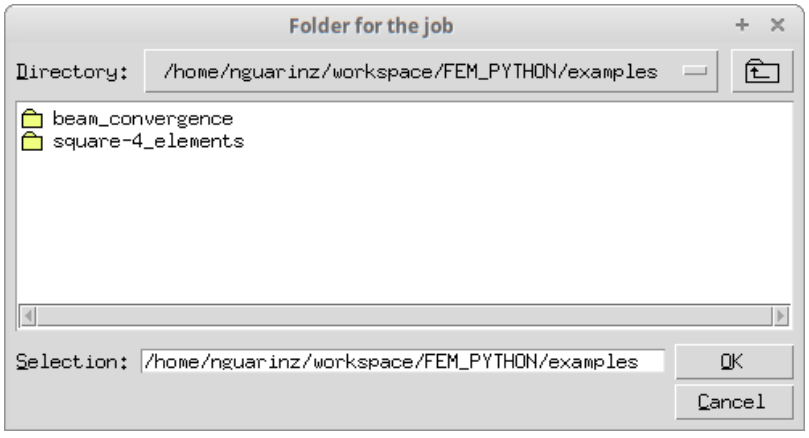


Fig. 2: Folder selection window.

select the folder and click ok.

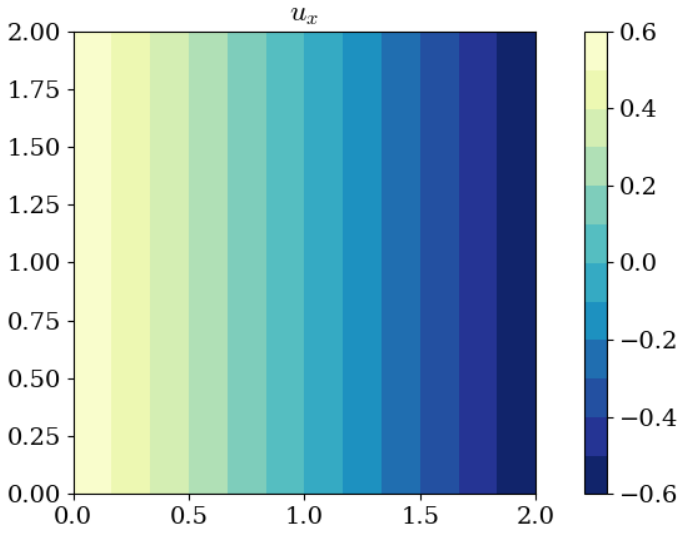
If you don't have `easygui` installed the software will ask you for the path to your folder. The path can be absolute or relative.

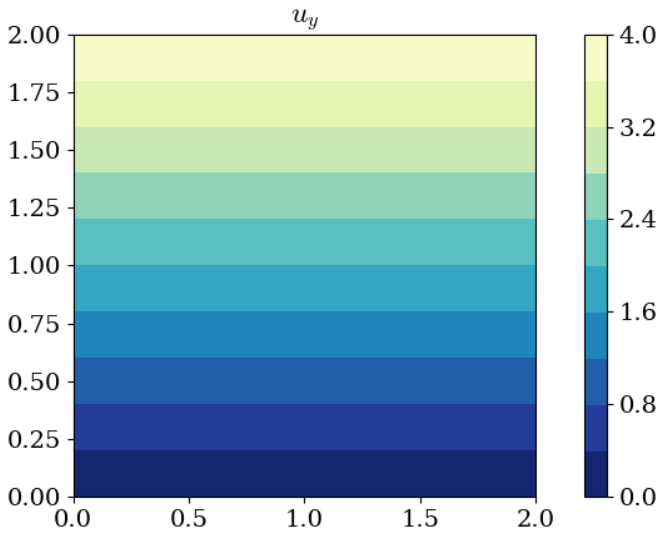
```
Enter folder (empty for the current one):
```

Then, you will see some information regarding your analysis

```
Number of nodes: 9
Number of elements: 4
Number of equations: 14
Duration for system solution: 0:00:00.006895
Duration for post processing: 0:00:01.466066
Analysis terminated successfully!
```

And, once the solution is achieved you will see displacements and stress solutions as contour plots, like the following





Interactive execution

You can also run the program interactively using a Python terminal, a good option is IPython.

In IPython you can run the program with

```
In [1]: from solidspy import solids_GUI

In [2]: UC = solids_GUI()
```

After running the code we have the nodal variables for post-processing. For example, we can print the displacement vector

```
In [3]: np.set_printoptions(threshold=np.nan)

In [4]: print(np.round(UC, 3))
[ 0.6 -0.6 -0.6  4.   0.6  4.  -0.6  2.  -0.   4.   0.6  2.  -0.   2. ]
```

where we first setup the printing option for IPython to show the full array and then rounded the array to 3 decimal places.

```
In [5]: U_mag = np.sqrt(UC[0::2]**2 + UG[1::2]**2)

In [6]: print(np.round(U_mag, 3))
[ 0.849  4.045  4.045  2.088  4.     2.088  2.   ]
```

2.2 Creation of a simple geometry using Gmsh

Author Juan Gómez

Date October, 2017

We want to create a mesh for the following geometry

The `.geo` file for this model is the following

```
// Parameters
L = 2.0;
H = 1.0;
h_1 = H/2;
h_2 = H/2;
lado_elem = 0.1;


// Points
Point(1) = {0.0, 0.0, 0, lado_elem};
Point(2) = {L, 0.0, 0, lado_elem};
Point(3) = {L, h_2, 0, lado_elem};
Point(4) = {L, H, 0, lado_elem};
Point(5) = {0, H, 0, lado_elem};
Point(6) = {0, h_2, 0, lado_elem};

// Lines
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
```

(continues on next page)

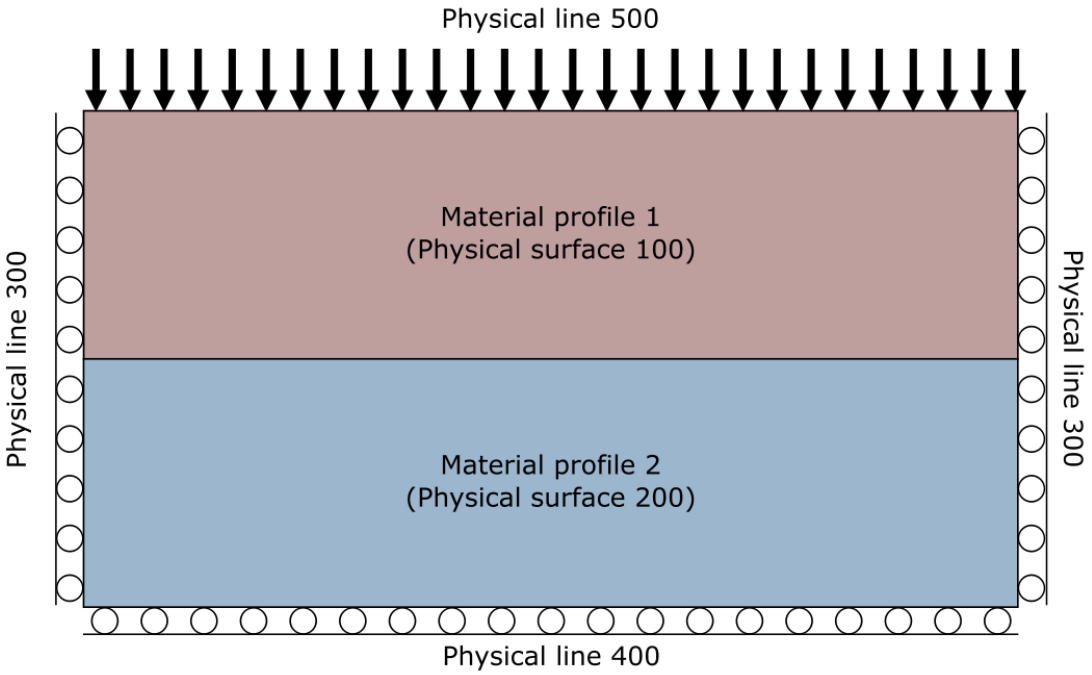


Fig. 3: Bilayer model.

(continued from previous page)

```
Line(4) = {4, 5};
Line(5) = {5, 6};
Line(6) = {6, 1};
Line(7) = {3, 6};

// Surfaces
Line Loop(8) = {1, 2, 7, 6};
Plane Surface(9) = {8};
Line Loop(10) = {-7, 5, 4, 3};
Plane Surface(11) = {10};

// Physical groups
Physical Surface(100) = {9};  // Material superior
Physical Surface(200) = {11};  // Material inferior
Physical Line(300) = {5, 6, 3, 2};  // Lineas laterales
Physical Line(400) = {1};  // Linea inferior
Physical Line(500) = {4};  // Linea superior (cargas)
```

2.3 Geometry in Gmsh and solution with SolidsPy

Author Nicolás Guarín-Zapata

Date April, 208

This document is a tutorial on how to generate a (specific) geometry using Gmsh [Gmsh2009] and its subsequent

processing for the generation of input files for a Finite Element program in Python. This document does not pretend to be an introduction to the management of Gmsh, for this we suggest the official tutorial [Gmsh_tut] (for text mode) or the official screencasts [Gmsh_scr] (for the graphical interface).

2.3.1 Model

The example to be solved corresponds to the determination of Efforts in a cylinder in the *Brazilian Test*. The Brazilian Test is a technique that is used for the indirect measurement of the resistance of rocks It is a simple and effective technique, and therefore it is commonly used for rock measurements. Sometimes this test is also used for concrete [D3967-16].

The following figure presents a scheme of the model to solve. Since the original model may present rigid body movements, it decides to use the symmetry of the problem. Then, the problem to solve is a quarter of the original problem and the surfaces lower e left present restrictions of *roller*.

Fig. 4: Schematic of the problem to be solved.

2.3.2 Creation of the geometry and mesh in Gmsh

As a first step, it is suggested to create a new file in Gmsh, as It shows in the following figure.

When creating a new document it is possible[1] for Gmsh to ask about which geometry kernel to use. We will not dwell on what the differences are and we will use `built-in`.

To create a model, we initially create the points. For that, let's go to the option: `Geometry> Elementary Entities> Add> Point`, as shown in the following figure. Then, the coordinates of the points in the pop-up window and "Add". Finally we can close the pop-up window and press `e`.

Later we create lines. For this, we go to the option: '' Geometry> Elementary Entities> Add> Straight line'', as shown in the following figure, and we select the initial points and endings for each line. At the end, we can press `e`.

We also create the circle arcs. For this, we go to the option: `Geometry> Elementary Entities> Add> Circle Arc`, as shown in the following figure, and we select the initial points, central and final for each arc (in that order). At the end, we can press `e`.

Since we already have a closed contour, we can define a surface. For this, we go to the option: `Geometry> Elementary Entities> Add> Plane Surface`, as shown in the following figure, and we select the contours in order. At the end, we can press '' e''.

Now, we need to define *physical groups*. Physical groups allow us to associate names to different parts of the model such as lines and surfaces. This will allow us to define the region in which we will resolve the model (and we will associate a material), the regions that have restricted movements (boundary conditions) and the regions on which we will apply the load. In our case we will have 4 groups physical:

- Region of the model, where we will define a material;
- Bottom edge, where we will restrict the displacement in y;
- Left edge, where we will restrict the displacement in x; and
- Top point, where we will apply the point load.

To define the physical groups we are going to `Geometry> Physical groups> Add> Plane Surface`, as shown in the next figure. In this case, we can leave the field of '' Name'' empty and allow Gmsh to name the groups for us, which will be numbers that we can then consult in the text file

[1] If the version is 3.0 or higher, this pop-up window will appear.

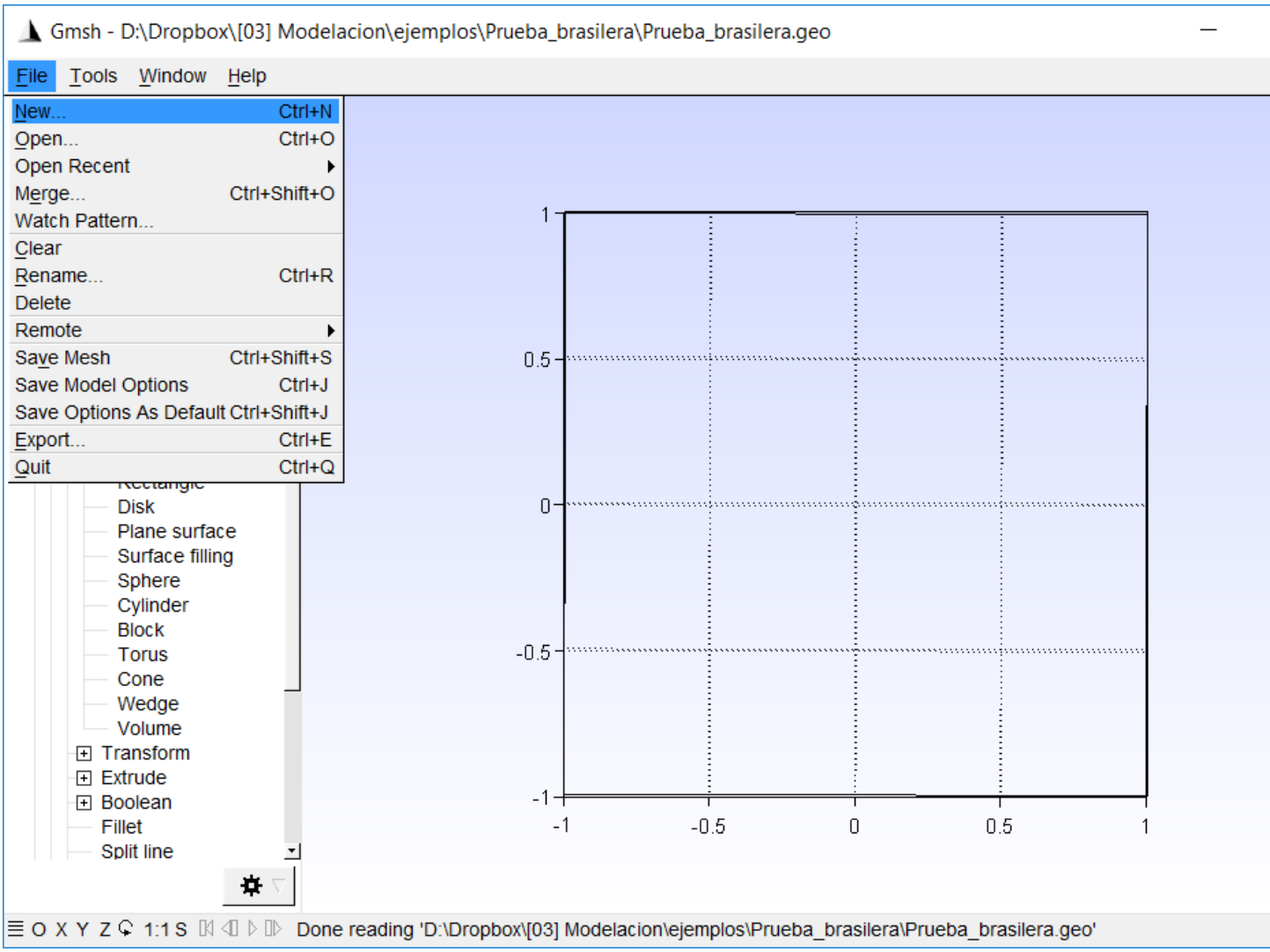


Fig. 5: Creation of a new file in Gmsh.

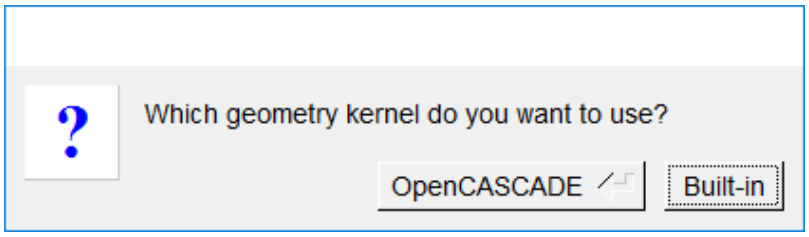


Fig. 6: Pop-up window asking for the geometry kernel.

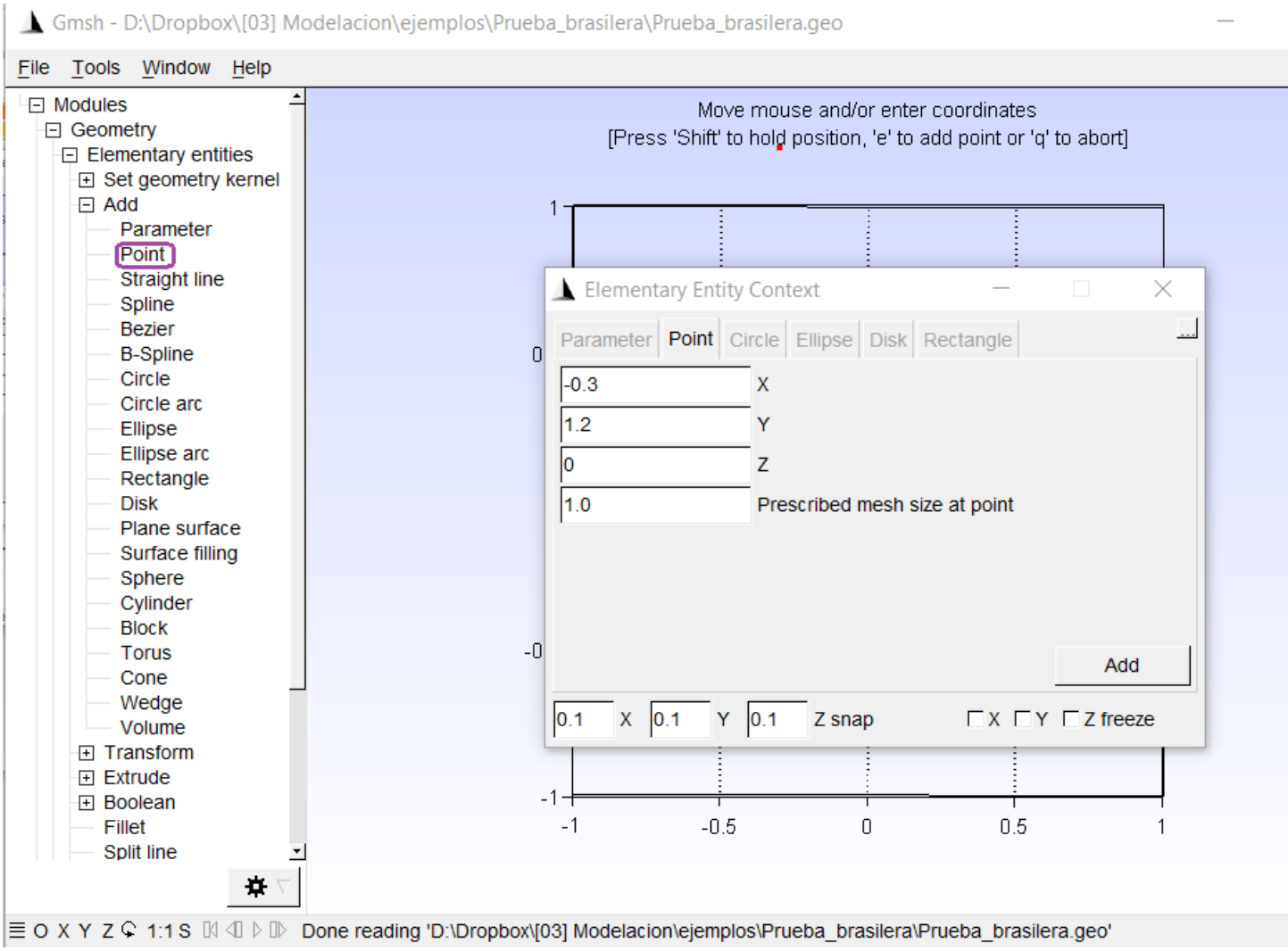


Fig. 7: Agregar puntos al modelo.

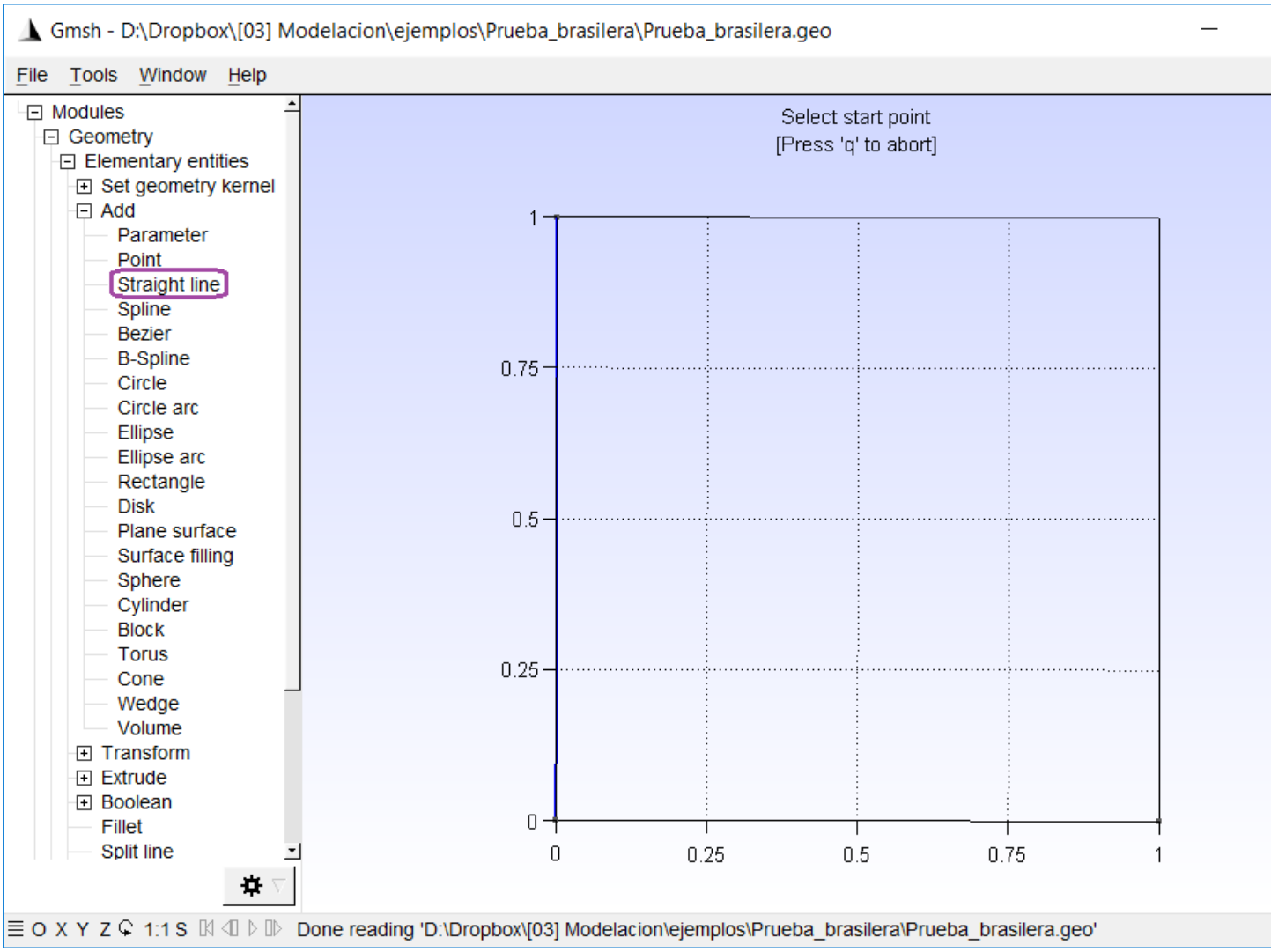


Fig. 8: Add straight lines to the model.

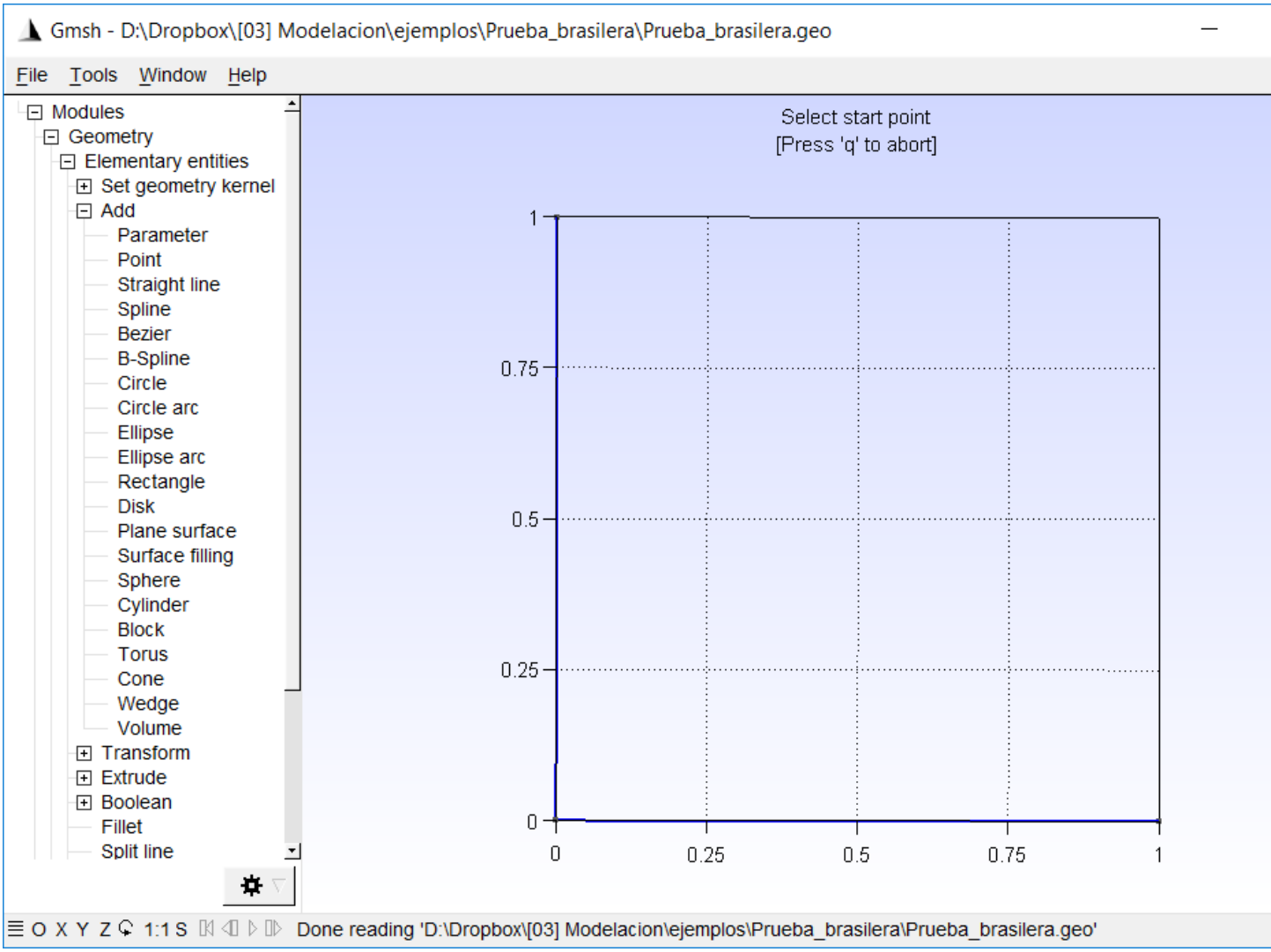


Fig. 9: Add arcs to the model.

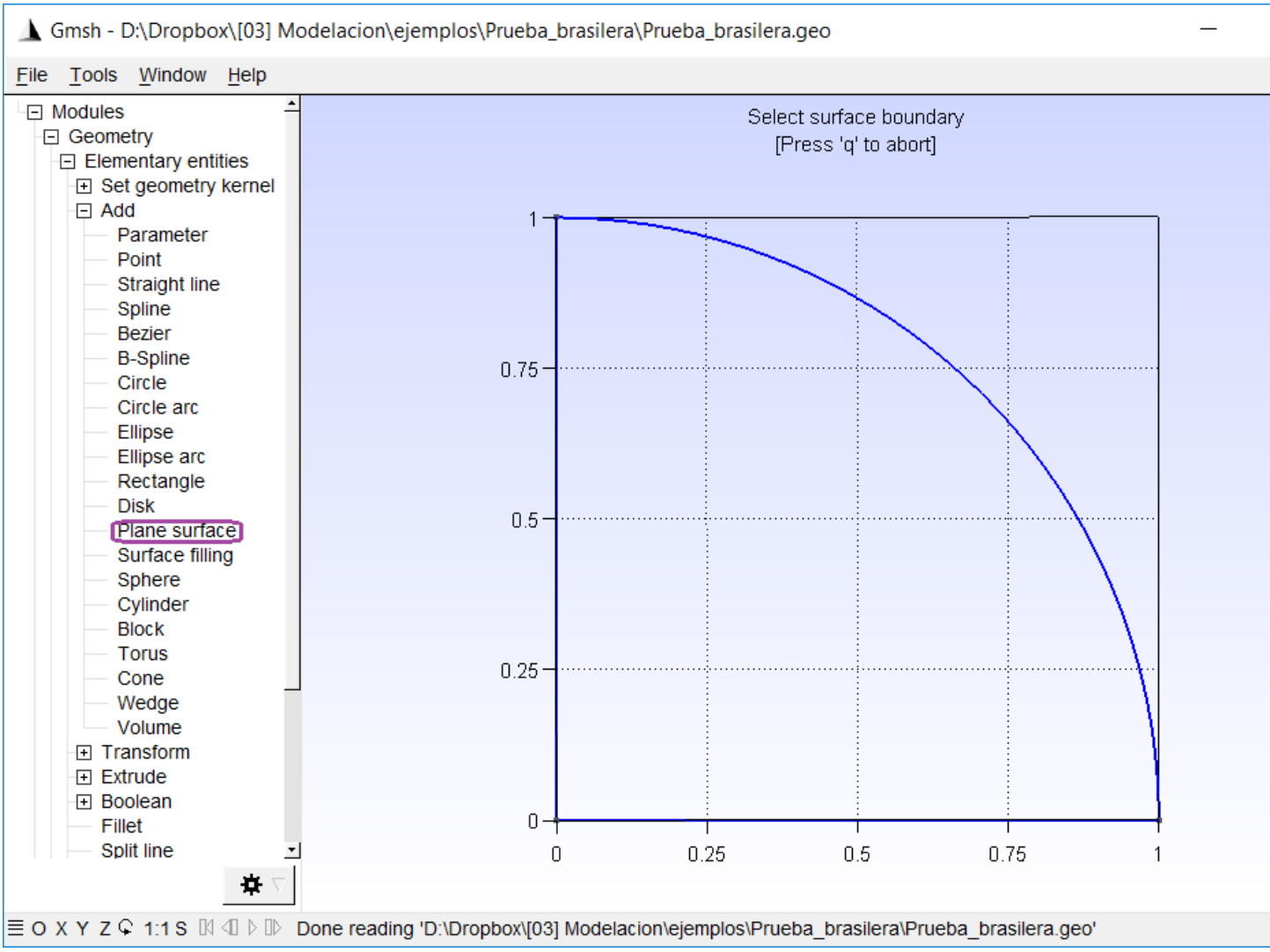


Fig. 10: Add surfaces.

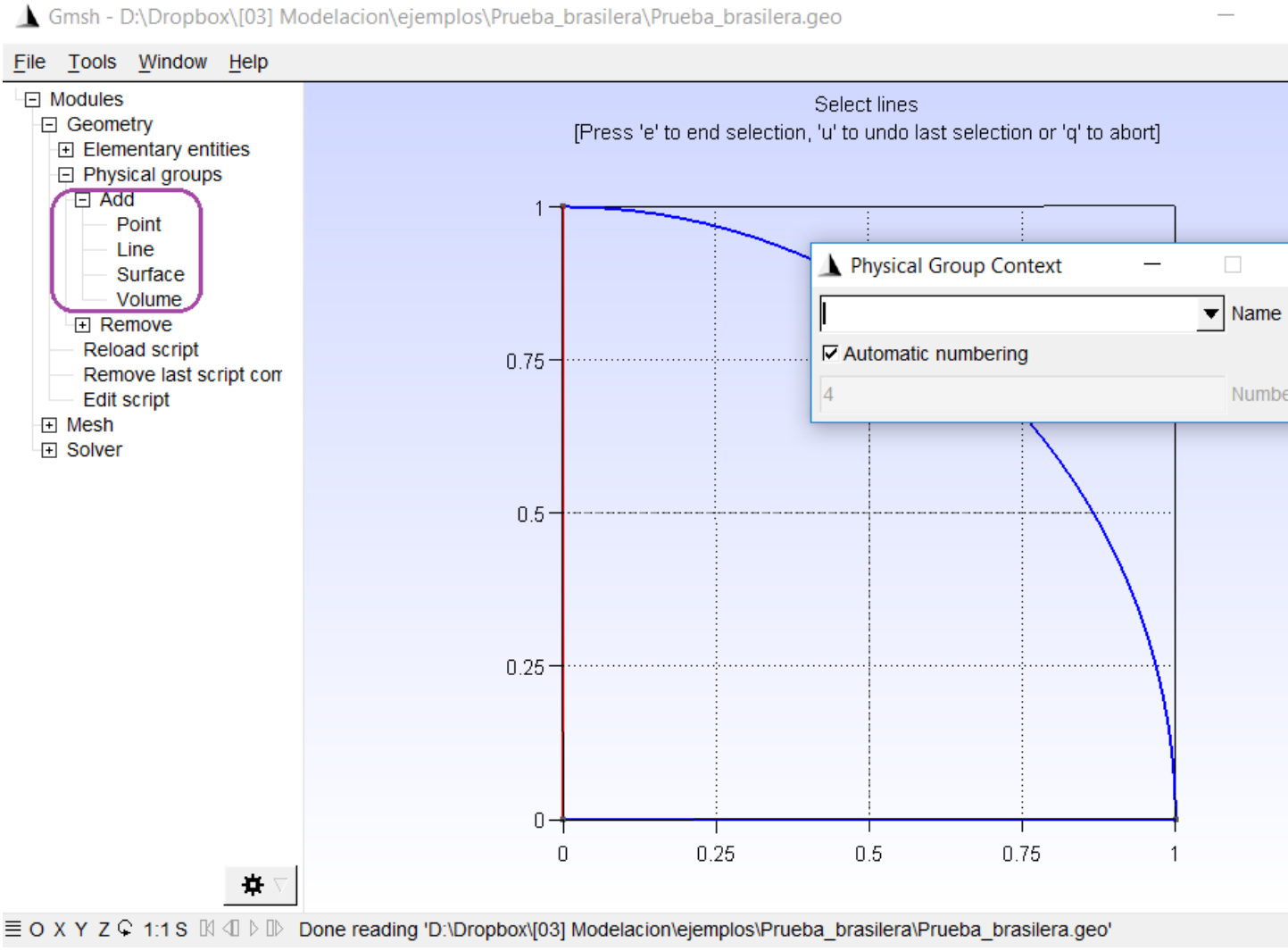


Fig. 11: Add physical groups.

After (slightly) editing the text file (`.geo`) this looks like this

```
L = 0.1;

// Points
Point(1) = {0, 0, 0, L};
Point(2) = {1, 0, 0, L};
Point(3) = {0, 1, 0, L};

// Lines
Line(1) = {3, 1};
Line(2) = {1, 2};
Circle(3) = {2, 1, 3};

// Surfaces
Line Loop(1) = {2, 3, 1};
Plane Surface(1) = {1};

// Physical groups
Physical Line(1) = {1};
Physical Line(2) = {2};
Physical Point(3) = {3};
Physical Surface(4) = {1};
```

We added a parameter `L`, which we can vary to to change the size of the elements when creating the mesh.

Now, we proceed to create the mesh. To do this, we go to `Mesh> 2D`. As we see in the figure below.

Additionally, we can change the configuration so that it shows the elements of the mesh in colors. For this, we are going to `Tools> Options> Mesh` and mark the box that indicates `Surface faces`.

We can then refine the mesh going to `Mesh> Refine by Splitting`, or by modifying the `L` parameter in the input file (.geo). As a last step, we want to save the mesh. To do this, go to `Mesh> Save`, or `File> Save Mesh`, as shows below.

2.3.3 Python script to generate input files

We need to create files with the information of the nodes (`nodes.txt`), elements (`eles.txt`), loads (`loads.txt`) and materials (`mater.txt`).

The following code generates the necessary input files for Run the finite element program in Python.

```
from __future__ import division, print_function
import meshio
import numpy as np


points, cells, point_data, cell_data, field_data = \
    meshio.read("Prueba_brasilera.msh")

# Element data
eles = cells["triangle"]
els_array = np.zeros([eles.shape[0], 6], dtype=int)
els_array[:, 0] = range(eles.shape[0])
els_array[:, 1] = 3
els_array[:, 3::] = eles

# Nodes
```

(continues on next page)

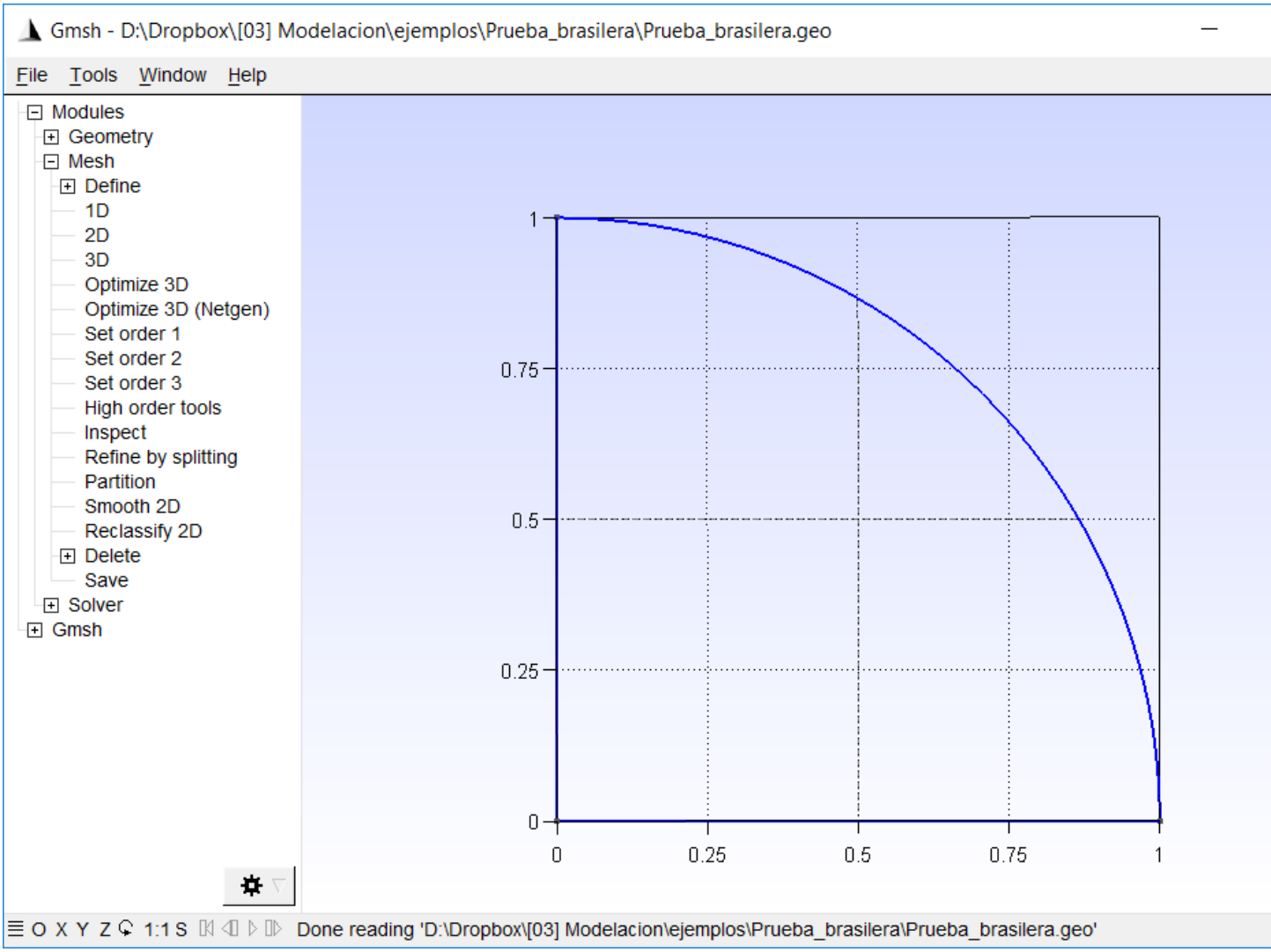


Fig. 12: Create the mesh.

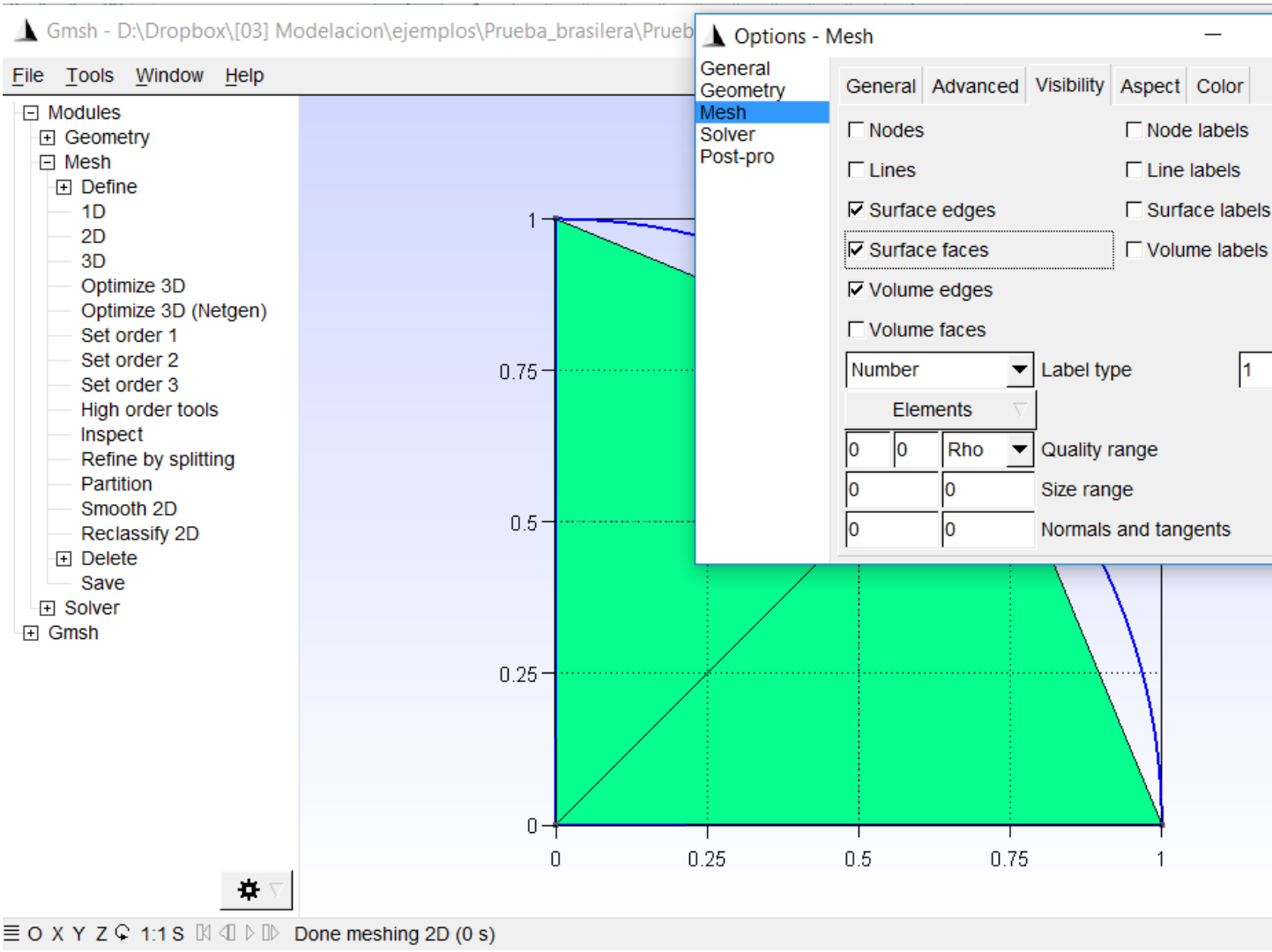


Fig. 13: Create the mesh.

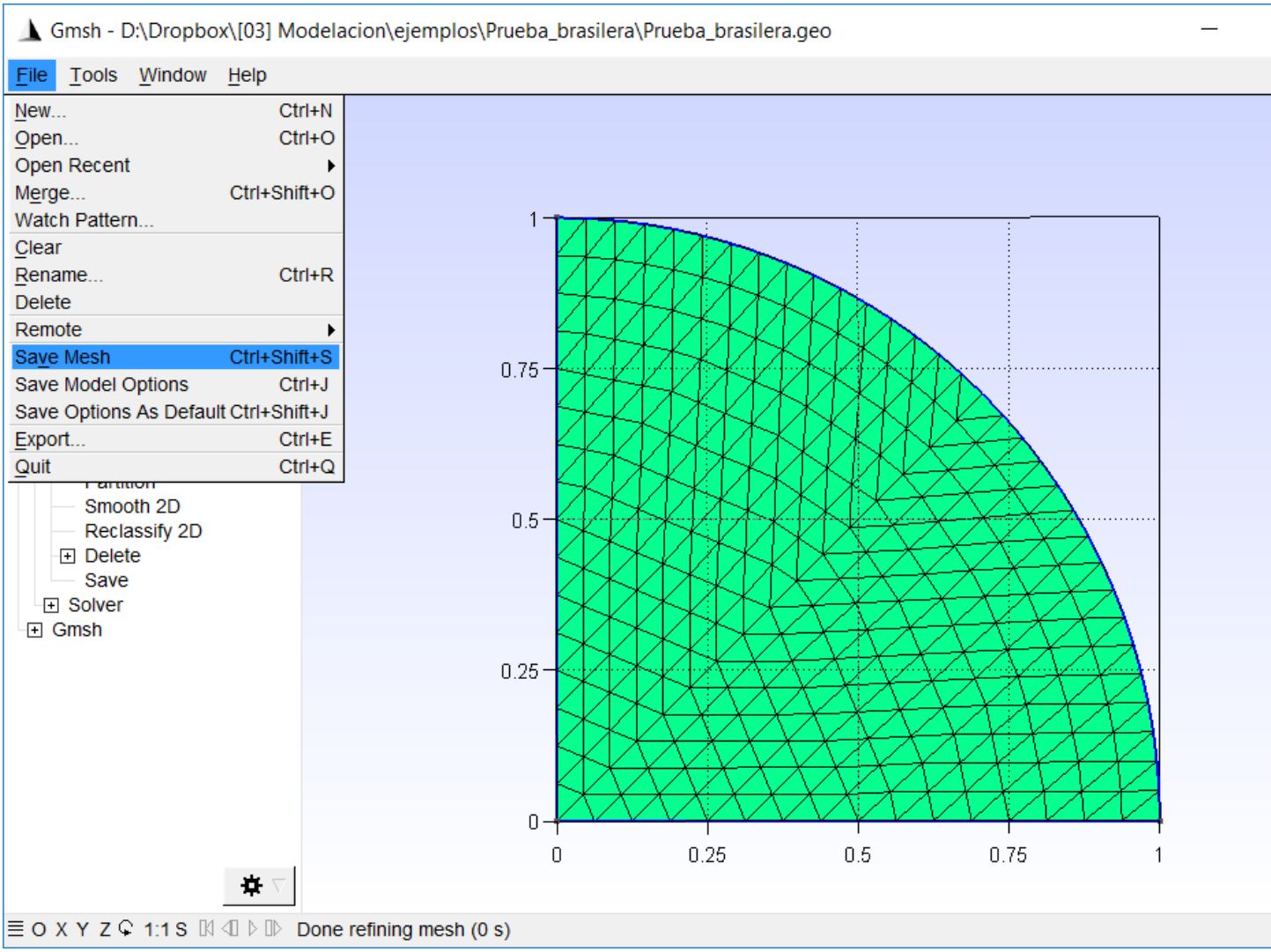


Fig. 14: Save the `.msh` file.

(continued from previous page)

```
nodes_array = np.zeros([points.shape[0], 5])
nodes_array[:, 0] = range(points.shape[0])
nodes_array[:, 1:3] = points[:, :2]

# Boundaries
lines = cells["line"]
bounds = cell_data["line"]["gmsh:physical"]
nbounds = len(bounds)

# Loads
id_cargas = cells["vertex"]
nloads = len(id_cargas)
load = -10e8 # N/m
loads_array = np.zeros((nloads, 3))
loads_array[:, 0] = id_cargas
loads_array[:, 1] = 0
loads_array[:, 2] = load

# Boundary conditions
id_izq = [cont for cont in range(nbounds) if bounds[cont] == 1]
id_inf = [cont for cont in range(nbounds) if bounds[cont] == 2]
nodes_izq = lines[id_izq]
nodes_izq = nodes_izq.flatten()
nodes_inf = lines[id_inf]
nodes_inf = nodes_inf.flatten()
nodes_array[nodes_izq, 3] = -1
nodes_array[nodes_inf, 4] = -1

#  Materials
mater_array = np.array([[70e9, 0.35],
                        [70e9, 0.35]])
maters = cell_data["triangle"]["gmsh:physical"]
els_array[:, 2]  = [1 for mater in maters if mater == 4]

# Create files
np.savetxt("eles.txt", els_array, fmt="%d")
np.savetxt("nodes.txt", nodes_array,
           fmt=("%d", "%.4f", "%.4f", "%d", "%d"))
np.savetxt("loads.txt", loads_array, fmt=("%d", "%.6f", "%.6f"))
np.savetxt("mater.txt", mater_array, fmt="%.6f")
```

Now, let's discuss the different parts of the code to see what it does each.

Header and reading the `.msh` file

The first part loads the necessary Python modules and reads the file of mesh that in this case is called `Prueba_brasilera.msh` (line 6 and 7). In order for Python to be able to read the file, it must be in the same directory as the Python file that will process it.

```
from __future__ import division, print_function
import meshio
import numpy as np


points, cells, point_data, cell_data, field_data = \
  meshio.read("Prueba_brasilera.msh")
```

Element data

The next section of the code creates the data for elements. The line 18 creates a variable ‘‘ eles‘‘ with the information of the nodes that make up each triangle. Line 11 creates an array (filled with zeros) with the number of rows equal to the number of elements (`eles.shape[0]`) and 6 columns[2]. Then we assign a number to each element, what we do on line 12 with `range(eles.shape[0])` and this we assign to column 0. All elements are triangles, that's why we should put 3 in column 1. Last, in this section, we assign the nodes of each element to the array with (line 19), and this assignment is made from column 3 to final with `els_array[:, 3::]`.

```
# Element data
eles = cells["triangle"]
els_array = np.zeros([eles.shape[0], 6], dtype=int)
els_array[:, 0] = range(eles.shape[0])
els_array[:, 1] = 3
els_array[:, 3::] = eles
```

Nodes data

In the next section we create the information related to the nodes. To do this, on line 17 we created an array `nodes_array` with 5 columns and as many rows as there are points in the model (`points.shape[0]`). Then, we assign the element type on line 18. And finally, we assign the information on the coordinates of the nodes on line 19 with `nodes_array[:, 1:3] = points[:, :2]`, where we are adding the information in columns 1 and 2.

```
# Nodes
nodes_array = np.zeros([points.shape[0], 5])
nodes_array[:, 0] = range(points.shape[0])
nodes_array[:, 1:3] = points[:, :2]
```

Boundary data

In the next section we find the line information. For this, we read the `cells` information in position `line`[3] (line 22). The array `lines` will then have the information of the nodes that form each line that is on the border of the model. Then, we read the information of the physical lines (line 23), and we calculate how many lines belong to the physical lines (line 24).

```
# Boundaries
lines = cells["line"]
bounds = cell_data["line"]["gmsh:physical"]
nbounds = len(bounds)
```

Load data

In the next section we must define the information of loads. In this case, the loads are assigned in a single point that we define as a physical group. On line 27 we read the nodes (in this case, one). Then, we create an array that has as many rows as loads (`nloads`) and 3 columns Assign the number of the node to which each load belongs (line 31), the charges in :math: x (line 32) and the loads in y and (line 33)

[2] For quadrilateral elements, 7 columns would be used, since each Element is defined by 4 nodes.
[3] `cells` is a dictionary and allows to store information associated with some keywords, in this case it is `lines`.

```
# Loads
id_cargas = cells["vertex"]
nloads = len(id_cargas)
load = -10e8 # N/m
loads_array = np.zeros((nloads, 3))
loads_array[:, 0] = id_cargas
loads_array[:, 1] = 0
loads_array[:, 2] = load
```

Boundary conditions

Now, we will proceed to apply the boundary conditions, that is, the model regions that have restrictions on displacements. Initially, we identify which lines have an identifier 1 (which would be the left side) with

```
id_izq = [cont for cont in range(nbounds) if bounds[cont] == 1]
```

This creates a list with the numbers (`cont`) for which the condition (`bounds[cont] == 1`). On line 46 we get the nodes that belong to these lines, however, this array has as many rows as lines on the left side and two columns. First we return this array as a one-dimensional array with `nodes_izq.flatten()`. Later, on line 42, we assign the value of -1 in the third column of the array for nodes that belong to the left side. In the same way, this process is repeated for the nodes at the bottom line.

```
# Boundary conditions
id_izq = [cont for cont in range(nbounds) if bounds[cont] == 1]
id_inf = [cont for cont in range(nbounds) if bounds[cont] == 2]
nodes_izq = lines[id_izq]
nodes_izq = nodes_izq.flatten()
nodes_inf = lines[id_inf]
nodes_inf = nodes_inf.flatten()
nodes_array[nodes_izq, 3] = -1
nodes_array[nodes_inf, 4] = -1
```

Materials

In the next section we assign the corresponding materials to each element. In this case, we only have one material. However, it present the example as if there were two different ones. First, we created a array with the material information where the first column represents the Young's module and the second the Poisson's relation (line 46). Then, we read the information of the physical groups of surfaces on line 48. Finally, we assign the value of 0 to the materials that have as physical group 4 (see file `.geo` above) and 1 to the others, which in this case will be zero (line 49). This information goes in the column 2 of the arrangement.

```
#  Materials
mater_array = np.array([[70e9, 0.35],
                        [70e9, 0.35]])
maters = cell_data["triangle"]["gmsh:physical"]
els_array[:, 2]  = [1 for mater in maters if mater == 4]
```

Export files

The last section uses the `numpy` function to export the files.

```
# Create files
np.savetxt("eles.txt", els_array, fmt="%d")
np.savetxt("nodes.txt", nodes_array,
           fmt=("%d", "%.4f", "%.4f", "%d", "%d"))
np.savetxt("loads.txt", loads_array, fmt=("%d", "%.6f", "%.6f"))
np.savetxt("mater.txt", mater_array, fmt="%.6f")
```

2.3.4 Solution using SolidsPy

To solve the model, we can type[4]

```
from solidspy import solids_GUI
disp = solids_GUI()
```

After running this program it will appear a pop-up window as shown below. In this window the directory we should locate the folder with the input files generated previously. Keep in mind that the appearance of this window may vary between operating systems. Also, we have notef that sometimes the pop-up window may be hidden by other windows on your desktop.

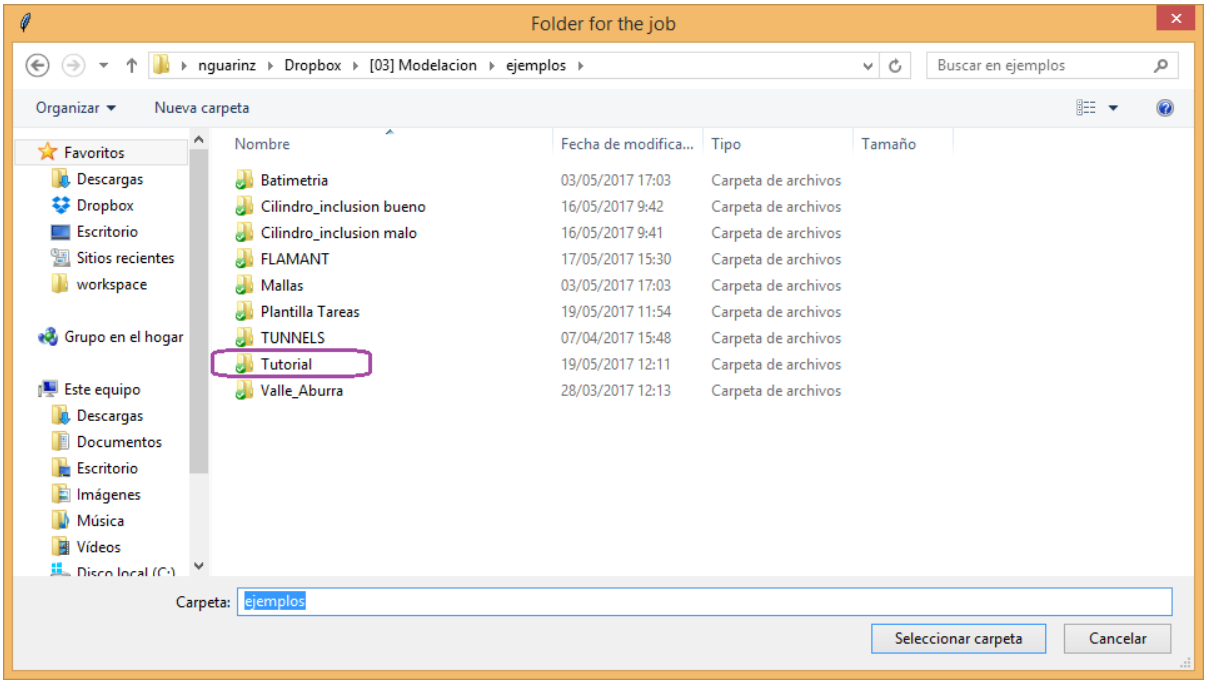


Fig. 15: Pop-up window to locate folder with input files.

At this point, the program must solve the model. If the input files are used without modifications the program should print a message similar to the following.

```
Number of nodes: 123
Number of elements: 208
Number of equations: 224
Duration for system solution: 0:00:00.086983
Duration for post processing: 0:00:00
Analysis terminated successfully!
```

[4] To make use of the graphical interface it must be installed `easygui`.

the times taken to solve the system can change a bit from one computer to another.

As a last step, the program generates graphics with the fields of displacements, deformations and stresses, as shown in the next figures.

Fig. 16: Horizontal displacement.

Fig. 17: Vertical displacement.

2.3.5 References

CHAPTER 3

Usage

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Types of Contributions

You can contribute in many ways:

4.1.1 Create FEM Analysis

If you run a Finite Element Analysis using SolidsPy, and want to share it with the community, submit a pull request to our sibling project SolidsPy-meshes.

4.1.2 Report Bugs

Report bugs at https://github.com/AppliedMechanics-EAFIT/SolidsPy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- If you can, provide detailed steps to reproduce the bug.
- If you don't have steps to reproduce the bug, just note your observations in as much detail as you can. Questions to start a discussion about the issue are welcome.

4.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "please-help" is open to whoever wants to implement it.

Please do not combine multiple feature enhancements into a single pull request.

4.1.5 Write Documentation

SolidsPy could always use more documentation, whether as part of the official SolidsPy docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/AppliedMechanics-EAFIT/SolidsPy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Contributor Guidelines

4.2.1 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4, 3.5, 3.6.

4.2.2 Coding Standards

- PEP8
- Functions over classes except in tests
- Quotes via http://stackoverflow.com/a/56190/5549
  - Use double quotes around strings that are used for interpolation or that are natural language messages
  - Use single quotes for small symbol-like strings (but break the rules if the strings contain quotes)
  - Use triple double quotes for docstrings and raw string literals for regular expressions even if they aren't needed.
  - Example:

```
LIGHT_MESSAGES = {
    'English': "There are %(number_of_lights)s lights.",
    'Pirate':  "Arr! Thar be %(number_of_lights)s lights."
}

def lights_message(language, number_of_lights):
    """Return a language-appropriate string reporting the light count."""
    return LIGHT_MESSAGES[language] % locals()

def is_pirate(message):
    """Return True if the given message sounds piratical."""
    return re.search(r"(?i)(arr|avast|yohoho)!", message) is not None
```

- Write new code in Python 3.

CHAPTER 5

Credits

5.1 Principal developers

- Nicolas Guarin-Zapata (@*nicoguaro*)
- Juan Gómez (@*jgomezc1*)

5.2 Contributions

- Edward Villegas Pulgarin (@*cosmoscalibur*)
- Guillaume Huet (@*guillaumehuet*)

CHAPTER 6

SolidsPy reference

SolidsPy have the following modules:

- `solids_GUI.py`: The main program;
- `preprocesor.py`: Pre-processing subroutines including Gmsh convertion functions using meshio
- `assemutil.py`: Assembly of elemental stiffness matrices ;
- `femutil.py`: Shape functions, its derivatives and general finite element method subroutines;
- `uelutil.py`: Elemental or local matrix subroutines for different elements; and
- `postprocesor.py`: Several results handling subroutines.

6.1 solids_GUI: simple interface

Computes the displacement solution for a finite element assembly of 2D solids under point loads using as input easy-to-create text files containing element, nodal, materials and loads data. The input files are created out of a Gmsh (.msh) generated file using the Python module `meshio`.

Created by Juan Gomez and Nicolas Guarin-Zapata as part of the courses:

- IC0283 Computational Modeling
- IC0602 Introduction to the Finite Element Method

Which are part of the Civil Engineering Department at Universidad EAFIT.

`solids_GUI.`**`solids_GUI`**(*plot_contours=True*, *compute_strains=False*, *folder=None*)
Run a complete workflow for a Finite Element Analysis

Parameters

- **`plot_contours`** (*`Bool (optional)`*) – Boolean variable to plot contours of the computed variables. By default it is True.

- **compute_strains** (*Bool (optional)*) – Boolean variable to compute Strains and Stresses at nodes. By default it is False.
- **folder** (*string (optional)*) – String with the path to the input files. If not provided it would ask for it in a pop-up window.

Returns

- **UC** (*ndarray (nnodes, 2)*) – Displacements at nodes.
- **E_nodes** (*ndarray (nnodes, 3), optional*) – Strains at nodes. It is returned when *compute_strains* is True.
- **S_nodes** (*ndarray (nnodes, 3), optional*) – Stresses at nodes. It is returned when *compute_strains* is True.

6.2 Assembly routines

Functions to assemble the system of equations for the Finite Element Analysis.

assemutil.**DME**(*nodes*, *elements*)
Counts active equations, creates BCs array IBC[] and the assembly operator DME[]

Parameters

- **nodes** (*ndarray.*) – Array with the nodal numbers and coordinates.
- **elements** (*ndarray*) – Array with the number for the nodes in each element.

Returns

- **DME** (*ndarray (int)*) – Assembly operator.
- **IBC** (*ndarray (int)*) – Boundary conditions array.
- **neq** (*int*) – Number of active equations in the system.

assemutil.**assembler**(*elements*, *mats*, *nodes*, *neq*, *DME*, *sparse=True*, *uel=None*)
Assembles the global stiffness matrix

Parameters

- **elements** (*ndarray (int)*) – Array with the number for the nodes in each element.
- **mats** (*ndarray (float)*) – Array with the material profiles.
- **nodes** (*ndarray (float)*) – Array with the nodal numbers and coordinates.
- **DME** (*ndarray (int)*) – Assembly operator.
- **neq** (*int*) – Number of active equations in the system.
- **sparse** (*boolean (optional)*) – Boolean variable to pick sparse assembler. It is True by default.
- **uel** (*callable function (optional)*) – Python function that returns the local stiffness matrix.

Returns **KG** – Array with the global stiffness matrix. It might be dense or sparse, depending on the value of _sparse_

Return type ndarray (float)

assemutil.**dense_assem**(*elements*, *mats*, *nodes*, *neq*, *DME*, *uel=None*)
Assembles the global stiffness matrix _KG_ using a dense storing scheme

Parameters

- **elements** (*ndarray (int)*) – Array with the number for the nodes in each element.
- **mats** (*ndarray (float)*) – Array with the material profiles.
- **nodes** (*ndarray (float)*) – Array with the nodal numbers and coordinates.
- **DME** (*ndarray (int)*) – Assembly operator.
- **neq** (*int*) – Number of active equations in the system.
- **uel** (*callable function (optional)*) – Python function that returns the local stiffness matrix.

Returns **KG** – Array with the global stiffness matrix in a dense numpy array.

Return type ndarray (float)

assemutil.**eqcounter**(*nodes*)

Counts active equations and creates BCs array IBC

Parameters **nodes** (*ndarray*) – Array with nodes coordinates and boundary conditions.

Returns

- **neq** (*int*) – Number of equations in the system after removing the nodes with imposed displacements.
- **IBC** (*ndarray (int)*) – Array that maps the nodes with number of equations.

assemutil.**loadasem**(*loads*, *IBC*, *neq*)

Assembles the global Right Hand Side Vector RHSG

Parameters

- **loads** (*ndarray*) – Array with the loads imposed in the system.
- **IBC** (*ndarray (int)*) – Array that maps the nodes with number of equations.
- **neq** (*int*) – Number of equations in the system after removing the nodes with imposed displacements.

Returns **RHSG** – Array with the right hand side vector.

Return type ndarray

assemutil.**retriever**(*elements*, *mats*, *nodes*, *i*, *uel=None*)

Computes the elemental stiffness matrix of element i

Parameters

- **elements** (*ndarray*) – Array with the number for the nodes in each element.
- **mats** (*ndarray.*) – Array with the material profiles.
- **nodes** (*ndarray.*) – Array with the nodal numbers and coordinates.
- **i** (*int.*) – Identifier of the element to be assembled.

Returns

- **kloc** (*ndarray (float)*) – Array with the local stiffness matrix.
- **ndof** (*int.*) – Number of degrees of fredom of the current element.

assemutil.**sparse_assem**(*elements*, *mats*, *nodes*, *neq*, *DME*, *uel=None*)

Assembles the global stiffness matrix _KG_ using a sparse storing scheme

The scheme used to assemble is COOrdinate list (COO), and it converted to Compressed Sparse Row (CSR) afterward for the solution phase [1]_.

Parameters

- **elements** (*ndarray (int)*) – Array with the number for the nodes in each element.
- **mats** (*ndarray (float)*) – Array with the material profiles.
- **nodes** (*ndarray (float)*) – Array with the nodal numbers and coordinates.
- **DME** (*ndarray (int)*) – Assembly operator.
- **neq** (*int*) – Number of active equations in the system.
- **uel** (*callable function (optional)*) – Python function that returns the local stiffness matrix.

Returns **KG** – Array with the global stiffness matrix in a sparse Compressed Sparse Row (CSR) format.

Return type ndarray (float)

References

6.3 FEM routines

Functions to compute kinematics variables for the Finite Element Analysis.

The elements included are:

1. 4 node bilinear quadrilateral.
2. 6 node quadratic triangle.
3. 3 node linear triangle.

The notation used is similar to the one used by Bathe [1]_.

References

femutil.**eletype**(*iet*)

Assigns number to degrees of freedom

According to iet assigns number of degrees of freedom, number of nodes and minimum required number of integration points.

Parameters **iet** (*int*) –

Type of element. These are:

1. 4 node bilinear quadrilateral.
2. 6 node quadratic triangle.

3. 3 node linear triangle. 5. 2 node spring. 6. 2 node truss element. 7. 2 node beam (3 DOF per node).

Returns

- **ndof** (*int*) – Number of degrees of freedom for the selected element.
- **nnodes** (*int*) – Number of nodes for the selected element.
- **ngpts** (*int*) – Number of Gauss points for the selected element.

femutil.**jacoper**(*dhdx*, *coord*)
Compute the Jacobian of the transformation evaluated at the Gauss point

Parameters

- **dhdx** (*ndarray*) – Derivatives of the interpolation function with respect to the natural coordinates.
- **coord** (*ndarray*) – Coordinates of the nodes of the element (nn, 2).

Returns **jaco_inv** – Jacobian of the transformation evaluated at *(r, s)*.

Return type ndarray (2, 2)

femutil.**sha3**(*x*, *y*)
Shape functions for a 3-noded triangular element

Parameters

- **x** (*float*) – x coordinate for a point within the element.
- **y** (*float*) – y coordinate for a point within the element.

Returns **N** – Array of interpolation functions.

Return type Numpy array

Examples

We can check evaluating at two different points, namely (0, 0) and (0, 0.5). Thus

```
>>> N = sha3(0, 0)
>>> N_ex = np.array([
...     [1, 0, 0, 0, 0, 0],
...     [0, 1, 0, 0, 0, 0]])
>>> np.allclose(N, N_ex)
True
```

and

```
>>> N = sha3(1/2, 1/2)
>>> N_ex = np.array([
...     [0, 0, 1/2, 0, 1/2, 0],
...     [0, 0, 0, 1/2, 0, 1/2]])
>>> np.allclose(N, N_ex)
True
```

femutil.**sha4**(*x*, *y*)
Shape functions for a 4-noded quad element

Parameters

- **x** (*float*) – x coordinate for a point within the element.
- **y** (*float*) – y coordinate for a point within the element.

Returns **N** – Array of interpolation functions.

Return type Numpy array

Examples

We can check evaluating at two different points, namely (0, 0) and (1, 1). Thus

```
>>> N = sha4(0, 0)
>>> N_ex = np.array([
...     [1/4, 0, 1/4, 0, 1/4, 0, 1/4, 0],
...     [0, 1/4, 0, 1/4, 0, 1/4, 0, 1/4]])
>>> np.allclose(N, N_ex)
True
```

and

```
>>> N = sha4(1, 1)
>>> N_ex = np.array([
...     [0, 0, 0, 0, 1, 0, 0, 0],
...     [0, 0, 0, 0, 0, 1, 0, 0]])
>>> np.allclose(N, N_ex)
True
```

femutil.**sha6**(*x*, *y*)

Shape functions for a 6-noded triangular element

Parameters

- **x** (*float*) – x coordinate for a point within the element.
- **y** (*float*) – y coordinate for a point within the element.

Returns **N** – Array of interpolation functions.

Return type Numpy array

Examples

We can check evaluating at two different points, namely (0, 0) and (0.5, 0.5). Thus

```
>>> N = sha6(0, 0)
>>> N_ex = np.array([
...     [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
...     [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
>>> np.allclose(N, N_ex)
True
```

and

```
>>> N = sha6(1/2, 1/2)
>>> N_ex = np.array([
...      [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
...      [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]])
>>> np.allclose(N, N_ex)
True
```

femutil.**stdm3NT**(*r*, *s*, *coord*)

Strain-displacement interpolator B for a 3-noded triang element

Parameters

- **r** (*float*) – r component in the natural space.
- **s** (*float*) – s component in the natural space.
- **coord** (*ndarray*) – Coordinates of the nodes of the element (3, 2).

Returns

- **det** (*float*) – Determinant evaluated at *(r, s)*.
- **B** (*ndarray*) – Strain-displacement interpolator evaluated at *(r, s)*.

femutil.**stdm4NQ**(*r*, *s*, *coord*)
Strain-displacement interpolator B for a 4-noded quad element

Parameters

- **r** (*float*) – r component in the natural space.
- **s** (*float*) – s component in the natural space.
- **coord** (*ndarray*) – Coordinates of the nodes of the element (4, 2).

Returns

- **ddet** (*float*) – Determinant evaluated at *(r, s)*.
- **B** (*ndarray*) – Strain-displacement interpolator evaluated at *(r, s)*.

femutil.**stdm6NT**(*r*, *s*, *coord*)
Strain-displacement interpolator B for a 6-noded triang element

Parameters

- **r** (*float*) – r component in the natural space.
- **s** (*float*) – s component in the natural space.
- **coord** (*ndarray*) – Coordinates of the nodes of the element (6, 2).

Returns

- **ddet** (*float*) – Determinant evaluated at *(r, s)*.
- **B** (*ndarray*) – Strain-displacement interpolator evaluated at *(r, s)*.

femutil.**str_el3**(*coord*, *ul*)
Compute the strains at each element integration point

This one is used for 3-noded triangular elements.

Parameters

- **coord** (*ndarray*) – Coordinates of the nodes of the element (nn, 2).
- **ul** (*ndarray*) – Array with displacements for the element.

Returns

- **epsGT** (*ndarray*) – Strain components for the Gauss points.
- **xl** (*ndarray*) – Configuration of the Gauss points after deformation.

femutil.**str_el4**(*coord*, *ul*)
Compute the strains at each element integration point

This one is used for 4-noded quadrilateral elements.

Parameters

- **coord** (*ndarray*) – Coordinates of the nodes of the element (4, 2).
- **ul** (*ndarray*) – Array with displacements for the element.

Returns

- **epsGT** (*ndarray*) – Strain components for the Gauss points.
- **xl** (*ndarray*) – Configuration of the Gauss points after deformation.

femutil.**str_el6**(*coord*, *ul*)

Compute the strains at each element integration point

This one is used for 6-noded triangular elements.

Parameters

- **coord** (*ndarray*) – Coordinates of the nodes of the element (6, 2).
- **ul** (*ndarray*) – Array with displacements for the element.

Returns

- **epsGT** (*ndarray*) – Strain components for the Gauss points.
- **xl** (*ndarray*) – Configuration of the Gauss points after deformation.

femutil.**umat**(*nu*, *E*)

2D Elasticity consitutive matrix in plane stress

For plane strain use effective properties.

Parameters

- **nu** (*float*) – Poisson coefficient (-1, 0.5).
- **E** (*float*) – Young modulus (>0).

Returns **C** – Constitutive tensor in Voigt notation.

Return type ndarray

Examples

```
>>> C = umat(1/3, 8/3)
>>> C_ex = np.array([
...     [3, 1, 0],
...     [1, 3, 0],
...     [0, 0, 1]])
>>> np.allclose(C, C_ex)
True
```

6.4 Numeric integration routines

Weights and coordinates for Gauss-Legendre quadrature [1]_. The values for triangles is presented in section 5.5 of Bathe book[2].

[2] Bathe, Klaus-Jürgen. Finite element procedures. Prentice Hall, Pearson Education, 2006.

References

gaussutil.**gpoints2x2**()
Gauss points for a 2 by 2 grid

Returns

- **xw** (*ndarray*) – Weights for the Gauss-Legendre quadrature.
- **xp** (*ndarray*) – Points for the Gauss-Legendre quadrature.

gaussutil.**gpoints3**()
Gauss points for a triangle element (3 points)

Returns

- **xw** (*ndarray*) – Weights for the Gauss-Legendre quadrature.
- **xp** (*ndarray*) – Points for the Gauss-Legendre quadrature.

gaussutil.**gpoints7**()
Gauss points for a triangle (7 points)

Returns

- **xw** (*ndarray*) – Weights for the Gauss-Legendre quadrature.
- **xp** (*ndarray*) – Points for the Gauss-Legendre quadrature.

6.5 Postprocessor subroutines

This module contains functions to preprocess the input files to compute a Finite Element Analysis.

preprocesor.**boundary_conditions**(*cells*, *cell_data*, *phy_lin*, *nodes_array*, *bc_x*, *bc_y*)
Impose nodal point boundary conditions as required by SolidsPy

Parameters

- **cell** (*dictionary*) – Dictionary created by meshio with cells information.
- **cell_data** (*dictionary*) – Dictionary created by meshio with cells data information.
- **phy_lin** (*int*) – Physical line where BCs are to be imposed.
- **nodes_array** (*int*) – Array with the nodal data and to be modified by BCs.
- **bc_y** (*bc_x,*) –

  Boundary condition flag along the x and y direction:

  – -1: restrained

  – 0: free

Returns **nodes_array** – Array with the nodal data after imposing BCs according to SolidsPy.

Return type int

preprocesor.**echomod**(*nodes*, *mats*, *elements*, *loads*, *folder=''*)
Create echoes of the model input files

preprocesor.**ele_writer**(*cells*, *cell_data*, *ele_tag*, *phy_sur*, *ele_type*, *mat_tag*, *nini*)
Extracts a subset of elements from a complete mesh according to the physical surface phy_sur and writes down the proper fields into an elements array.

Parameters

- **cell** (*dictionary*) – Dictionary created by meshio with cells information.
- **cell_data** (*dictionary*) – Dictionary created by meshio with cells data information.
- **ele_tag** (*string*) – Element type according to meshio convention, e.g., quad9 or line3.
- **phy_sur** (*int*) – Physical surface for the subset.
- **ele_type** (*int*) – Element type.
- **mat_tag** (*int*) – Material profile for the subset.
- **ndof** (*int*) – Number of degrees of freedom for the elements.
- **nnode** (*int*) – Number of nodes for the element.
- **nini** (*int*) – Element id for the first element in the set.

Returns

- **nf** (*int*) – Element id for the last element in the set
- **els_array** (*int*) – Elemental data.

preprocesor.**initial_params**()
Read initial parameters for the simulation

The parameters to be read are:

- folder: location of the input files.
- name: name for the output files (if echo is True).
- echo: echo output files.

preprocesor.**loading**(*cells*, *cell_data*, *phy_lin*, *P_x*, *P_y*)
Impose nodal boundary conditions as required by SolidsPy

Parameters

- **cell** (*dictionary*) – Dictionary created by meshio with cells information.
- **cell_data** (*dictionary*) – Dictionary created by meshio with cells data information.
- **phy_lin** (*int*) – Physical line where BCs are to be imposed.
- **nodes_array** (*int*) – Array with the nodal data and to be modified by BCs.
- **P_y** (*P_x,*) – Load components in x and y directions.

Returns **nodes_array** – Array with the nodal data after imposing BCs according to SolidsPy.

Return type int

preprocesor.**node_writer**(*points*, *point_data*)
Write nodal data as required by SolidsPy

Parameters

- **points** (*dictionary*) – Nodal points
- **point_data** (*dictionary*) – Physical data associatted to the nodes.

Returns **nodes_array** – Array with the nodal data according to SolidsPy.

Return type ndarray (int)

preprocesor.**readin**(*folder=''*)
Read the input files

preprocesor.**rect_grid**(*length*, *height*, *nx*, *ny*, *eletype=None*)
Generate a structured mesh for a rectangle

The coordinates of the nodes will be defined in the domain [-length/2, length/2] x [-height/2, height/2].

Parameters

- **length** (*float*) – Length of the domain.
- **height** (*gloat*) – Height of the domain.
- **nx** (*int*) – Number of elements in the x direction.
- **ny** (*int*) – Number of elements in the y direction.
- **eletype** (*None*) – It does nothing right now.

Returns

- **x** (*ndarray (float)*) – x-coordinates for the nodes.
- **y** (*ndarray (float)*) – y-coordinates for the nodes.
- **els** (*ndarray*) – Array with element data.

Examples

```
>>> x, y, els = rect_grid(2, 2, 2, 2)
>>> x
array([-1.,  0.,  1., -1.,  0.,  1., -1.,  0.,  1.])
>>> y
array([-1., -1., -1.,  0.,  0.,  0.,  1.,  1.,  1.])
>>> els
array([[0, 1, 0, 0, 1, 4, 3],
       [1, 1, 0, 1, 2, 5, 4],
       [2, 1, 0, 3, 4, 7, 6],
       [3, 1, 0, 4, 5, 8, 7]])
```

6.6 Postprocessor subroutines

This module contains functions to postprocess results.

postprocesor.**complete_disp**(*IBC*, *nodes*, *UG*)
Fill the displacement vectors with imposed and computed values.

IBC [ndarray (int)] IBC (Indicator of Boundary Conditions) indicates if the nodes has any type of boundary conditions applied to it.

UG [ndarray (float)] Array with the computed displacements.

nodes [ndarray (float)] Array with number and nodes coordinates

Returns **UC** – Array with the displacements.

Return type (nnodes, 2) ndarray (float)

postprocesor.**eigvals**(*mat*, *tol=1e-06*)
Eigenvalues and eigenvectors for a 2x2 symmetric matrix/tensor

Parameters

- **mat** (*ndarray*) – Symmetric matrix.
- **tol** (*float (optional)*) – Tolerance for considering a matrix diagonal.

Returns

- **eig1** (*float*) – First eigenvalue.
- **eig2** (*float*) – Second eigenvalue.
- **vec1** (*ndarray*) – First eigenvector.
- **vec2** (*ndarray*) – Second eigenvector

Examples

```
>>> mat = np.array([[5, 6],
...                 [6, 9]])
>>> eig1, eig2, vec1, vec2 =  eigvals(mat)
>>> np.allclose(eig1, 7 + 2*np.sqrt(10))
True
>>> np.allclose(eig2, 7 - 2*np.sqrt(10))
True
>>> np.allclose(vec1, np.array([-0.584710284663765, -0.8112421851755609]))
True
>>> np.allclose(vec2, np.array([-0.8112421851755609,0.584710284663765]))
True
```

postprocesor.**energy**(*disp*, *stiff*)
Computes the potential energy for the current solution.

Parameters

- **disp** (*ndarray (float)*) – Array with the computed displacements.
- **stiff** (*ndarray (float)*) – Global stiffness matrix.

Returns **el_energy** – Total energy in the system. $-\frac{1}{2}U^T KU$

Return type scalar (float)

postprocesor.**fields_plot**(*elements*, *nodes*, *disp*, *E_nodes=None*, *S_nodes=None*)
Plot contours for displacements, strains and stresses

Parameters

- **nodes** (*ndarray (float)*) –

  Array with number and nodes coordinates: *number coordX coordY BCX BCY*
- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.
- **disp** (*ndarray (float)*) – Array with the displacements.
- **E_nodes** (*ndarray (float)*) – Array with strain field in the nodes.
- **S_nodes** (*ndarray (float)*) – Array with stress field in the nodes.

postprocesor.**mesh2tri**(*nodes*, *elements*)
Generate a matplotlib.tri.Triangulation object from the mesh

Parameters

- **nodes** (*ndarray (float)*) –

  Array with number and nodes coordinates: *number coordX coordY BCX BCY*

- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.

Returns **tri** – An unstructured triangular grid consisting of npoints points and ntri triangles.

Return type Triangulation

postprocesor.**plot_node_field**(*field*, *nodes*, *elements*, *plt_type='contourf'*, *levels=12*, *savefigs=False*, *title=None*, *figtitle=None*, *filename=None*)
Plot the nodal displacement using a triangulation

Parameters

- **field** (*ndarray (float)*) – Array with the field to be plotted. The number of columns determine the number of plots.
- **nodes** (*ndarray (float)*) – Array with number and nodes coordinates *number coordX coordY BCX BCY*
- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.
- **plt_type** (*string (optional)*) – Plot the field as one of the options: `pcolor` or `contourf`.
- **levels** (*int (optional)*) – Number of levels to be used in `contourf`.
- **savefigs** (*bool (optional)*) – Allow to save the figure.
- **title** (*Tuple of strings (optional)*) – Titles of the plots. If not provided the plots will not have a title.
- **figtitle** (*Tuple of strings (optional)*) – Titles of the plotting windows. If not provided the windows will not have a title.
- **filename** (*Tuple of strings (optional)*) – Filenames to save the figures. Only used when *savefigs=True*. If not provided the name of the figures would be "outputk.pdf", where *k* is the number of the column.

postprocesor.**plot_truss**(*nodes*, *elements*, *mats*, *stresses=None*, *max_val=4*, *min_val=0.5*, *savefigs=False*, *title=None*, *figtitle=None*, *filename=None*)
Plot a truss and encodes the stresses in a colormap

Parameters

- **UC** (*(nnodes, 2) ndarray (float)*) – Array with the displacements.
- **nodes** (*ndarray (float)*) – Array with number and nodes coordinates *number coordX coordY BCX BCY*
- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.
- **mats** (*ndarray (float)*) – Array with material profiles.
- **loads** (*ndarray (float)*) – Array with loads.

- **tol** (*float (optional)*) – Minimum difference between cross-section areas of the members to be considered different.
- **savefigs** (*bool (optional)*) – Allow to save the figure.
- **title** (*Tuple of strings (optional)*) – Titles of the plots. If not provided the plots will not have a title.
- **figtitle** (*Tuple of strings (optional)*) – Titles of the plotting windows. If not provided the windows will not have a title.
- **filename** (*Tuple of strings (optional)*) – Filenames to save the figures. Only used when *savefigs=True*. If not provided the name of the figures would be "output.pdf", where *k* is the number of the column.

postprocesor.**principal_dirs**(*field*)

Compute the principal directions of a tensor field

Parameters **field** (*ndarray*) – Tensor field. The tensor is written as "vector" using Voigt notation.

Returns

- **eigs1** (*ndarray*) – Array with the first eigenvalues.
- **eigs2** (*ndarray*) – Array with the second eigenvalues.
- **vecs1** (*ndarray*) – Array with the first eigenvectors.
- **vecs2** (*ndarray*) – Array with the Second eigenvector.

postprocesor.**strain_nodes**(*nodes*, *elements*, *mats*, *UC*)

Compute averaged strains and stresses at nodes

First, the variable is extrapolated from the Gauss point to nodes for each element. Then, these values are averaged according to the number of element that share that node. The theory for this technique can be found in [1]_.

Parameters

- **nodes** (*ndarray (float)*) – Array with nodes coordinates.
- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.
- **mats** (*ndarray (float)*) – Array with material profiles.
- **UC** (*ndarray (float)*) – Array with the displacements. This one contains both, the computed and imposed values.

Returns **E_nodes** – Strains evaluated at the nodes.

Return type ndarray

References

postprocesor.**stress_truss**(*nodes*, *elements*, *mats*, *disp*)

Compute axial stresses in truss members

Parameters

- **nodes** (*ndarray (float)*) – Array with nodes coordinates.
- **elements** (*ndarray (int)*) – Array with the node number for the nodes that correspond to each element.

- **mats** (*ndarray (float)*) – Array with material profiles.
- **disp** (*ndarray (float)*) – Array with the displacements. This one contains both, the computed and imposed values.

Returns **stress** – Stresses for each member on the truss

Return type ndarray

Examples

The following examples are taken from [1]_. In all the examples $A = 1$, $E = 1$.

```
>>> import assemutil as ass
>>> import solidspy.solutil as sol
```

```
>>> def fem_sol(nodes, elements, mats, loads):
...     DME , IBC , neq = ass.DME(nodes, elements)
...     KG = ass.assembler(elements, mats, nodes, neq, DME)
...     RHSG = ass.loadasem(loads, IBC, neq)
...     UG = sol.static_sol(KG, RHSG)
...     UC = complete_disp(IBC, nodes, UG)
...     return UC
```

Exercise 3.3-18

The axial stresses in this example are

$$[\sigma] = \left[\frac{1}{2}, \frac{\sqrt{3}}{4}, \frac{1}{4}\right]$$

```
>>> nodes = np.array([
... [0, 0.0,  0.0, 0, -1],
... [1, -1.0,  0.0, -1, -1],
... [2,  -np.cos(np.pi/6),  -np.sin(np.pi/6),  -1,  -1],
... [3,  -np.cos(np.pi/3),  -np.sin(np.pi/3),  -1,  -1]])
>>> mats = np.array([[1.0, 1.0]])
>>> elements = np.array([
... [0, 6, 0, 1, 0],
... [1, 6, 0, 2, 0],
... [2, 6, 0, 3, 0]])
>>> loads = np.array([[0, 1.0, 0]])
>>> disp = fem_sol(nodes, elements, mats, loads)
>>> stress = stress_truss(nodes, elements, mats, disp)
>>> stress_exact = np.array([1/2, np.sqrt(3)/4, 1/4])
>>> np.allclose(stress_exact, stress)
True
```

Exercise 3.3-19

The axial stresses in this example are

$$[\sigma] = \left[\frac{1}{\sqrt{2}+2}, \frac{\sqrt{2}}{\sqrt{2}+1}, \frac{1}{\sqrt{2}+2}\right]$$

```
>>> nodes = np.array([
...     [0, 0.0,  0.0, 0, 0],
...     [1, -1.0,  -1.0, -1, -1],
...     [2,  0.0,  -1.0,  -1,  -1],
...     [3,  1.0, -1.0,  -1,  -1]])
>>> mats = np.array([[1.0, 1.0]])
>>> elements = np.array([
...     [0, 6, 0, 1, 0],
...     [1, 6, 0, 2, 0],
...     [2, 6, 0, 3, 0]])
>>> loads = np.array([[0, 0, 1]])
>>> disp = fem_sol(nodes, elements, mats, loads)
>>> stress = stress_truss(nodes, elements, mats, disp)
>>> stress_exact = np.array([
...      1/(np.sqrt(2) + 2),
...      np.sqrt(2)/(np.sqrt(2) + 1),
...      1/(np.sqrt(2) + 2)])
>>> np.allclose(stress_exact, stress)
True
```

Exercise 3.3-22

The axial stresses in this example are

$$[\sigma] = \left[\frac{1}{3\sqrt{2}}, \frac{5}{12}, \frac{1}{2^{\frac{3}{2}}}, \frac{1}{12}, -\frac{1}{3\sqrt{2}}\right]$$

```
>>> cathetus = np.cos(np.pi/4)
>>> nodes = np.array([
...     [0, 0.0,  0.0, 0, 0],
...     [1, -1.0,  0.0, -1, -1],
...     [2,  -cathetus,  cathetus,  -1,  -1],
...     [3,  0.0, 1.0,  -1,  -1],
...     [4,  cathetus, cathetus,  -1,  -1],
...     [5,  1.0, 0.0,  -1,  -1]])
>>> mats = np.array([[1.0, 1.0]])
>>> elements = np.array([
...     [0, 6, 0, 1, 0],
...     [1, 6, 0, 2, 0],
...     [2, 6, 0, 3, 0],
...     [3, 6, 0, 4, 0],
...     [4, 6, 0, 5, 0]])
>>> loads = np.array([[0, cathetus, -cathetus]])
>>> disp = fem_sol(nodes, elements, mats, loads)
>>> stress = stress_truss(nodes, elements, mats, disp)
>>> stress_exact = np.array([
...      1/(3*np.sqrt(2)),
...      5/12,
...      1/2**(3/2),
...      1/12,
...      -1/(3*np.sqrt(2))])
>>> np.allclose(stress_exact, stress)
True
```

References

postprocesor.**tri_plot**(*tri*, *field*, *title=''*, *levels=12*, *savefigs=False*, *plt_type='contourf'*, *filename='solution_plot.pdf'*)

Plot contours over triangulation

Parameters

- **tri** (*ndarray (float)*) – Array with number and nodes coordinates: *number coordX coordY BCX BCY*
- **field** (*ndarray (float)*) – Array with data to be plotted for each node.
- **title** (*string (optional)*) – Title of the plot.
- **levels** (*int (optional)*) – Number of levels to be used in `contourf`.
- **savefigs** (*bool (optional)*) – Allow to save the figure.
- **plt_type** (*string (optional)*) – Plot the field as one of the options: `pcolor` or `contourf`
- **filename** (*string (optional)*) – Filename to save the figures.

6.7 Solver routines

Utilities for solution of FEM systems

solutil.**static_sol**(*mat*, *rhs*)

Solve a static problem [mat]{u_sol} = {rhs}

Parameters

- **mat** (*array*) – Array with the system of equations. It can be stored in dense or sparse scheme.
- **rhs** (*array*) – Array with right-hand-side of the system of equations.

Returns **u_sol** – Solution of the system of equations.

Return type array

6.8 Element subroutines

Each UEL subroutine computes the local stiffness matrix for a given finite element.

New elements can be added by including additional subroutines.

uelutil.**uel3ntrian**(*coord*, *enu*, *Emod*)

Triangular element with 3 nodes

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (3, 2).
- **enu** (*float*) – Poisson coefficient (-1, 0.5).
- **Emod** (*float*) – Young modulus (>0).

Returns **kl** – Local stiffness matrix for the element (6, 6).

Return type ndarray

Examples

```
>>> coord = np.array([
...          [0, 0],
...          [1, 0],
...          [0, 1]])
>>> stiff = uel3ntrian(coord, 1/3, 8/3)
>>> stiff_ex = 1/2 * np.array([
...              [4, 2, -3, -1, -1, -1],
...              [2, 4, -1, -1, -1, -3],
...              [-3, -1, 3, 0, 0, 1],
...              [-1, -1, 0, 1, 1, 0],
...              [-1, -1, 0, 1, 1, 0],
...              [-1, -3, 1, 0, 0, 3]])
>>> np.allclose(stiff, stiff_ex)
True
```

uelutil.**uel4nquad**(*coord*, *enu*, *Emod*)

Quadrilateral element with 4 nodes

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (4, 2).
- **enu** (*float*) – Poisson coefficient (-1, 0.5).
- **Emod** (*float*) – Young modulus (>0).

Returns **kl** – Local stiffness matrix for the element (8, 8).

Return type ndarray

Examples

```
>>> coord = np.array([[-1, -1], [1, -1], [1, 1], [-1, 1]])
>>> stiff = uel4nquad(coord, 1/3, 8/3)
>>> stiff_ex = 1/6 * np.array([
...              [ 8,  3, -5,  0, -4, -3,  1,  0],
...              [ 3,  8,  0,  1, -3, -4,  0, -5],
...              [-5,  0,  8, -3,  1,  0, -4,  3],
...              [ 0,  1, -3,  8,  0, -5,  3, -4],
...              [-4, -3,  1,  0,  8,  3, -5,  0],
...              [-3, -4,  0, -5,  3,  8,  0,  1],
...              [ 1,  0, -4,  3, -5,  0,  8, -3],
...              [ 0, -5,  3, -4,  0,  1, -3,  8]])
>>> np.allclose(stiff, stiff_ex)
True
```

uelutil.**uel6ntrian**(*coord*, *enu*, *Emod*)

Triangular element with 6 nodes

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (6, 2).
- **enu** (*float*) – Poisson coefficient (-1, 0.5).
- **Emod** (*float*) – Young modulus (>0).

Returns **kl** – Local stiffness matrix for the element (12, 12).

Return type ndarray

Examples

```
>>> coord = np.array([
...          [0, 0],
...          [1, 0],
...          [0, 1],
...          [0.5, 0],
...          [0.5, 0.5],
...          [0, 0.5]])
>>> stiff = uel6ntrian(coord,1/3, 8/3)
>>> stiff_ex = 1/6 * np.array([
...             [12, 6, 3, 1, 1, 1, -12, -4, 0, 0, -4, -4],
...             [6, 12, 1, 1, 1, 3, -4, -4, 0, 0, -4, -12],
...             [3, 1, 9, 0, 0, -1, -12, -4, 0, 4, 0, 0],
...             [1, 1, 0, 3, -1, 0, -4, -4, 4, 0, 0, 0],
...             [1, 1, 0, -1, 3, 0, 0, 0, 0, 4, -4, -4],
...             [1, 3, -1, 0, 0, 9, 0, 0, 4, 0, -4, -12],
...             [-12, -4, -12, -4, 0, 0, 32, 8, -8, -8, 0, 8],
...             [-4, -4, -4, -4, 0, 0, 8, 32, -8, -24, 8, 0],
...             [0, 0, 0, 4, 0, 4, -8, -8, 32, 8, -24, -8],
...             [0, 0, 4, 0, 4, 0, -8, -24, 8, 32, -8, -8],
...             [-4, -4, 0, 0, -4, -4, 0, 8, -24, -8, 32, 8],
...             [-4, -12, 0, 0, -4, -12, 8, 0, -8, -8, 8, 32]])
>>> np.allclose(stiff, stiff_ex)
True
```

uelutil.**uelbeam2DU**(*coord*, *I*, *Emod*)

2D-2-noded beam element without axial deformation

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (2, 2).
- **A** (*float*) – Cross section area.
- **Emod** (*float*) – Young modulus (>0).

Returns **kl** – Local stiffness matrix for the element (4, 4).

Return type ndarray

uelutil.**uelspring**(*coord*, *enu*, *Emod*)

1D-2-noded Spring element

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (2, 2).
- **enu** (*float*) – Fictitious parameter.
- **Emod** (*float*) – Stiffness coefficient (>0).

Returns **kl** – Local stiffness matrix for the element (4, 4).

Return type ndarray

Examples

```
>>> coord = np.array([
...          [0, 0],
...          [1, 0]])
>>> stiff = uelspring(coord, 1/3, 8/3)
>>> stiff_ex = 8/3 * np.array([
...     [1, 0, -1, 0],
...     [0, 0, 0, 0],
...     [-1, 0, 1, 0],
...     [0, 0, 0, 0]])
>>> np.allclose(stiff, stiff_ex)
True
```

uelutil.**ueltruss2D**(*coord*, *A*, *Emod*)

2D-2-noded truss element

Parameters

- **coord** (*ndarray*) – Coordinates for the nodes of the element (2, 2).
- **A** (*float*) – Cross section area.
- **Emod** (*float*) – Young modulus (>0).

Returns **kl** – Local stiffness matrix for the element (4, 4).

Return type ndarray

Examples

```
>>> coord = np.array([
...          [0, 0],
...          [1, 0]])
>>> stiff = ueltruss2D(coord, 1.0 , 1.0)
>>> stiff_ex =  np.array([
...     [1, 0, -1, 0],
...     [0, 0, 0, 0],
...     [-1, 0, 1, 0],
...     [0, 0, 0, 0]])
>>> np.allclose(stiff, stiff_ex)
True
```

CHAPTER 7

History

7.1 1.0.15 (2018-05-09)

- Fix element ordering in *rectgrid* and doctests.

7.2 1.0.14 (2018-05-08)

- Add Jacobian checks.
- Pytest catch exceptions.

7.3 1.0.13 (2018-05-07)

- Update meshio syntax for physical groups.
- Add citation information to package.

7.4 1.0.12 (2018-04-16)

- Documentation built with Sphinx.
- Docs in ReadTheDocs.

7.5 1.0.0 (2017-07-17)

- First release on PyPI.

CHAPTER 8

Roadmap

Pending . . .

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[D3967-16] ASTM D3967–16 (2016), Standard Test Method for Splitting Tensile Strength of Intact Rock Core Specimens, ASTM International, www.astm.org.

[Gmsh2009] Geuzaine, Christophe, y Jean-François Remacle (2009), *Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities*. International Journal for Numerical Methods in Engineering, 79.11.

[Gmsh_tut] Geuzaine, Christophe, y Jean-François Remacle (2017), Gmsh Official Tutorial. Accessed: April 18, 2018 http://gmsh.info/doc/texinfo/gmsh.html#Tutorial.

[Gmsh_scr] Geuzaine, Christophe, y Jean-François Remacle (2017), Gmsh Official Screencasts. Accessed: April 18, 2018de http://gmsh.info/screencasts/.

Python Module Index

a

assemutil, 36

f

femutil, 38

g

gaussutil, 42

p

postprocesor, 45
preprocesor, 43

s

solids_GUI, 35
solutil, 51

u

uelutil, 51

Index

A

assembler() (*in module assemutil*), 36
assemutil (*module*), 36

B

boundary_conditions() (*in module preprocesor*), 43

C

complete_disp() (*in module postprocesor*), 45

D

dense_assem() (*in module assemutil*), 36
DME() (*in module assemutil*), 36

E

echomod() (*in module preprocesor*), 43
eigvals() (*in module postprocesor*), 45
ele_writer() (*in module preprocesor*), 43
eletype() (*in module femutil*), 38
energy() (*in module postprocesor*), 46
eqcounter() (*in module assemutil*), 37

F

femutil (*module*), 38
fields_plot() (*in module postprocesor*), 46

G

gaussutil (*module*), 42
gpoints2x2() (*in module gaussutil*), 43
gpoints3() (*in module gaussutil*), 43
gpoints7() (*in module gaussutil*), 43

I

initial_params() (*in module preprocesor*), 44

J

jacoper() (*in module femutil*), 39

L

loadasem() (*in module assemutil*), 37
loading() (*in module preprocesor*), 44

M

mesh2tri() (*in module postprocesor*), 46

N

node_writer() (*in module preprocesor*), 44

P

plot_node_field() (*in module postprocesor*), 47
plot_truss() (*in module postprocesor*), 47
postprocesor (*module*), 45
preprocesor (*module*), 43
principal_dirs() (*in module postprocesor*), 48

R

readin() (*in module preprocesor*), 44
rect_grid() (*in module preprocesor*), 45
retriever() (*in module assemutil*), 37

S

sha3() (*in module femutil*), 39
sha4() (*in module femutil*), 39
sha6() (*in module femutil*), 40
solids_GUI (*module*), 35
solids_GUI() (*in module solids_GUI*), 35
solutil (*module*), 51
sparse_assem() (*in module assemutil*), 37
static_sol() (*in module solutil*), 51
stdm3NT() (*in module femutil*), 40
stdm4NQ() (*in module femutil*), 41
stdm6NT() (*in module femutil*), 41
str_el3() (*in module femutil*), 41
str_el4() (*in module femutil*), 41
str_el6() (*in module femutil*), 42
strain_nodes() (*in module postprocesor*), 48
stress_truss() (*in module postprocesor*), 48

T

tri_plot() (*in module postprocesor*), 51

U

uel3ntrian() (*in module uelutil*), 51
uel4nquad() (*in module uelutil*), 52
uel6ntrian() (*in module uelutil*), 52
uelbeam2DU() (*in module uelutil*), 53
uelspring() (*in module uelutil*), 53
ueltruss2D() (*in module uelutil*), 54
uelutil (*module*), 51
umat() (*in module femutil*), 42