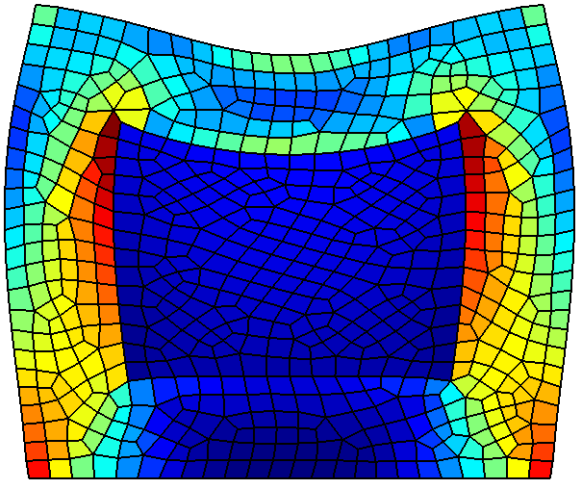
LUND UNIVERSITY



MESHING AND VISUALISATION ROUTINES IN THE PYTHON VERSION OF CALFEM

ANDREAS EDHOLM

Structural Mechanics

Master's Dissertation

Department of Construction Sciences
Structural Mechanics

ISRN LUTVDG/TVSM--13/5187--SE (1-86)
ISSN 0281-6679

MESHING AND VISUALISATION ROUTINES IN THE PYTHON VERSION OF CALFEM

Master's Dissertation by

ANDREAS EDHOLM

Supervisor:

Jonas Lindemann, *PhD,*
LUNARC, Lund

Examiner:

Ola Dahlblom, *Professor,*
Div. of Structural Mechanics

Printed by Media-Tryck LU, Lund, Sweden, February 2013 *(Pl).*

For information, address:
Division of Structural Mechanics, LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: http://www.byggmek.lth.se

Preface

I would like to thank project supervisor Jonas Lindemann for the suggestions and feedback provided by him during this project.

Lund, February 2013

Andreas Edholm

Abstract

This report is the result of a masters project conducted at the Division of Structural Mechanics at Lund University. The purpose of the project was to integrate a FEM mesh generator with the Python version of the code library CALFEM and improve the mesh visualisation capabilities of said library. This report describes how the mesher software Gmsh and the visualisation library visvis were integrated with CALFEM for Python. It also describes the three Python modules pycalfem_GeoData, pycalfem_mesh, and pycalfem_vis that were created as part of the project.

Contents

Chapter 1

Introduction

1.1 Software related to this project

CALFEM The work in this project consists of additions to the code library CALFEM for Python. The name stands for "Computer Aided Learning of the Finite Element Method", and its purpose is to serve as a learning tool for students of FEM. It was originally developed for FORTRAN and later redeveloped into a MATLAB toolbox.[1] Later still a Python version was developed by Andreas Ottosson in 2009 as part of his masters thesis at Lund University.[2]

The Python modules for CALFEM are named pycalfem and therefore the Python version of CALFEM is sometimes informally called "PyCALFEM".

Triangle Triangle is a FEM mesh generator for 2D triangular meshes. It is the meshing tool that is used in the older meshing routines in both the MATLAB and Python versions of CALFEM. It can create triangular meshes from 2D geometry with holes. Triangle is developed by Jonathan Richard Shewchuk.[3]

Gmsh Gmsh is a 3D finite element grid generator.[4] It is developed by Christophe Geuzaine, Jean-François Remacle and other contributors. It is used in the new meshing module described in this report.

Gmsh has a large set of features. It has a graphical interface that users can use to create and mesh geometry. Geometry can also be generated from scripts that can be written by hand. The scripts have file extension `geo`. Meshed geometry produces `msh`-files that contain nodes, elements, and other information about the FEM mesh.

Gmsh can be executed, and told to mesh geometry in `geo`-files, via the command-line. This is how Gmsh is used in this project.

Apart from these features Gmsh also has post-processing and visualisation features that were not used in this project.

Visvis Visvis is a graphical plotting library for Python.[5] It is an object oriented library built on top of OpenGL. It has some syntactic similarities to MATLAB, and is able to

plot in both 2D and 3D. It can draw 3D meshes and can easily be extended with custom drawable objects. Visvis is dependent on the libraries Numpy and PyOpenGL. It also needs a supported GUI backend, such as wxPython or PyQt.

1.2 Problem description

The project was divided into three goals; Meshing, visualisation, and general improvements.

Meshing The original mesh generation capabilities of CALFEM were weak. It could only use Triangle, which is only capable of generating 2D meshes with triangular elements.

The first priority of this project was therefore to find and incorporate an alternative meshing tool that could be used in the same way as Triangle, but with greater capabilities. The mesher tool had to be executable from a Python script, and be able to mesh all the element types available in CALFEM, which includes 3D hexahedral elements.

Since Triangle was neatly incorporated in CALFEM it was decided to attempt to make user-interaction with the new mesher as similar as possible to the way Triangle was used, i.e. it should be called with a function that returns the same data structures as trimesh2d, which is the function that meshes using Triangle.

Visualisation The biggest issue with the visualisation functionality in CALFEM for Python was that the draw functions assumed that the mesh consists of either 2D line elements of triangle elements. This meant that other element types could not be drawn.

Another issue was that all drawing was done in a pop-up window that could not be embedded in graphical user interfaces. The reason for this was that drawing functions were in the class ElementView, which inherits from wx.Frame. wx.Frame is a class in the wxPython GUI library and represents windows. This meant that it was impossible to embed graphs drawn with ElementView in other windows. It also meant that users had to use wxPython if they wanted to plot results with the built-in functions.

The second goal was to improve the visualisation capabilities of CALFEM for Python, so that all available element types could be drawn and so that graphs could be embedded in different GUI libraries.

At the beginning of the project it was decided that in order to solve these issues the existing draw functions would need to be remade, or a new set of draw functions would need to be written.

Improvements The final goal was to implement functionality that already existed in CALFEM for MATLAB but was not yet implemented in Python. In particular, the missing functions listed in appendix F of the original CALFEM for Python masters thesis[2] were to be implemented. This goal was of low priority, and was only to be done if the other two goals were completed within reasonable time.

Chapter 2

Development

The plan was to develop the parts of project in the order described in the previous chapter, i.e. meshing first, then visualisation, and finally general improvements to CALFEM for Python if any time remained. Due to the expected high reliance on external libraries, which had not been evaluated before the project started, it was very difficult to determine what functionality was possible to include, and how long it would take to implement functionality missing in the libraries. it was therefore decided early on to not write a formal requirements document. Another reason for this was because the goal was to match the functionality of CALFEM for MATLAB, so it could serve as a guideline and minimum requirements document.

2.1 Integration of mesh generation tools

The first objective was to implement a mesher. The obvious solution was to use an existing mesher program, of which there were many. There were a handful of requirements that such a mesher would have to fulfil. The requirements were:

- Must be callable from the command-line or otherwise be able to communicate with python programs.
- Must be able to mesh all the element types that CALFEM can handle.
- Must work in 2D and 3D.
- Must be free to use without licensing costs.
- Must be able to mesh geometry with holes.
- Optional: A graphical user interface. Especially if geometry can be built in the GUI.
- Optional: Easy to use.

In order to find such a mesher an online list of mesh generation software was consulted.[6] It was found that only the grid generator Gmsh fulfilled all requirements.

CALFEM for MATLAB already had a set of functions for meshing geometry. Early on in development this MATLAB-mesher was considered a guideline for the direction of development of the new mesher. It was soon dropped when it became apparent that the design ideas in the MATLAB-mesher did not work very well in Python or with Gmsh. The

idea of storing geometry data in structs (the MATLAB equivalent of Python dictionaries) was taken and developed into the class GeoData, but otherwise there are few similarities.

The new meshing functionality went through a couple of iterations before it arrived at its current shape. The class GeoData was not planned initially. It was created after it was decided that it was not enough to represent geometry with dictionaries or lists, as in MATLAB and the older Python function trimesh2d had done. The reason was that Triangle took geometry that consisted of points and straight lines, which could be represented with two lists, while Gmsh could take many different sorts of curves, surfaces and volumes.

The classes GeoData and GmshMesher were written to make this plethora of choices easily accessible. Since Gmsh geometry can be fairly complex, it has many restrictions on how to define geometry in the correct way. For example, there are "line loops" that are used for defining the boundaries of surfaces, and these line loops must be constructed from curves that are directed in a counter-clockwise fashion. It seemed like it would be a cumbersome chore for users to keep all these line loops and curves in mind, so GmshMesher was written so that it would automatically take care of such things. This is described in more detail in the Results section.

2.2 Development of visualisation functions

As mentioned, the visualisation functionality in CALFEM for Python had many issues. One of the issues was the drawing required a wxPython GUI, which meant that no other GUI library could be used. The first plan was to simply rewrite the existing class ElementView so that it was based on the wxPython Panel class instead of the Frame class. This would make plots embeddable in wx GUIs. ElementView could then be rewritten for other GUI libraries so that those other libraries could be used instead of wxPython.

That approach had some big flaws. The biggest being that rewriting the drawing code to different libraries would be time-consuming and lead to much duplicate code. It was therefore decided to instead write new drawing functions using an external plotting library rather than attempt to rewrite the existing functions. The plotting library would need to fulfil the following criteria:

- Able to draw triangles and quads with different-coloured faces and interpolated vertex colouring.
- Able to draw splines, B-splines, and ellipse arcs. Alternatively, plot arbitrary curves in 2D and 3D. This would be necessary in order to draw the different curve types that Gmsh can handle.
- Few library dependencies. This would be important on school computers where users might not have the administrative right to install new libraries.
- Embeddable in atleast wxPython and PyQT.
- Optional: Faster than ElementView.

In order to find such a library the internet was searched. Many libraries fulfilled some of these criteria. Matplotlib and Mayavi were good candidates that missed one or two important

criteria. In the end, graphics library visvis was chosen because it fulfilled all of the above criteria.

Visvis is a Python library with a plotting syntax similar to MATLAB. Because of this it was decided to avoid new classes and instead use convenience functions that work similarly to the MATLAB plotting functions. The functions that were written are described in the next chapter, in the pycalfem_vis section.

Chapter 3

Implementation

The results of the project are three new modules called pycalfem_GeoData, pycalfem_mesh, and pycalfem_vis. There were also a few additions to the module pycalfem and some example scripts that explain how to use the new modules.

Figure 3.1 contains a diagram of the dependencies of the new modules in CALFEM for Python. Similarly, figure 3.2 has a diagram of the classes. Note that "private" classes, functions, or attributes are not included in the diagrams, neither are external modules or libraries. This means that, for example, the dependency of pycalfem_vis on the library visvis is not shown.

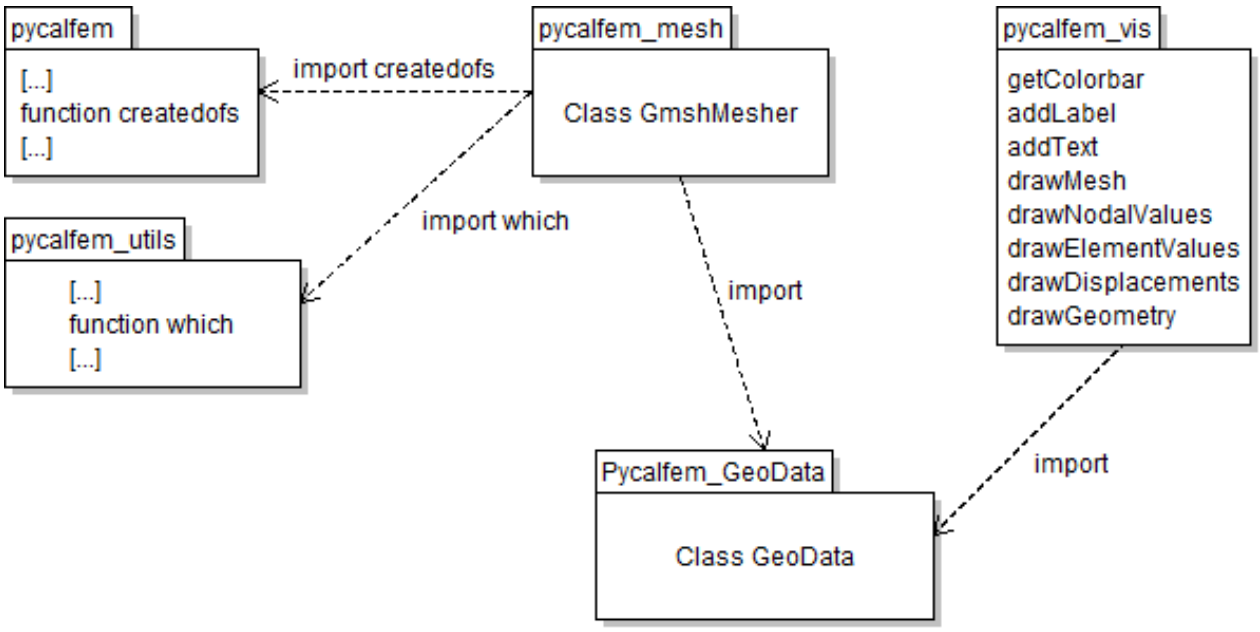


Figure 3.1: Dependencies between the new modules pycalfem_GeoData, pycalfem_mesh, and pycalfem_vis in CALFEM for Python

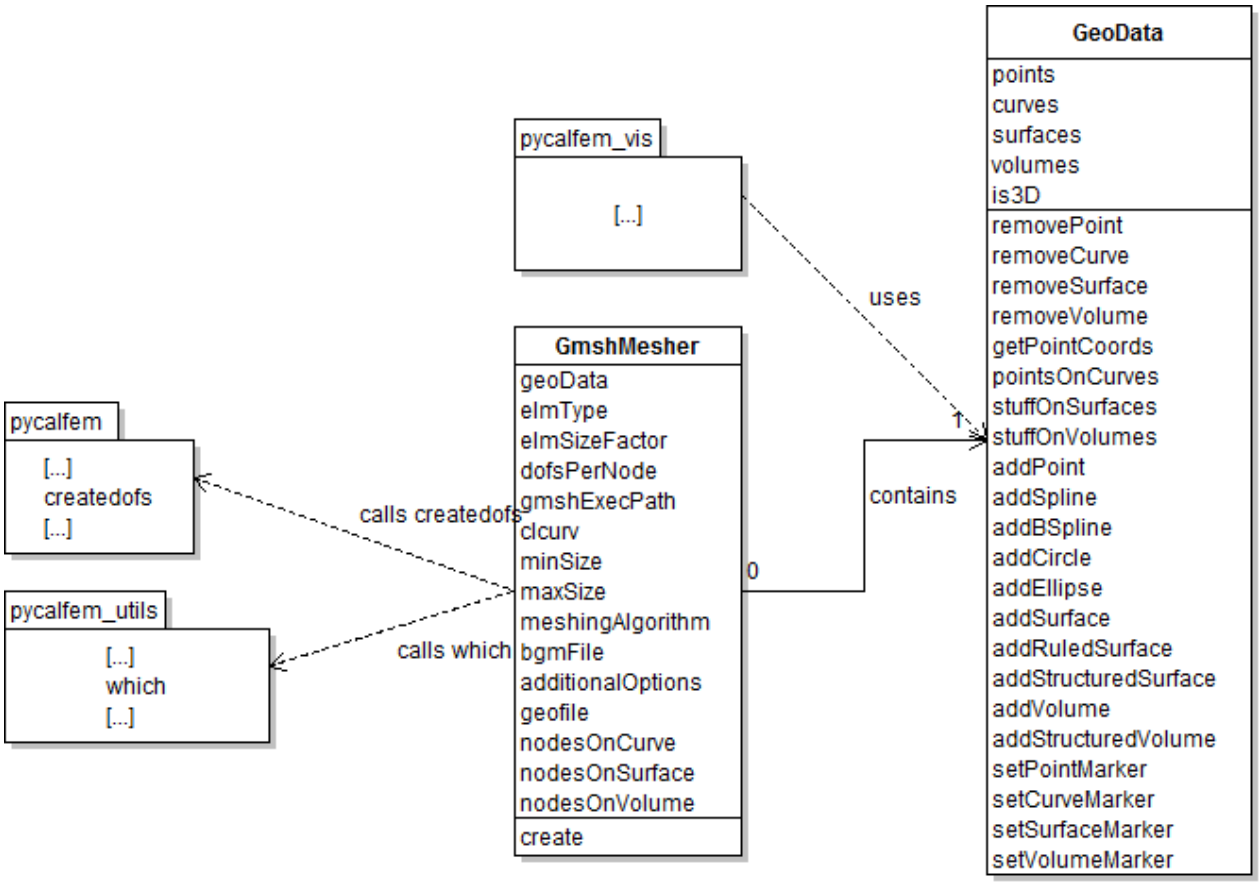


Figure 3.2: Class diagram of the new classes. The squares with tabs on the top are modules. Only "public" methods, attributes, and the functions that are called by the classes are shown.

3.1 Minimal example

Before describing the new modules in detail, here is a simple example of them in use. In the example we use all the three modules and the library visvis to create, mesh, and draw some geometry. The geometry is a flat triangle.

We begin by using addPoint to create three points at xy-coordinates `(0, 0)`, `(5, 0)`, and `(2.5, 4)`. These points are automatically assigned ID numbers, starting from 0.

Then we connect the points with three splines (in this example the splines are straight lines). The first parameter of addSpline is a list of point IDs that the spline connects. The other parameter seen here is marker, which is used for marking curves so that boundary conditions may be applied later. Like the points, our splines are automatically assigned ID numbers.

The last step of creating geometry is adding a surface with addSurface. The parameter is a list of curve IDs that define the outer boundary of the surface.

Then we can create a GmshMesher object. Its parameters are the GeoData object g, the file path to the executable file `gmsh.exe`, and a size factor that determines the size of the elements. For this example we assume that `gmsh.exe` is placed in a folder called `gmsh` in the current working directory (typically the same as our python script). Meshing is done by calling the create method, which returns the mesh in the shape of coords, edof, dofs, bdofs, and elementmarkers.

Drawing geometry is done by calling the function drawGeometry. We also call the visvis function figure, which opens a new figure window and sets it to the current figure, so that the next drawing is not done in the same window.

We draw the mesh by calling drawmesh. It has four necessary parameters. These are the node coordinate coords, the element topology edof, the number of dofs per node dofsPerNode, and the element type elType. Element type 2 is triangle (3 is quadrangle).

Finally, it is necessary to make the script enter the application loop of a GUI back-end. If this is not done the figures that have been plotted will disappear when the script terminates. The two final lines of code make visvis find a GUI back-end and enter its application loop.

```
import pycalfem_GeoData
import pycalfem_mesh
import pycalfem_vis as pcv
import visvis as vv

g = pycalfem_GeoData.GeoData()

g.addPoint([0  , 0])                #0
g.addPoint([5  , 0])                #1
g.addPoint([2.5, 4])                #2

g.addSpline([0, 1])             #0
g.addSpline([1, 2])             #1
g.addSpline([2, 0], marker=10) #2

g.addSurface([0,1,2])

mesher = pycalfem_mesh.GmshMesher(geoData = g,
                                  gmshExecPath = "gmsh/gmsh.exe"
                                  elSizeFactor = 0.15)

coords, edof, dofs, bdofs, elementmarkers = mesher.create()

pcv.drawGeometry(g)  #Draw geometry
vv.figure()          #create new figure window
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=1, elType=2)

app = vv.use()       #Get application object
app.Run()            #Run app main loop.
```

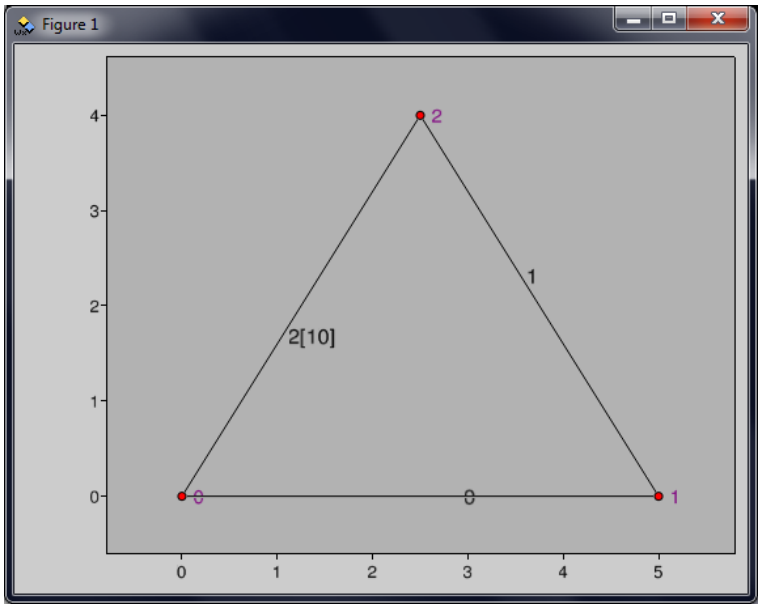


Figure 3.3: Plot of the geometry of the minimal example.

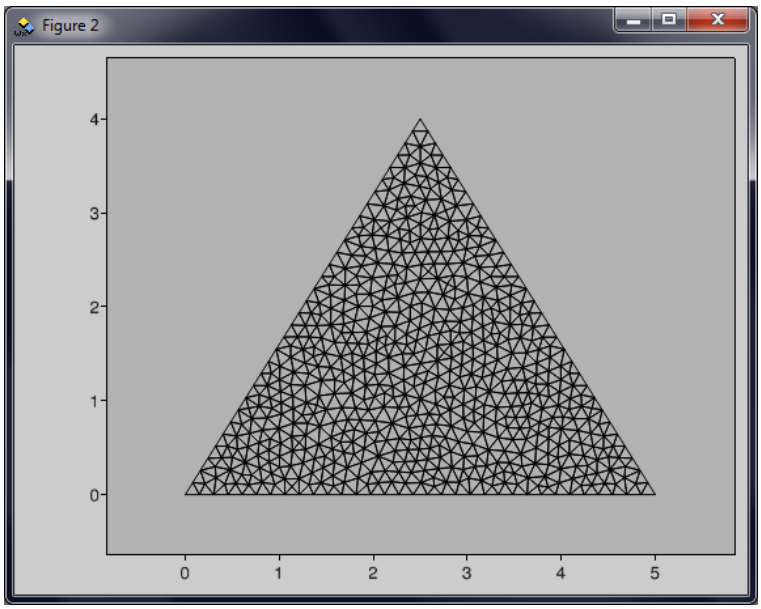


Figure 3.4: Plot of the mesh of the minimal example.

3.2 pycalfem_GeoData

The module pycalfem_GeoData is fairly straight-forward in purpose and structure. It contains a single class, GeoData, that simply holds the geometric data of a model. The user builds

geometry by calling methods like addPoint, addSpline, addSurface, etc. These methods do some (limited) checks on input to make sure it is correctly formatted.

Apart from geometry, the class also keeps track of markers and ID-numbers of all geometric entities.

The purpose of the class is to make construction of geometry as easy as possible for the user. The mesher Gmsh is much more complicated than the old mesher Triangle. However, it should still be easy to create simple geometry. By hiding the representation of the geometry behind adder- and remover-methods, the user can interact with the data without full knowledge of how it is represented. It also means that the representation can be changed in the future without breaking existing programs.

Class GeoData

Attributes

GeoData objects have a the following "public" attributes:

- points
- curves
- surfaces
- volumes
- is3D

Even though they are not prefixed by an underscore to mark them as private these attributes should not be accessed by the user directly. Instead, the user should add, remove and query geometric entities by calling the various methods in GeoData. The reason they are "public" is because they are directly accessed by the GmshMesher class and the pycalfem_vis module.

Methods

These are the methods, grouped by functionality:

- addPoint(coord, ID=`None`, marker=0, elSize=1)

Method addPoint adds a point to the geometry. The only parameter that must be defined is coord, which should be a lists of two or three coordinates. If three coordinates are given GeoData will assume that the model is in 3D, which among other things makes the coordinates of the mesh 3D.

ID is the ID number of this entity. It must be a non-negative integer. If no ID is supplied the point will be given an ID, starting from 0.

The parameter marker is used for specifying in which region an entity is.

Parameter elSize sets the preferred element size at this point. Users can set this to make the mesh more or less dense at different points. Mesh density may also be controlled with the parameters elSizeFactor, minSize and maxSize of the GmshMesher constructor.

- addSpline(points, ID=None, marker=0, elOnCurve=None, elDistribType=None, elDistribVal=None)
- addBSpline(points, ID=None, marker=0, elOnCurve=None, elDistribType=None, elDistribVal=None)
- addCircle(points, ID=None, marker=0, elOnCurve=None, elDistribType=None, elDistribVal=None)
- addEllipse(points, ID=None, marker=0, elOnCurve=None, elDistribType=None, elDistribVal=None)

These methods add curves to the geometry model. All curves share the same pool of ID-numbers, so users should not assign the same ID to a Spline and a Circle, for example.

The parameter marker is used for specifying in which region an entity is. When GmshMesher meshes geometry, dofs on this region will be placed in the return value dictionary bdofs so that users may apply boundary conditions.

The parameters elOnCurve, elDistribType, elDistribVal are used for structured meshes. Parameter elOnCurve sets the number of elements on the curve. If elOnCurve is defined the curve counts as a "structured" curve and can be used as a boundary curve on structured surfaces.

The other two parameters define how the elements are distributed along the curve. elDistribType can have values "bump" or "progression". "Bump" means elements are denser or less dense near the middle of the curve, while "progression" means elements are larger the further along the curve they are. elDistribVal should be a float with value typically around 0.1 to 2.0.

For example, if elDistribType = "progression" and elDistribVal = 1.1 the sides of each element will be 1.1 times that of the element that precedes it on the curve.

- addSurface(outerLoop, holes=[], ID=None, marker=0)
- addRuledSurface(outerLoop, ID=None, marker=0)
- addStructuredSurface(outerLoop, ID=None, marker=0)

These methods create surfaces from lists of curve IDs. The parameter outerLoop is a list of curve IDs that define the outer boundary of the surface.

Method addSurface creates a flat surface that can have holes in it. The parameter holes is a list of lists of curve IDs and define inner holes on the surface.

Conversely, addRuledSurface and addStructuredSurface make curved surfaces. The difference between the two is that addStructuredSurface takes a list of four structured curves, while addRuledSurface can use three or four curves that do not have to be structured.

The parameter marker is used for specifying in which region an entity is. When GmshMesher meshes geometry in 2D, apart from adding nodes to bdofs, elements in this region will be placed in the list elementMarkers so that users may look up which marker a given element has. Users can use this information to set different properties (such as thickness or thermal conductivity) to elements in different regions.

- addVolume(outerSurfaces, holes=[], ID=None, marker=0)
- addStructuredVolume(outerSurfaces, ID=None, marker=0)

These methods create volumes from lists of surface IDs. The parameter outerLoop is a list of surface IDs that define the outer boundary of the volume.

The differences between the two functions are that addVolume can make arbitrary volumes with internal cavities (holes), while addStructuredVolume must use six structured surfaces and may not have holes. I.e. it creates hexahedral "super elements".

Parameter *marker* is used for specifying in which region an entity is. When GmshMesher meshes geometry in 3D, elements in this region will be placed in the list elementMarkers so that users may look up which marker a given element has.

- setPointMarker(ID, marker)
- setCurveMarker(ID, marker)
- setSurfaceMarker(ID, marker)
- setVolumeMarker(ID, marker)

These methods let the user specify markers after entities have been constructed.

- removePoint(ID)
- removeCurve(ID)
- removeSurface(ID)
- removeVolume(ID)

These four methods remove the entity with the given ID. As mentioned, all curves share the same ID pool. Same for surfaces and volumes.

- getPointCoords(IDs=`None`)

This method returns a list of coordinates of the points specified in IDs, which is a list of integers. Each row of the returned list contains another list with the x, y, and z coordinates of the point.

- pointsOnCurves(IDs)
- stuffOnSurfaces(IDs)
- stuffOnVolumes(IDs)

These methods get the geometric entities that exist on some other entity. For example, pointsOnCurves(5) returns a list containing the point IDs of the points that define curve 5.

The oddly named stuffOnSurfaces and stuffOnVolumes return more lists. The IDs of every subentity is returned. For example, stuffOnVolumes(8) returns three lists with IDs of the subentities that make up volume 8: One list of points, one list of curves, and one list of surfaces.

Internals

Attributes

GeoData has an attribute is3D which is `False` by default, but is set to `True` if addPoint is called with the parameter coord as a list of three values.

Internally, when any of the add* methods are called, the parameters are processed (to make sure they have valid values) and inserted in one of the object attributes points, curves, surfaces, or volumes. These attributes are dictionaries containing nested lists. The keys to these dictionaries are the entity IDs.

The purpose of representing geometric attributes in this way is to make the meshing code in GmshMesher cleaner and easier to read. In one way this is bad design, because much of the logic of the routines that write geo-files is built into the structure of GeoData.

For example, the values in the dictionary points are lists that look like:

```
[[x, y, z], elSize, marker]
```

I.e. a point is a list of three elements; A list of float coordinates, a float element size, and an integer marker. Note that the names are not attribute names, only comments that make it easier to remember which indices represent what.

Similarly, curves is a dictionary whose elements look like:

```
[curvTypestring, [p1, p2, ... pn], marker, elementsOnCurve,
distributionString, distributionVal]
```

In this case curvTypestring is a string with the name of the type of curve, such as "Spline" or "Circle". This value is used when GmshMesher writes geo-files. The second value is a list of the point IDs that define the curve. The third value is the marker of this curve. It is used by GmshMesher when it writes "Physical Lines" in geo-files. The fourth value, elementsOnCurve, is an integer that says how many elements are placed along the curve. This and the following values are `None` if the parameter elOnCurve in the add* method is not defined (this means the curve is not structured). DistributionString is either "bump" or "progression" and defines how elements are distributed on a structured curve. The last value, distributionVal, is the number of elements along the structured curve.

Next up is the dictionary surfaces. Its values are:

```
[SurfaceTypeString, [c1, c2 ... cn], [[c1, c2 ... cm], ... [c1, ... ck]],
ID, marker, isStructured]
```

SurfaceTypeString is the name of the surface type, which can be "Plane Surface" or "Ruled Surface". These are the types of surface that can exist in Gmsh geo-files. Note that they are not the same as the three types of surfaces that can be added (addStructuredSurface and addRuledSurface both make "Ruled Surface", but have different requirements - a structured surface needs four structured boundary curves while ruled surface can have three or four

curves of any type). The second and third values are lists of integer curve IDs. One is a simple list of the curves that make the boundary, while the other is a nested list of curves that make holes in the surface. Only plane surfaces (created with addSurface) may have holes. Values four and five are the ID and marker numbers of the surface. The last value, isStructured, is a boolean that GmshMesher uses to determine whether a surface is structured or not.

Finally, the dictionary volumes looks like:

```
[[s1, s2 ..], [[s1,s2...],[s1,s2..],..], ID, marker, isStructured]
```

This is entirely analogous to the previous dictionary, surfaces, except volumes do not have a name-value because GmshMesher only writes one type of volume (which can be structured or not).

Methods

The functions _getNewPointID, _getNewCurveID, _getNewSurfaceID, _getNewVolumeID, and _smallestFreeKey find the a new (free) ID number if a geometric entity is added without an ID. Note that _smallestFreeKey is only called if, for example, we are adding a new curve and _curveIDspecified = `True`. This means that some curve has been given an ID number by the user and we can not simply increment _nextcurveID to find our next ID number, since it might be taken. The function _smallestFreeKey must thus search the curves dictionary for an unused ID number (key).

The method _checkIfProperStructuredQuadBoundary is called from inside addStructured-Surface. It takes a list of curve IDs (integers) as parameter and finds whether the number of elements corresponding curves are correct. There must be four curves arranged in a loop, like a square, and the number of elements (i.e. distributionVal) on two opposite curves must be the same.

The methods _addCurve, _addSurf, and _addVolume are called from the adder methods. They decrease the amount of duplicate code by handling things that must be done whenever an entity is added, such as requesting a free ID number or making sure that the input is correct. It is in these methods that entries are inserted into the dictionaries points, curves, etc.

_subentitiesOnEntities and _subentityHolesOnEntities are helper methods for the methods pointsOnCurves, stuffOnSurfaces and stuffOnVolumes. They get a Set of the ID numbers of all the "sub-entities" of a set of entities. For example, all the curves on the boundaries of a set of surfaces. The parameters are IDs, entityDict, and index. IDs is a list or set of IDs. Parameter entityDict is a reference to one of the dictionaries that hold the geometric entities. As mentioned, geometric entities are nested lists stored in dictionaries. The parameter index should be the index where sub-entities are stored (for example the points of a curve).

3.3 pycalfem_mesh

Like the previous module pycalfem_mesh contains a single class, GmshMesher. Apart from the constructor, GmshMesher has only one method of interest to the user. That method is create, which executes Gmsh in order to mesh the geometry.

GmshMesher writes the geometry from GeoData objects into geo-files, which is the file format that Gmsh parses to create geometry. After Gmsh is done meshing the geometry it writes a `msh`-file. This `msh`-file is then parsed by GmshMesher to make the mesh in the format CALFEM for Python uses (i.e. arrays/dictionaries called coords, edof, dofs, bdofs, etc).

GmshMesher does some preprocessing of input geometry to make things easier for the user (compared to writing geometry directly in a Gmsh geo-file). For example, by automatically writing line loops and flipping the directions of curves so that all 2D surfaces point in the positive z-axis. This is important because elements that point the wrong way cause faulty FEM calculations that are difficult to diagnose.

GmshMesher has a few "hidden" attributes. They are called nodesOnCurve, nodesOnSurface, and nodesOnVolume. These attributes are defined when create is called (i.e. when a mesh is created). As their names imply they are dictionaries that say which nodes are on a given curve, surface, or volume. These attributes may change or be removed in future versions of pycalfem_mesh. There is no equivalent attribute nodeOnPoint for points.

Attributes

GmshMesher objects have a the following "public" attributes:

- geoData
  A GeoData object or a path string.
- elType
  Integer that defines which element type the mesh will have. Some of these are:
  2 - 3-node triangle,
  3 - 4-node quadrangle,
  5 - 8-node hexahedron,
  16 - 8-node second order quadrangle
  The complete list is in the Gmsh manual, page 89.[7]
- elSizeFactor
  Element sizes are multiplied by this number.
- dofsPerNode
  Integer. Number of degrees of freedom per node.
- gmshExecPath
  Path string to the location of gmsh.exe. Both relative and absolute paths are accepted. If `None` the script will look in the PATH environment variable. (Optional)
- clcurv
  If `True`, this tells Gmsh to try to make the mesh denser at boundaries with high curvature. Experimental feature of Gmsh. (Optional)

- minSize
  Minimum size of elements. (Optional)
- maxSize
  Maximum size of elements. (Optional)
- meshingAlgorithm
  String that informs Gmsh which meshing algorithm to use. ”meshadapt”, ”del2d”, ”front2d”, ”del3d”, ”front3d”, etc. (Optional)
- additionalOptions
  A string with any additional command line options for Gmsh. This string is simply added to the command that executes Gmsh. (Optional)

These are all set in the constructor.

Methods

The only method in GmshMesher that is public is

- create(is3D=`False`)

After GmshMesher has been initialised create can be called to mesh the geometry.

The optional parameter is3D only needs to be set in one scenario; When meshing a 3D model and the attribute geoData is a text string instead of a GeoData object. In this case the string should be the path to a `geo-file`. Normally GmshMesher can ask the GeoData object whether it is a 2D or 3D model.

create executes Gmsh and processes the `msh`-file it writes. The function returns the following values which, apart from the last one, are the same as CALFEM for Python function trimesh2D returns.

- coords
  Array of node coordinates. One row per node.
- edof
  Array with element topology. One row per element.
- dofs
  Array of node dofs. One row per node.
- bdofs
  Boundary dofs. Dictionary containing lists of dofs for each boundary marker. Dictionary key is marker number.
- elementmarkers
  List of integer markers. Row i contains the marker of element i. Element markers can be used to determine in which region an element lies.

Internals

The meat of the class GmshMesher is the method create which runs Gmsh, processes its `msh`-file output and returns the mesh data. What it does not do is convert geometric data in GeoData into the geo-file that Gmsh reads. This is done by the method _writeGeoFile and six other helper methods.

Whenever the ID numbers of geometric entities are written in the geo-file, they need to be converted. The issue is that the IDs represent indices that start at 0, while the corresponding enumeration in the geo-file starts at 1. Curves may even be referenced by *negative* indices (Curve directions are important to Gmsh, and a negative number means the curve direction is reversed). The conversions are handled by _offsetIndices(lst, offset) and _formatList(lst, offset), which increase/decrease indices and turn a list of integers into a corresponding string (such as `[2,4,5]` to "2, 4, 5"), respectively.

The helper function _insertInSetDict(dictionary, key, values) simply finds a set in dictionary with the key and adds the values to it. The helper function is used in both create and _writeGeoFile.

The method _writeGeoFile goes through each entity type (points, curves, surfaces, and volumes) in the GeoData object and writes, in a geo-file, a script that will create the same entities in Gmsh.

Surfaces are particularly difficult, and subsequently have a large portion of the code dedicated to them, such as the helper methods _writeLineLoop and _makeCounterClockwise. The big issue is that surfaces are defined by the loops of curves that are their boundary. Gmsh requires that the *direction* of the curves (as defined by their start- and end-points) go the same way along the entire loop. In geo-files the loops are written as a list of curve IDs, and the direction may be flipped by putting a minus-sign in front of the ID-number of the curve. The problem with this is twofold. One, this kind of complexity is contrary to simplicity that is expected from CALFEM. Two, it is simply impossible to "flip" a curve with a minus sign in Python because the curve can have ID$= 0$, and $0 = -0$. The solution is to pre-process the list of curves that define a surface so that the curves form a correct loop. This is what _writeLineLoop does. It makes the indices start at 1 instead of 0. Then it goes through the list of curve IDs, flipping (multiplying by -1) some of them based on their start- and end-points.

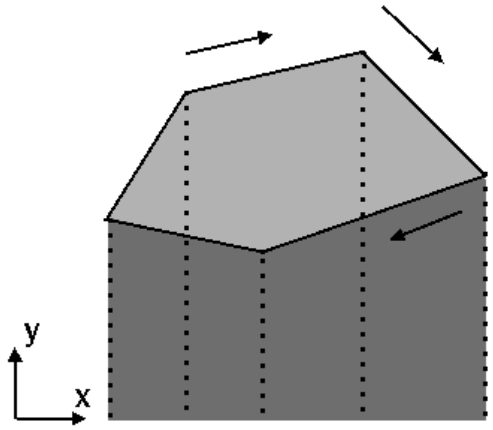


Figure 3.5: Area under a loop. The arrows point from the start point to the end point of the lines. If a line points right ($x_{start} < x_{end}$) the area under the curve is positive, otherwise negative.

All this is still not enough. If the geometry is 2D, then the curve loops must also be *counter-clockwise*, or the surface normals will point in the negative z-axis. This would reverse the node order of the elements, which in turn would ruin FEM calculations. The method _makeCounterClockwise reverses the direction of a loop if it is clockwise by multiplying the curve IDs by -1. It determines the direction of the loop by using a simple trick.[†] The trick consists of calculating the sum of the signed area under the curves of the loop. If the area is positive then the loop is clockwise. The method only works for polygons with straight lines, so splines are approximated with straight lines between their control points and ellipses/circles are broken down into two straight lines as described in figure 3.6. The area under a line from (x_{start}, y_{start}) to (x_{end}, y_{end}) is $(x_{end} - x_{start}) \cdot (y_{end} + ystart)/2$. The minus sign means that lines that go left give positive areas, while lines that go right give negative areas. As figure 3.5 demonstrates, the summed area will be positive if the loop is clockwise.

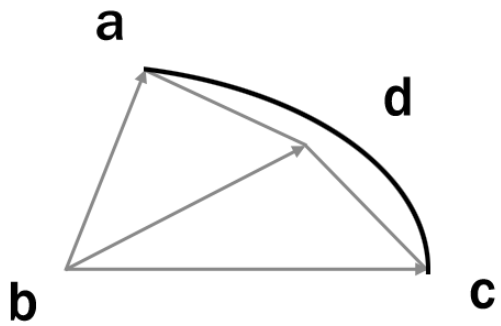


Figure 3.6: Approximation of an ellipse segment as two straight lines **ad** and **dc**. The points a, b, and c are know and d is calculated. Vector **bd** bisects the angle **abc**. The length of **bd** is the mean of the lengths of **ba** and **bc**.

[†]Trick found on `http://stackoverflow.com/questions/1165647/`

3.4 pycalfem_vis

The module pycalfem_vis contains functions that draw geometry or meshes. Unlike the visualisation tools in standard CALFEM for Python, pycalfem_vis does not use a class of its own to handle drawing. Instead, it uses the graphics library visvis.

The module has a number of functions for drawing, and can draw geometry, meshes, deformed meshes, meshes coloured by nodal values, and meshes coloured by element values. It is also possible to add texts and labels.

The module contains no public classes, but a number of functions. After these functions have been called in a script, it is necessary to make the script enter the application loop of a GUI back-end. If this is not done the figures that have been plotted will disappear when the script terminates. This can be done by adding the following at the end of the script (assuming visvis is imported as vv):

```
app = vv.use()
app.Run()
```

This makes visvis find a GUI back-end and enter its application loop. It is also possible to do calculations and plot while inside an application loop, as in example `Mesh_Ex_09.py`.

Functions

All the functions have a parameter called axes. Axes objects in visvis represent spaces wherein drawable objects exist. Meshes and labels must be added to an Axes to be drawn. If the parameter axes is `None` the current Axes will be used automatically.

- getColorbar(axes=`None`)

The function getColorbar returns the Colorbar object, if there is any. This lets users access the label attribute of the Colorbar.

- addLabel(text, pos, angle=0, fontName=`None`, fontSize=9, color='k', bgcolor=`None`, axes=`None`)
- addText(text, pos, angle=0, fontName=`None`, fontSize=9, color='k', bgcolor=`None`, axes=`None`)

These functions add labels and texts to a figure/axes. Labels are placed in screen space while Texts exist in world space. Parameter pos is the xyz-position of the addition. Texts in 2D and Labels use 2D coordinates.

Parameter angle is the counter-clockwise rotation of the text in degrees. Parameters fontName, fontSize, and color define the look and size of the text, while bgcolor sets the background colour behind it. The fontName may be "mono", "sans" or "serif".

Colours in visvis use the same syntax as MATLAB, i.e. black is "k" and red is "r" and so on. It is also possible to define colours as 3-tuples with RGB-values between 0 and 1. For example, green is $(0, 1, 0)$.

- drawMesh(coords, edof, dofsPerNode, elType, axes=`None`, axesAdjust=`True`, title="Mesh", color=(0,0,0), faceColor=(1,1,1), filled=`False`)
- drawNodalValues(nodeVals, coords, edof, dofsPerNode, elType, clim=`None`, axes=`None`, axesAdjust=`True`, doDrawMesh=`True`, title="Node Values")
- drawElementValues(ev, coords, edof, dofsPerNode, elType, displacements=`None`, clim=`None`, axes=`None`, axesAdjust=`True`, doDrawMesh=`True`, doDrawUndisplacedMesh=`False`, magnfac=1.0, title="Element Values")
- drawDisplacements(displacements, coords, edof, dofsPerNode, elType, nodeVals=`None`, clim=`None`, axes=`None`, axesAdjust=`True`, doDrawUndisplacedMesh=`True`, magnfac=1.0, title=`None`)

These functions draw meshes in various ways. Most parameters are the same in these functions.

Parameter coords is an array of node coordinates, each row has the xy- or xyz-coordinates of a node. Parameter edof contains the element degrees of freedom. One row per element. Parameters dofsPerNode and elType are the number of dofs per node and the Gmsh element type, respectively. If axesAdjust is set to `False` the view will not be altered to fit the objects added to the scene, which it does by default. Parameter title is the title label drawn at the top of the figure.

Other parameters only exist in some of the functions. Parameters color and faceColor define the colours of the mesh edge wire and the faces between the edges. The boolean parameter filled decides whether faces are drawn or not. Parameter clim is a 2-tuple containing the minimum and maximum values of the Colorbar.

If doDrawMesh is set to `False` the edge wire will not be drawn. Parameter ev is a a list or array of element values. One value per element. Parameter displacements is an N-by-2 or N-by-3 array. Row i contains the x, y, z displacements of node i.

If doDrawUndisplacedMesh is `True` the mesh wire of the undeformed mesh will be drawn under the deformed mesh. Parameter magnfac is a magnification factor by which displacements are multiplied. It can be used if displacements are too small to be seen.

- drawGeometry(geoData, axes=`None`, axesAdjust=`True`, drawPoints=`True`, labelPoints=`True`, labelCurves=`True`, title=`None`, fontSize=11, N=20)

This function draws the geometry contained in the GeoData object geoData. It will not work if GeoData is a path string. Only points and curves are drawn, not surfaces or volumes.

Most parameters are the same as in the above functions. The exceptions are drawPoints, labelPoints and labelCurves which determine whether points and curves are drawn and labeled, respectively.

Curves are labeled like "$a(b)[c]$", where a is the curve ID, b is the number of prescribed elements on the curve (if the curve is structured) and c is the marker of this curve. To reduce clutter, b and c are not written out if the curve is not structured or does not have a marker.

Points are similarly labeled "$a[c]$".

Internals

As mentioned, pycalfem_vis uses the visualisation library visvis. The functions drawMesh, drawNodalValues, drawElementValues, and drawDisplacements draw the mesh by creating a visvis Mesh object m and putting it in the Axis. If no Axis is supplied in the parameters the current Axis will be used or one will be created. A Mesh object represents a 3D polygonal mesh. These four functions call the helper function _preMeshDrawPrep which does various preparations for creating the Mesh. For example, it converts the coordinates of the nodes to 3D if they are 2D. It also determines whether the mesh has three-sided or four-sided faces. If the elements are 3D, it will also breaks these into their component faces.

Mesh objects have several attributes. Apart from vertices and faces they have attributes that control the way the mesh is rendered, such as faceShading, edgeShading, specular, etc.[†] In all the draw functions these properties are set so that the mesh is drawn without shadows or other lighting effects. This also includes setting the Axis.light0[*] attributes ambient= 1.0 and diffuse= 0.0. These control the level of ambient and diffuse lighting in the scene. 0 diffuse means there are no shadows.

The function drawElementValues does not create a Mesh. Instead it uses a custom object called _elementsWobject to represent the mesh. The reason for this is that Mesh objects are drawn with vertex colours, i.e the colours on the faces are interpolated from the colour values of the mesh vertices/nodes. Element values need to be drawn as a single colour per face/element, which means a custom drawing routine is required.

_makeColorBar is a helper function that creates a visvis Colorbar object. Colorbars in visvis represent the colour scale that shows the mapping between node/element values and colours. For unknown reasons it is possible to create several overlapping Colorbars in visvis, so the function getColorbar (which is called by _makeColorBar) finds the current active one if it exists in order to sidestep this issue.

Curves and drawGeometry

The function drawGeometry draws the points and curves of a GeoData object. To aid it, it uses a number of helper functions called _catmullspline, _bspline, _circleArc, and _ellipseArc. As their names imply they return curves that can be plotted.

The "splines" in Gmsh are more precisely Catmull-Rom splines. In order to draw geometry the way it will be interpreted by Gmsh it is necessary to construct a parametric representation of these splines. This is done in the function _catmullspline, which takes a list of control points as parameter. The function starts by creating extra control points at the start and end of the list. This is necessary because Catmull-Rom splines are calculated with the following formula.

[†]The Mesh class is described at `http://code.google.com/p/visvis/wiki/cls_Mesh`

[*]The Axis class is described at `http://code.google.com/p/visvis/wiki/cls_Axes`

$$q(t) = 0.5 \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix} \tag{3.1}$$

where $q(t)$ is a parametric point $[x(t), y(t), z(t)]$ on the curve. t is a parameter between 0 and 1. P_i is a control point and i goes from 1 to $N-2$ if N is the number of control points.†

This formula creates a curve $q(t)$ which connects control points P_i and P_{i+1}. The other two points determine the tangent of the curve at these points. The whole curve is calculated by applying this formula to every group of four point in the list of control points.

Note that the curve starts at the second point and ends at the second-to-last point. This means an extra set of points are needed on the ends. These two points are created by either mirroring the second control point in the first or by duplicating the second-to-last point, depending on whether the curve is open or closed.* See figure 3.7.

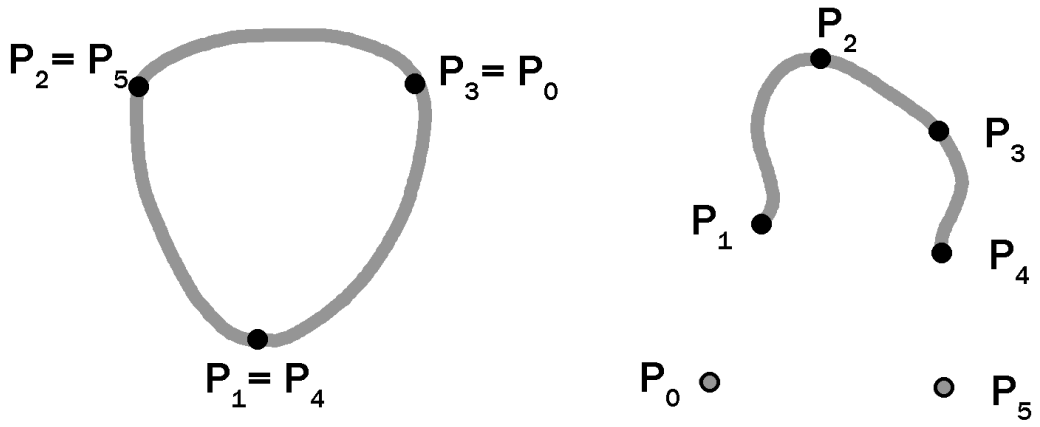


Figure 3.7: Creation of extra control points for Catmull-Rom splines. The example on the left is a closed spline with four control points ($P_1 - P_4$), so the extra points P_0 and P_5 are created by duplicating the second points at either end and appending them to the other end. The example on the left also has four control points, but the spline is open. In this case the extra points are created by mirroring the second points in the first (or second-to-last mirrored in the last) and appending the mirrored points.

The function _bspline works similarly as _catmullspline. It returns a uniform cubic B-spline. This type of curve also needs an extra set of points at the ends to look like they do in Gmsh. The extra points are determined in exactly the same way as before.

A parametric representation of the B-spline is calculated with the formula

†This formula can be found at `http://www.mvps.org/directx/articles/catmull/`

*This idea was taken from `http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-curve-closed.html`

$$q(t) = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix} \tag{3.2}$$

where P_i is a control point and i goes from 1 to $N-2$ if N is the number of control points. s is a parameter between 0 and 1.[‡]

Ellipse arcs and circle arcs are calculated by the function _ellipseArc. In Gmsh ellipse arcs are defined by four points; start point, center point, end point, and a point on the major axis of the ellipse (due to symmetry circle arcs do not need the last point). This posed an unexpectedly large obstacle during development, because no formula for determining arbitrary ellipse arcs from these points could be found. However, the problem could be solved in the case of 2D axis-aligned ellipses. Therefore, _ellipseArc starts by transforming these points to a 2D axis-aligned ellipse. See figure 3.8.

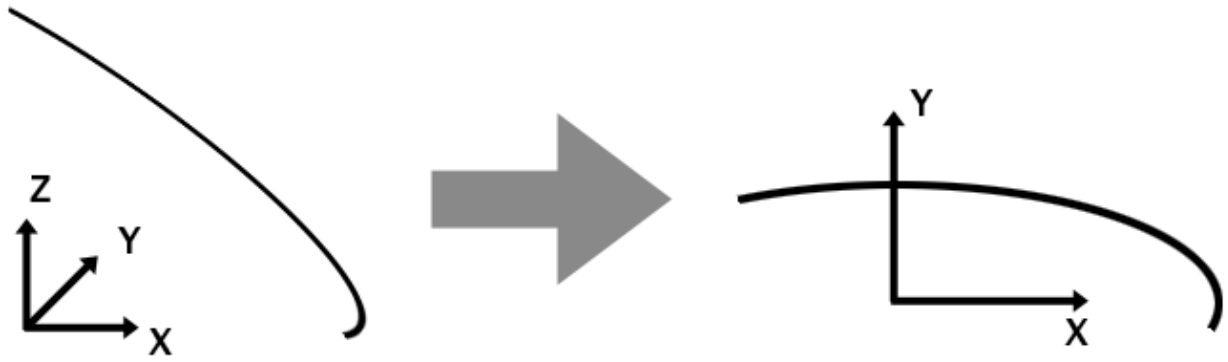


Figure 3.8: The four points that define an ellipse arc are transformed to a 2D coordinate system where the ellipse is axis aligned.

At this point we want to determine the parameters a and b (the lengths of the major and minor axes) of the ellipse equation

$$x = a \cdot cos(t) \tag{3.3}$$
$$y = b \cdot sin(t) \tag{3.4}$$

where $0 < t < 2\pi$. This is done with

[‡]See `http://www.siggraph.org/education/materials/HyperGraph/modeling/splines/b_spline.htm`

$$a = \sqrt{\frac{(y_e \cdot x_s)^2 - (x_e \cdot y_s)^2}{y_e^2 - y_s^2}} \tag{3.5}$$

$$b = \sqrt{\frac{(y_e \cdot x_s)^2 - (x_e \cdot y_s)^2}{y_e^2 - y_s^2} \cdot \frac{x_e^2 - x_s^2}{y_s^2 - y_e^2}} \tag{3.6}$$

.

where (x_s, y_s) is the start point and (x_e, y_e) is the end point of the ellipse segment. If memory serves, these equations were determined by feeding the ellipse equation (and some conditions like $-\pi < t < \pi$) into Wolfram Alpha and cleaning up the resulting equations.

We then determine a range for the parameter t with

$$t_s = atan2(\frac{y_s}{b}, \frac{x_s}{a}) \tag{3.7}$$

$$t_e = atan2(\frac{y_e}{b}, \frac{x_e}{a}) \tag{3.8}$$

where t_s is the value of t at the start point, and t_e is t for the end point. The function $atan2(x, y)$ gives the angle from the negative x-axis (between $-pi$ and pi) of a vector (x, y). However, since the limits of t is $-pi$ and pi, we do not yet know whether the shortest "distance" between t_s and t_e is direct or if it crosses the discontinuity at the limit. We find out by simply calculating the parameter distances and choosing the shortest distance. See figure 3.9.

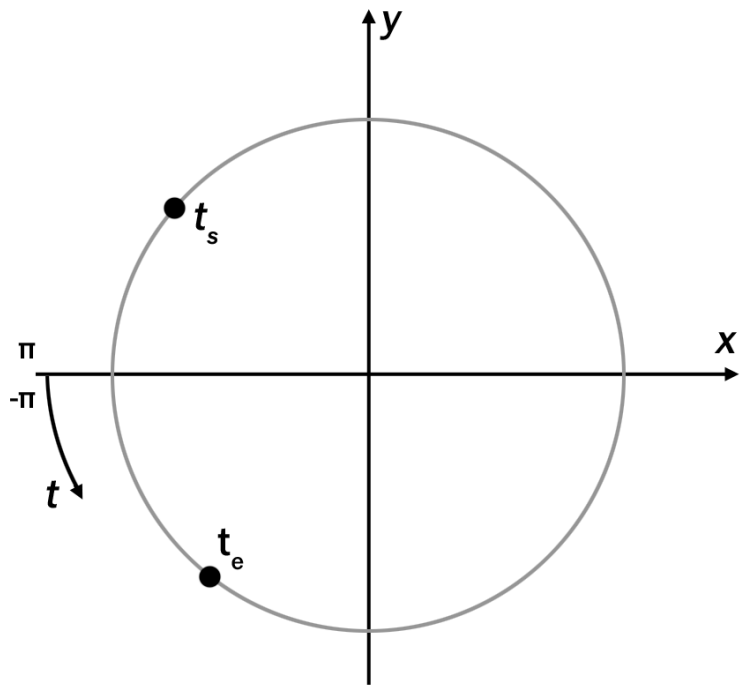


Figure 3.9: The parameter t determines a point on an ellipse in a similar way an angle determines a point on a circle (but not quite the same way). The shortest distance between two points can cross the discontinuity at $t = \pm\pi$ like in this example.

This gives us the range of t that makes the ellipse. We pass values t from this range to the ellipse equation (equations (3.3) and (3.4)) which gives us vertices on the ellipse. This is where the ellipse segment lies in this 2D coordinate system, so we transform the vertices on the curve back to the original coordinate system.

3.5 Example scripts

A handful of example scripts were written. These demonstrate how to use the new modules. As of writing there are ten scripts named `Mesh_Ex_*.py`, where `*` is a number between 01 and 10. These are all in the appendix. Most of the examples are fairly similar to each other and do not have to be read in order, but the later examples are less thoroughly commented.

3.6 Other changes to CALFEM for Python

Functions planqe and planqs were added to the existing pycalfem module. These functions calculate the stiffness matrix and element stresses/strains of quadrilateral elements. The

function plani4e[†], was also added.

[†]Converted from CALFEM for MATLAB by Eskil Andreasson and given to me in private communication

Chapter 4

Performance test

In order to test the performance of the mesher modules some experiments were conducted. The purpose was to see how fast the mesher could mesh different types of meshes and compare performance with the older trimesh2d routine, which uses `Triangle.exe` to mesh.

The experiments were done with various different values for the GmshMesher parameter elmSizeFactor, alternatively the parameter maxArea for trimesh2d().

The tables below show the number of elements in the mesh and the mean time to mesh. Due to the long mesh times the mean values were calculated from just three repetitions. The table columns are *Elements* (the number of elements), *mesh time* (the total meshing time in seconds), *Gmsh* (seconds spent waiting on Gmsh to mesh the geometry and write a msh-file), *parse .msh* (time used for parsing the textttmsh-file), and finally *parse/Gmsh ratio* (A procent ratio of the two previous times).

The first three tables (4.1, 4.2, 4.3) describe unstructured meshing in 2D. By comparing the mesh times in table 4.2 and table 4.3 we see that GmshMesher is slightly slower than than the function trimesh2d. We can also see in table 4.2 that the amount of time spent parsing the output of Gmsh is about the same as the time Gmsh took to mesh the geometry. Comparing this to table 4.1 shows that meshing quadrangular elements takes more time than meshing triangles (about 10 times longer at 1 million elements), and that most of the time is used by Gmsh. This is not surprising since the running time of the parsing algorithm is roughly linearly proportional to the number of elements, and thus does not change with element type. In Gmsh quadrangles are created by merging triangles, which is apparently a slightly time-consuming process.

The test was also done on a structured mesh. The mesh had the same shape as the previous tests (a flat square), but the elements were distributed in a grid pattern instead of unstructured distribution. Comparing tables 4.1 and 4.4 hints that structured meshes are made more quickly than unstructured meshes (atleast for 2D quadrangles).

Similar tests were also done on the 3D dice-shaped geometry in `Mesh_Ex_04.py`. In this case the meshed geometry was structured and had hexahedral elements. Again, the mean time was calculated from the mean running times of three repetitions. The results are in table 4.5, which do not seem to show anything conclusive, except that meshing in 3D is slightly slow and that the time spent on parsing the msh-files of 3D meshes is remarkably longer than

2D quads unstructured (Gmsh)				
Elements	mesh time (s)	Gmsh (s)	parse .msh (s)	parse/Gmsh ratio
977	0.45	0.36	0.09	25 %
2,032	0.70	0.54	0.15	28 %
4,194	1.79	1.47	0.31	21 %
8,079	2.40	1.81	0.59	33 %
16,342	5.87	4.67	1.21	26 %
33,361	16.19	13.74	2.44	18 %
66,018	31.74	26.85	4.88	18 %
132,700	82.89	73.07	9.81	13 %
266,112	308.01	288.03	19.96	7 %
532,756	650.23	609.78	40.43	7 %
1,064,747	1116.82	1036.29	80.48	8 %

Table 4.1: Meshing times for an unstructured mesh with 2D quadrangular elements. Meshing was done by `Gmsh.exe` by calling create in GmshMesher.

2D triangles unstructured (Gmsh)				
Elements	mesh time (s)	Gmsh (s)	parse .msh (s)	parse/Gmsh ratio
1,018	0.25	0.18	0.07	37 %
2,066	0.34	0.22	0.12	55 %
4,282	0.60	0.35	0.25	73 %
8,486	0.99	0.49	0.49	100 %
16,930	2.06	1.05	1.00	95 %
34,152	3.38	1.44	1.94	135 %
68,658	6.80	2.88	3.92	136 %
137,574	14.25	6.36	7.89	124 %
275,310	28.42	12.43	15.97	128 %
552,702	60.98	26.05	34.91	134 %
1,106,748	124.37	57.91	66.42	115 %
2,212,960	262.29	123.85	138.36	112 %
4,436,214	609.88	300.00	309.73	103 %

Table 4.2: Meshing times for an unstructured mesh with 2D triangular elements. Meshing was done by `Gmsh.exe` by calling create in GmshMesher

2D triangles unstructured (Triangle)	
Elements	mesh time (s)
1,542	0.05
3,272	0.07
6,300	0.11
1,303	0.14
2,540	0.24
5,123	0.42
10,187	0.79
20,417	1.39
40,763	2.88
81,525	5.70
162,798	11.01
325,704	23.11
651,175	44.90
1,302,622	92.24
2,604,770	183.81

Table 4.3: Meshing times for an unstructured mesh with 2D triangular elements. Meshing was done by `Triangle.exe` by calling trimesh2d

for 2D meshes.

Even though the main purpose of these experiments was to test the performance of the mesher, it was noted that the visualisation function drawMesh failed to draw meshes with more than around 1 million faces, possibly due to memory allocation issues.

2D quads structured (Gmsh)				
Elements	mesh time (s)	Gmsh (s)	parse .msh (s)	parse/Gmsh ratio
1,296	0.28	0.16	0.12	73 %
2,401	0.36	0.17	0.19	113 %
4,096	0.52	0.21	0.31	149 %
6,561	0.72	0.22	0.49	223 %
10,000	1.02	0.26	0.75	286 %
14,641	1.40	0.31	1.09	348 %
20,736	1.90	0.38	1.52	398 %
28,561	2.60	0.48	2.11	444 %
38,416	3.45	0.61	2.84	469 %
50,625	4.52	0.75	3.76	501 %
65,536	5.77	0.92	4.85	528 %
83,521	7.32	1.14	6.17	542 %
104,976	9.14	1.40	7.73	553 %
130,321	11.34	1.71	9.62	563 %
160,000	13.85	2.08	11.76	566 %
194,481	16.89	2.50	14.37	575 %

Table 4.4: Meshing times for a structured mesh with 2D quadrangular elements. Meshing was done by `Gmsh.exe` by calling create in GmshMesher.

3D hexahedrons structured (Gmsh)				
Elements	mesh time (s)	Gmsh (s)	parse .msh (s)	parse/Gmsh ratio
1,536	0.53	0.21	0.32	154 %
3,750	1.49	0.39	1.10	284 %
7,776	3.81	0.85	2.95	347 %
14,406	8.95	1.92	7.02	365 %
24,576	19.24	4.04	15.17	375 %
39,366	40.51	9.50	30.95	326 %
60,000	72.54	15.57	56.97	366 %
87,846	127.87	28.39	99.28	350 %
124,416	214.44	46.74	167.34	358 %
171,366	357.56	82.62	274.22	332 %

Table 4.5: Meshing times for a structured mesh with 3D hexahedral elements. Meshing was done by `Gmsh.exe` by calling create in GmshMesher.

Chapter 5

Differences between Gmsh and the meshing modules

Geometry in Gmsh is defined in script files with the extension geo. Other properties that influence meshing are set as parameters when running Gmsh from the command-line. The modules pycalfem_GeoData and pycalfem_mesh emulate these geo-files and properties, in order to let users access some Gmsh meshing functionality from python scripts.

5.0.1 Geometry

Gmsh reads geometry data from geo-files, which are human-readable scripts that tell Gmsh how to construct the geometry. geo-files can be written by hand or generated by Gmsh when a user makes geometry in its graphical user interface. geo-files are simply a series of commands that construct geometry. The python class GeoData has methods that correspond to a subset of these commands.

For example, to define a triangular surface between three points where one side has $marker = 5$ you might write the following in a Gmsh geo-file:

```
Point(1) = {2, 1, 0, 0.3};
Point(2) = {3, 1, 0, 0.3};
Point(3) = {2, 2, 0, 0.3};
Line(1) = {1,2}
Line(2) = {2,3}
Line{3} = {3,1}
LineLoop(1) = {1,2,3}
Plane Surface(1) = {1}
Physical Line(5) = {2};
```

By comparison, with the new modules you would write (in a python script)

```
g = GeoData()
g.addPoint([2, 1], elSize=0.3)
```

```
g.addPoint([3, 1], elSize=0.3)
g.addPoint([2, 2], elSize=0.3)
g.addSpline([0,1])
g.addSpline([1,2], marker=5)
g.addSpline([2,0])
g.addSurface([0,1,2])
```

GeoData is the object that receives commands and stores the geometry.

There are several differences. One of the important ones is that indices start at 1 in Gmsh and 0 in Python. Points in Gmsh have four parameters and an ID-number. The parameters are the x-, y-, z-positions of the point and element size at the point. In GeoData, the Point has a different set of parameters, some of which are optional. These are *position, ID, marker*, and elSize.

There are different types of curves in Gmsh. In the top example we used `Line`, which is a straight line between two points. Since a spline with two points is functionally the same as a straight line, there are no lines in GeoData. Instead splines are used. The remaining Gmsh curve types (`BSpline, Circle, Ellipse`) are defined similarly in both Gmsh and GeoData.

Whereas in Gmsh markers are applied with the command `Physical Line(5) = {2}`, in GeoData markers are set when the entity is added with the parameter marker.

In Gmsh it is necessary to define `LineLoop`s. `LineLoop`s are used for specifying the surface boundary when surfaces are created. In GeoData there are no line loops. Surfaces use lists of curves instead (the parameters outerLoop and holes of addSurface, for example).

`Plane Surface` in Gmsh corresponds to the method addSurface in GeoData. There is also `Ruled Surface` that corresponds to addRuledSurface. GeoData has a third surface type called addStructuredSurface which is the same as a combined `Ruled Surface` and `Transfinite Surface`.

In 2D, one very important difference is that Gmsh requires that the Line Loops that define surfaces are counter clockwise. Otherwise the surface normals of elements on the surface will point in the negative z-direction, which will ruin calculations. Since that is bothersome the class GmshMesher will automatically reorient surfaces so that the surface is pointing in the correct way.

The curves can be ordered in a clockwise or counter-clockwise order. The direction of the curves themselves (as defined by the order of the points that make the curve) does not matter - with one exception. The only situation where the order of points in a curve matters is when the curve is "structured" and has "progression" distribution, i.e when nodes are placed in a geometric progression along the curve. In this situation the order of the nodes must be reversed to reverse the direction of the progression.

Curves are made structured in different ways in Gmsh and GeoData. For example, in Gmsh we could create a structured four-sided surface with the following script:

```
Point(1) = {0, 0, 0, 1};
Point(2) = {1.2, 0, 0, 1};
Point(3) = {1, 1.3, 0, 1};
Point(4) = {0, 1, 0, 1};
Spline(1) = {1, 2};
Transfinite Line{1} = 11 Using Bump 0.2;
Spline(2) = {2, 3};
Transfinite Line{2} = 21 Using Progression 1.1;
Spline(3) = {3, 4};
Transfinite Line{3} = 11 Using Bump 0.2;
Spline(4) = {1, 4};
Transfinite Line{4} = 21 Using Progression 1.1;
Line Loop(1) = {1, 2, 3, -4};
Ruled Surface(1) = {1};
Transfinite Surface{1} = {1, 2, 3, 4};
```

In Python we would instead write:

```
g = GeoData()
    #Add Points:
g.addPoint([0,0])
g.addPoint([1.2, 0])
g.addPoint([1, 1.3])
g.addPoint([0, 1])
    #Add Splines:
g.addSpline([0,1], elOnCurve=10, elDistribType="bump",        elDistribVal=0.2)
g.addSpline([1,2], elOnCurve=20, elDistribType="progression", elDistribVal=1.1)
g.addSpline([2,3], elOnCurve=10, elDistribType="bump",        elDistribVal=0.2)
g.addSpline([0,3], elOnCurve=20, elDistribType="progression", elDistribVal=1.1)
    #Add Surface:
g.addStructuredSurface([0,1,2,3])
```

Gmsh has the command `Transfinite Line` to make a curve structured. In GeoData a curve is made structured by defining the parameter elOnCurve when the curve is added.

For structured surfaces Gmsh has the command `Transfinite Surface`, which turns a surface into a structured surface. In GeoData there is a method called addStructuredSurface which both creates a surface and makes it structured (The surface must have four edges).

Note that the `Line Loop` in the top example contains curve -4, i.e. the direction of curve 4 is reversed to make the loop. This is not necessary in the corresponding command addStructuredSurface due to the preprocessing mentioned above (Also remember that Python uses 0-based indices, so the curve is 3, not 4, in Python).

5.0.2 How to make quadrangular elements

In Gmsh you would add the following command to a geo-file to make all elements quad-shaped:

```
Mesh.RecombineAll = 1;
```

In Python you set the element type when you create the GmshMesher object that handles meshing. In the example the element type (elType) is 3, which corresponds to quads with four nodes. If a user wants second order quads with eight nodes, they would set elType = 16. To do the same directly in Gmsh would be slightly more complicated.

```
mesher = GmshMesher(geoData = g,
                    elType = 3,
                    dofsPerNode= 1)
```

Chapter 6

Conclusions

The new modules have yet to be tested in the wild, so it remains to be seen if they are beneficial to FEM students and other users. However, it is already possible to make some comparisons between expectations at the start of the project and the results.

One noticeable difference from the initial plan is that the new visualisation functionality did not replace the old. In one way this is unfortunate, since it increases the size of the library and makes the design less similar to CALFEM for MATLAB. As a learning tool it is especially important that the library is simple and does not overwhelm the user with functionality they do not need. On the other hand, having pycalfem_vis separate from the rest of the package means it does not break any existing code. It also means it is possible for a user to to make full use of the capabilities of visvis if they want to.

The plans to convert many MATLAB functions to Python did not happen. However, the conversion process is time-consuming and unchallenging. Therefore it is not particularly well suited to be a major part of a degree project.

Chapter 7

Future Work

There is much work to be done still.

2D line elements have been completely neglected in this project in favour of 3D elements and quads. As a result the mesh module has not been tested for that sort of element, and it is not possible to draw line elements in the new visualisation module. This neglect should be amended.

Many parts of CALFEM for Python assume that the mesh is in 2D or assumes triangular elements. Notably, function applyBC can only apply boundary conditions in the first two dimensions.

There is some visualisation functionality that is still missing from the new visualisation module. For example, it is not possible to draw isolines, element flux as arrows, or principal stresses.

In general, very few MATLAB functions have been converted in this project. This still needs to be done.

Gmsh entities `Compound Line` and `Compound Surface` could be implemented in classes GeoData and GmshMesher. `Compound Line/Surface` allow several curves/surfaces to be treated as a single curve/surface. However, it is doubtful whether the added functionality would outweigh the issue of feature creep.

Chapter 8

Example Case

In this chapter we will go through an example case in depth. The figure below illustrates the example, which consists of a thick square with a stiff shell and flexible centre. The square has a downward force applied on the top.

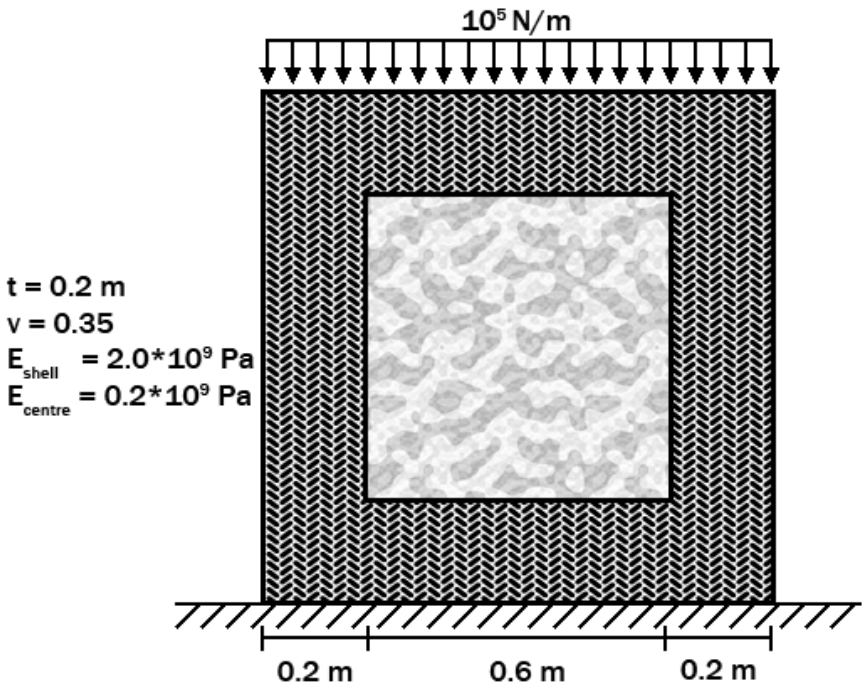


Figure 8.1: The problem case.

8.1 The script

To start with, we import everything we need. In this case we use named imports because we want to be able to see which modules we are calling.

```
import pycalfem_GeoData
import pycalfem_mesh
import pycalfem_vis as pcv
import visvis as vv
from pycalfem import *
from pycalfem_utils import *
```

8.1.1 Defining the geometry

The next step is to define our problem geometry. In ordinary CALFEM, this would consist of two arrays containing points and lines. Here, however, things are slightly more complicated due to the larger variety of curves and surfaces we can make, but for this example we'll keep things fairly simple.

We begin by creating a GeoData object that will hold our geometry.

```
g = pycalfem_GeoData.GeoData()
```

Next, we add geometric points. The function addPoint has four parameters, but only one is non-optional, the coordinates. We add one point for each corner of the squares in figure 8.1. One of the other parameters is ID, which is a number that identifies a point. Since we do not specify IDs, the points will automatically receive IDs starting from 0.

```
g.addPoint([0, 0]) #0
g.addPoint([1, 0]) #1
g.addPoint([1, 1]) #2
g.addPoint([0, 1]) #3
g.addPoint([0.2, 0.2]) #4
g.addPoint([0.8, 0.2]) #5
g.addPoint([0.8, 0.8]) #6
g.addPoint([0.2, 0.8]) #7
```

We add curves that connect our points. There are several types of curves available. In this case we use splines, which pass through all the points listed in the first parameter, which is a list of point IDs. We also set markers for the curves that correspond to the bottom and top of the big square. We will use these markers to apply boundary conditions. Like the points, curves also have an ID parameter that is automatically filled in if unspecified.

```
g.addSpline([0, 1], marker=70)        #0
g.addSpline([2, 1])                   #1
g.addSpline([3, 2], marker=90)        #2
g.addSpline([0, 3])                   #3
g.addSpline([4, 5])                  #4
g.addSpline([5, 6])                  #5
g.addSpline([6, 7])                  #6
g.addSpline([7, 4])                  #7
```

The final part of defining 2D geometry is adding surfaces. The method addSurface has four parameters, outerLoop, holes, ID and marker.

Here we add two surfaces. The first surface is the outer shell of the square. Its outer boundary is defined by the curves 0, 1, 2 and 3. Its inner boundary (or "hole") is defined by the curves 4, 5, 6 and 7. Note that a surface may have many holes, so the parameter hole is a list of lists, rather than a simple list.

The center square is similarly defined. Both surfaces are also given a marker. These markers will be used for setting different material properties to elements in these regions.

```
g.addSurface([0,1,2,3], holes=[[4,5,6,7]], marker = 55)
g.addSurface([4,5,6,7], marker = 66)
```

8.1.2 Meshing the geometry

At this point we may mesh the geometry. This is done by creating a GmshMesher object and calling its create method. GmshMesher has a large number of optional parameters that influence the result of the meshing process. Below we define the necessary parameters. The first parameter is geoData, which is simply the GeoData object we have made (It may also be a string containing a file path to a Gmsh geo-file). The second parameter is the path to the Gmsh executable file. Here, it is undefined so the program will attempt to find Gmsh by looking in the PATH environment variable. Parameter elSizeFactor is a number that multiplies the size of elements (size means the length of the sides of the elements). The final parameters dofsPerNode and elType are the number of degrees of freedom per node and element type. Element type 3 means quadrangular elements.

The method create returns a number of values that define the mesh. If you have used trimesh2d you will be familiar with all of these except elementmarkers. The value coords contains the node coordinates of the mesh. Meanwhile, edof, dofs and bdofs contain the degrees of freedom by element, node, and boundary marker.

```
elType = 3 #Element type 3 is quad.
dofsPerNode = 2 #Degrees of freedom per node.

mesher = pycalfem_mesh.GmshMesher(geoData = g,
                                  gmshExecPath = None,
                                  elSizeFactor = 0.04,
                                  elType = elType,
                                  dofsPerNode= dofsPerNode)

coords, edof, dofs, bdofs, elementmarkers = mesher.create()
```

8.1.3 Solving the problem

Solving problems is done in the same way with or without the new modules from this project. The sole difference is the existence of the aforementioned elementmarkers. However, for com-

pleteness sake we demonstrate how to solve the problem here.

The first step is to define problem constants, such as thickness t, Poisson's ratio v, Young's modulus E, and material matrix D. Since we want different material properties in the hard shell and soft center we need two material matrices D1 and D2. We place these in a Python dictionary which we call Ddict. The keys to the dictionary are the markers that we applied to the two surfaces (55 and 66). This will let us access the correct material matrix of a given element via the array elementmarkers.

We also set the problem type ptype= 1, which means this is a plane stress problem (2 means plane strain). The list ep contains the problem type and thickness, and will be used as a parameter in the next step.

```
t = 0.2
v = 0.35
E1 = 2e9
E2 = 0.2e9
ptype = 1
ep = [ptype, t]
D1 = hooke(ptype, E1, v)
D2 = hooke(ptype, E2, v)
Ddict = {55 : D1, 66 : D2}
```

The second step consists of assembling the stiffness matrix K. First, we get the total number of degrees of freedom, nDofs, from the size of the array dofs and initialise an empty matrix K. We also extract the x- and y-coordinates of the nodes in each element by calling the coordxtr function.

Next, we iterate over each element and add the element stiffness matrixKe to K. Each row of edof, ex, ey, and elementmarkers correspond to the dofs, x-coordinates, y-coordinates, and element markers of an element. These are passed to the function planqe, which returns the element stiffness matrix Ke of a quadrangular solid element. Ke is added to K by the function assem.

```
nDofs = size(dofs)
K = zeros([nDofs,nDofs])
ex, ey = coordxtr(edof, coords, dofs)
for eltopo, elx, ely, elMarker in zip(edof, ex, ey, elementmarkers):
    Ke = planqe(elx, ely, ep, Ddict[elMarker])
    assem(eltopo, K, Ke)
```

We prepare boundary conditions by calling the function **applybc**. The variable bc is an array containing prescribed dofs, while bcVal contains the prescribed values. These are used later.

The function applybc also takes parameter bdofs, which is a dictionary of which dofs belong to which boundary marker. The value 70 is the marker of the boundary we want to apply the boundary condition to. 0.0 is the value we apply. The function also has a parameter

dimension that determines in which direction the boundary condition is applied. The default value is 0, which means "in all dimensions" (1 means x, 2 means y).

The boundary condition we apply here locks the bottom of the square in place.

```
bc = array([],'i')
bcVal = array([],'i')
bc, bcVal = applybc(bdofs,bc,bcVal, 70, 0.0)
```

The force on top of the square is applied with the following commands. First, we initialise an empty force array. Then, we call applyforce to apply the force -10^6 in the y-direction to all nodes with marker$= 90$.

```
f = zeros([nDofs,1])
applyforce(bdofs, f, 90, value = -10e5, dimension=2)
```

We get the solution to the problem by calling solveq. The first return value a is an array containing the displacement of every dof. The second return value r contains the reaction forces.

```
a,r = solveq(K,f,bc,bcVal)
```

We want to measure the effective stress in the deformed object. This can be done in a number of ways. Here, we begin by extracting the element displacements by calling extractEldisp. We also initialise an empty list vonMises, which will hold the effective stress of each element.

In the for-loop we calculate stresses es in the quadrilateral elements with the planqs function. The list es contains normal stresses σ_x and σ_y, and shear stress τ_{xy}. These values are inserted into the von Mises stress equation $\sigma_v = \sqrt{\sigma_x^2 - \sigma_x\sigma_y + \sigma_y^2 + 3\tau_{xy}^2}$, which gives us an effective stress value.

```
ed = extractEldisp(edof,a)
vonMises = []
for i in range(edof.shape[0]):
    es, et = planqs(ex[i,:], ey[i,:], ep, Ddict[elementmarkers[i]], ed[i,:])
    vonMises.append( math.sqrt( pow(es[0],2) - es[0]*es[1] + pow(es[1],2) +
                    3*pow(es[2],2) ) )
```

8.1.4 Visualising the results

We start by drawing the geometry. The function drawGeometry belongs to the pycalfem_vis module, which we imported as pcv. The function has many parameters, but the only necessary one is the first one, which is a GeoData object that holds the geometry to be drawn.

```
pcv.drawGeometry(g, title="Geometry")
```

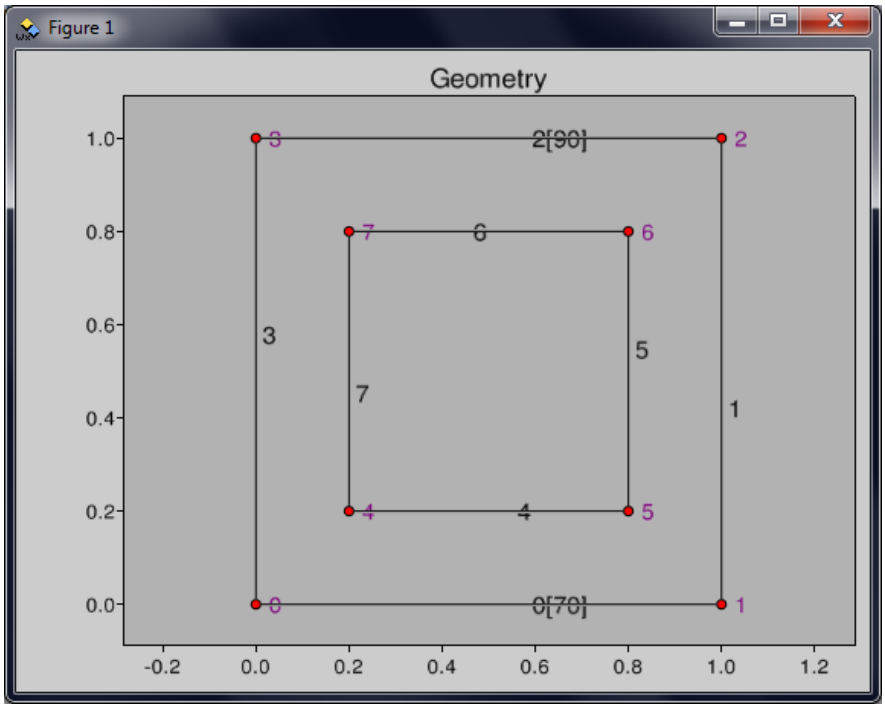


Figure 8.2: The geometry.

Next, we draw the mesh. We want this to be drawn in a new window, so we call figure from the visvis library (which we imported as vv).

The function drawMesh draws the mesh. The parameter coords and edof are the coordinates and element dofs. Reminder: We got these values when we meshed the geometry. The other two parameters were defined right before that.

```
vv.figure()
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
             elType=elType, filled=True, title="Mesh")
```

We also draw the deformed mesh, i.e. the solution to the problem we solved. This is done by calling drawDisplacements. The first parameter is the displacement solution a. The parameter doDrawUndisplacedMesh determines whether the undeformed mesh will be drawn superimposed on the deformed mesh.

```
vv.figure()
pcv.drawDisplacements(a, coords, edof, dofsPerNode, elType,
                      doDrawUndisplacedMesh=False, title="Displacements")
```

Next, we draw the von Mises effective stresses of the elements by calling drawElementValues. Like the other visualisation functions this one has many parameters. The first one is a list (or array) of scalar element values. the variable a belongs to the parameter displacements

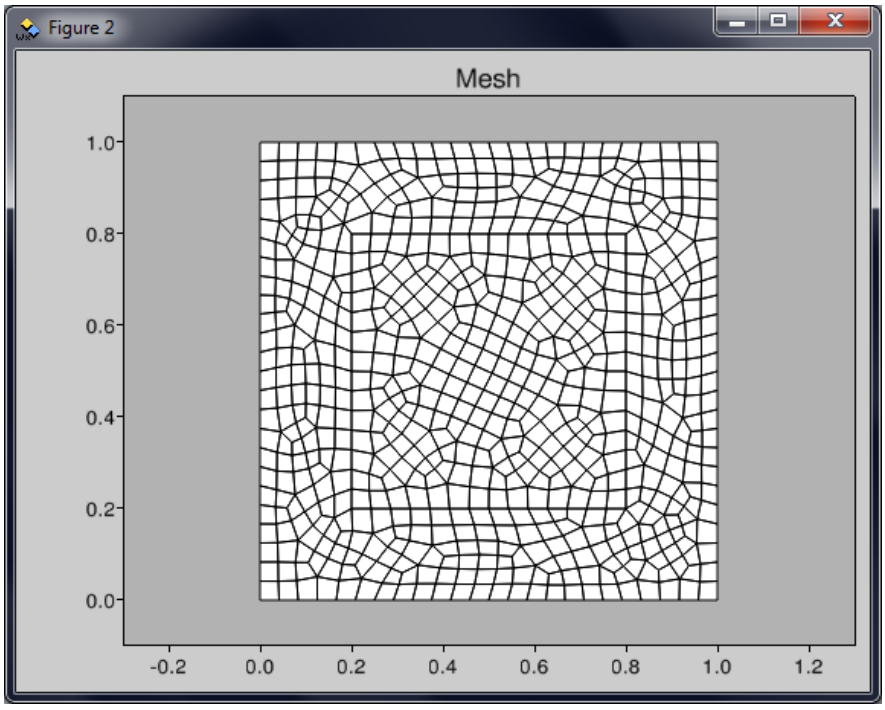


Figure 8.3: The mesh.

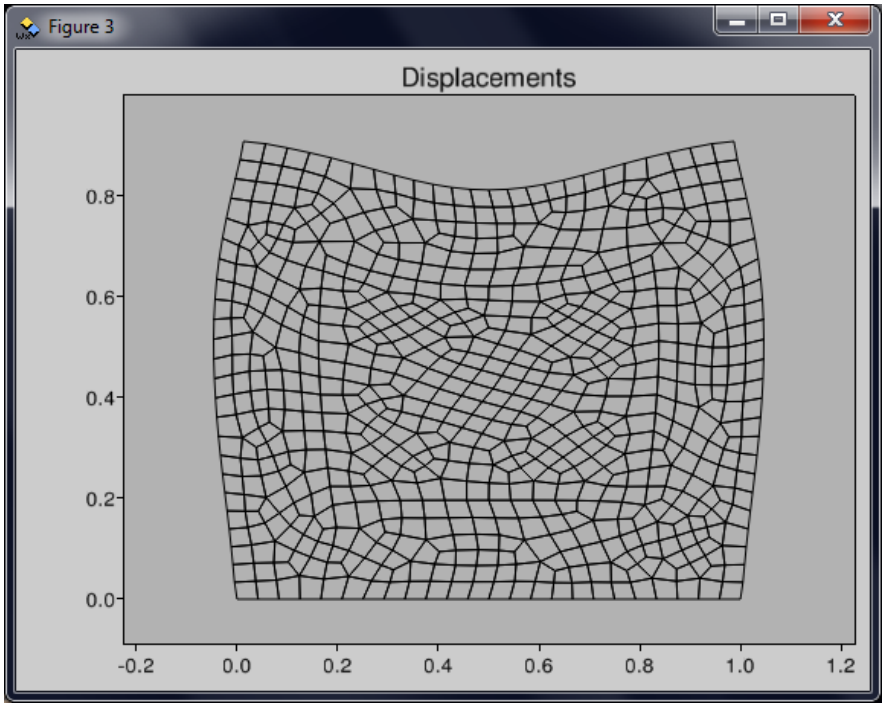


Figure 8.4: The deformed mesh.

which (as the name implies) contains the displacements of the dofs. The boolean parameters doDrawMesh determines whether the black borders around elements are drawn.

The function getColorbar gets the visvis Colorbar object that can be seen on the right of figure 8.5. We use the reference to this object to set its text label to "Effective stress".

```
vv.figure()
pcv.drawElementValues(vonMises, coords, edof, dofsPerNode, elmType, a,
                      doDrawMesh=True, doDrawUndisplacedMesh=False,
                      title="Effective Stress")
pcv.getColorbar().SetLabel("Effective stress")
```

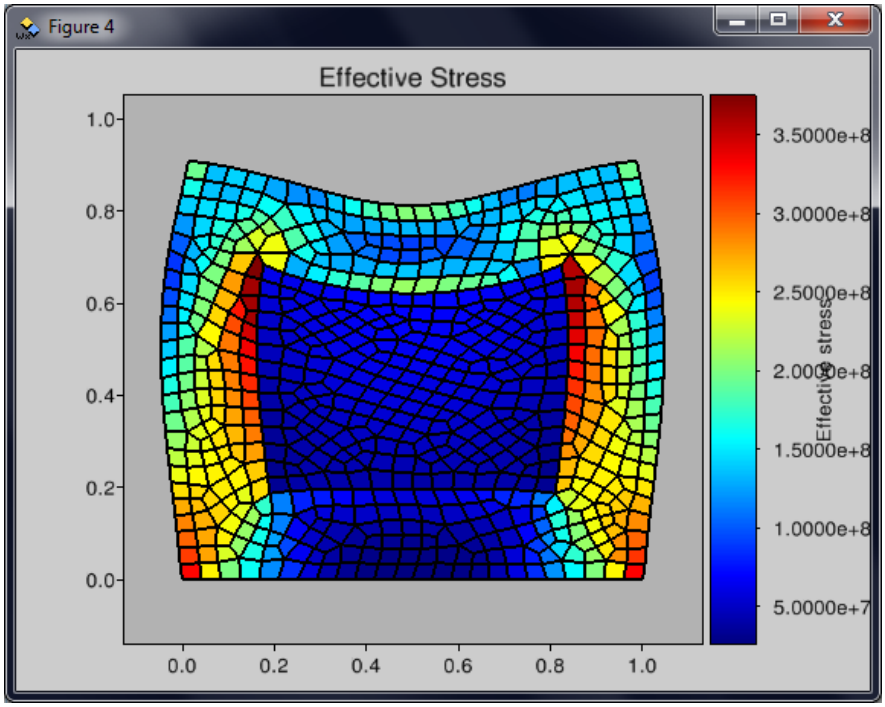


Figure 8.5: The deformed mesh with von Mises effective stress.

Finally, we start the application loop of the graphical user interface that houses our figures. If we did not do this step, the figures would disappear when the script terminates. The visvis function use automatically gets a Python GUI backend (such as wxPython or PyQt if they exist on the computer), and Run enters its main loop. See `Mesh_Ex_09.py` for an example on how to embed visvis figures in a GUI.

```
app = vv.use()
app.Run()
```

References

[1] Matti Ristinmaa, Göran Sandberg, and Karl-Gunnar Olsson, *http://science.uniserve.edu.au/pubs/callab/vol5/ristin.html*, Retrieved on 2 Feb 2013

[2] Andreas Ottosson, *Implementation of CALFEM for Python*, Wallin & Dalholm Digital AB, Lund, Sweden, August, 2010

[3] Jonathan Richard Shewchuk, *https://www.cs.cmu.edu/~quake/triangle.html*, Retrieved on 25 Dec 2012

[4] C. Geuzaine and J.-F. Remacle, *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*, International Journal for Numerical Methods in Engineering 79(11), pp. 1309-1331, 2009

[5] *http://code.google.com/p/visvis/*, Retrieved on 25 Dec 2012

[6] Robert Schneiders, *http://www.robertschneiders.de/meshgeneration/software.html*

[7] C. Geuzaine and J.-F. Remacle *Gmsh Reference Manual*, 19 June 2012

Appendix A

Examples

A.1 Mesh_Ex_01.py

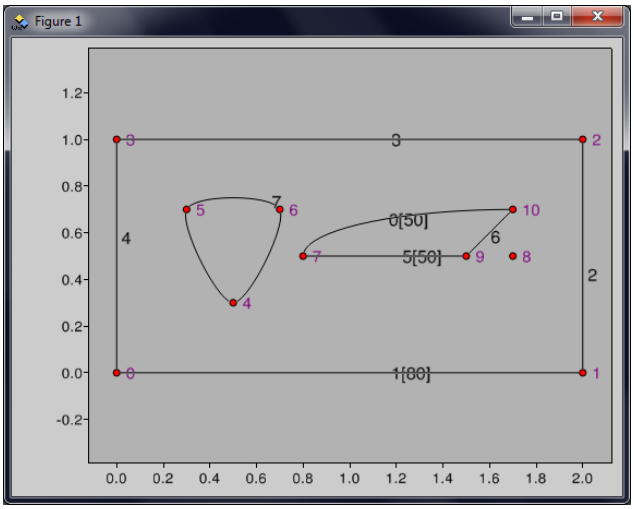


Figure A.1: Geometry of example 01

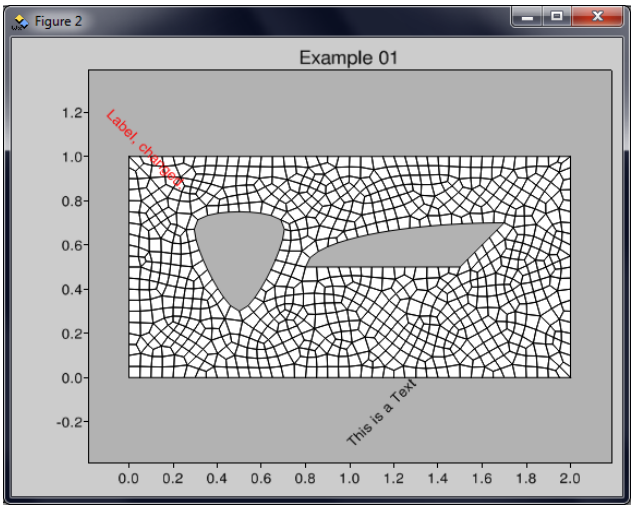


Figure A.2: Mesh of example 01

```
'''Example 01
Shows how to create simple geometry from splines and ellipse
    arcs, and how to mesh a quad mesh in GmshMesher.
Also demonstrates drawGeometry(), drawMesh, and drawing texts
    and labels in a figure.
'''

from pycalfem_GeoData import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

#DEFINE GEOMETRY:
g = GeoData() #Create a GeoData object that holds the geometry.

#Add points:
# The first parameter is the coordinates. These can be in 2D or
    3D.
# The other parameters are not defined in this example. These
    parameters are
# ID, marker, and elSize.
# Since we do not specify an ID the points are automatically
    assigned IDs, starting from 0.
g.addPoint([0, 0])
g.addPoint([2, 0])
g.addPoint([2, 1])
g.addPoint([0, 1])
g.addPoint([0.5, 0.3])
g.addPoint([0.3, 0.7])
```

```
g.addPoint([0.7, 0.7])
g.addPoint([0.8, 0.5])
g.addPoint([1.7, 0.5])
g.addPoint([1.5, 0.5])
g.addPoint([1.7, 0.7])

#Add curves:
# There are four types of curves. In this example we create an
    ellipse arc and some splines.
# The first parameter is a list of point IDs that define the
    curve.
# Curves can have have IDs and markers. In this example the IDs
    are undefined so the curves are
# automatically assigned IDs. The markers can be used for
    identifying regions/boundaries in the
# model.
g.addEllipse([7,8,9,10], marker=50)   #0 An ellipse arc. Read
    the function doc for more information. The four points are [
    start, center, majorAxis, end]
g.addSpline([0, 1], marker=80)        #1 A spline. Splines pass
    through the points in the first parameter.
g.addSpline([2, 1])                   #2
g.addSpline([3, 2])                   #3
g.addSpline([0, 3])                   #4
g.addSpline([7, 9], marker=50)        #5
g.addSpline([10, 9])                  #6
g.addSpline([4, 5, 6, 4])             #7 This is a closed spline
    . The start and end points are the same (4).

#Add a surface:
# Surfaces are defined by its curve boundaries.
# The first parameter is a list of curve IDs that specify the
    outer boundary of the surface.
# The second parameter is a list of lists of curve IDs that
    specify holes in the surface.
# In this example there are two holes.
# The boundaries and holes must be closed paths. We can see that
     [7] is closed because curve 7
# is a closed spline.
# addSurface creates a flat surface, so all curves must lie on
    the same plane.
g.addSurface([4,3,2,1], [[7], [5,6,0]])

#MESHING:
elType = 3 #Element type 3 is quad. (2 is triangle. See user
    manual for more element types)
dofsPerNode= 1 #Degrees of freedom per node.
```

```
mesher = GmshMesher(geoData = g,
                    gmshExecPath = None, #Path to gmsh.exe. If
                        None then the system PATH variable is
                        queried. Both relative and absolute paths
                         work, like "gmsh\gmsh.exe".
                    elSizeFactor = 0.05, #Factor that changes
                        element sizes.
                    elType = elType,
                    dofsPerNode= dofsPerNode)

#Mesh the geometry:
# The first four return values are the same as those that
    trimesh2d() returns.
# coords is as list of node coordinates.
# edof is the element topology (element degrees of freedom)
# dofs is a lists of all degrees of freedom
# bdofs is a dictionary of boundary dofs (dofs of geometric
    entities with markers).
# elementmarkers is a list of markers, and is used for finding
    the marker of a given element (index).
coords, edof, dofs, bdofs, elementmarkers = mesher.create()


#VISUALISATION:

#Hold left mouse button to pan.
#Hold right mouse button to zoom.
pcv.drawGeometry(g)#Draws the geometry. Note that surfaces and
    volumes are not drawn at all by this function.

vv.figure() #New figure window

pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=True, title="Example␣01") #Draws the
    mesh.

pcv.addText("This␣is␣a␣Text", pos=(1, -0.3), angle=45) #Adds a
    text in world space

ourLabel = pcv.addLabel("This␣is␣a␣Label", pos=(100,200), angle
    =-45) #Adds a label in the screen space
ourLabel.text = "Label,␣changed." #We can change the attributes
    of labels and texts, such as color, text, and position.
ourLabel.textColor = 'r' #Make it red. (1,0,0) would also have
    worked.
ourLabel.position = (20,30)

# Enter main loop:
```

```
app = vv.use()
app.Create()
app.Run()
```

A.2 Mesh_Ex_02.py

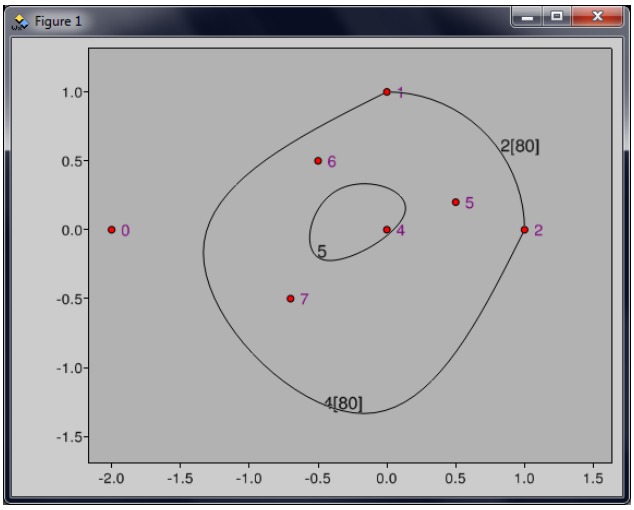


Figure A.3: Geometry of example 02

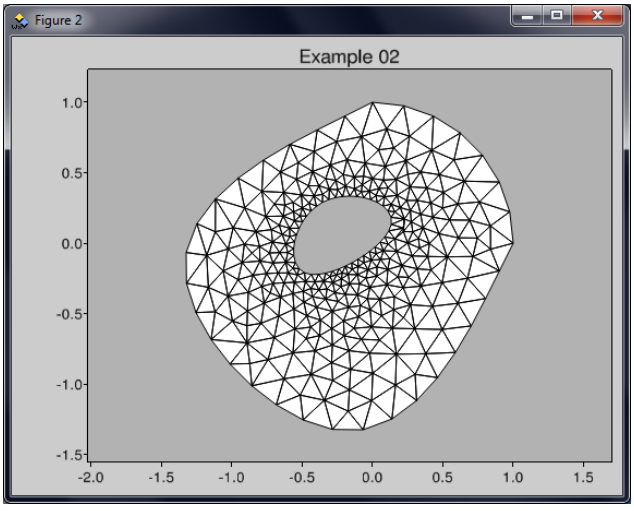


Figure A.4: Mesh of example 02

```
'''Example 02
Creating geometry from B-Splines and circle arcs.
```

```
Also shows how to set ID numbers for geometry entities and how
    to specify element density.
'''

from pycalfem_GeoData import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

#DEFINE GEOMETRY:
g = GeoData()

#Add points:
#In this example we set the IDs manually.
g.addPoint([ -2,  0], ID=0)
g.addPoint([  0,  1], ID=1, elSize=5) #elSize determines the
    size of the elements near this point.
g.addPoint([  1,  0], 2, elSize=5)    # elSize is 1 by default.
    Larger number means less dense mesh.
g.addPoint([  0, -2], 3)              # Size means the length of
     the sides of the elements.
g.addPoint([  0,  0], 4, elSize=5)
g.addPoint([ .5, .2], 5)
g.addPoint([-.5, .5], 6)
g.addPoint([-.7,-.5], 7)
#Add curves:
g.addCircle([1, 4, 2], 2) #The 3 points that define the circle
    arc are [start, center, end]. The arc must be smaller than Pi
    .
g.addBSpline([5,6,7,5], 5) # BSplines are similar to Splines,
    but do not necessarily pass through the control points.
g.addBSpline([1,0,3,2], 4)
#Add surface:
g.addSurface([4,2], [[5]])

#Markers do not have to be set when the curve is created. It can
     be done afterwards.
# Set marker=80 for curves 2 and 4:
for curveID in [2, 4]:
    g.setCurveMarker(curveID, 80)

#MESHING:
elType = 2 #Element type 2 is triangle. (3 is quad. See user
    manual for more element types)
dofsPerNode= 2 #Degrees of freedom per node.

mesher = GmshMesher(geoData = g,
```

```
                        gmshExecPath = None, #Path to gmsh.exe. If
                            None then the system PATH variable is
                            queried. Relative and absolute paths work
                            .
                        elSizeFactor = 0.05, #Factor that changes
                            element sizes.
                        elType = elType,
                        dofsPerNode= dofsPerNode)

#Mesh the geometry:
# The first four return values are the same as those that
    trimesh2d() returns.
# value elementmarkers is a list of markers, and is used for
    finding the marker of a given element (index).
coords, edof, dofs, bdofs, elementmarkers = mesher.create()

#VISUALISATION:
#Hold left mouse button to pan.
#Hold right mouse button to zoom.
pcv.drawGeometry(g, labelCurves=True)#Draws the geometry.

vv.figure() #New figure window

pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=True, title="Example 02") #Draws the
    mesh.

vv.gca().axis.showGrid = True #Show grid

# Enter main loop:
app = vv.use()
app.Create()
app.Run()
```

A.3 Mesh_Ex_03.py

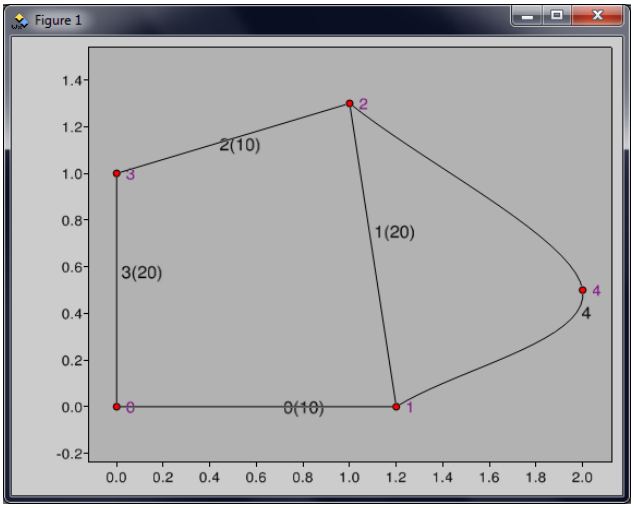


Figure A.5: Geometry of example 03

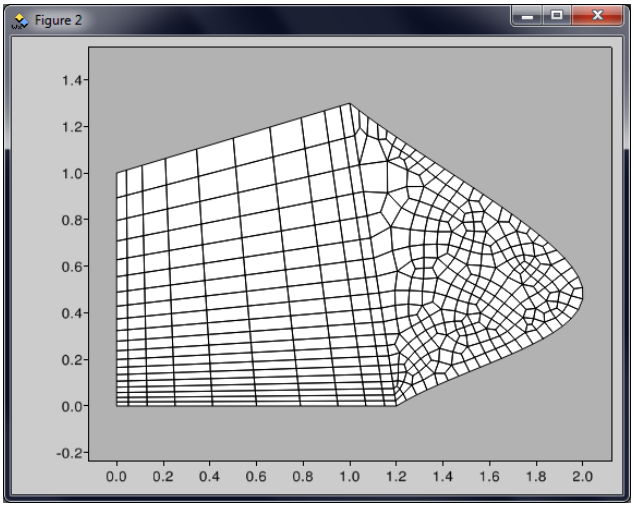


Figure A.6: Mesh of example 03

```
'''Example 03
Shows structured meshing in 2D.
'''

from pycalfem_GeoData import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv
```

```
#DEFINE GEOMETRY:
g = GeoData()

#Add Points:
g.addPoint([0,0])
g.addPoint([1.2, 0])
g.addPoint([1, 1.3])
g.addPoint([0, 1])
g.addPoint([2, 0.5])

#Add Splines:
#The first four curves are structured curves, i.e the number of
    nodes along the curves is pre-determined.
#Parameter elOnCurve states how many elements are placed along
    the curve.
#Parameters elDistribType and elDistribVal are optional
    parameters that specify how elements are distributed.
#   "bump" means elements are bunched up at the ends or the
    middle of the curve.
#       In this case elDistribVal is smaller than 1, so elements
     crowd at the edges.
#   "progression" means each element along the curve is larger/
    smaller than the previous one.
#       A larger elDistribVal makes the elements larger at the
    end of the curves.
g.addSpline([0,1], elOnCurve=10, elDistribType="bump",
    elDistribVal=0.2)
g.addSpline([1,2], elOnCurve=20, elDistribType="progression",
    elDistribVal=1.1)
g.addSpline([2,3], elOnCurve=10, elDistribType="bump",
    elDistribVal=0.2)
g.addSpline([0,3], elOnCurve=20, elDistribType="progression",
    elDistribVal=1.1) #Change order of points to reverse
    progression distribution
g.addSpline([2, 4, 1])

#Add Surfaces:
#A structured surface must contain 4 curves that have the
    parameter 'elOnCurve' defined.
#The number of elements on two opposite curves must be the same
    (In this case, curves 0 & 2 and 1 & 3).
g.addStructuredSurface([0,1,2,3])
g.addSurface([4,1])

#MESHING:
elType = 3 #Element type 3 is quad. (2 is triangle. See user
    manual for more element types)
```

```
dofsPerNode= 1 #Degrees of freedom per node.

mesher = GmshMesher(geoData = g,
                    gmshExecPath = None, #Path to gmsh.exe. If
                        None then the system PATH variable is
                        queried. Both relative and absolute paths
                         work.
                    elSizeFactor = 0.05, #Factor that changes
                        element sizes.
                    elType = elType,
                    dofsPerNode= dofsPerNode)

#Mesh the geometry:
coords, edof, dofs, bdofs, elementmarkers = mesher.create()

#VISUALISATION:
pcv.drawGeometry(g)#Draws the geometry.
vv.figure() #New figure window
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=True) #Draws the mesh.
# Enter main loop:
app = vv.use()
app.Create()
app.Run()
```

A.4 Mesh_Ex_04.py

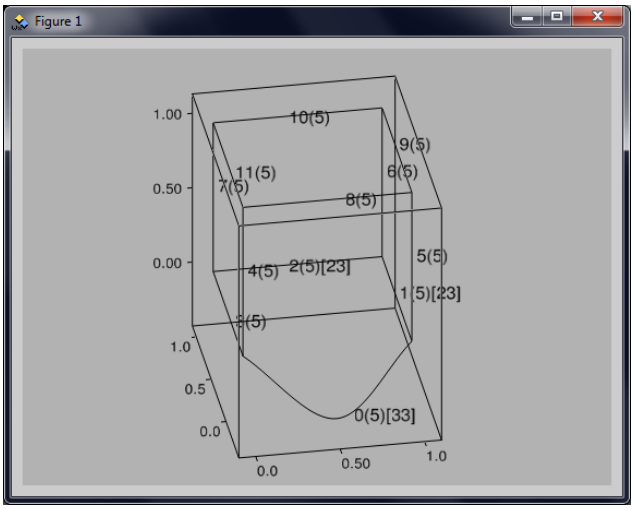


Figure A.7: Geometry of example 04

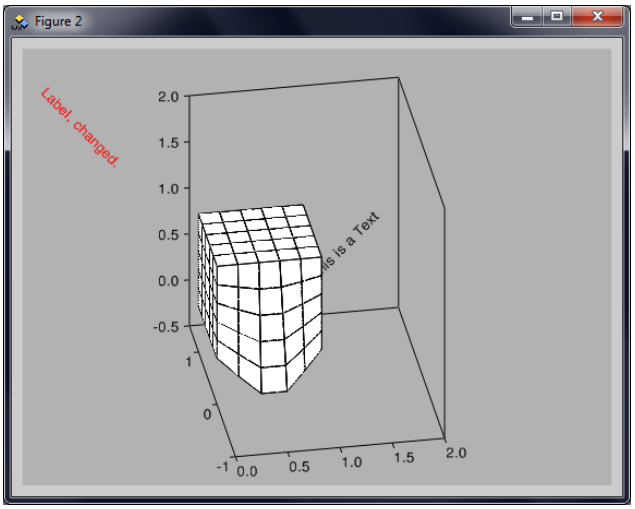


Figure A.8: Mesh of example 04

```
'''Example 04.
Structured 3D meshing. Adding texts and labels to figures.
    Altering axis properties.
'''

from pycalfem_GeoData import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

g = GeoData()

#Add Points:
g.addPoint([0, 0, 0], ID=0)
g.addPoint([0.5, -0.3, -0.3], 1)
g.addPoint([1, 0, 0], 2)
g.addPoint([1, 1, 0], 3)
g.addPoint([0, 1, 0], 4, marker = 11) #Set some markers no
    reason.
g.addPoint([0, 0, 1], 5, marker = 11) #(markers can be given to
    points as well as curves and surfaces)
g.addPoint([1, 0, 1], 6, marker = 11)
g.addPoint([1, 1, 1], 7)
g.addPoint([0, 1, 1], 8)

#Add splines:
g.addSpline([0, 1, 2], 0, marker = 33, elOnCurve = 5)
g.addSpline([2, 3], 1, marker = 23, elOnCurve = 5)
g.addSpline([3, 4], 2, marker = 23, elOnCurve = 5)
```

```
g.addSpline([4, 0], 3, elOnCurve = 5)
g.addSpline([0, 5], 4, elOnCurve = 5)
g.addSpline([2, 6], 5, elOnCurve = 5)
g.addSpline([3, 7], 6, elOnCurve = 5)
g.addSpline([4, 8], 7, elOnCurve = 5)
g.addSpline([5, 6], 8, elOnCurve = 5)
g.addSpline([6, 7], 9, elOnCurve = 5)
g.addSpline([7, 8], 10, elOnCurve = 5)
g.addSpline([8, 5], 11, elOnCurve = 5)

#Add surfaces:
g.addStructuredSurface([0, 1, 2, 3], 0, marker=45)
g.addStructuredSurface([8, 9, 10, 11], 1)
g.addStructuredSurface([0, 4, 8, 5], 2, marker=55)
g.addStructuredSurface([1, 5, 9, 6], 3, marker=55)
g.addStructuredSurface([2, 6, 10, 7], 4)
g.addStructuredSurface([3, 4, 11, 7], 5)

#Add Volume:
#addStructuredVolume() takes three args. The first is a list of
    surface IDs (structured surfaces).
# The surfaces should make a hexahedron (i.e. 6 surfaces). Other
     kinds of structured volumes than hexahedra will
# not work for hexahedral elements, which is the only type of 3D
     element that CALFEM handles.
#The two optional parameters are the volume ID and volume marker
    .
g.addStructuredVolume([0,1,2,3,4,5], 0, marker=90)

elType = 5 #Element type 5 is hexahedron. (See user manual for
    more element types)
dofsPerNode= 1 #Degrees of freedom per node.

mesher = GmshMesher(geoData = g,
                    gmshExecPath = None,
                    elType = elType,
                    dofsPerNode= dofsPerNode)

#Mesh the geometry:
coords, edof, dofs, bdofs, _ = mesher.create()

#Hold Left Mouse button to rotate.
#Hold right mouse button to zoom.
#Hold SHIFT and left mouse button to pan.
#Hold SHIFT and right mouse button to change the field of view.
#Hold Ctrl and left mouse button to roll the camera.
pcv.drawGeometry(g, drawPoints=False)#Draws the geometry.
```

```
vv.figure()
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=True) #Draws the mesh.

pcv.addText("This is a Text", pos=(1, 0.5, 0.5), angle=45) #
    Adds a text in world space
ourLabel = pcv.addLabel("This is a Label", pos=(20,30), angle
    =-45) #Adds a label in the screen space
ourLabel.text = "Label, changed." #We can change the attributes
    of labels and texts, such as color and position.
ourLabel.textColor = 'r' #Make it red. (1,0,0) would also have
    worked.

vv.gca().axis.showBox = 0 #Matlab style axes (three axes in the
    background instead of a cube)
vv.gca().SetLimits(rangeX=(0,2), rangeY=(-1,1.5), rangeZ
    =(-0.5,2), margin=0.02) #Change the limits of the axes.

# Enter main loop:
app = vv.use()
app.Create()
app.Run()
```

A.5 Mesh_Ex_05.py

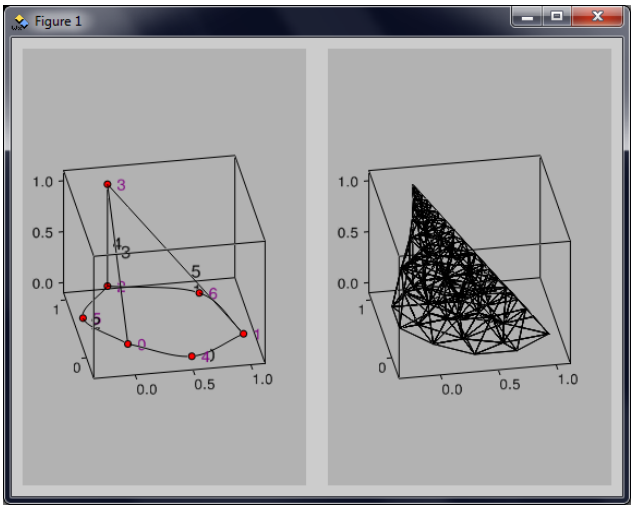


Figure A.9: Geometry of example 05

```
'''Example 05
```

```
'''
This example shows how to make an unstructured 3D mesh (
    tetrahedron elements, which calfem cant actually use).
It also demonstrates how to do subplots and create two axes that
     are viewed from the same camera.
'''

from pycalfem_GeoData import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

g = GeoData()

g.addPoint([0, 0, 0],       0)
g.addPoint([1, 0, 0],       1)
g.addPoint([0, 1, 0],       2)
g.addPoint([0, 1, 1],       3, elSize=0.1)
g.addPoint([0.5, -0.3, 0], 4)
g.addPoint([-0.3, 0.5, 0], 5)
g.addPoint([0.75, 0.75, 0],6)

g.addSpline([0,4,1])
g.addSpline([1,6,2])
g.addSpline([2,5,0])
g.addSpline([0,3])
g.addSpline([3,2])
g.addSpline([3,1])

g.addRuledSurface([0,1,2])
g.addRuledSurface([0,5,3])
g.addRuledSurface([1,5,4])
g.addRuledSurface([2,3,4])

g.addVolume([0,1,2,3])

elType = 4 #Element type 4 is tetrahedron. (See user manual for
    more element types).
dofsPerNode= 1 #Degrees of freedom per node.

mesher = GmshMesher(geoData = g,
                    gmshExecPath = None,
                    elType = elType,
                    elSizeFactor = 0.3,
                    dofsPerNode= dofsPerNode)

#Mesh the geometry:
coords, edof, dofs, bdofs, elementmarkers = mesher.create()
```

```
#VISUALISATION:

#Create two axes that are viewed from the same camera:
vv.figure()
a1 = vv.subplot(121)
a2 = vv.subplot(122)
cam = vv.cameras.ThreeDCamera()
a1.camera = a2.camera = cam
#Draw:
pcv.drawGeometry(g, axes=a1)#Draws the geometry.
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=False, axes=a2) #Draws the mesh.

# Enter main loop:
app = vv.use()
app.Create()
app.Run()
```

A.6 Mesh_Ex_06.py

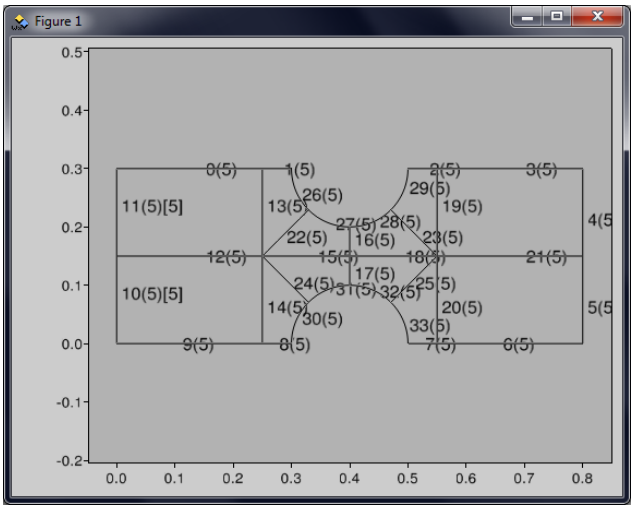


Figure A.10: Geometry of example 06

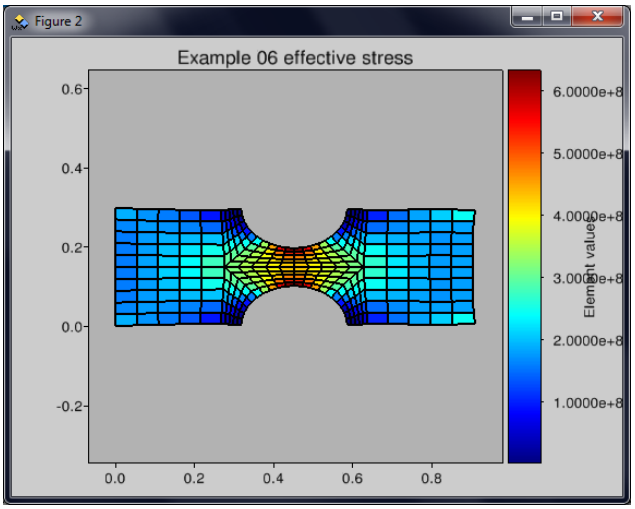


Figure A.11: Element values (effective stress) of example 06

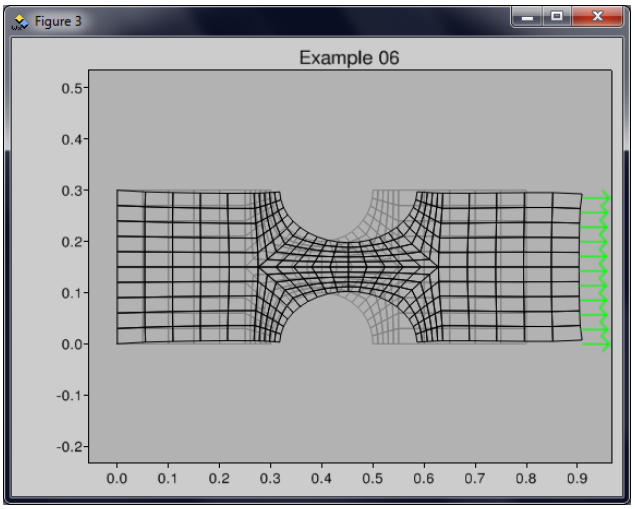


Figure A.12: Deformed mesh of example 06

```
'''Example 06
Solves a plane stress 2D problem using a structured mesh.
Shows how to draw von Mises effective stress as an element value
    with drawElementValues().
Shows use of GmshMesher attribute 'nodesOnCurve' (dictionary
   that says which nodes are on a given geometry curve)
'''

from pycalfem import *
from pycalfem_utils import *
from pycalfem_GeoData import *
```

```
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv
from math import sqrt

# ----- Problem constants
t = 0.2
v = 0.35
E = 2.1e9
ptype = 1
ep = [ptype,t]
D=hooke(ptype, E, v)

print "Creating geometry..."
g = GeoData()

s2 = 1/sqrt(2) #Just a shorthand. We use this to make the circle
    arcs.

points = [[0, 3], [2.5, 3], [3, 3], [4-s2, 3-s2], [4, 2],     #
   0-4
          [4+s2, 3-s2], [5, 3], [5.5, 3], [8,3], [0, 1.5],    #
             5-9
          [2.5, 1.5], [4, 1.5], [5.5, 1.5], [8, 1.5], [0, 0], #
             10-14
          [2.5, 0], [3, 0], [4-s2, s2], [4, 1], [4+s2, s2],   #
             15-19
          [5, 0], [5.5, 0], [8,0], [4,3], [4,0]]              #
             20-24
for xp, yp in points:
    g.addPoint([xp*0.1, yp*0.1])

splines = [[0,1], [1,2], [6,7], [7,8], [8,13],          #0-4
           [13,22], [22,21], [21,20], [16,15], [15,14], #5-9
           [14,9], [9,0], [9,10], [10,1], [10, 15],     #10-14
           [10,11], [11,4], [11,18], [11,12], [12,7],   #15-19
           [12,21], [12,13], [3,10], [5,12], [10,17],   #20-24
           [12,19]]                                     #25
for s in splines:
    g.addSpline(s, elOnCurve=5)

g.setCurveMarker(ID=4,  marker=7) #Assign marker 7 to the
   splines on the right.
g.setCurveMarker(ID=5,  marker=7) # We will apply a force on
   nodes with marker 7.
g.setCurveMarker(ID=10, marker=5) #Assign marker 5 to the
   splines on the left.
```

```
g.setCurveMarker(ID=11, marker=5) # The nodes with marker 5 will
    be locked in place.

# Points in circle arcs are [start, center, end]
circlearcs = [[2, 23, 3], [3, 23, 4], [4, 23, 5], [5, 23, 6],
              #26-29
               [16, 24, 17], [17, 24, 18], [18, 24, 19], [19, 24,
                   20]]   #30-33
for c in circlearcs:
    g.addCircle(c, elOnCurve=5)

g.addStructuredSurface([11,12,13,0]) #0
g.addStructuredSurface([14, 12, 10, 9])
g.addStructuredSurface([8, 30, 24, 14])
g.addStructuredSurface([24, 31, 17, 15])
g.addStructuredSurface([15, 16, 27, 22]) #4
g.addStructuredSurface([22, 26, 1, 13])
g.addStructuredSurface([16, 18, 23, 28])
g.addStructuredSurface([19, 2, 29, 23])
g.addStructuredSurface([19, 21, 4, 3]) #8
g.addStructuredSurface([20, 6, 5, 21])
g.addStructuredSurface([25, 20, 7, 33])
g.addStructuredSurface([32, 17, 18, 25]) #11


print "Meshing␣geometry..."
elType = 3 #3 Quads
dofsPerNode = 2

mesher = GmshMesher(geoData = g,
                    elType = elType,
                    dofsPerNode=dofsPerNode,
                    gmshExecPath = None)
coords, edof, dofs, bdofs, elementmarkers = mesher.create()

print "Assembling␣system␣matrix..."
nDofs = size(dofs)
ex, ey = coordxtr(edof, coords, dofs)
K = zeros([nDofs,nDofs])

for eltopo, elx, ely in zip(edof, ex, ey):
    Ke = planqe(elx, ely, ep, D)
    assem(eltopo, K, Ke)

print "Solving␣equation␣system..."
f = zeros([nDofs,1])

bc = array([],'i')
```

```
bcVal = array([],'i')

bc, bcVal = applybc(bdofs, bc, bcVal, 5, 0.0, 0)

applyforce(bdofs, f, 7, 10e5, 1)

a,r = solveq(K,f,bc,bcVal)

print "Computing element forces..."
ed = extractEldisp(edof,a)

vonMises = []
for i in range(edof.shape[0]): #For each element:
    es, et = planqs(ex[i,:], ey[i,:], ep, D, ed[i,:]) #Determine
        element stresses and strains in the element.
    vonMises.append( math.sqrt( pow(es[0],2) - es[0]*es[1] + pow
       (es[1],2) + 3*es[2] ) ) #calc and append effective stress
        to list.
    ## es: [sigx sigy tauxy]


print "Visualising..."
pcv.drawGeometry(g, drawPoints=False, labelCurves=True)

vv.figure()
pcv.drawElementValues(vonMises, coords, edof, dofsPerNode,
   elType, a, doDrawMesh=True, doDrawUndisplacedMesh=False,
   title="Example 06 effective stress")

vv.figure()
pcv.drawDisplacements(a, coords, edof, dofsPerNode, elType,
   doDrawUndisplacedMesh=True, title="Example 06")

# Make use of attribute 'nodesOnCurve' in GmshMesher to draw
   some arrows on the right hand side of the mesh:
rightSideNodes = set()
for curveID in [4,5]: #4 and 5 are the IDs of the curves where
   we applied the forces.
    rightSideNodes = rightSideNodes.union(set(mesher.
       nodesOnCurve[curveID])) #Get the nodes, without
       duplicates.
for i in rightSideNodes:
    x = coords[i,0] + a[i*2, 0]     #Position of the node with
       displacements.
    y = coords[i,1] + a[i*2+1, 0]
    pcv.addText("\rightarrow", (x, y), fontSize=20, color='g') #
       A poor man's force indicator. Could also use vv.plot()
```

```
# Enter main loop:
app = vv.use()
app.Create()
app.Run()

print "Done."
```

A.7 Mesh_Ex_07.py

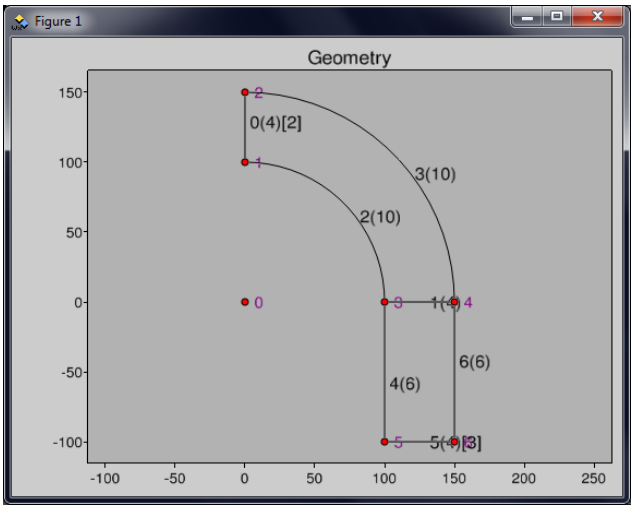


Figure A.13: Geometry of example 07

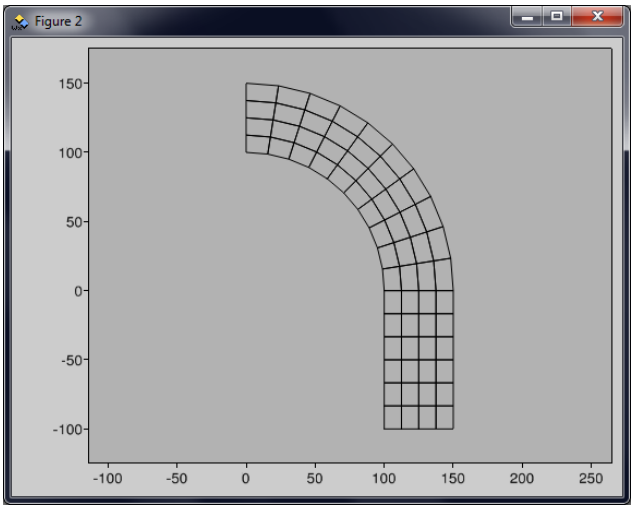


Figure A.14: Mesh of example 07

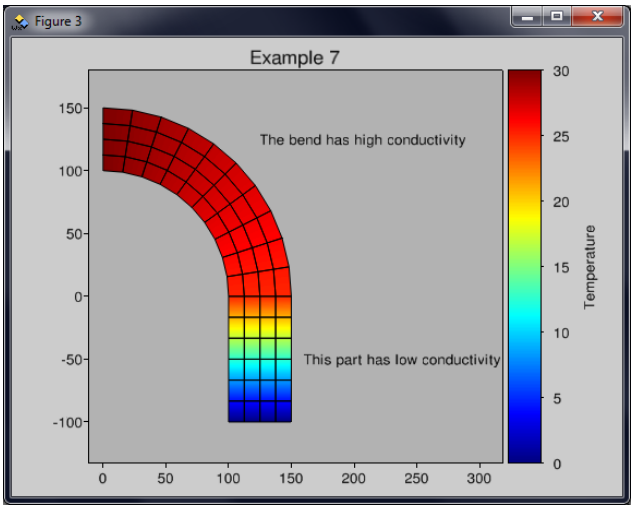


Figure A.15: Node values (temperature) of example 07

```
'''Example 07
Meshing 8-node-isoparametric elements (second order incomplete
    quads).
Shows use of surfacemarkers/elementmarkers to apply different
    properties to elements in different regions.
'''
from pycalfem_GeoData import *
from pycalfem_mesh import *
from pycalfem_vis import *
import visvis as vv
from pycalfem import *
from pycalfem_utils import *

# ---- Problem constants
kx1 = 100
ky1 = 100
kx2 = 10
ky2 = 10
t = 1.0
n = 2 #Gauss points or integration points
ep = [t, n]

D1 = matrix([
     [kx1, 0.],
     [0., ky1]])
D2 = matrix([
     [kx2, 0.],
     [0., ky2]])
```

```
Ddict = {10 : D1, 11 : D2} #markers 10 & 11 will be used to
    specify different regions with different conductivity.

# ---- Create Geometry
g = GeoData()

#Add Points:
points = [[0,0], [0,100], [0,150], [100,0], [150,0], [100,-100],
    [150,-100]]
for p in points:
    g.addPoint(p)

#Add Splines:
g.addSpline([1,2], marker=2, elOnCurve=4)
g.addSpline([3,4], elOnCurve=4)
g.addCircle([1,0,3], elOnCurve = 10)
g.addCircle([2,0,4], elOnCurve = 10)
g.addSpline([3,5], elOnCurve = 6)
g.addSpline([5,6], marker=3, elOnCurve = 4)
g.addSpline([6,4], elOnCurve = 6)

#Add Surfaces:
# When we set markers for surfaces, and have 2D elements, we can
    find which region
# an element is in via the list 'elementmarkers', which is
    returned by GmshMesher.create()
g.addStructuredSurface([0,2,1,3], marker = 10)
g.addStructuredSurface([1,4,5,6], marker = 11)

elType = 16 #Element type 16 is 8-node-quad. (See gmsh manual
    for more element types)
dofsPerNode= 1 #Degrees of freedom per node.

mesher = GmshMesher(geoData = g,
                    gmshExecPath = None, #Path to gmsh.exe. If
                        None then the system PATH variable is
                        queried. Relative and absolute paths work
                        .
                    elType = elType,
                    dofsPerNode= dofsPerNode)
coords, edof, dofs, bdofs, elementmarkers = mesher.create()

print "Assembling system matrix..."
nDofs = size(dofs)
ex, ey = coordxtr(edof, coords, dofs)

K = zeros([nDofs,nDofs])
```

```
for eltopo, elx, ely, elMarker in zip(edof, ex, ey,
    elementmarkers):
    #Calc element stiffness matrix: Conductivity matrix D is
        taken
    # from Ddict and depends on which region (which marker) the
        element is in.
    Ke = flw2i8e(elx, ely, ep, Ddict[elMarker])
    assem(eltopo, K, Ke)


print "Solving equation system..."
f = zeros([nDofs,1])

bc = array([],'i')
bcVal = array([],'i')

bc, bcVal = applybc(bdofs,bc,bcVal,2,30.0)
bc, bcVal = applybc(bdofs,bc,bcVal,3,0.0)

a,r = solveq(K,f,bc,bcVal)


print "Computing element forces..."
ed = extractEldisp(edof,a)

for i in range(shape(ex)[0]):
    es, et, eci = flw2i8s(ex[i,:], ey[i,:], ep, Ddict[
        elementmarkers[i]], ed[i,:])
    #Do something with es, et, eci here.

print "Visualising..."
drawGeometry(g, title="Geometry")

vv.figure()
drawMesh(coords, edof, dofsPerNode, elType, filled=False)
#8-node quads are drawn as simple quads.

vv.figure()
drawNodalValues(a, coords, edof, dofsPerNode, elType, title="
    Example 7")
getColorbar().SetLabel("Temperature")
addText("The bend has high conductivity", (125,125))
addText("This part has low conductivity", (160,-50))

# Enter main loop:
app = vv.use()
app.Create()
app.Run()
```

```
print "Done."
```

A.8 Mesh_Ex_08.py

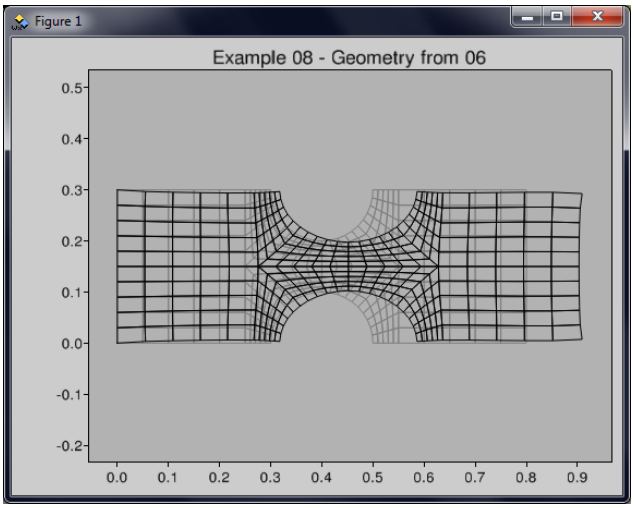


Figure A.16: Deformed mesh of example 08

```
'''Example 08
Shows how to load geometry from a gmsh geo file.
The geometry is the same as in example 06.
'''

from pycalfem import *
from pycalfem_utils import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

# ---- Create Geometry
g = "examplegeo\ex8.geo" #Relative path to the geo file that
    contains the geometry.

elType = 3 #3 Quads
dofsPerNode = 2

mesher = GmshMesher(geoData = g,
                    gmshExecPath = None,
                    elType = elType,
                    dofsPerNode= dofsPerNode)
```

```
coords, edof, dofs, bdofs, elementmarkers = mesher.create()

#The rest of the example is the same as example 06.
t = 0.2
v = 0.35
E = 2.1e9
ptype = 1
ep = [ptype,t]
D=hooke(ptype, E, v)

print "Assembling system matrix..."
nDofs = size(dofs)
ex, ey = coordxtr(edof, coords, dofs)
K = zeros([nDofs,nDofs])

for eltopo, elx, ely in zip(edof, ex, ey):
    Ke = planqe(elx, ely, ep, D)
    assem(eltopo, K, Ke)

print "Solving equation system..."
f = zeros([nDofs,1])
bc = array([],'i')
bcVal = array([],'i')
bc, bcVal = applybc(bdofs, bc, bcVal, 5, 0.0, 0)
applyforce(bdofs, f, 7, 10e5, 1)
a,r = solveq(K,f,bc,bcVal)

print "Visualising..."
pcv.drawDisplacements(a, coords, edof, dofsPerNode, elType,
   doDrawUndisplacedMesh=True, title="Example 08 - Geometry from
   06")

# Enter main loop:
app = vv.use()
app.Create()
app.Run()

print "Done."
```

A.9 Mesh_Ex_09.py

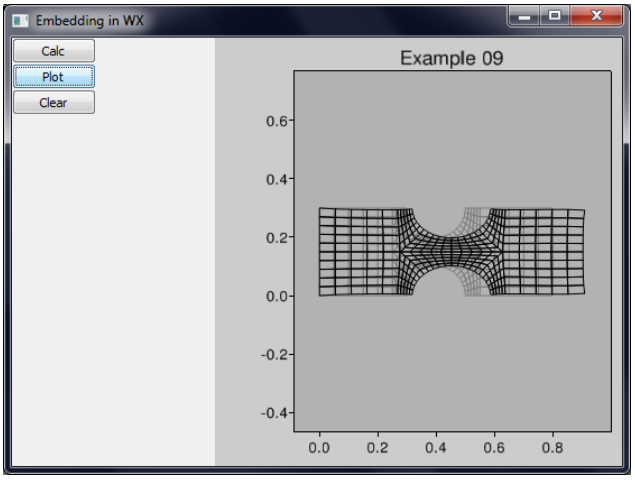


Figure A.17: Deformed mesh of example 09

```
'''Example 09
Shows how to embed visvis figures in a wxPython GUI.
Visvis can also be embedded in Qt4 (PyQt), GTK, and FLTK.
Based on http://code.google.com/p/visvis/wiki/
    example_embeddingInWx
'''

import wx
from pycalfem import *
from pycalfem_utils import *
from pycalfem_mesh import *
import pycalfem_vis as pcv
import visvis as vv

# Create a visvis app instance, which wraps a wx application
    object.
# This needs to be done *before* instantiating the main window.
app = vv.use('wx')


class MainWindow(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "Embedding in WX",
            size=(560, 420))

        # Make a panel with buttons
        self.panel = wx.Panel(self)
```

```
        but1 = wx.Button(self.panel, -1, 'Calc')
        but2 = wx.Button(self.panel, -1, 'Plot')
        but3 = wx.Button(self.panel, -1, 'Clear')

        #Make panel sizer and embed stuff
        self.panelsizer = wx.BoxSizer(wx.VERTICAL)
        self.panelsizer.Add(but1)
        self.panelsizer.Add(but2)
        self.panelsizer.Add(but3)
        self.panel.SetSizer(self.panelsizer)

        # Make figure using "self" as a parent
        Figure = app.GetFigureClass()
        self.fig = Figure(self)

        # Make window sizer and embed stuff
        self.sizer = wx.BoxSizer(wx.HORIZONTAL)
        self.sizer.Add(self.panel, 1, wx.EXPAND)
        self.sizer.Add(self.fig._widget, 2, wx.EXPAND)

        # Make callback
        but1.Bind(wx.EVT_BUTTON, self._Calc)
        but2.Bind(wx.EVT_BUTTON, self._Plot)
        but3.Bind(wx.EVT_BUTTON, self._Clear)

        # Apply window sizers
        self.SetSizer(self.sizer)
        self.SetAutoLayout(True)
        self.Layout()

        # Finish
        self.Show()


    def _Calc(self, event):
        #Calculations are taken from example 08.
        # Constants:
        t = 0.2
        v = 0.35
        E = 2.1e9
        ptype = 1
        ep = [ptype,t]
        D=hooke(ptype, E, v)

        # Create Geometry:
        g = "examplegeo\ex8.geo"
        self.elType = 3 #3 Quads
        self.dofsPerNode = 2
```

```
            mesher = GmshMesher(geoData = g,
                                gmshExecPath = None, #"gmsh\gmsh.exe
                                    "
                                elType = self.elType,
                                dofsPerNode= self.dofsPerNode)
            self.coords, self.edof, self.dofs, self.bdofs, _ =
                mesher.create()

            # Assem systems matrix:
            nDofs = size(self.dofs)
            ex, ey = coordxtr(self.edof, self.coords, self.dofs)
            K = zeros([nDofs,nDofs])
            for eltopo, elx, ely in zip(self.edof, ex, ey):
                Ke = planqe(elx, ely, ep, D)
                assem(eltopo, K, Ke)

            # Solve:
            f = zeros([nDofs,1])
            bc = array([],'i')
            bcVal = array([],'i')
            bc, bcVal = applybc(self.bdofs, bc, bcVal, 5, 0.0, 0)
            applyforce(self.bdofs, f, 7, 10e5, 1)
            self.a, _ = solveq(K,f,bc,bcVal)


        def _Plot(self, event):
            # Make sure our figure is the active one
            # If only one figure, this is not necessary.
            #vv.figure(self.fig.nr)

            # Clear it:
            vv.clf()

            # Plot:
            pcv.drawDisplacements(self.a, self.coords, self.edof,
                self.dofsPerNode, self.elType, doDrawUndisplacedMesh=
                True, title="Example 09")

        def _Clear(self, event):
            vv.clf() #Clear current figure

    # Two ways to create the application and start the main loop
    if True:
        # The visvis way. Will run in interactive mode when used in
            IEP or IPython.
        app.Create()
        m = MainWindow()
        app.Run()
```

```
else:
    # The native way.
    wxApp = wx.App()
    m = MainWindow()
    wxApp.MainLoop()
```

A.10 Mesh_Ex_10.py

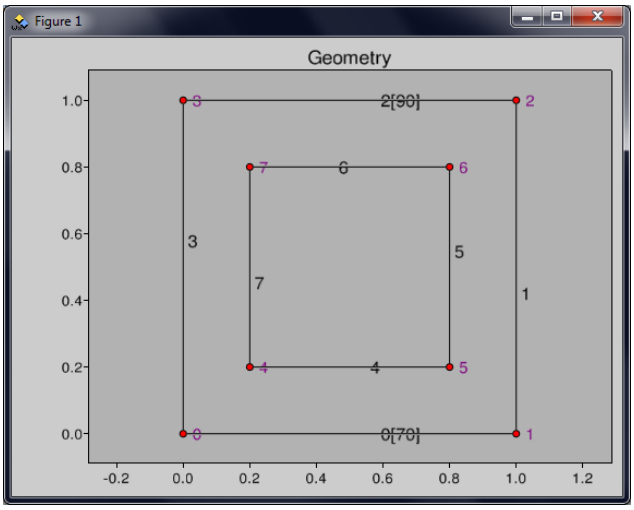


Figure A.18: Geometry of example 10. Same as the use case

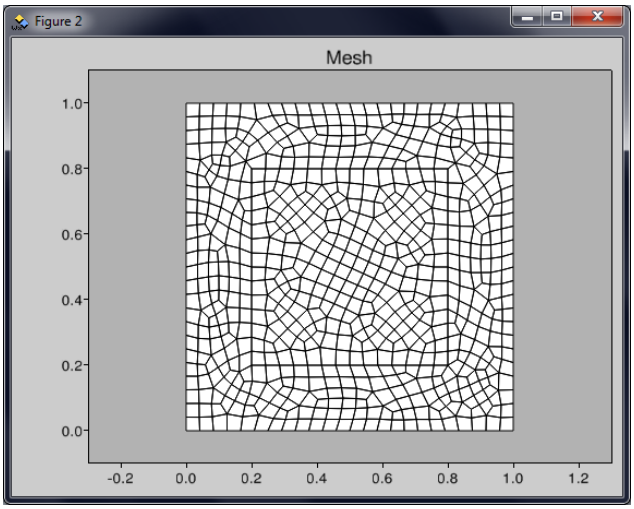


Figure A.19: Mesh of example 10. Same as the use case

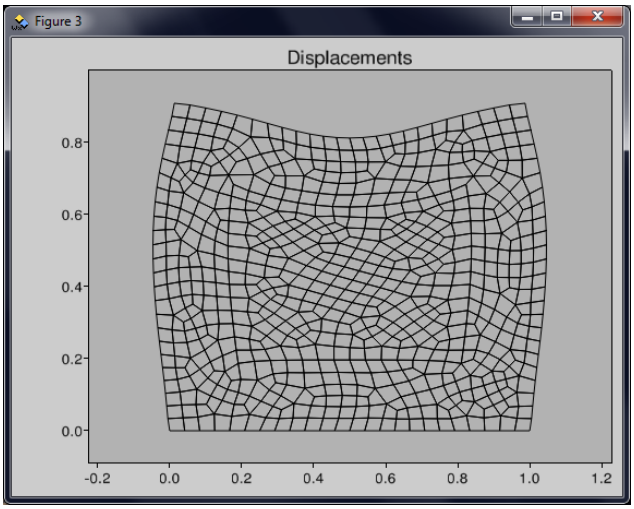


Figure A.20: Deformed mesh of example 10. Same as the use case

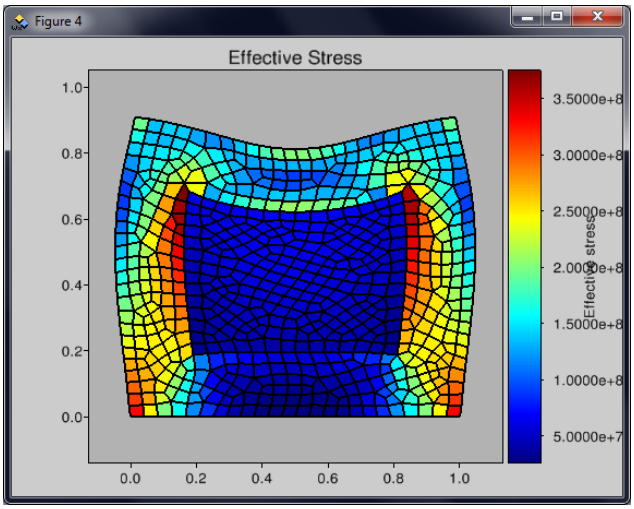


Figure A.21: Element values (effective stress) of example 10. Same as the use case

```
'''Example 10
The use case from the user manual.
The example does not contain anything that is not covered in the
    previous examples.
'''

import pycalfem_GeoData
import pycalfem_mesh
import pycalfem_vis as pcv
import visvis as vv
from pycalfem import *
```

```
from pycalfem_utils import *

#DEFINE GEOMETRY:
g = pycalfem_GeoData.GeoData() #Create a GeoData object that
    holds the geometry.

#Add points:
g.addPoint([0, 0])      #0
g.addPoint([1, 0])      #1
g.addPoint([1, 1])      #2
g.addPoint([0, 1])      #3
g.addPoint([0.2, 0.2])  #4
g.addPoint([0.8, 0.2])  #5
g.addPoint([0.8, 0.8])  #6
g.addPoint([0.2, 0.8])  #7

#Add curves:
g.addSpline([0, 1], marker=70)        #0
g.addSpline([2, 1])                   #1
g.addSpline([3, 2], marker=90)        #2
g.addSpline([0, 3])                   #3
g.addSpline([4, 5])                  #4
g.addSpline([5, 6])                  #5
g.addSpline([6, 7])                  #6
g.addSpline([7, 4])                  #7

#Add surfaces:
g.addSurface([0,1,2,3], holes=[[4,5,6,7]], marker = 55)
g.addSurface([4,5,6,7], marker = 66)

#MESHING:
elType = 3 #Element type 3 is quad.
dofsPerNode = 2 #Degrees of freedom per node.

mesher = pycalfem_mesh.GmshMesher(geoData = g,
                                  gmshExecPath = None, #Path to gmsh.
                                     exe. If None then the system PATH
                                      variable is queried. Relative
                                     and absolute paths work.
                                  elSizeFactor = 0.04, #Factor that
                                     changes element sizes.
                                  elType = elType,
                                  dofsPerNode= dofsPerNode)

#Mesh the geometry:
# The first four return values are the same as those that
    trimesh2d() returns.
```

```
# value elementmarkers is a list of markers, and is used for
    finding the marker of a given element (index).
coords, edof, dofs, bdofs, elementmarkers = mesher.create()


#SOLVE:
t = 0.2
v = 0.35
E1 = 2e9
E2 = 0.2e9
ptype = 1
ep = [ptype,t]
D1 = hooke(ptype, E1, v)
D2 = hooke(ptype, E2, v)
Ddict = {55 : D1, 66 : D2}

nDofs = size(dofs)
K = zeros([nDofs,nDofs])
ex, ey = coordxtr(edof, coords, dofs)

for eltopo, elx, ely, elMarker in zip(edof, ex, ey,
    elementmarkers):
    #Calc element stiffness matrix: Conductivity matrix D is
        taken
    # from Ddict and depends on which region (which marker) the
        element is in.
    Ke = planqe(elx, ely, ep, Ddict[elMarker])
    assem(eltopo, K, Ke)

bc = array([],'i')
bcVal = array([],'i')
bc, bcVal = applybc(bdofs,bc,bcVal, 70, 0.0)

f = zeros([nDofs,1])
applyforce(bdofs, f, 90, value = -10e5, dimension=2)
a,r = solveq(K,f,bc,bcVal)

ed = extractEldisp(edof,a)
vonMises = []
for i in range(edof.shape[0]): #For each element:
    es, et = planqs(ex[i,:], ey[i,:], ep, Ddict[elementmarkers[i
        ]], ed[i,:]) #Determine element stresses and strains in
        the element.
    vonMises.append( math.sqrt( pow(es[0],2) - es[0]*es[1] + pow
        (es[1],2) + 3*pow(es[2],2) ) )


#VISUALISATION:
```

```
pcv.drawGeometry(g, title="Geometry")
vv.figure() #New figure window
pcv.drawMesh(coords=coords, edof=edof, dofsPerNode=dofsPerNode,
    elType=elType, filled=True, title="Mesh") #Draws the mesh.
vv.figure()
pcv.drawDisplacements(a, coords, edof, dofsPerNode, elType,
    doDrawUndisplacedMesh=False, title="Displacements")
vv.figure()
pcv.drawElementValues(vonMises, coords, edof, dofsPerNode,
    elType, a, doDrawMesh=True, doDrawUndisplacedMesh=False,
    title="Effective Stress")
pcv.getColorbar().SetLabel("Effective stress")

# Enter main loop:
app = vv.use()
#app.Create()
app.Run()
```

Appendix B

Status of existing CALFEM functions

Summary status of existing element functions. Functions replaced by "-" are not yet implemented. Functions with an "*" will most likely be removed from the next release of CALFEM. An implementation has been started for functions with a "†" but they are not completed yet.

Python	MATLAB
spring1e(ep)	spring1e(ep)
spring1s(ep, ed)	spring1s(ep, ed)
bar1e(ep)	bar1e(ep)
bar1s(ep, ed)	bar1s(ep, ed)
bar2e(ex, ey, ep)	bar2e(ex, ey, ep)
bar2g(ex, ey, ep, N)	bar2g(ex, ey, ep, N)
bar2s(ex, ey, ep, ed)	bar2s(ex, ey, ep, ed)
bar3e(ex, ey, ez, ep)	bar3e(ex, ey, ez, ep)
bar3s(ex, ey, ez, ep, ed)	bar3s(ex, ey, ez, ep, ed)
flw2te(ex, ey, ep, D, eq)	flw2te(ex, ey, ep, D, eq)
flw2ts(ex, ey, D, ed)	flw2ts(ex, ey, D, ed)
flw2qe(ex, ey, ep, D, eq)	flw2qe(ex, ey, ep, D, eq)
flw2qs(ex, ey, ep, D, ed, eq)	flw2qs(ex, ey, ep, D, ed, eq)
flw2i4e(ex, ey, ep, D, eq)	flw2i4e(ex, ey, ep, D, eq)
flw2i4s(ex, ey, ep, D, ed)	flw2i4s(ex, ey, ep, D, ed)
flw2i8e(ex, ey, ep, D, eq)	flw2i8(ex, ey, ep, D, eq)
flw2i8s(ex, ey, ep, D, ed)	flw2i8s(ex, ey, ep, D, ed)
flw3i8e(ex, ey, ez, ep, D, eq)	flw3i8e(ex, ey, ez, ep, D, eq)
flw3i8s(ex, ey, ez, ep, D, ed)	flw3i8s(ex, ey, ez, ep, D, ed)
plante(ex, ey, ep, D, eq)	plante(ex, ey, ep, D, eq)
plants(ex, ey, ep, D, ed)	plants(ex, ey, ep, D, ed)
plantf(ex, ey, ep, es)	plantf(ex, ey, ep, es)
planqe(ex, ey, ep, D, eq)	planqe(ex, ey, ep, D, eq)*
planqs(ex, ey, ep, D, ed, eq)	planqs(ex, ey, ep, D, ed, eq) *
-	planre(ex, ey, ep, D, eq)
-	planrs(ex, ey, ep, D, ed)

-	plantce(ex, ey, ep, eq) *
-	plantcs(ex, ey, ep, ed) *
plani4e(ex, ey, ep, D, eq)	plani4e(ex, ey, ep, D, eq)
-	plani4s(ex, ey, ep, D, ed)
-	plani4f(ex, ey, ep, es)
-	plani8e(ex, ey, ep, D, eq) *
-	plani8s(ex, ey, ep, D, ed) *
-	plani8f(ex, ey, ep, es) *
-	soli8e(ex, ey, ez, ep, D, eq) *
-	soli8s(ex, ey, ez, ep, D, ed) *
-	soli8f(ex, ey, ez, ep, es) *
beam2e(ex, ey, ep, eq)	beam2e(ex, ey, ep, eq)
beam2s(ex, ey, ep ,ed, eq, np)	beam2s(ex, ey, ep, ed, eq, n)
beam2t(ex, ey, ep, eq)	beam2t(ex, ey, ep, eq)
beam2ts(ex, ey, ep, ed, eq, np)	beam2ts(ex, ey, ep, ed, eq, n)
beam2w(ex, ey, ep, eq)	beam2w(ex, ey, ep, eq)
beam2ws(ex, ey, ep, ed, eq)	beam2ws(ex, ey, ep, ed, eq)
beam2g(ex, ey, ep, N, eq)	beam2g(ex, ey, ep, N, eq)
beam2gs(ex, ey, ep, ed, N, eq)	beam2gs(ex, ey, ep, ed, N, eq)
beam2d(ex, ey, ep)	beam2d(ex, ey, ep)
-	beam2ds(ex, ey, ep, ed, ev, ea)
beam3e(ex, ey, ez, eo, ep, eq)	beam3e(ex, ey, ez, eo, ep, eq)
beam3s(ex, ey, ez, eo, ep, ed, eq, n)	beam3s(ex, ey, ez, eo, ep, ed, eq, n)
platre(ex, ey, ep, D, eq)	platre(ex, ey, ep, D, eq)
-	platrs(ex, ey, ep, D, ed)
-	red(A, b)
hooke(ptype, E, v)	hooke(ptype, E, v)
-	mises(ptype, mp, est, st)
-	dmises(ptype, mp, es, st)
assem(edof, K, Ke, f, fe)	assem(edof, K, Ke, f, fe)
coordxtr(edof, coords, dofs)	cordxtr(Edof, Coord, Dof, nen)
-	eigen(K, M, b)
extract(edof, a)	extract(edof, a)
-	insert(edof, f, ef)
solveq(K, f, bcPresc, bcVal)	solveq(K, f, bc)
statcon(K, f, cd)	statcon(K, f, b)
-	dyna2(w2, xi, f, g, dt)
-	dyna2f(w2, xi, f, p, dt)
-	freqresp(D, dt)
-	gfunc(G, dt)
-	ritz(K, M, f, m, b)
-	spectra(a, xi, dt, f)
-	step1(K, C, d0, ip, f, pbound)
-	step2(K, C, d0, v0, ip, f, pdisp)
-	sweep(K, C, M, p, w)

-	eldia2(ex, ey, es, plotpar, sfac, eci)
eldisp2(ex, ey, ed, magnfac, showmesh) †	eldisp2(Ex, Ey, Ed, plotpar, sfac)
eldraw2(ex, ey) †	eldraw2(Ex, Ey, plotpar, elnum)
-	elflux2(Ex, Ey, Es, plotpar, sfac)
eliso2(ex, ey, ed, showmesh) †	eliso2(Ex, Ey, Ed, isov, plotpar)
-	elprinc2(Ex, Ey, Es, plotpar, sfac)
-	pltscalb2(sfac, magnitude, plotpar)
scalfact2(ex, ey, ed, rat) †	scalfact2(ex, ey, ed, rat)