abapy Documentation

Release 2.3.0

Ludovic Charleux

June 28, 2016

Contents

Abaqus Python "AbaPy" contains tools to build, postprocess and plot automatic finite element simulations using Abaqus. It is d

- *mesh*: contains finite element mesh utilities allowing building, modifying, ploting and exporting to various formats.
- *postproc*: contains utilities to read in Abaqus odb files and represent history and field outputs using custom classes. These classes allow easy calculations, export, storage (using pickle/cPickle).
- *materials*: contains functions to preprocess materials before finite element simulations.
- *indentation*: contains all indentation dedicated tools.
- *advanced_examples*: contains examples using abapy and other python packages to perform research higher level tasks.

Contributors:

- Ludovic Charleux
- Laurent Bizet
- Arnaud Faivre
- Moustapha Issack
- Vincent Keryvin

For citation, please use the following link:

Installation can be performed in many ways, here a two:

- The right way:

```
pip install git+https://github.com/lcharleux/abapy.git
```

- If you are contributing to the module, you can just clone the repository:

```
git clone https://github.com/lcharleux/abapy.git
```

And remember to add the abapy/abapy directory to your `PYTHONPATH`. For example, the following code can be used under Linux (in `.bashrc` or `.profile`):

```
export PYTHONPATH=$PYTHONPATH:yourpath/abapy
```

Contents:

CHAPTER 1

Tutorial

This tutorial introduces the main reasons to use Abapy and explains how to do so. In order to follow the tutorial, following components are required:

- Abaqus
- Python (2.5 or above) and its modules Numpy, Scipy, Matplotlib, SQLAlchemy.

1.1 Introduction

Indentation testing is used widely as an example in this tutorial but everything can be transposed easily to any other problem. Let's start with an existing axisymmetric conical indentation simulations defined in the following INP file: `indentation_axi.inp`. The model includes following features:

- Axisymmetric solids.
- Conical rigid indenter.
- Von Mises elastic-plastic sample.
- Frictionless contact.
- Non linear geometry effects.

The simulation can be launched directly using the command-line:

```
abaqus job=indentation-axi
```

Note: The INP file can also be imported in Abaqus/CAE through file/import/model and then by chosing .inp. Then create a job and launch it.

The simulation is normaly very fast because the meshes are rather coarse. When it is completed, you can open the resulting ODB file using abaqus viewer to have a look at it. Then, then you can start to go deeper into the ODB structure. Use a terminal or DOS shell to launch the following command:

```
abaqus viewer -noGUI
```

You are now in the Python interface of Abaqus/Viewer.

Note: The are several ways to access Python in Abaqus. `abaqus python` is the standard way, it is faster since it doesn't require a licence token. However, you will often need packages that are not available in `abaqus python`,

then you will have to use `abaqus viewer -noGUI` or `abaqus cae -noGUI` which both allow access to everything that is available in Abaqus/viewer and Abaqus/CAE.

Now that you have access to Python inside Abaqus, you can open the odb file using:

```
>>> from odbAccess import openOdb
>>> odb = openOdb('indentation_axi.odb')
```

Then you can have a look at the structure of the odb object mainly using the `print` and `dir` commands.

```
>>> dir(odb)
['AcousticInfiniteSection', 'AcousticInterfaceSection', 'ArbitraryProfile', 'BeamSection', 'BoxProfil
>>> # OK let's have a look inside jobData
>>> print odb.jobData
({'analysisCode': ABAQUS_STANDARD, 'creationTime': 'Mon Apr 29 15:44:54 CEST 2013', 'machineName': '
>>> print odb.diagnosticData
({'analysisErrors': 'OdbSequenceAnalysisError object', 'analysisWarnings': 'OdbSequenceAnalysisWarnin
>>> print odb.diagnosticData.jobStatus
JOB_STATUS_COMPLETED_SUCCESSFULLY
>>> # Now we know that the simulation was successful
>>> # Let's now have a look to history outputs
>>> print odb.steps
{'LOADING': 'OdbStep object', 'UNLOADING': 'OdbStep object'}
>>> print odb.steps['LOADING']
({'acousticMass': -1.0, 'acousticMassCenter': (), 'description': '', 'domain': TIME, 'eliminatedNodal
>>> print odb.steps['LOADING'].frames[-1]
({'cyclicModeNumber': None, 'description': 'Increment     20: Step Time =    1.000', 'domain': TIME,
>>> print odb.steps['LOADING'].historyRegions['Assembly ASSEMBLY']
({'description': 'Output at assembly ASSEMBLY', 'historyOutputs': 'Repository object', 'loadCase': No
>>> # And then to field outputs
>>> print odb.steps['LOADING'].frames[-1].fieldOutputs['U'].values[0]
({'baseElementType': '', 'conjugateData': None, 'conjugateDataDouble': 'unknown', 'data': array([-2.5
```

At this point, you must understand that you can find back every single input as well as every output in through Python. The question is now how to do it painlessly. Abapy was originaly made to solve this problem even if it can perform many other tasks like preprocessing and data management.

1.2 First example

Let's now try to get back the load *vs.* disp curve and plot it using Abapy. First, we have to emphasize that Abaqus Python is not the best place to run any calculation or to plot things. There are many reasons for that among which:

- Abaqus uses old versions of Python, sometimes 5 years behind the current stable versions.
- It can be painful or even impossible to install packages on Abaqus/Python, for example on servers where you don't have admin rights.

This point is essential to understand how Abapy is built. Every function or class that can be used inside Abaqus/Python does not rely on third party packages like Numpy, even if it could be of great utility. On the other hand, classes that work on Abaqus/Python can have methods that use numpy locally because they are not intended to be used inside Abaqus. Then, the post processing with Abapy is intended to be done in two steps:

- Step 1: grabbing raw data inside Abaqus/Python and save it, mainly using serialization built-in module Pickle.
- Step 2: Processing raw data inside a standard Python on which third party modules are available.

Now that we clarified this point, we will work with both Abaqus/Python and Python. The second one will progressively take more importance and become our main concern. We can make a first script that will be executed inside Abaqus/Python. It aims to find where to find the indenter displacement of the force applied on it.

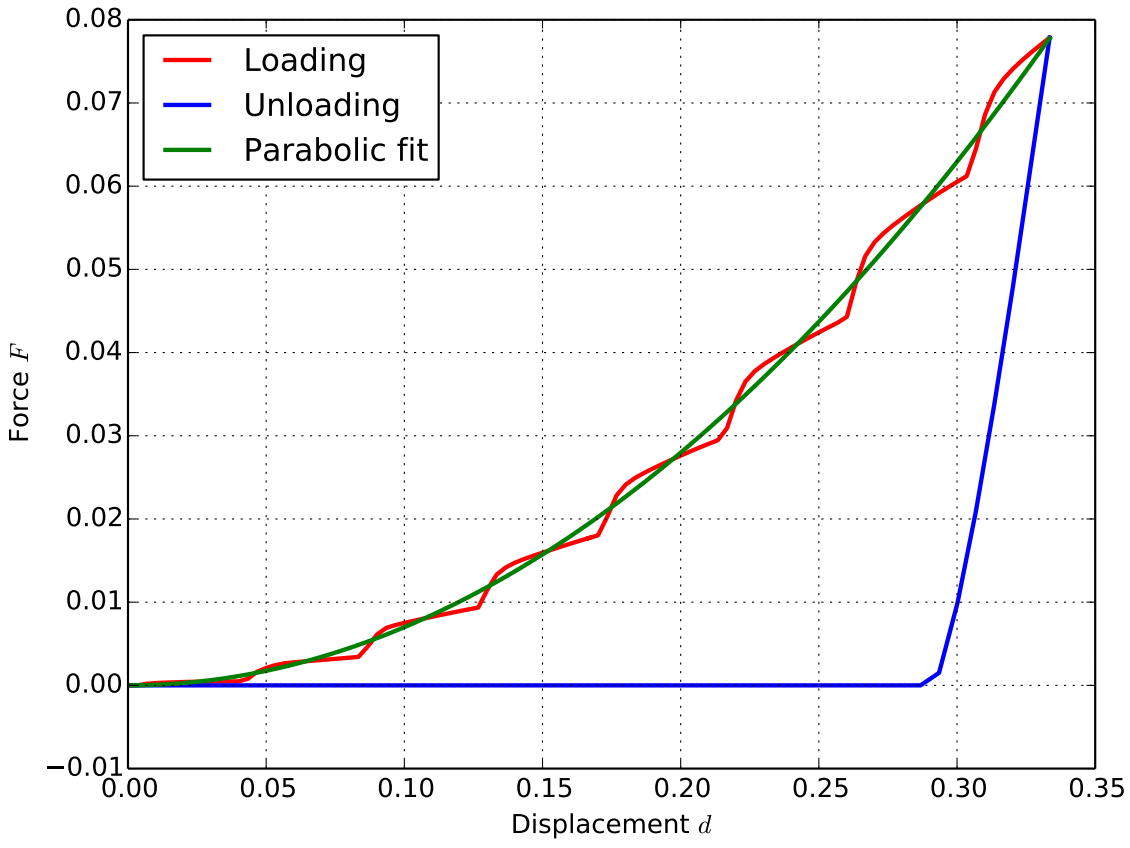
```
# ABAQUS/PYTHON POST PROCESSING SCRIPT
# Run using abaqus python / abaqus viewer -noGUI / abaqus cae -noGUI

# Packages (Abaqus, Abapy and built-in only here)
from odbAccess import openOdb
from abapy.misc import dump
from abapy.postproc import GetHistoryOutputByKey as gho

# Setting up some pathes
workdir = 'workdir'
name = 'indentation_axi'

# Opening the Odb File
odb = openOdb(workdir + '/' + name + '.odb')

# Finding back the position of the reference node of the indenter. Its number is stored inside a node
ref_node_label = odb.rootAssembly.instances['I_INDENTER'].nodeSets['REF_NODE'].nodes[0].label

# Getting back the reaction forces along Y (RF2) and displacements along Y (U2) where they are record
RF2 = gho(odb, 'RF2')
U2  = gho(odb, 'U2')

# Packing data
data = {'ref_node_label': ref_node_label, 'RF2':RF2, 'U2':U2}

# Dumping data
dump(data, workdir + '/' + name + '.pckl')

# Closing Odb
odb.close()
```

Dowload link: `first_example_abq.py`

Then, we can make a second script that is made to work in regular Python

```
# PYTHON POST PROCESSING SCRIPT
# Run using python

# Packages
from abapy.misc import load
import matplotlib.pyplot as plt
import numpy as np

# Setting up some pathes
workdir = 'workdir'
name = 'indentation_axi'

# Getting back raw data
data = load(workdir + '/' + name + '.pckl')

# Post processing
ref_node_label = data['ref_node_label']
force_hist = -data['RF2']['Node I_INDENTER.{0}'.format(ref_node_label)]
disp_hist = -data['U2']['Node I_INDENTER.{0}'.format(ref_node_label)]
```

```
time, force_l = force_hist[0,1].plotable() # Getting back force  during loading
time, disp_l = disp_hist[0,1].plotable()   # Getting backdisplacement during loading
time, force_u = force_hist[2].plotable() # Getting back force  during unloading
time, disp_u = disp_hist[2].plotable()   # Getting backdisplacement during unloading

# Dimensional analysis (E. Buckingham. Physical review, 4, 1914.) shows that the loading curve must b


C_factor = (force_hist[1] / disp_hist[1]**2).average()
disp_fit = np.array(disp_hist[0,1].toArray()[1])
force_fit = C_factor * disp_fit**2


plt.figure()
plt.clf()
plt.plot(disp_l, force_l, 'r-', linewidth = 2., label = 'Loading')
plt.plot(disp_u, force_u, 'b-', linewidth = 2., label = 'Unloading')
plt.plot(disp_fit, force_fit, 'g-', linewidth = 2., label = 'Parabolic fit')
plt.xlabel('Displacement $d$')
plt.ylabel('Force $F$')
plt.grid()
plt.legend(loc = 'upper left')
plt.show()
```



The loading curve is very noisy because the mesh is coarse. Anyway, the loading phase is theoretically parabolic and so, it can be averaged and give a quite good result even if looks ugly.

1.3 Fancier example

Now we want to do fancier things like wrapping all the Abaqus/Python part inside the regular Python part in order to change the problem into a simple function or method evaluation. On a more mechanical point of view, we want to be able to compute the loading prefactor C for any material parameters usable in the von Mises elastic plastic law. We create 2 files:

```
# FANCIER_EXAMPLE: computing the loading prefactor of the indentation load vs. disp curve.
# Run using python

# Packages
from abapy.misc import load
import matplotlib.pyplot as plt
import numpy as np




def compute_C(E = 1. , nu = 0.3, sy = 0.01, abqlauncher = '/opt/Abaqus/6.9/Commands/abaqus', workdir
  '''
  Computes the load prefactor C using Abaqus.

  Inputs:
  * E: sample's Young modulus.
  * nu: samples's Poisson's ratio
  * sy: sample's yield stress (von Mises yield criterion).
  * abqlauncher: absolute path to abaqus launcher.
  * wordir: path to working directory, can be relative.
  * name: name of simulation files.
  * frames: number of frames per step, increase if the simulation does not complete.

  Returns:
  * Load prefactor C (float)
  '''
  import time, subprocess, os
  t0 = time.time() # Starting time recording
  path = workdir + '/' + name
  # Reading the INP target file
  f = open('indentation_axi_target.inp', 'r')
  inp = f.read()
  f.close()
  # Replace the targets in the file
  inp = inp.replace('#E', '{0}'.format(E))
  inp = inp.replace('#NU', '{0}'.format(nu))
  inp = inp.replace('#SY', '{0}'.format(sy))
  inp = inp.replace('#FRAME', '{0}'.format(1./frames))
  # Creating a new inp file
  f = open(path  + '.inp', 'w')
  f.write(inp)
  f.close()
  print 'Created INP file: {0}.inp'.format(path)
  # Then we run the simulation
  print 'Running simulation in Abaqus'
  p = subprocess.Popen( '{0} job={1} input={1}.inp interactive ask_delete=OFF'.format(abqlauncher, na
  trash = p.communicate()

  # Now we test run the post processing script
  print 'Post processing the simulation in Abaqus/Python'
```

```
  p = subprocess.Popen( [abqlauncher,  'viewer', 'noGUI=fancier_example_abq.py'], cwd = '.',stdout =
  trash = p.communicate()
  # Getting back raw data
  data = load(workdir + '/' + name + '.pckl')
  # Post processing
  print 'Post processing the simulation in Python'
  if data['completed']:
    ref_node_label = data['ref_node_label']
    force_hist = -data['RF2']['Node I_INDENTER.{0}'.format(ref_node_label)]
    disp_hist = -data['U2']['Node I_INDENTER.{0}'.format(ref_node_label)]
    trash, force_l = force_hist[0,1].plotable() # Getting back force  during loading
    trash, disp_l = disp_hist[0,1].plotable()   # Getting backdisplacement during loading
    trash, force_u = force_hist[2].plotable() # Getting back force  during unloading
    trash, disp_u = disp_hist[2].plotable()   # Getting backdisplacement during unloading
    C_factor = (force_hist[1] / disp_hist[1]**2).average()

  else:
    print 'Simulation aborted, probably because frame number is to low'
    C_factor = None
  t1 = time.time()
  print 'Time used: {0:.2e} s'.format(t1-t0)
  return C_factor

# Setting up some pathes
workdir = 'workdir'
name = 'indentation_axi_fancier'
abqlauncher = '/opt/Abaqus/6.9/Commands/abaqus'

# Setting material parameters
E = 1.
nu = 0.3
sy = 0.01
frames = 50

# Testing it all
C = compute_C(E = E, nu = nu, sy = sy, frames = frames)
print 'C = {0}'.format(C)
```

Dowload link: `fancier_example.py`

```
# ABAQUS/PYTHON POST PROCESSING SCRIPT
# Run using abaqus python / abaqus viewer -noGUI / abaqus cae -noGUI

# Packages (Abaqus, Abapy and built-in only here)
from odbAccess import openOdb
from abapy.misc import dump
from abapy.postproc import GetHistoryOutputByKey as gho
from abaqusConstants import JOB_STATUS_COMPLETED_SUCCESSFULLY


# Setting up some pathes
workdir = 'workdir'
name = 'indentation_axi_fancier'

# Opening the Odb File
odb = openOdb(workdir + '/' + name + '.odb')

# Testing job status
```

```
data = {}
status = odb.diagnosticData.jobStatus
if status == JOB_STATUS_COMPLETED_SUCCESSFULLY:
  data['completed'] = True
  # Finding back the position of the reference node of the indenter. Its number is stored inside a no
  ref_node_label = odb.rootAssembly.instances['I_INDENTER'].nodeSets['REF_NODE'].nodes[0].label

  # Getting back the reaction forces along Y (RF2) and displacements along Y (U2) where they are reco
  RF2 = gho(odb, 'RF2')
  U2  = gho(odb, 'U2')

  # Packing data
  data['ref_node_label'] = ref_node_label
  data['RF2'] = RF2
  data['U2']  = U2


else:
  data['completed'] = False
# Dumping data
dump(data, workdir + '/' + name + '.pckl')
# Closing Odb
odb.close()
```

Dowload link: `fancier_example_abq.py`

Then, executing `fancier_example.py` gives:

```
>>> execfile('fancier_example.py')
Created INP file: workdir/indentation_axi_fancier.inp
Running simulation in Abaqus
Abaqus License Manager checked out the following licenses:
Abaqus/Standard checked out 5 tokens.
<55 out of 60 licenses remain available>.
Abaqus License Manager checked out the following licenses:
Abaqus/Standard checked out 5 tokens.
<55 out of 60 licenses remain available>.
Post processing the simulation in Abaqus/Python
Abaqus License Manager checked out the following license(s):
"cae" release 6.9 from epua-e172.univ-savoie.fr
<5 out of 6 licenses remain available>.
Post processing the simulation in Python
Time used: 2.93e+01 s
C = 0.6993227005
```

Now you know how how to control the simulation through regular Python but important things are still missing. For example:

- The coarse mesh limits the accuracy of the simulation, a parametric mesh allowing to adjust speed and precision freely would be welcomed.
- What if you need more that the single load prefactor ? A class having many important outputs instead of a function could be nice.
- Simulations take time and it's always frustrating to do them two times, data persistence could of great use here.

All this can be addressed using Abapy with other third party packages like Numpy, Scipy, Matplotlib and SQLAlchemy. The advanced example provides details explanations on this point.

CHAPTER 2

Mesh

Mesh processing tools.

2.1 Nodes

class abapy.mesh.**Nodes**(*labels=[], x=[], y=[], z=[], sets={}, dtf='f', dti='I'*)

Manages nodes for finite element modeling pre/postprocessing and further graphical representations.

Parameters

- **labels** (*list of int > 0*) – list of node labels.
- **x** (*list floats*) – x coordinate of nodes.
- **y** (*list floats*) – y coordinate of nodes.
- **z** (*list floats*) – z coordinate of nodes.
- **sets** (*dict with str keys and where values are list of ints > 0.*) – node sets
- **dti** (*'I' or 'H'*) – int data type in array.array
- **dtf** (*'f' or 'd'*) – float data type in array.array

```
>>> from abapy.mesh import Nodes
>>> labels = [1,2]
>>> x = [0., 1.]
>>> y = [0., 2.]
>>> z = [0., 0.]
>>> sets = {'mySet': [1,2]}
>>> nodes = Nodes(labels = labels, x = x, y = y, z = z, sets = sets)
>>> nodes
<Nodes class instance: 2 nodes>
>>> print nodes
Nodes class instance:
Nodes:
Label x       y       z
1     0.0     0.0     0.0
2     1.0     2.0     0.0
Sets:
Label Nodes
myset 1,2
>>> from abapy.mesh import Nodes
```

```
>>> labels = range(1,11) # 10 nodes
>>> x = labels
>>> y = labels
>>> z = [0. for i in x]
>>> nodes = Nodes(labels=labels, x=x, y=y, z=z)
>>> nodes.add_set('myset',[4,5,6,9]) # A set
>>> print nodes
Nodes class instance:
Nodes:
Label x       y       z
1     1.0     1.0     0.0
2     2.0     2.0     0.0
3     3.0     3.0     0.0
4     4.0     4.0     0.0
5     5.0     5.0     0.0
6     6.0     6.0     0.0
7     7.0     7.0     0.0
8     8.0     8.0     0.0
9     9.0     9.0     0.0
10    10.0    10.0    0.0
Sets:
Label Nodes
myset 4,5,6,9
>>> print nodes[5] # requesting node 5
Nodes class instance:
Nodes:
Label x       y       z
5     5.0     5.0     0.0
Sets:
Label Nodes
myset 5
>>> print nodes[5,4,10]  # requesting nodes 5, 4 and 10. Note that output has ordered nodes and
Nodes class instance:
Nodes:
Label x       y       z
4     4.0     4.0     0.0
5     5.0     5.0     0.0
10    10.0    10.0    0.0
Sets:
Label Nodes
myset 5,4
>>> print nodes['myset'] # requesting nodes using set key
Nodes class instance:
Nodes:
Label x       y       z
4     4.0     4.0     0.0
5     5.0     5.0     0.0
6     6.0     6.0     0.0
9     9.0     9.0     0.0
Sets:
Label Nodes
myset 4,5,6,9
>>> print nodes['myset',10] # mixed request: nodes in myset AND node 10.
Nodes class instance:
Nodes:
Label x       y       z
4     4.0     4.0     0.0
5     5.0     5.0     0.0
```

```
    6     6.0     6.0     0.0
    9     9.0     9.0     0.0
    10    10.0    10.0    0.0
    Sets:
    Label Nodes
    myset 4,5,6,9
    >>> print nodes[1:9:2] # slice
    Nodes class instance:
    Nodes:
    Label x       y       z
    1     1.0     1.0     0.0
    3     3.0     3.0     0.0
    5     5.0     5.0     0.0
    7     7.0     7.0     0.0
    Sets:
    Label Nodes
    myset 5
```

2.1.1 Add/remove/get data

Nodes.**add_node**(*label=None*, *x=0.0*, *y=0.0*, *z=0.0*, *toset=None*)

Adds one node to Nodes instance.

Parameters

- **label** – If None, label is automatically chosen to be the highest existing label + 1 (default: None). If label (and susequently node) already exists, a warning is printed and the node is not added and sets that could be created ar not created.
- **x** (*float*) – x coordinate of node.
- **y** (*float*) – y coordinate of node.
- **z** (*float*) – z coordinate of node.
- **tosets** – set(s) to which the node should be added. If a set does not exist, it is created. If None, the node is not added to any set.

```
    >>> from abapy.mesh import Nodes
    >>> nodes = Nodes()
    >>> nodes.add_node(label = 10, x = 0., y = 0., z = 0., toset = 'firstSet')
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    10  0.0     0.0     0.0
    Sets:
    Label       Nodes
    firstset    10
    >>> nodes.add_node(x = 0., y = 0., z = 0., toset = ['firstSet', 'secondSet'])
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    10  0.0     0.0     0.0
    11  0.0     0.0     0.0
    Sets:
    Label       Nodes
```

```
    firstset    10,11
    secondset   11
```

Nodes.**drop_node**(*label*)

Removes one node to Nodes instance. The node is also removed from sets, if a set happens to be empty, it is also removed.

Parameters **label** (*int > 0*) – node be removed's label.

```
>>> from abapy.mesh import Nodes
>>> nodes = labels = [1,2]
>>> x = [0., 1.]
>>> y = [0., 2.]
>>> z = [0., 0.]
>>> sets = {'mySet': [2]}
>>> nodes = Nodes(labels = labels, x = x, y = y, z = z, sets = sets)
>>> print nodes
Nodes class instance:
Nodes:
Label       x       y       z
1   0.0     0.0     0.0
2   1.0     2.0     0.0
Sets:
Label       Nodes
myset       2
>>> nodes.drop_node(2)
>>> print nodes
Nodes class instance:
Nodes:
Label       x       y       z
1   0.0     0.0     0.0
Sets:
Label       Nodes
```

Nodes.**add_set**(*label*, *nodes*)

Adds a node set to the Nodes instance or appends nodes to existing node sets.

Parameters

- **label** (*string*) – set to be added's label.
- **nodes** (*int or list of ints*) – nodes to be added in the set.

Note: set labels are always lower case in this class to be case insensitive. This way to proceed is coherent with Abaqus.

```
>>> from abapy.mesh import Nodes
>>> nodes = Nodes()
>>> labels = [1,2]
>>> x = [0., 1.]
>>> y = [0., 2.]
>>> z = [0., 0.]
>>> sets = {'mySet': 2}
>>> nodes = Nodes(labels = labels, x = x, y = y, z = z, sets = sets)
>>> print nodes
Nodes class instance:
Nodes:
Label       x       y       z
```

```
    1   0.0     0.0     0.0
    2   1.0     2.0     0.0
    Sets:
    Label       Nodes
    myset       2
    >>> nodes.add_set('MYSET',1)
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    1   0.0     0.0     0.0
    2   1.0     2.0     0.0
    Sets:
    Label       Nodes
    myset       2,1
    >>> nodes.add_set('MyNeWseT',[1,2])
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    1   0.0     0.0     0.0
    2   1.0     2.0     0.0
    Sets:
    Label       Nodes
    myset       2,1
    mynewset    1,2
```

Nodes.**add_set_by_func**(*name*, *func*)

Creates a node set using a function of x, y, z and labels (given as `numpy.array`). Must get back a boolean array of the same size.

Parameters

- **name** (*string*) – set label.
- **func** (*function*) – function of x, y ,z and labels

```
    >>> mesh.nodes.add_set_by_func('setlabel', lambda x, y, z, labels: x == 0.)
```

Nodes.**drop_set**(*label*)

Drops a set without removing elements and nodes.

Parameters **label** (*string*) – label of the to be removed.

```
    >>> from abapy.mesh import Nodes
    >>> labels = range(1,11)
    >>> x = labels
    >>> y = labels
    >>> z = [0. for i in x]
    >>> nodes = Nodes(labels=labels, x=x, y=y, z=z)
    >>> nodes.add_set('myset',[4,5,6,9])
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    1   1.0     1.0     0.0
    2   2.0     2.0     0.0
    3   3.0     3.0     0.0
    4   4.0     4.0     0.0
    5   5.0     5.0     0.0
```

```
    6   6.0     6.0     0.0
    7   7.0     7.0     0.0
    8   8.0     8.0     0.0
    9   9.0     9.0     0.0
    10  10.0    10.0    0.0
    Sets:
    Label       Nodes
    myset       4,5,6,9
    >>> nodes.drop_set('someSet')
    Info: sets someset does not exist and cannot be dropped.
    >>> nodes.drop_set('MYSET')
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    1   1.0     1.0     0.0
    2   2.0     2.0     0.0
    3   3.0     3.0     0.0
    4   4.0     4.0     0.0
    5   5.0     5.0     0.0
    6   6.0     6.0     0.0
    7   7.0     7.0     0.0
    8   8.0     8.0     0.0
    9   9.0     9.0     0.0
    10  10.0    10.0    0.0
    Sets:
    Label       Nodes
```

2.1.2 Modifications

Nodes.**translate**(*x=0.0*, *y=0.0*, *z=0.0*)

Translates all the nodes.

Parameters

- **x** (*float*) – translation along x value.
- **y** (*float*) – translation along y value.
- **z** (*float*) – translation along z value.

```
    >>> from abapy.mesh import Nodes
    >>> nodes = Nodes()
    >>> labels = [1,2]
    >>> x = [0., 1.]
    >>> y = [0., 2.]
    >>> z = [0., 0.]
    >>> sets = {'mySet': 2}
    >>> nodes = Nodes(labels = labels, x = x, y = y, z = z, sets = sets)
    >>> nodes.translate(x = 1., y=0., z = -4.)
    >>> print nodes
    Nodes class instance:
    Nodes:
    Label       x       y       z
    1   1.0     0.0     -4.0
    2   2.0     2.0     -4.0
    Sets:
```

```
    Label       Nodes
    myset       2
```

Nodes.**apply_displacement**(*disp*)

Applies a displacement field to the nodes.

Parameters **disp** (VectorFieldOutput instance) – displacement field.

```
    from abapy.mesh import Mesh, Nodes, RegularQuadMesh
    import matplotlib.pyplot as plt
    from numpy import cos, sin, pi
    from copy import copy
    def function(x, y, z, labels):
      r0 = 1.
      theta = 2 * pi * x
      r = y + r0
      ux = -x + r * cos(theta)
      uy = -y + r * sin(theta)
      uz = 0. * z
      return ux, uy, uz
    N1, N2 = 100, 25
    l1, l2 = .75, 1.
    Ncolor = 20
    mesh = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
    vectorField = mesh.nodes.eval_vectorFunction(function)
    mesh.nodes.apply_displacement(vectorField)
    field = vectorField.get_coord(2) # we chose to plot coordinate 2
    field2 = vectorField.get_coord(2) # we chose to plot coordinate 2
    x,y,z = mesh.get_edges() # Mesh edges
    X,Y,Z,tri = mesh.dump2triplot()
    xb,yb,zb = mesh.get_border() # mesh borders
    xe, ye, ze = mesh.get_edges()
    fig = plt.figure(figsize=(10,10))
    fig.gca().set_aspect('equal')
    plt.axis('off')
    plt.plot(xb,yb,'k-', linewidth = 2.)
    plt.plot(xe, ye,'k-', linewidth = .5)
    plt.tricontour(X,Y,tri,field.data, Ncolor, colors = 'black')
    color = plt.tricontourf(X,Y,tri,field.data, Ncolor)
    plt.colorbar(color)
    plt.show()
```

Nodes.**closest_node**(*label*)

Finds the closest node of an existing node.

Parameters **label** (*int > 0*) – node label to be used.

Return type label (int > 0) and distance (float) of the closest node.

Nodes.**replace_node**(*old*, *new*)

Replaces a node of given label (old) by another existing node (new).

Mesh.**apply_reflection**(*point=(0.0, 0.0, 0.0)*, *normal=(1.0, 0.0, 0.0)*)

Applies a reflection symmetry to the mesh instance. The reflection plane is defined by a point and a normal direction.

Parameters

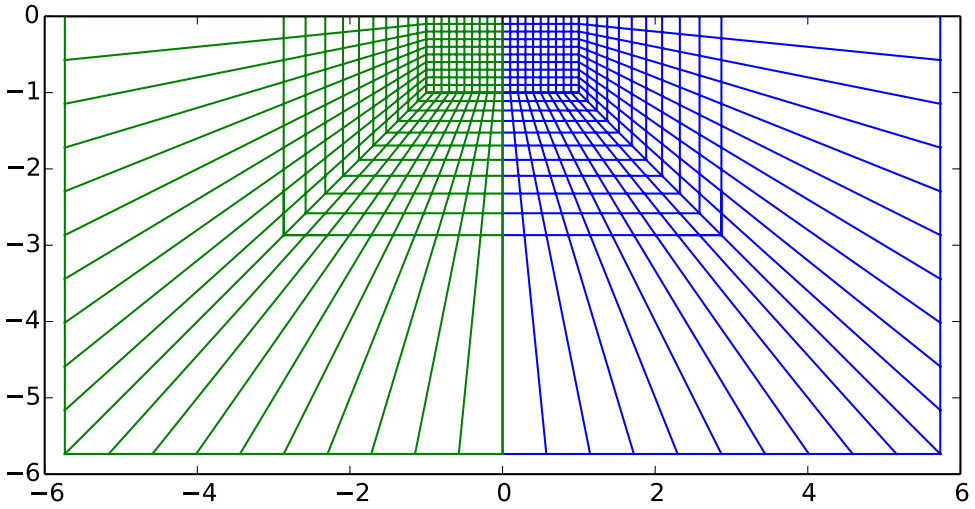
- **point** (*tuple or list containing 3 floats*) – coordinates of a point of the reflection plane.

- **normal** (*tuple or list containing 3 floats*) – normal vector to the reflection plane

Return type Mesh instance

..note: This method can lead to coherence problems with surfaces, this problem will be addressed in the future. Surfaces are removed by this operation untill this problem is solved.

```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import ParamInfiniteMesh
import matplotlib.pyplot as plt

point  = (0., 0., 0.)
normal = (1., 0., 0.)
m0 = ParamInfiniteMesh()
x0, y0, z0 = m0.get_edges()
m1 = m0.apply_reflection(normal = normal, point = point)
x1, y1, z1 = m1.get_edges()
plt.plot(x0, y0)
plt.plot(x1, y1)
plt.gca().set_aspect('equal')
plt.show()
```



2.1.3 Export

Nodes.**dump2inp**()

Dumps Nodes instance to Abaqus INP format.

Return type string

```
>>> from abapy.mesh import Nodes
>>> nodes = Nodes()
>>> labels = [1,2]
>>> x = [0., 1.]
>>> y = [0., 2.]
>>> z = [0., 0.]
>>> sets = {'mySet': 2}
>>> nodes = Nodes(labels = labels, x = x, y = y, z = z, sets = sets)
>>> out = nodes.dump2inp()
```

2.1.4 Tools

`Nodes.`**`eval_function`**(*function*)

Evals a function at each node and returns a `FieldOutput` instance.

Parameters **`function`** (*`function`*) – a function with arguments x, y and z (float `numpy.arrays` containing nodes coordinates) and labels (int `numpy.array`). Field should not depend on labels but on some vicious problem, it could be useful. The function should return 1 array.

Return type `'FieldOutput'` instance.

```
from abapy.mesh import Mesh, Nodes, RegularQuadMesh
import matplotlib.pyplot as plt
from numpy import cos, pi
def function(x, y, z, labels):
 r = (x**2 + y**2)**.5
 return cos(2*pi*x)*cos(2*pi*y)/(r+1.)
N1, N2 = 100, 25
l1, l2 = 4., 1.
Ncolor = 20
mesh = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
field = mesh.nodes.eval_function(function)
x,y,z = mesh.get_edges() # Mesh edges
X,Y,Z,tri = mesh.dump2triplot()
xb,yb,zb = mesh.get_border()
fig = plt.figure(figsize=(16,4))
fig.gca().set_aspect('equal')
fig.frameon = True
plt.plot(xb,yb,'k-', linewidth = 2.)
plt.xticks([0,l1],['$0$', '$l_1$'], fontsize = 15.)
plt.yticks([0,l2],['$0$', '$l_2$'], fontsize = 15.)
plt.tricontourf(X,Y,tri,field.data, Ncolor)
plt.tricontour(X,Y,tri,field.data, Ncolor, colors = 'black')
plt.show()
```

`Nodes.`**`eval_vectorFunction`**(*function*)

Evals a vector function at each node and returns a `VectorFieldOutput` instance.

Parameters **`function`** (*`function`*) – a vector function with arguments x, y and z (float `numpy.arrays` containing nodes coordinates) and labels (int `numpy.array`). Field should not depend on labels but on some vicious problem, it could be useful. The function should return 3 arrays.

Return type `'FieldOutput'` instance.

```
from abapy.mesh import Mesh, Nodes, RegularQuadMesh
import matplotlib.pyplot as plt
from numpy import cos, sin, pi
def function(x, y, z, labels):
  r0 = 1.
  theta = 2 * pi * x
  r = y + r0
  ux = -x + r * cos(theta)
  uy = -y + r * sin(theta)
  uz = 0. * z
  return ux, uy, uz
N1, N2 = 100, 25
l1, l2 = 1., 1.
Ncolor = 20
mesh = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
vectorField = mesh.nodes.eval_vectorFunction(function)
field = vectorField.get_coord(1) # we chose to plot coordinate 1
field2 = vectorField.get_coord(2) # we chose to plot coordinate 1
field3 = vectorField.norm() # we chose to plot norm
fig = plt.figure(figsize=(16,4))
ax = fig.add_subplot(131)
ax2 = fig.add_subplot(132)
ax3 = fig.add_subplot(133)
ax.set_aspect('equal')
ax2.set_aspect('equal')
ax3.set_aspect('equal')
ax.set_xticks([])
ax.set_yticks([])
ax2.set_xticks([])
ax2.set_yticks([])
ax3.set_xticks([])
ax3.set_yticks([])
ax.set_frame_on(False)
ax2.set_frame_on(False)
ax3.set_frame_on(False)
ax.set_title(r'$V_1$')
ax2.set_title(r'$V_2$')
ax3.set_title(r'$\sqrt{\vec{V}^2}$')
ax3.set_title(r'$||\vec{V}||$')
x,y,z = mesh.get_edges() # Mesh edges
xt,yt,zt = mesh.convert2tri3().get_edges() # Triangular mesh edges
xb,yb,zb = mesh.get_border()
X,Y,Z,tri = mesh.dump2triplot()
ax.plot(xb,yb,'k-', linewidth = 2.)
ax.tricontourf(X,Y,tri,field.data, Ncolor)
ax.tricontour(X,Y,tri,field.data, Ncolor, colors = 'black')
ax2.plot(xb,yb,'k-', linewidth = 2.)
ax2.tricontourf(X,Y,tri,field2.data, Ncolor)
ax2.tricontour(X,Y,tri,field2.data, Ncolor, colors = 'black')
ax3.plot(xb,yb,'k-', linewidth = 2.)
ax3.tricontourf(X,Y,tri,field3.data, Ncolor)
ax3.tricontour(X,Y,tri,field3.data, Ncolor, colors = 'black')
ax.set_xlim([-.1*l1,1.1*l1])
ax.set_ylim([-.1*l2,1.1*l2])
ax2.set_xlim([-.1*l1,1.1*l1])
ax2.set_ylim([-.1*l2,1.1*l2])
ax3.set_xlim([-.1*l1,1.1*l1])
ax3.set_ylim([-.1*l2,1.1*l2])
```

```
    plt.show()
```

Nodes.**eval_tensorFunction**(*function*)

Evaluates a tensor function at each node and returns a `tensorFieldOutput` instance.

Parameters **function** (*function*) – a tensor function with arguments x, y and z (float `numpy.arrays` containing nodes coordinates) and labels (int `numpy.array`). Field should not depend on labels but on some vicious problem, it could be useful. The function should return 6 arrays corresponding to indices ordered as follows: 11, 22, 33, 12, 13, 23.

Return type `'FieldOutput'` instance.

```
    from abapy.mesh import Mesh, Nodes, RegularQuadMesh
    import matplotlib.pyplot as plt
    from numpy import cos, sin, pi, linspace
    def boussinesq(r, z, theta, labels):
      '''
      Stress solution of the Boussinesq point loading of semi infinite elastic half space for a forc
      '''
      from math import pi
      from numpy import zeros_like
      nu = 0.3
      rho = (r**2 + z**2)**.5
      s_rr = -1./(2. * pi * rho**2) * ( (-3. * r**2 * z)/(rho**3) + (1.-2. * nu)*rho / (rho + z) )
      #s_rr = 1./(2.*pi) *( (1-2*nu) * ( r**-2 -z / (rho * r**2)) - 3 * z * r**2 / rho**5 )
      s_zz = 3. / (2. *pi ) * z**3 / rho**5
      s_tt = -( 1. - 2. * nu) / (2. * pi * rho**2 ) * ( z/rho - rho / (rho + z) )
      #s_tt = ( 1. - 2. * nu) / (2. * pi ) * ( 1. / r**2 -z/( rho * r**2) -z / rho**3  )
      s_rz = -3./ (2. * pi) * r * z**2 / rho **5
      s_rt = zeros_like(r)
      s_zt = zeros_like(r)
      return s_rr, s_zz, s_tt, s_rz, s_rt, s_zt

      return ux, uy, uz


    N1, N2 = 50, 50
    l1, l2 = 1., 1.
    Ncolor = 200
    levels = linspace(0., 10., 20)

    mesh = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
    # Finding the node located at x = y =0.:
    nodes = mesh.nodes
    for i in xrange(len(nodes.labels)):
      if nodes.x[i] == 0. and nodes.y[i] == 0.: node = nodes.labels[i]
    mesh.drop_node(node)
    tensorField = mesh.nodes.eval_tensorFunction(boussinesq)
    field = tensorField.get_component(22) # sigma_zz
    field2 = tensorField.vonmises() # von Mises stress
    field3 = tensorField.pressure() # pressure

    fig = plt.figure(figsize=(16,4))
    ax = fig.add_subplot(131)
    ax2 = fig.add_subplot(132)
    ax3 = fig.add_subplot(133)
    ax.set_aspect('equal')
    ax2.set_aspect('equal')
    ax3.set_aspect('equal')
```

```
    ax.set_xticks([])
    ax.set_yticks([])
    ax2.set_xticks([])
    ax2.set_yticks([])
    ax3.set_xticks([])
    ax3.set_yticks([])
    ax.set_frame_on(False)
    ax2.set_frame_on(False)
    ax3.set_frame_on(False)
    ax.set_title(r'$\sigma_{zz}$')
    ax2.set_title(r'Von Mises $\sigma_{eq}$')
    ax3.set_title(r'Pressure $p$')
    xt,yt,zt = mesh.convert2tri3().get_edges() # Triangular mesh edges
    xb,yb,zb = mesh.get_border()

    X,Y,Z,tri = mesh.dump2triplot()

    ax.plot(xb,yb,'k-', linewidth = 2.)
    ax.tricontourf(X,Y,tri,field.data, levels = levels)
    ax.tricontour(X,Y,tri,field.data, levels = levels, colors = 'black')
    ax2.plot(xb,yb,'k-', linewidth = 2.)
    ax2.tricontourf(X,Y,tri,field2.data, levels = levels)
    ax2.tricontour(X,Y,tri,field2.data, levels = levels, colors = 'black')
    ax3.plot(xb,yb,'k-', linewidth = 2.)
    ax3.tricontourf(X,Y,tri,field3.data, levels = sorted(-levels))
    ax3.tricontour(X,Y,tri,field3.data, levels = sorted(-levels), colors = 'black')
    ax.set_xlim([-.1*l1,1.1*l1])
    ax.set_ylim([-.1*l2,1.1*l2])
    ax2.set_xlim([-.1*l1,1.1*l1])
    ax2.set_ylim([-.1*l2,1.1*l2])
    ax3.set_xlim([-.1*l1,1.1*l1])
    ax3.set_ylim([-.1*l2,1.1*l2])
    plt.show()
```

Nodes.**boundingBox**(*margin=0.1*)

Returns the dimensions of a cartesian box containing the mesh with a relative margin.

Parameters **margin** (*float*) – relative margin of the box. O. means no margin, 0.1 is default.

Return type tuple containing 3 tuples with x, y and z limits of the box.

2.2 Mesh

class abapy.mesh.**Mesh**(*nodes=None*, *connectivity=[]*, *space=[]*, *labels=[]*, *name=None*, *sets={}*, *surfaces={}*)

Manages meshes for finite element modeling pre/postprocessing and further graphical representations.

Parameters

- **nodes** (*Nodes class instance or None*) – nodes container. If None, a void Nodes instance will be used. The values of dti and dtf used by nodes are extended to mesh.
- **labels** – elements labels
- **connectivity** (*list of lists each containing ints > 0*) – elements connectivities using node labels

- **space** (*list of ints in [1,2,3]*) – elements embedded spaces. This formulation is simple and allows to distinguish 1D elements (space = 1), surface elements (space = 2) and volumic elements (space = 3)
- **name** (*list of strings*) – elements names used, for example in a FEM code: 'CAX4, C3D8, ...'
- **sets** (*dict with string keys and list of ints > 0 values*) – element sets
- **surface** – dict with str keys containing tuples with 2 elements, the first being the name of an element set and the second the number of the face.

```
>>> from abapy.mesh import Mesh, Nodes
>>> mesh = Mesh()
>>> nodes = mesh.nodes
>>> # Adding some nodes
>>> nodes.add_node(label = 1, x = 0. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 2, x = 1. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 3, x = 1. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 4, x = 0. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 5, x = 2. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 6, x = 2. ,y = 1. , z = 0.)
>>> # Adding some elements
>>> mesh.add_element(label=1, connectivity = (1,2,3,4), space =2, name = 'QUAD4', toset='mySet'
>>> mesh.add_element(label=2, connectivity = (2,5,6,3), space =2, name = 'QUAD4' )
>>> print mesh[1]
Mesh class instance:
Elements:
Label Connectivity            Space   Name
1     [1L, 2L, 3L, 4L]        2D      QUAD4
Sets:
Label Elements
myset 1
>>> print mesh[1,2] # requesting elements with labels 1 and 2
Mesh class instance:
Elements:
Label Connectivity            Space   Name
1     [1L, 2L, 3L, 4L]        2D      QUAD4
2     [2L, 5L, 6L, 3L]        2D      QUAD4
Sets:
Label Elements
myset 1
>>> print mesh[1:2:1] # requesting elements with labels in range(1,2,1)
Mesh class instance:
Elements:
Label Connectivity            Space   Name
1     [1L, 2L, 3L, 4L]        2D      QUAD4
Sets:
Label Elements
myset 1
>>> print mesh['mySet']
Mesh class instance:
Elements:
Label Connectivity            Space   Name
1     [1L, 2L, 3L, 4L]        2D      QUAD4
Sets:
Label Elements
myset 1
```

```
>>> print mesh['myset'] # requesting elements that belong to set 'myset'
Mesh class instance:
Elements:
Label Connectivity            Space   Name
1     [1L, 2L, 3L, 4L]        2D      QUAD4
Sets:
Label Elements
myset 1
>>> print mesh['ImNoSet']
Mesh class instance:
Elements:
Label Connectivity            Space   Name
Sets:
Label Elements
```

2.2.1 Add/remove/get data

Mesh.**add_element**(*connectivity*, *space*, *label=None*, *name=None*, *toset=None*)

Adds an element.

Parameters

- **connectivity** (*list of int > 0*) – element connectivity using node labels.
- **space** (*int in [1,2,3]*) – element embedded space, can be 1 for lineic element, 2 for surfacic element and 3 for volumic element.
- **name** (*string*) – element name used in fem code.
- **toset** (*string or list of strings*) – set(s) to which element is to be added. If a set does not exist, it is created.

```
>>> from abapy.mesh import Mesh, Nodes
>>> mesh = Mesh()
>>> nodes = mesh.nodes
>>> # Adding some nodes
... nodes.add_node(label = 1, x = 0. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 2, x = 1. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 3, x = 1. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 4, x = 0. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 5, x = 2. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 6, x = 2. ,y = 1. , z = 0.)
>>> # Adding some elements
... mesh.add_element(label=1, connectivity = (1,2,3,4), space =2, name = 'QUAD4', toset='mySet'
>>> mesh.add_element(label=2, connectivity = (2,5,6,3), space =2, name = 'QUAD4', toset = ['mySe
>>> print mesh
Mesh class instance:
Elements:
Label       Connectivity            Space   Name
1   [1L, 2L, 3L, 4L]        2D      QUAD4
2   [2L, 5L, 6L, 3L]        2D      QUAD4
Sets:
Label       Elements
myotherset  2
myset       1,2
```

Mesh.**drop_element**(*label*)

Removes one element to Mesh instance. The element is also removed from sets and surfaces, if a set or surface happens to be empty, it is also removed.

Parameters **label** (*int > 0*) – element to be removed's label.

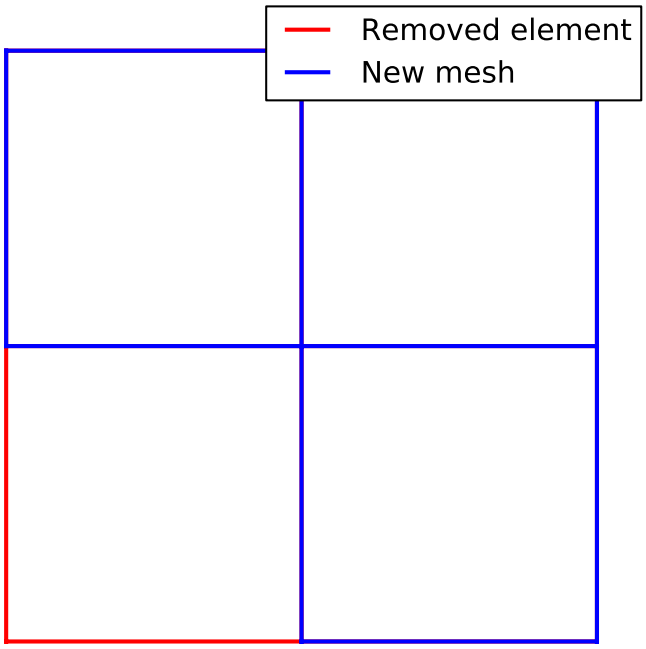
```
>>> from abapy.indentation import ParamInfiniteMesh
>>> from copy import copy
>>>
>>> # Let's create a mesh containing a surface:
... m = ParamInfiniteMesh(Na = 2, Nb = 2)
>>> print m.surfaces
{'samp_surf': [('top_elem', 3)]}
>>> elem_to_remove = copy(m.sets['top_elem'])
>>> # Let's remove all elements in the surface:
... for e in elem_to_remove:
...   m.drop_element(e)
... # We can see that sets and surfaces are removed when they become empty
...
>>> print m.surfaces
{}
```

Mesh.**drop_node**(*label*)

Drops a node from mesh.nodes instance. This method differs from to the nodes.drop_node element because it removes the node but also all elements containing the node in the mesh instance.

Parameters **label** (*int > 0*) – node to be removed's label.

```
from abapy.mesh import RegularQuadMesh
from matplotlib import pyplot as plt
# Creating a mesh
m = RegularQuadMesh(N1 = 2, N2 = 2)
x0, y0, z0 = m.get_edges()
# Finding the node located at x = y =0.:
nodes = m.nodes
for i in xrange(len(nodes.labels)):
  if nodes.x[i] == 0. and nodes.y[i] == 0.: node = nodes.labels[i]
# Removing this node
m.drop_node(node)
x1, y1, z1 = m.get_edges()
bbx, bby, bbz = m.nodes.boundingBox()
plt.figure()
plt.clf()
plt.gca().set_aspect('equal')
plt.axis('off')
plt.xlim(bbx)
plt.ylim(bby)
plt.plot(x0,y0, 'r-', linewidth = 2., label = 'Removed element')
plt.plot(x1,y1, 'b-', linewidth = 2., label = 'New mesh')
plt.legend()
plt.show()
```



Mesh.**add_set**(*label*, *elements*)

Adds a new set or appends elements to an existing set.

Parameters

- **label** (*string*) – set label to be added.
- **elements** (*int > 0 or list of int > 0*) – element(s) that belong to the step.

```
>>> from abapy.mesh import Mesh, Nodes
>>> mesh = Mesh()
>>> nodes = mesh.nodes
>>> # Adding some nodes
>>> nodes.add_node(label = 1, x = 0. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 2, x = 1. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 3, x = 1. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 4, x = 0. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 5, x = 2. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 6, x = 2. ,y = 1. , z = 0.)
>>> # Adding some elements
>>> mesh.add_element(label=1, connectivity = (1,2,3,4), space =2, name = 'QUAD4')
>>> mesh.add_element(label=2, connectivity = (2,5,6,3), space =2, name = 'QUAD4')
>>> # Adding sets
>>> mesh.add_set(label = 'niceSet', elements = 1)
>>> mesh.add_set(label = 'veryNiceSet', elements = [1,2])
>>> mesh.add_set(label = 'simplyTheBestSet', elements = 1)
>>> mesh.add_set(label = 'simplyTheBestSet', elements = 2)
>>> print mesh
Mesh class instance:
```

```
    Elements:
    Label       Connectivity            Space   Name
    1   [1L, 2L, 3L, 4L]        2D      QUAD4
    2   [2L, 5L, 6L, 3L]        2D      QUAD4
    Sets:
    Label       Elements
    niceset     1
    veryniceset 1,2
    simplythebestset    1,2
```

`Mesh.`**`drop_set`**(*label*)

Goal: drops a set without removing elements and nodes. Inputs:

•label: set label to be dropped, must be string.

`Mesh.`**`add_surface`**(*label*, *description*)

Adds or expands an element surface (*i. e.* a group a element faces). Surfaces are used to define contact interactions in simulations.

Parameters

- **label** (*string*) – surface label.
- **description** (*list containing tuples each containing a string and an int*) – list of (element set label , face number) tuples.

```
    >>> from abapy.mesh import RegularQuadMesh
    >>> mesh = RegularQuadMesh()
    >>> mesh.add_surface('topsurface', [ ('top', 1) ])
    >>> mesh.add_surface('topsurface', [ ('top', 2) ])
    >>> mesh.surfaces
    {'topsurface': [('top', 1), ('top', 2)]}
```

`Mesh.`**`node_set_to_surface`**(*surface*, *nodeSet*)

Builds a surface from a node set.

Parameters

- **surface** (*string*) – surface label
- **nodeSet** (*string*) – nodeSet label

```
    from abapy import mesh

    m = mesh.RegularQuadMesh(N1 = 4, N2 = 4, l1 = 2., l2 = 6.)
    m.nodes.sets = {}
    m.nodes.add_set_by_func("top_nodes", lambda x,y,z,labels : y == 6.)
    m.node_set_to_surface("top_surface", "top_nodes")
```

`Mesh.`**`replace_node`**(*old*, *new*)

Replaces a node of given label (old) by another existing node (new). This version of `replace_node` differs from the version of the `Nodes` class because it also updates elements connectivity. When working with mesh (an not only nodes), this version should be used.

Parameters

- **old** (*int > 0*) – node label to be replaced.
- **new** (*int > 0*) – node label of the node replacing old.

```
    >>> from abapy.mesh import RegularQuadMesh
    >>> N1, N2 = 1,1
```

```
>>> mesh = RegularQuadMesh(N1, N2)
>>> mesh.replace_node(1,2)
Info: element 1 maybe have become degenerate due du node replacing.
>>> print mesh
Mesh class instance:
Elements:
Label       Connectivity            Space   Name
1   [2L, 4L, 3L]    2D      QUAD4
Sets:
Label       Elements
```

Mesh.**simplify_nodes**(*crit_distance=1e-10*)

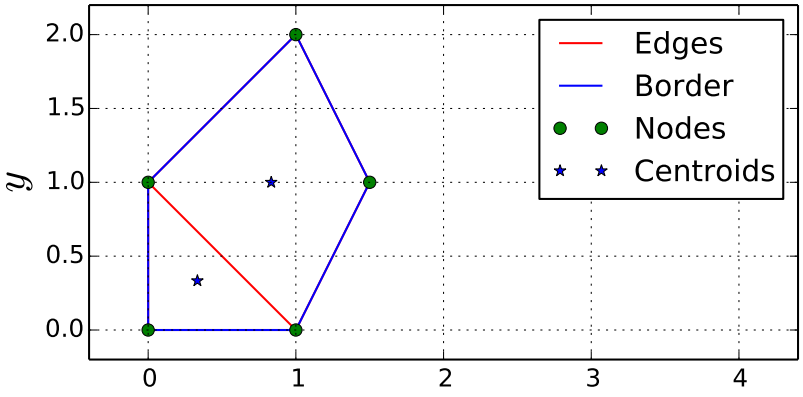
Mesh.**add_field**(*field*, *label*)
Add a field to the mesh.

2.2.2 Useful data

Mesh.**centroids**()
Returns a dictionnary containing the coordinates of all the nodes belonging to earch element.

```
from abapy.mesh import Mesh
from matplotlib import pyplot as plt
import numpy as np

N1,N2 = 10,5 # Number of elements
l1, l2 = 4., 2. # Mesh size
fs = 20. # fontsize
mesh = Mesh()
nodes =  mesh.nodes
nodes.add_node(label = 1, x = 0., y = 0.)
nodes.add_node(label = 2, x = 1., y = 0.)
nodes.add_node(label = 3, x = 0., y = 1.)
nodes.add_node(label = 4, x = 1.5, y = 1.)
nodes.add_node(label = 5, x = 1., y = 2.)
mesh.add_element(label = 1, connectivity = [1,2,3], space = 2)
mesh.add_element(label = 2, connectivity = [2,4,5,3], space = 2)


centroids = mesh.centroids()

plt.figure(figsize=(8,3))
plt.gca().set_aspect('equal')
nodes = mesh.nodes
xn, yn, zn = np.array(nodes.x), np.array(nodes.y), np.array(nodes.z) # Nodes coordinates
xe,ye,ze = mesh.get_edges() # Mesh edges
xb,yb,zb = mesh.get_border() # Mesh border


plt.plot(xe,ye,'r-',label = 'Edges')
plt.plot(xb,yb,'b-',label = 'Border')
plt.plot(xn,yn,'go',label = 'Nodes')
plt.xlim([-.1*l1,1.1*l1])
plt.ylim([-.1*l2,1.1*l2])
plt.xlabel('$x$',fontsize = fs)
plt.ylabel('$y$',fontsize = fs)
plt.plot(centroids[:,0], centroids[:,1], '*', label = "Centroids")
```

```
    plt.legend()
    plt.grid()
    plt.show()
```

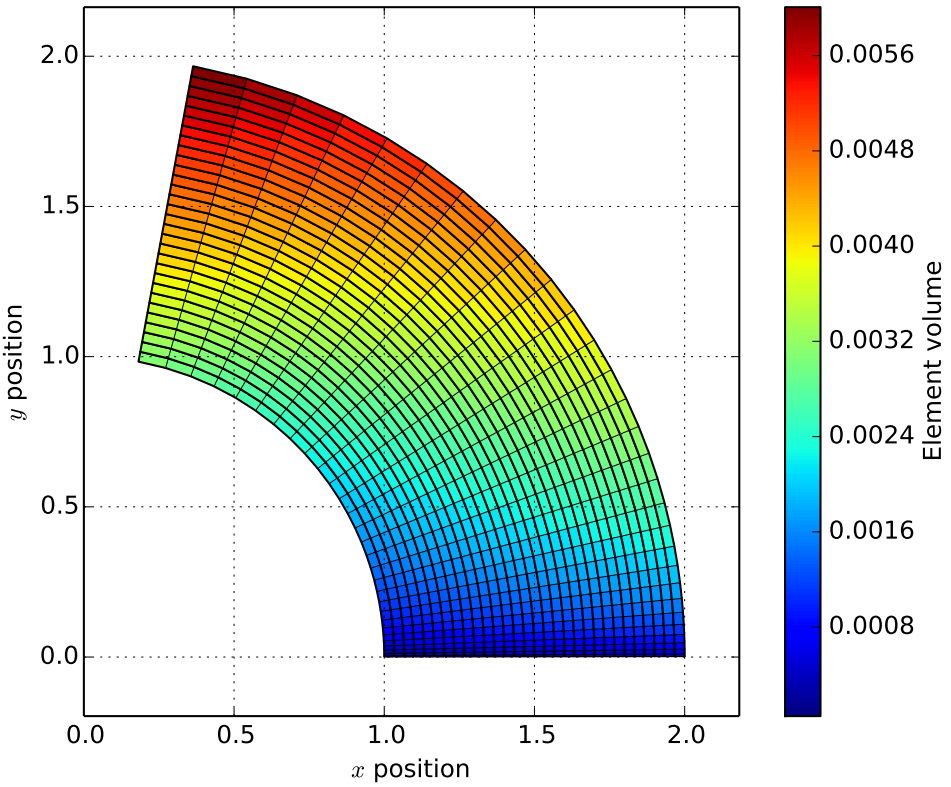


Mesh.**volume**()

Returns a dictionnary containing the volume of all the elements.

```
    from abapy.mesh import RegularQuadMesh
    from abapy.indentation import IndentationMesh
    import matplotlib.pyplot as plt
    from matplotlib.path import Path
    import matplotlib.patches as patches
    import matplotlib.collections as collections
    import numpy as np
    from matplotlib import cm
    from scipy import interpolate


    def function(x, y, z, labels):
      r0 = 1.
      theta = .5 * np.pi * x
      r = y + r0
      ux = -x + r * np.cos(theta**2)
      uy = -y + r * np.sin(theta**2)
      uz = 0. * z
      return ux, uy, uz
    N1, N2 = 30, 30
    l1, l2 = .75, 1.



    m = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
    vectorField = m.nodes.eval_vectorFunction(function)
    m.nodes.apply_displacement(vectorField)
    patches = m.dump2polygons()
    volume = m.volume()
    bb = m.nodes.boundingBox()
    patches.set_facecolor(None) # Required to allow face color
    patches.set_array(volume)
    patches.set_linewidth(1.)
    fig = plt.figure(0)
    plt.clf()
    ax = fig.add_subplot(111)
```

```
    ax.set_aspect("equal")
    ax.add_collection(patches)
    plt.legend()
    cbar = plt.colorbar(patches)
    plt.grid()
    plt.xlim(bb[0])
    plt.ylim(bb[1])
    plt.xlabel("$x$ position")
    plt.ylabel("$y$ position")
    cbar.set_label("Element volume")
    plt.show()
```



2.2.3 Modifications

Mesh.**extrude**(*N=1*, *l=1.0*, *quad=False*, *mapping={}*)

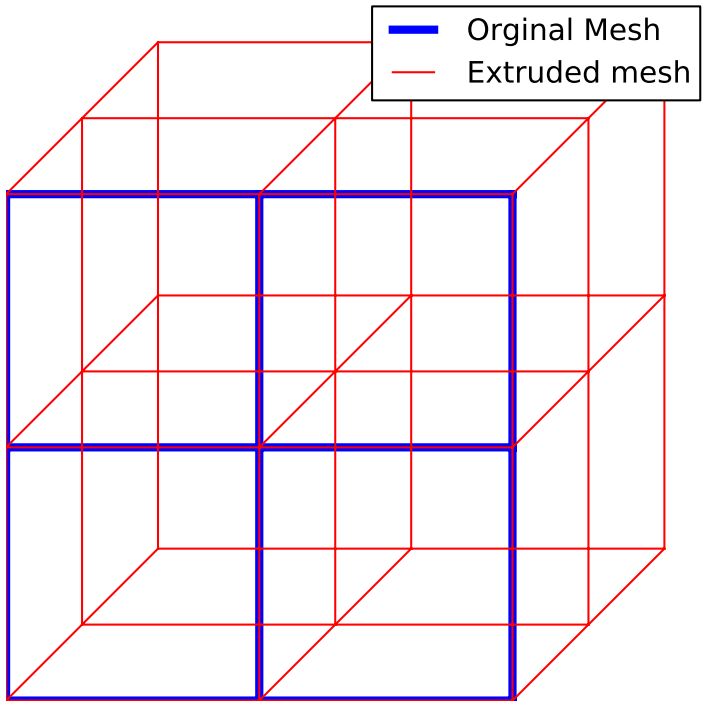
Extrudes a mesh in z direction. The method is made to be applied to 2D mesh, it may work on shell elements but may lead to inside out elements.

Parameters

- **N** (*int*) – number of ELEMENTS along z, must > 0.
- **l** (*float*) – length of the extrusion along z, should be > 0 to avoid inside out elements.
- **quad** (*boolean*) – specifies if quadratic elements should be used instead of linear elements (default). Doesn't work yet. Linear and quadratic elements should not be mixed in the same mesh.

- **mapping** (*boolean*) – gives the way to translate element names during extrusion. Example: {'CAX4':'C3D8','CAX3':'C3D6'}. If 2D element name is not in the mapping, names will be chosen in the basic continuum elements used by Abaqus: 'C3D6' and 'C3D8'.

```
from abapy.mesh import RegularQuadMesh, Mesh
from matplotlib import pyplot as plt

m = RegularQuadMesh(N1 = 2, N2 =2)
m.add_set('el_set',[1,2])
m.add_surface('my_surface',[('el_set',2), ])
m2 = m.extrude(l = 1., N = 2)
x,y,z = m.get_edges()
x2,y2,z2 = m2.get_edges()

# Adding some 3D "home made" perspective:
zx, zy = .3, .3
for i in xrange(len(x2)):
  if x2[i] != None:
    x2[i] += zx * z2[i]
    y2[i] += zy * z2[i]

# Plotting stuff
plt.figure()
plt.clf()
plt.gca().set_aspect('equal')
plt.axis('off')
plt.plot(x,y, 'b-', linewidth  = 4., label = 'Orginal Mesh')
plt.plot(x2,y2, 'r-', linewidth  = 1., label = 'Extruded mesh')
plt.legend()
plt.show()
```



Mesh.**sweep**(*N=1*, *sweep_angle=45.0*, *quad=False*, *mapping={}*, *extrude=False*)

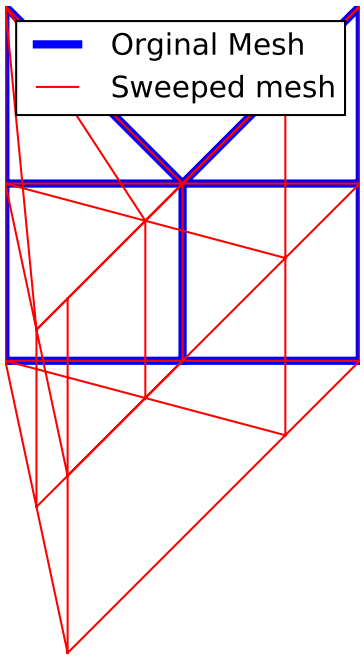
Sweeps a mesh in around z axis. The method is made to be applied to 2D mesh, it may work on shell elements but may lead to inside out elements.

Parameters

- **N** (*int*) – number of ELEMENTS along z, must > 0.
- **sweep_angle** (*float*) – sweep angle around the z axis in degrees. Should be > 0 to avoid inside out elements.
- **quad** (*boolean*) – specifies if quadratic elements should be used instead of linear elements (default). Doesn't work yet. Linear and quadratic elements should not be mixed in the same mesh.
- **mapping** (*dictionary*) – gives the way to translate element names during extrusion. Example: {'CAX4':'C3D8','CAX3':'C3D6'}. If 2D element name is not in the mapping, names will be chosen in the basic continuum elements used by Abaqus: 'C3D6' and 'C3D8'.
- **extrude** (*boolean*) – if True, this param will modify the transformation used to produce the sweep. The result will be a mixed form of sweep and extrusion useful to produce pyramids. When using this option, the sweep angle must be lower than 90 degrees.

```
    from abapy.mesh import RegularQuadMesh, Mesh
    from matplotlib import pyplot as plt
    from array import array
    from abapy.indentation import IndentationMesh

    m = RegularQuadMesh(N1 = 2, N2 =2)
```

```
    m.connectivity[2] = array(m.dti,[5, 7, 4])
    m.connectivity[3] = array(m.dti,[5, 6, 9])
    m.add_set('el_set',[1,2])
    m.add_set('el_set2',[2,4])
    m.add_surface('my_surface',[('el_set',1),])
    m2 = m.sweep(sweep_angle = 70., N = 2, extrude=True)
    x,y,z = m.get_edges()
    x2,y2,z2 = m2.get_edges()

    # Adding some 3D "home made" perspective:
    zx, zy = .3, .3
    for i in xrange(len(x2)):
      if x2[i] != None:
        x2[i] += zx * z2[i]
        y2[i] += zy * z2[i]

    # Plotting stuff
    plt.figure()
    plt.clf()
    plt.gca().set_aspect('equal')
    plt.axis('off')
    plt.plot(x,y, 'b-', linewidth  = 4., label = 'Orginal Mesh')
    plt.plot(x2,y2, 'r-', linewidth  = 1., label = 'Sweeped mesh')
    plt.legend()
    plt.show()
```



Mesh.**union**(*other_mesh*, *crit_distance=None*, *simplify=True*)

Computes the union of 2 Mesh instances. The second operand's labels are increased to be compatible with the

first. All sets are kepts and merged if they share the same name. Nodes which are too close (< crit_distance) are merged. If crit_distance is None, the defautl value value of `simplify_mesh` is used.

Parameters

- **other_mesh** (`Mesh` instance) – mesh to be added to current mesh.
- **crit_distance** (*float > 0*) – critical distance under which nodes are considered identical.

`Mesh`.**apply_reflection**(*point=(0.0, 0.0, 0.0)*, *normal=(1.0, 0.0, 0.0)*)

Applies a reflection symmetry to the mesh instance. The reflection plane is defined by a point and a normal direction.

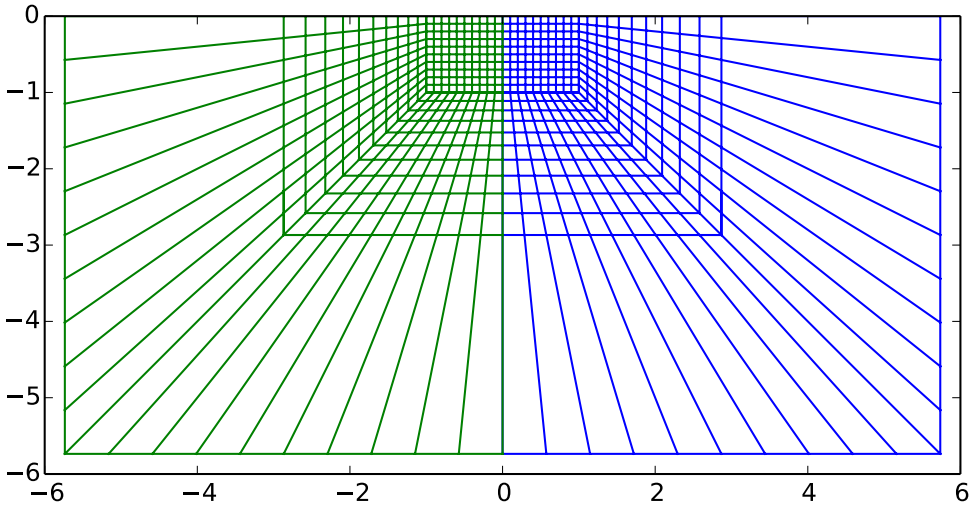
Parameters

- **point** (*tuple or list containing 3 floats*) – coordinates of a point of the reflection plane.
- **normal** (*tuple or list containing 3 floats*) – normal vector to the reflection plane

Return type `Mesh` instance

..note: This method can lead to coherence problems with surfaces, this problem will be addressed in the future. Surfaces are removed by this operation untill this problem is solved.

```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import ParamInfiniteMesh
import matplotlib.pyplot as plt

point  = (0., 0., 0.)
normal = (1., 0., 0.)
m0 = ParamInfiniteMesh()
x0, y0, z0 = m0.get_edges()
m1 = m0.apply_reflection(normal = normal, point = point)
x1, y1, z1 = m1.get_edges()
plt.plot(x0, y0)
plt.plot(x1, y1)
plt.gca().set_aspect('equal')
plt.show()
```



2.2.4 Export

`Mesh.`**`dump2inp`**`()`

Dumps the whole mesh (*i. e.* elements + nodes) to Abaqus INP format.

Return type string

```
>>> from abapy.mesh import Mesh, Nodes
>>> mesh = Mesh()
>>> nodes = mesh.nodes
>>> # Adding some nodes
>>> nodes.add_node(label = 1, x = 0. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 2, x = 1. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 3, x = 1. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 4, x = 0. ,y = 1. , z = 0.)
>>> nodes.add_node(label = 5, x = 2. ,y = 0. , z = 0.)
>>> nodes.add_node(label = 6, x = 2. ,y = 1. , z = 0.)
>>> # Adding some elements
>>> mesh.add_element(label=1, connectivity = (1,2,3,4), space =2, name = 'QUAD4')
>>> mesh.add_element(label=2, connectivity = (2,5,6,3), space =2, name = 'QUAD4')
>>> # Adding sets
>>> mesh.add_set(label = 'veryNiceSet', elements = [1,2])
>>> # Adding surfaces
>>> mesh.add_surface(label = 'superNiceSurface', description = [ ('veryNiceSet', 2) ])
>>> out = mesh.dump2inp()
```

`Mesh.`**`dump2vtk`**`(`*`path=None`*`)`

Dumps the mesh to the VTK format. VTK format can be visualized using Mayavi2 or Paraview. This method is particularly useful for 3D mesh. For 2D mesh, it may be more efficient to work with matplotlib using methods like: get_edges, get_border and dump2triplot.

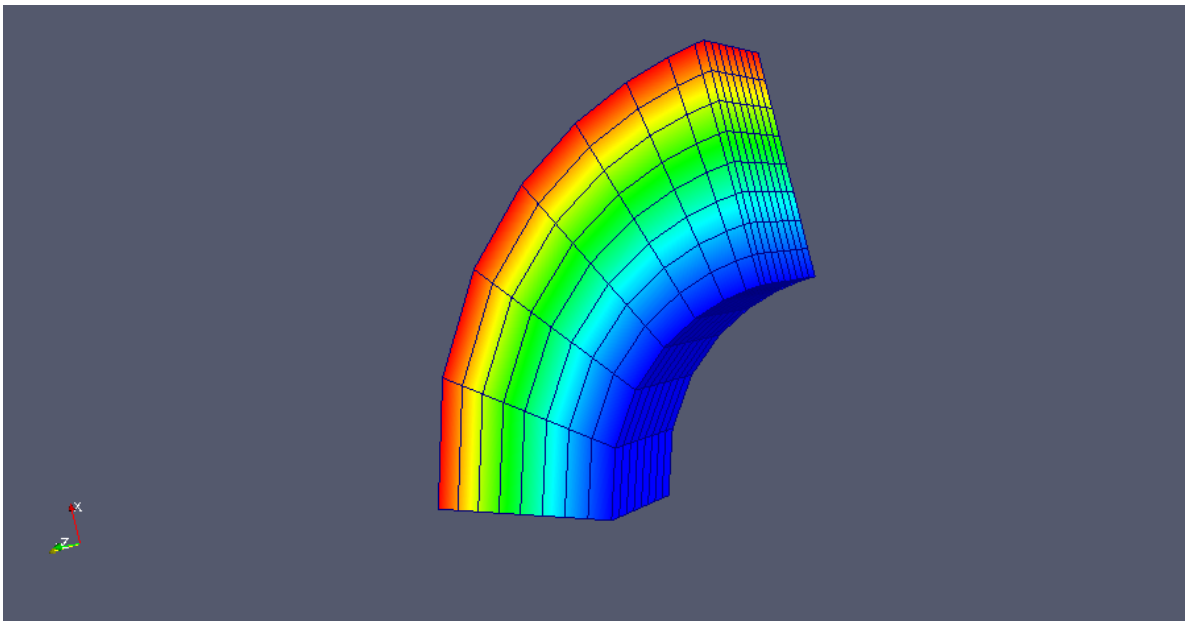
Parameters **path** – if None, return a string containing the VTK data. If not, must be a path to a file where the data will be written.

Return type string or None.

```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import IndentationMesh
import numpy as np

def tensor_function(x, y, z, labels):
  """
  Vector function used to produced the displacement field.
  """
  r0 = 1.
  theta = .5 * np.pi * x
  r = y + r0
  s11 = z + x
  s22 = z + y
  s33 = x**2
  s12 = y**2
  s13 = x + y
  s23 = z
  return s11, s22, s33, s12, s13, s23

def vector_function(x, y, z, labels):
  """
  Vector function used to produced the displacement field.
  """
  r0 = 1.
  theta = .5 * np.pi * x
  r = y + r0
  ux = -x + r * np.cos(theta**2)
  uy = -y + r * np.sin(theta**2)
  uz = 0. * z
  return ux, uy, uz

def scalar_function(x, y, z, labels):
  """
  Scalar function used to produced the plotted field.
  """
  return x**2 + y**2
#MESH GENERATION
N1, N2, N3 = 8, 8, 8
l1, l2, l3 = .75, 1., .5
m = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2).extrude(N = N3, l = l3)
#FIELDS GENERATION
s = m.nodes.eval_tensorFunction(tensor_function)
m.add_field(s, "s")
u = m.nodes.eval_vectorFunction(vector_function)
m.add_field(u, "u")
m.nodes.apply_displacement(u)
f = m.nodes.eval_function(scalar_function)
m.add_field(f, "f")
m.dump2vtk("Mesh-dump2vtk.vtk")
```

•VTK output: `Mesh-dump2vtk.vtk`

•Paraview plot:



2.2.5 Ploting tools

`Mesh.`**`convert2tri3`**(*mapping=None*)

Converts 2D elements to 3 noded triangles only.

Parameters **`mapping`** (*`dict with string keys and values`*) – gives the mapping of element name changes to be applied when elements are splitted. Example: mapping = {'CAX4':'CAX3'}

Return type Mesh instance containing only triangular elements.

Note: This function was mainly developped to allow easy ploting in matplotlib using `matplotlib.plyplot.triplot`, `matplotlib.plyplot.tricontour` and `matplotlib.plyplot.contourf` which rely on full triangle meshes. On a practical point of view, it easily used wrapped inside the `abapy.Mesh.dump2triplot` methods which rewrites connectivity in an easier to plot way.

```
from abapy.mesh import Mesh, Nodes, RegularQuadMesh
import matplotlib.pyplot as plt
from numpy import cos, pi
def function(x, y, z, labels):
 r = (x**2 + y**2)**.5
 return cos(2*pi*x)*cos(2*pi*y)/(r+1.)
N1, N2 = 30, 30
l1, l2 = 1., 1.
Ncolor = 10
mesh = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
field = mesh.nodes.eval_function(function)
fig = plt.figure(figsize=(16,4))
ax = fig.add_subplot(131)
```

```
    ax2 = fig.add_subplot(132)
    ax3 = fig.add_subplot(133)
    ax.set_aspect('equal')
    ax2.set_aspect('equal')
    ax3.set_aspect('equal')
    ax.set_xticks([])
    ax.set_yticks([])
    ax2.set_xticks([])
    ax2.set_yticks([])
    ax3.set_xticks([])
    ax3.set_yticks([])
    ax.set_frame_on(False)
    ax2.set_frame_on(False)
    ax3.set_frame_on(False)
    ax.set_title('Orginal Mesh')
    ax2.set_title('Triangularized Mesh')
    ax3.set_title('Field')
    x,y,z = mesh.get_edges() # Mesh edges
    xt,yt,zt = mesh.convert2tri3().get_edges() # Triangular mesh edges
    xb,yb,zb = mesh.get_border()
    X,Y,Z,tri = mesh.dump2triplot()
    ax.plot(x,y,'k-')
    ax2.plot(xt,yt,'k-')
    ax3.plot(xb,yb,'k-', linewidth = 2.)
    ax3.tricontourf(X,Y,tri,field.data, Ncolor)
    ax3.tricontour(X,Y,tri,field.data, Ncolor, colors = 'black')
    ax.set_xlim([-.1*l1,1.1*l1])
    ax.set_ylim([-.1*l2,1.1*l2])
    ax2.set_xlim([-.1*l1,1.1*l1])
    ax2.set_ylim([-.1*l2,1.1*l2])
    ax3.set_xlim([-.1*l1,1.1*l1])
    ax3.set_ylim([-.1*l2,1.1*l2])
    plt.show()
```

`Mesh.`**`dump2triplot`**(*use_3D=False*)

Allows any 2D mesh to be triangulized and formated in a suitable way to be used by triplot, tricontour and tricontourf in matplotlib.pyplot. This is the best way to produce clean 2D plots of 2D meshs. Returns 4 arrays/lists: x, y and z coordinates of nodes and triangles connectivity. It can be directly used in matplotlib.pyplot using:

Return type 4 lists

```
    >>> import matplotlib.pyplot as plt
    >>> from abapy.mesh import RegularQuadMesh
    >>> plt.figure()
    >>> plt.axis('off')
    >>> plt.gca().set_aspect('equal')
    >>> mesh = RegularQuadMesh(N1 = 10 , N2 = 10)
    >>> x,y,z,tri = mesh.dump2triplot()
    >>> plt.triplot(x,y,tri)
    >>> plt.show()
```

`Mesh.`**`get_edges`**(*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *zmin=None*, *zmax=None*)

Returns the list of edges composing the meshed domain. Edges are given as x and y lists with None separator for faster ploting in `matplotlib.pyplot`.

Return type 3 lists of coordinates directly plotable in matplotlib

`Mesh.`**`get_border`**(*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *zmin=None*, *zmax=None*)

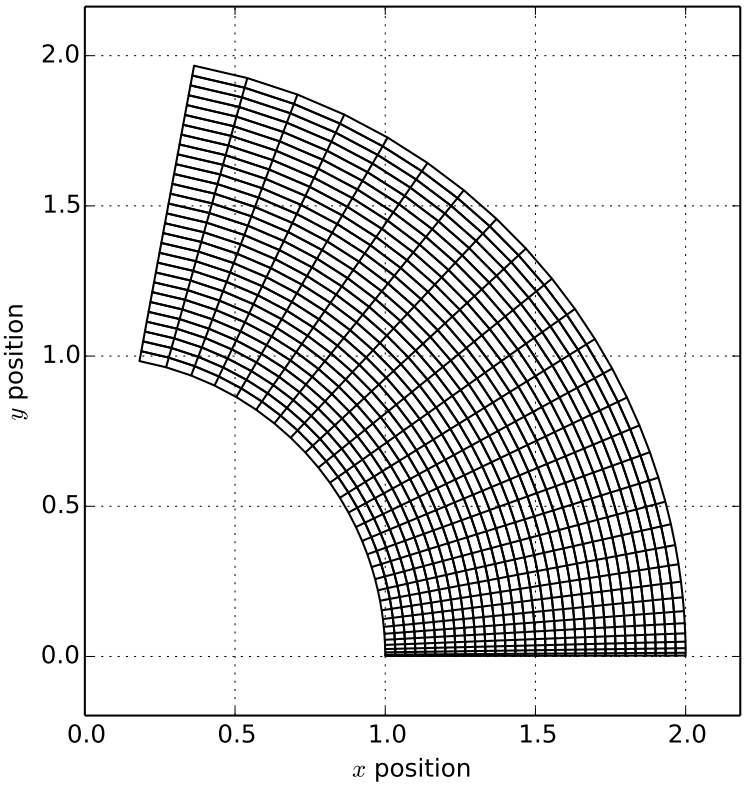
`Mesh.`**`dump2polygons`**(*edge_color='black'*, *edge_width=1.0*, *face_color=None*, *use_3D=False*)

Returns 2D elements as matplotlib poly collection.

Parameters

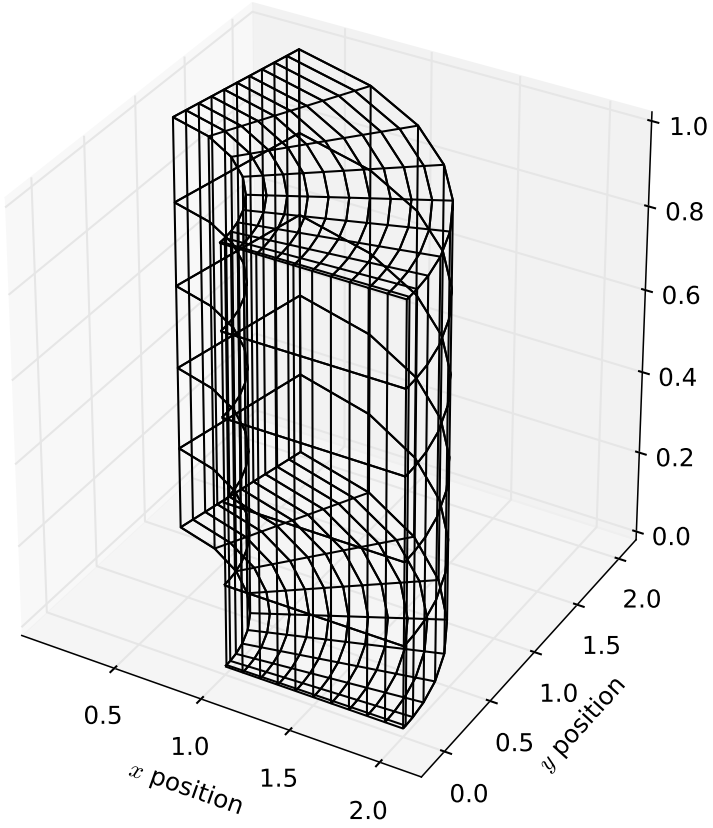
- **edge_color** – edge color.
- **edge_width** – edge width.
- **face_color** – face color.
- **use_3D** – True for 3D polygon export.

```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import IndentationMesh
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches
import matplotlib.collections as collections
import numpy as np
from matplotlib import cm
from scipy import interpolate


def function(x, y, z, labels):
  r0 = 1.
  theta = .5 * np.pi * x
  r = y + r0
  ux = -x + r * np.cos(theta**2)
  uy = -y + r * np.sin(theta**2)
  uz = 0. * z
  return ux, uy, uz
N1, N2 = 30, 30
l1, l2 = .75, 1.



m = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
vectorField = m.nodes.eval_vectorFunction(function)
m.nodes.apply_displacement(vectorField)
patches = m.dump2polygons()
bb = m.nodes.boundingBox()
patches.set_linewidth(1.)
fig = plt.figure(0)
plt.clf()
ax = fig.add_subplot(111)
ax.set_aspect("equal")
ax.add_collection(patches)
plt.grid()
plt.xlim(bb[0])
plt.ylim(bb[1])
plt.xlabel("$x$ position")
plt.ylabel("$y$ position")
plt.show()
```




```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import IndentationMesh
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches
import matplotlib.collections as collections
import mpl_toolkits.mplot3d as a3
import numpy as np
from matplotlib import cm
from scipy import interpolate


def function(x, y, z, labels):
  r0 = 1.
  theta = .5 * np.pi * x
  r = y + r0
  ux = -x + r * np.cos(theta**2)
  uy = -y + r * np.sin(theta**2)
  uz = 0. * z
  return ux, uy, uz
N1, N2, N3 = 10, 10, 5
l1, l2, l3 = .75, 1., 1.



m = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
m = m.extrude(l = l3, N = N3 )
```

```
    vectorField = m.nodes.eval_vectorFunction(function)
    m.nodes.apply_displacement(vectorField)
    patches = m.dump2polygons(use_3D = True,
                              face_color = None,
                              edge_color = "black")
    bb = m.nodes.boundingBox()
    patches.set_linewidth(1.)

    fig = plt.figure(0)
    plt.clf()
    ax = a3.Axes3D(fig)
    ax.set_aspect("equal")
    ax.add_collection3d(patches)
    plt.xlim(bb[0])
    plt.ylim(bb[1])
    plt.xlabel("$x$ position")
    plt.ylabel("$y$ position")
    plt.show()
```



`Mesh.`**`draw`**(*ax*, *field_func=None*, *disp_func=None*, *cmap=None*, *cmap_levels=20*, *cbar_label='Field'*, *cbar_orientation='horizontal'*, *edge_color='black'*, *edge_width=1.0*, *node_style='k.'*, *node_size=1.0*, *contour=False*, *contour_colors='black'*, *alpha=1.0*)

Draws a 2D mesh in a given matplotlib axes instance.

Parameters

- **`ax`** – matplotlib axes instance.
- **`field_func`** (*function or None*) – a function that defines how to used existing

fields to produce a FieldOutput instance.

- **disp_func** (*function*) – a function that defines how to used existing fields to produce a VectorFieldOutput instance used as a diplacement field.
- **cmap** – matplotlib colormap.
- **cmap_levels** – number of levels in the colormap
- **cbar_label** (*string*) – colorbar label.
- **cbar_orientation** – "horizontal" or "vertical".
- **edge_color** – valid matplotlib color for the edges of the mesh.
- **edge_width** – mesh edge width.
- **node_style** – nodes plot style.
- **node_size** – nodes size.
- **contour** (*boolean*) – plot field contour.
- **contour_colors** – contour colors to use, colormap of fixed color.
- **alpha** – alpha lvl of the gradiant plot.

```
from abapy.mesh import RegularQuadMesh
from abapy.indentation import IndentationMesh
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches
import matplotlib.collections as collections
import numpy as np
from matplotlib import cm
from scipy import interpolate

def vector_function(x, y, z, labels):
  """
  Vector function used to produced the displacement field.
  """
  r0 = 1.
  theta = .5 * np.pi * x
  r = y + r0
  ux = -x + r * np.cos(theta**2)
  uy = -y + r * np.sin(theta**2)
  uz = 0. * z
  return ux, uy, uz

def scalar_function(x, y, z, labels):
  """
  Scalar function used to produced the plotted field.
  """
  return x**2 + y**2
#MESH GENERATION
N1, N2 = 30, 30
l1, l2 = .75, 1.
m = RegularQuadMesh(N1 = N1, N2 = N2, l1 = l1, l2 = l2)
#FIELDS GENERATION
u = m.nodes.eval_vectorFunction(vector_function)
m.add_field(u, "u")
f = m.nodes.eval_function(scalar_function)
m.add_field(f, "f")
```

```
    #PLOTS
    fig = plt.figure(0)
    plt.clf()
    ax = fig.add_subplot(1,1,1)
    m.draw(ax,
        disp_func  = lambda fields : fields["u"],
        field_func = lambda fields : fields["f"],
        cmap = cm.jet,
        cbar_orientation = "vertical",
        contour = False,
        contour_colors = "black",
        alpha = 1.,
        cmap_levels = 10,
        edge_width = .1)
    ax.set_aspect("equal")
    plt.grid()
    plt.xlabel("$x$ position")
    plt.ylabel("$y$ position")
    plt.show()
```

2.3 Mesh generation

2.3.1 `RegularQuadMesh` functions

abapy.mesh.**RegularQuadMesh**(*N1=1*, *N2=1*, *l1=1.0*, *l2=1.0*, *name='QUAD4'*, *dtf='f'*, *dti='I'*)

Generates a 2D regular quadrangle mesh.

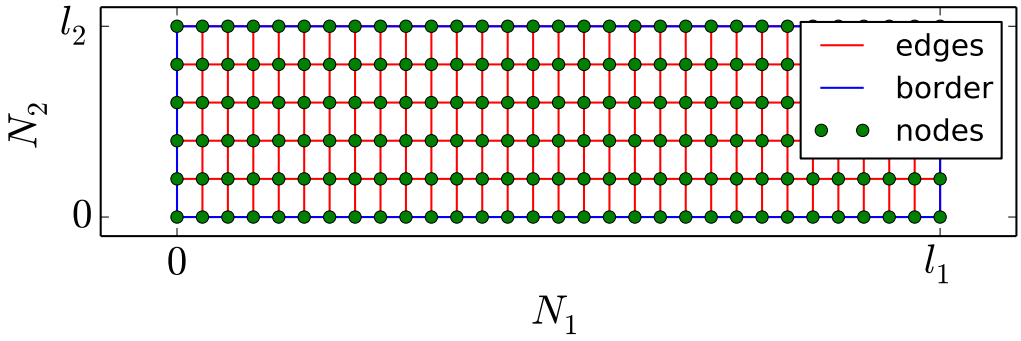
Parameters

- **N1** (*int > 0*) – number of elements respectively along y.
- **N2** (*int > 0*) – number of elements respectively along y.
- **l1** (*float*) – length of the mesh respectively along x.
- **l2** (*float*) – length of the mesh respectively along y.
- **name** (*string*) – elements names, for example 'CPS4'.
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array

Return type Mesh instance

```
    from abapy.mesh import RegularQuadMesh
    from matplotlib import pyplot as plt
    N1,N2 = 30,5 # Number of elements
    l1, l2 = 4., 1. # Mesh size
    fs = 20. # fontsize
    mesh = RegularQuadMesh(N1,N2,l1,l2)
    plt.figure(figsize=(8,3))
    plt.gca().set_aspect('equal')
    nodes = mesh.nodes
    xn, yn, zn = nodes.x, nodes.y, nodes.z # Nodes coordinates
    xe,ye,ze = mesh.get_edges() # Mesh edges
    xb,yb,zb = mesh.get_border() # Mesh border
    plt.plot(xe,ye,'r-',label = 'edges')
    plt.plot(xb,yb,'b-',label = 'border')
```

```
    plt.plot(xn,yn,'go',label = 'nodes')
    plt.xlim([-.1*l1,1.1*l1])
    plt.ylim([-.1*l2,1.1*l2])
    plt.xticks([0,l1],['$0$', '$l_1$'],fontsize = fs)
    plt.yticks([0,l2],['$0$', '$l_2$'],fontsize = fs)
    plt.xlabel('$N_1$',fontsize = fs)
    plt.ylabel('$N_2$',fontsize = fs)
    plt.legend()
    plt.show()
```



abapy.mesh.**RegularQuadMesh_like**(*x_list=[0.0, 1.0], y_list=[0.0, 1.0], name='QUAD4', dtf='f', dti='I'*)

Generates a 2D regular quadrangle mesh from 2 lists of positions. This version of RegularQuadMesh is an alternative to the normal one in some cases where fine tuning of x, y positions is required.

Parameters

- **x_list** (*list, array.array or numpy.array*) – list of x values
- **y_list** (*list, array.array or numpy.array*) – list of y values
- **name** (*string*) – elements names, for example 'CPS4'.
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array

Return type Mesh instance

2.3.2 Other meshes

abapy.mesh.**TransitionMesh**(*N1=4, N2=2, l1=1.0, l2=1.0, direction='y+', name='CAX4', crit_distance=1e-06*)

A mesh transition to manage them all...

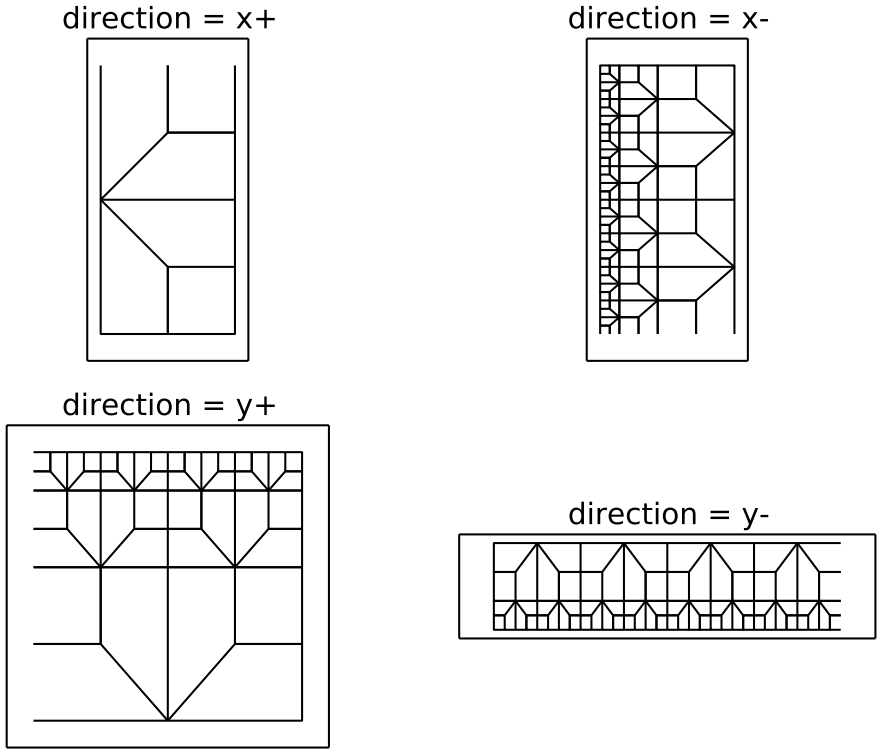
Parameters

- **N1** (*int*) – starting number of elements, must be multiple of 4.
- **N2** (*int*) – ending number of elements, must be lower than N1 and multiple of 2.
- **l1** (*float*) – length of the mesh in the x direction.
- **l2** (*float*) – length of the mesh in the y direction.
- **direction** (*str*) – direction of mesh. Must be in ("x+", "x-", "y+", "y-").

- **name** (*str*) – name of the element in the export procedures.
- **crit_distance** (*float*) – critical distance in union process.

```
from abapy.mesh import TransitionMesh
from matplotlib import pyplot as plt

fig = plt.figure(0)
plt.clf()

ax = fig.add_subplot(221)
ax.set_aspect("equal")
ax.set_title('direction = x+')
m = TransitionMesh(N1 = 4, N2 = 2,l1 = 1., l2 = 2., direction = "x+")
patches = m.dump2polygons()
bb = m.nodes.boundingBox()
patches.set_linewidth(1.)
ax.add_collection(patches)
plt.xlim(bb[0])
plt.ylim(bb[1])
plt.xticks([])
plt.yticks([])

ax = fig.add_subplot(222)
ax.set_aspect("equal")
ax.set_title('direction = x-')
m = TransitionMesh(N1 =32, N2 = 4,l1 = 1., l2 = 2., direction = "x-")
patches = m.dump2polygons()
bb = m.nodes.boundingBox()
patches.set_linewidth(1.)
ax.add_collection(patches)
plt.xlim(bb[0])
plt.ylim(bb[1])
plt.xticks([])
plt.yticks([])

ax = fig.add_subplot(223)
ax.set_aspect("equal")
ax.set_title('direction = y+')
m = TransitionMesh(N1 = 16, N2 = 2,l1 = 1, l2 = 1., direction = "y+")
patches = m.dump2polygons()
bb = m.nodes.boundingBox()
patches.set_linewidth(1.)
ax.add_collection(patches)
plt.xlim(bb[0])
plt.ylim(bb[1])
plt.xticks([])
plt.yticks([])

ax = fig.add_subplot(224)
ax.set_aspect("equal")
ax.set_title('direction = y-')
m = TransitionMesh(N1 =32, N2 = 8,l1 = 4., l2 = 1., direction = "y-")
patches = m.dump2polygons()
bb = m.nodes.boundingBox()
patches.set_linewidth(1.)
ax.add_collection(patches)
plt.xlim(bb[0])
plt.ylim(bb[1])
```

```
    plt.xticks([])
    plt.yticks([])

    plt.show()
```



Note: see also in `abapy.indentation` for indentation dedicated meshes.

CHAPTER 3

Materials

Material definitions.

3.1 Elastic materials

class abapy.materials.**Elastic**(*labels='mat'*, *E=1.0*, *nu=0.3*, *dtf='d'*)
Represents an istotrop linear elastic material used for FEM simulations

Parameters

- **E** (*float, list, array.array*) – Young's modulus.
- **nu** (*float, list, array.array*) – Poisson's ratio.

Note: All inputs must have the same length or an exception will be raised.

dump2inp()
Returns materials in INP format suitable with abaqus input files.

Return type string

3.2 Elastic-plastic materials

class abapy.materials.**VonMises**(*labels='mat'*, *E=1.0*, *nu=0.3*, *sy=0.01*, *dtf='d'*)
Represents von Mises materials used for FEM simulations

Parameters

- **E** (*float, list, array.array*) – Young's modulus.
- **nu** (*float, list, array.array*) – Poisson's ratio.
- **sy** (*float, list, array.array*) – Yield stress.

Note: All inputs must have the same length or an exception will be raised.

```
>>> from abapy.materials import VonMises
>>> m = VonMises(labels='myMaterial',E=1,nu=0.45, sy=0.01)
>>> print m.dump2inp()
...
```

dump2inp()
Returns materials in INP format suitable with abaqus input files.

Return type string

class abapy.materials.**Hollomon**(*labels='mat', E=1.0, nu=0.3, sy=0.01, n=0.2, kind=1, dtf='d'*)
Represents von Hollom materials (i. e. power law haderning and von mises yield criterion) used for FEM simulations.

Parameters

- **E** (*float, list, array.array*) – Young's modulus.
- **nu** (*float, list, array.array*) – Poisson's ratio.
- **sy** (*float, list, array.array*) – Yield stress.
- **n** – hardening exponent
- **kind** (*int*) – kind of equation to be used (see below). Default is 1.

Note: All inputs must have the same length or an exception will be raised.

Several sets of equations are refered to as Hollomon stress-strain law. In all cases, we the strain decomposition $\epsilon = \epsilon_e + \epsilon_p$ is used and the elastic part is described by $\sigma = E\epsilon_e = E\epsilon$. Only the plastic parts (i. e. $\sigma > \sigma_y$) differ:

•kind 1:

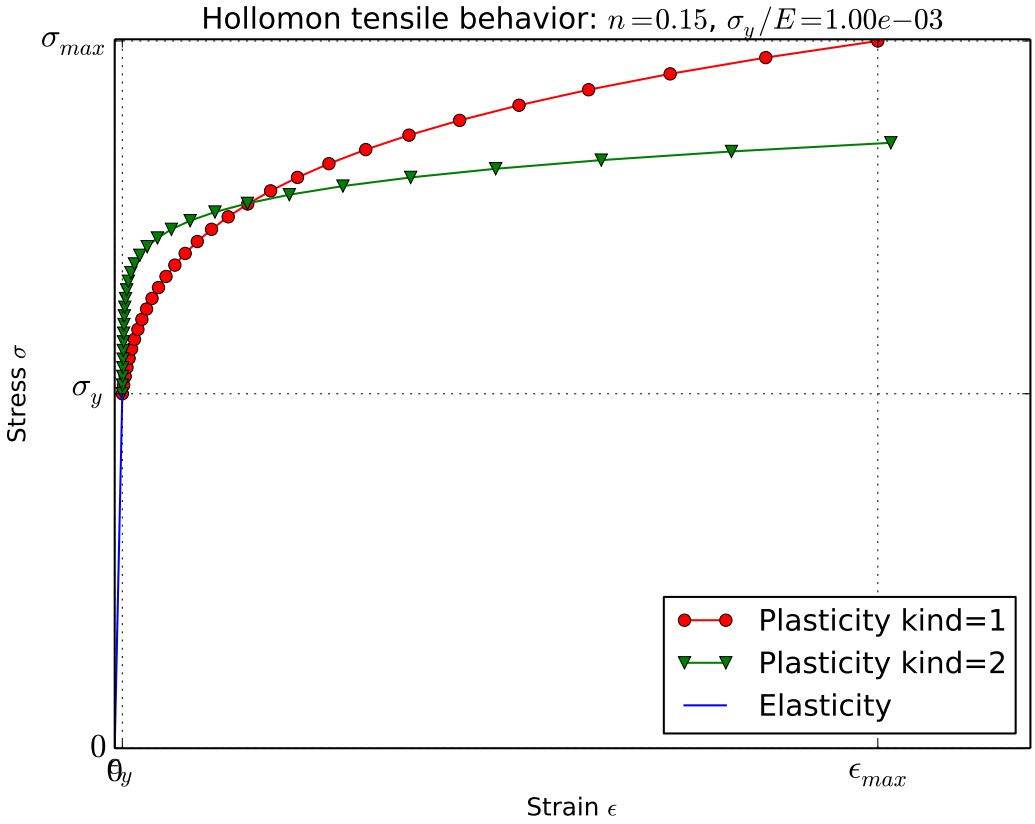
$$\sigma = \sigma_y \left(\epsilon E/\sigma_y\right)^n \tag{3.1}$$

•kind 2:

$$\sigma = \sigma_y \left(1 + \epsilon_p\right)^n = E\epsilon_e \tag{3.2}$$

```
from abapy.materials import Hollomon
import matplotlib.pyplot as plt

E = 1.        # Young's modulus
sy = 0.001      # Yield stress
n = 0.15    # Hardening exponent
nu = 0.3
eps_max = .1 # maximum strain to be computed
N = 30      # Number of points to be computed (30 is a low value useful for graphical reasons, i
mat1 = Hollomon(labels = 'my_material', E=E, nu=nu, sy=sy, n=n)
table1 = mat1.get_table(0, N=N, eps_max=eps_max)
eps1 = table1[:,0]
sigma1 = table1[:,1]
sigma_max1 = max(sigma1)

mat2 = Hollomon(labels = 'my_material', E=E, nu=nu, sy=sy, n=n, kind = 2)
table2 = mat2.get_table(0, N=N, eps_max=eps_max)
eps2 = table2[:,0]
sigma2 = table2[:,1]
```

```
    sigma_max2 = max(sigma2)


    plt.figure()
    plt.clf()
    plt.title('Hollomon tensile behavior: $n = {0:.2f}$, $\sigma_y / E = {1:.2e}$'.format(n, sy/E))
    plt.xlabel('Strain $\epsilon$')
    plt.ylabel('Stress $\sigma$')
    plt.plot(eps1, sigma1, 'or-', label = 'Plasticity kind=1')
    plt.plot(eps2, sigma2, 'vg-', label = 'Plasticity kind=2')
    plt.plot([0., sy / E], [0., sy], 'b-', label = 'Elasticity')
    plt.xticks([0., sy/E, eps_max], ['$0$', '$\epsilon_y$', '$\epsilon_{max}$'], fontsize = 16.)
    plt.yticks([0., sy, sigma_max1], ['$0$', '$\sigma_y$', '$\sigma_{max}$'], fontsize = 16.)
    plt.grid()
    plt.legend(loc = "lower right")
    plt.show()
```



dump2inp (*eps_max=10.0*, *N=100*)
Returns materials in INP format suitable with abaqus input files.

Parameters

- **eps_max** (*float*) – maximum strain to be computed.
- **N** (*int*) – number of points to be computed.

Return type string

get_table (*position=0*, *eps_max=10.0*, *N=100*)
Returns the tabular data corresponding to the tensile stress strain law using log spacing. :param position:

indice of the concerned material (default is 0). :type position: int :param eps_max: maximum strain to be computed. If kind is 1, eps_max is the total strain, if kind is 2, eps_max is the plastic strain. :type eps_max: float :param N: number of points to be computed. :type N: int :rtype: `numpy.array`

class `abapy.materials.`**`DruckerPrager`**(*labels='mat'*, *E=1.0*, *nu=0.3*, *sy=0.01*, *beta=10.0*, *psi=None*, *k=1.0*, *dtf='d'*)

Represents Drucker-Prager materials used for FEM simulations

Parameters

- **E** (*float, list, array.array*) – Young's modulus.
- **nu** (*float, list, array.array*) – Poisson's ratio.
- **sy** (*float, list, array.array*) – Compressive yield stress.
- **beta** (*float, list, array.array*) – Friction angle in degrees.
- **psi** (*float, list, array.array or None*) – Dilatation angle in degress. If psi = beta, the plastic flow is associated. If psi = None, the associated flow is automatically be chosen.
- **k** (*float, list, array.array*) – tension vs. compression asymmetry. For k = 1., not asymmetry, for k=0.778 maximum possible asymmetry.

Note:

All inputs must have the same length or an exception will be raised.

...

`dump2inp`()

Returns materials in INP format suitable with abaqus input files.

Return type string

class `abapy.materials.`**`Bilinear`**(*labels='mat'*, *E=1.0*, *nu=0.3*, *Ssat=1000.0*, *n=100.0*, *sy=100.0*, *dtf='d'*)

Represents von Mises materials used for FEM simulations

Parameters

- **E** (*float, list, array.array*) – Young's modulus.
- **nu** (*float, list, array.array*) – Poisson's ratio.
- **Ssat** (*float, list, array.array*) – Saturation stress.
- **n** (*float, list, array.array*) – Slope of the first linear plastic law
- **Sy** (*float, list, array.array*) – Stress at zero plastic strain

---

Note: All inputs must have the same length or an exception will be raised.

`dump2inp`()

Returns materials in INP format suitable with abaqus input files.

Return type string

CHAPTER 4

Post Processing

Finite Element Modeling post processing tools.

4.1 Field Outputs

4.1.1 Scalar fields

class abapy.postproc.**FieldOutput**(*position='node'*, *data=None*, *labels=None*, *dti='I'*, *dtf='f'*)

Scalar output representing a field evaluated on nodes or elements referenced by their labels. A FieldOutput instance cannot be interpreted with its mesh. On initiation, labels and data will be reordered to have labels sorted.

Parameters

- **position** (*'node' or 'element'*) – location of the field evaluation
- **data** (*list, array.array, numpy.array containing floats*) – value of the field where it is evaluated
- **labels** (*list, array.array, numpy.array containt ints or None.*) – labels of the nodes/elements where the field is evaluated. If None, labels will be [1,2,...,len(data)+1]
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array

```
>>> from abapy.postproc import FieldOutput
>>> data = [-1.,5.,3.]
>>> labels = [1,3,2]
>>> fo = FieldOutput(data=data, labels = labels, position = 'node')
>>> print fo # data is sorted by labels
FieldOutput instance
Position = node
Label Data
1     -1.0
2     3.0
3     5.0
>>> print fo[1:2] # slicing
FieldOutput instance
Position = node
Label Data
1     -1.0
```

```
>>> print fo[2] # indexing
FieldOutput instance
Position = node
Label Data
2     3.0
>>> print fo[1,3] # multiple indexing
FieldOutput instance
Position = node
Label Data
1     -1.0
3     5.0
>>> print fo*2 # multiplication
FieldOutput instance
Position = node
Label Data
1     -2.0
2     6.0
3     10.0
>>> fo2 = fo**2  #power
>>> print fo2
FieldOutput instance
Position = node
Label Data
1     1.0
2     9.0
3     25.0
>>> print fo * fo2
FieldOutput instance
Position = node
Label Data
1     -1.0
2     27.0
3     125.0
>>> print fo + fo2
FieldOutput instance
Position = node
Label Data
1     0.0
2     12.0
3     30.0
>>> print abs(fo)
FieldOutput instance
Position = node
Label Data
1     1.0
2     3.0
3     5.0
```

Note: If dti='H' is chosen, labels are stored as unsigned 16 bits ints. If more than 65k labels are stored, an OverFlow error will be raised.

Add/remove/get data

`FieldOutput.`**`add_data`**(*label*, *data*)

Adds one point to a FieldOutput instance. Label must not already exist in the current FieldOutput, if not so, nothing will be changed. Data and label will be inserted in self.data, self.labels in order to keep self.labels sorted.

```
>>> from abapy.postproc import FieldOutput
>>> data = [5.5, 2.2]
>>> labels = [1,4]
>>> temperature = FieldOutput(labels = labels, data = data, position = 'node')
>>> temperature.add_data(2, 5.)
>>> temperature.data # labels are sorted
array('f', [5.5, 5.0, 2.200000047683716])
>>> temperature.labels # data was sorted like labels
array('I', [1L, 2L, 4L])
```

Parameters

- **label** – labels of the nodes/elements where the field is evaluated.
- **data** (*float*) – value of the field where it is evaluated

FieldOutput.**get_data**(*label*)

Returns data at a location with given label.

Parameters **label** (*int > 0*) – location's label.

Return type float

Note: Requesting data at a label that does not exist in the instance will just lead in a warning but if label is negative or is not int, then an Exception will be raised.

VTK Export

FieldOutput.**dump2vtk**(*name='fieldOutput', header=True*)

Converts the FieldOutput instance to VTK format which can be directly read by Mayavi2 or Paraview. This method is very useful to quickly and efficiently plot 3D mesh and fields.

Parameters

- **name** (*string*) – name used for the field in the output.
- **header** (*boolean*) – if True, adds the location header (eg. CELL_DATA / POINT_DATA)

Return type string

```
from abapy.postproc import FieldOutput
from abapy.mesh import Mesh, Nodes
x = [0.,1.,0.]
y = [0.,0.,1.]
z = [0.,0.,0.]
labels = [1,2,3]
nodes = Nodes(x=x,y=y,z=z, labels=labels)
mesh = Mesh(nodes=nodes)
mesh.add_element(label = 1 , connectivity = [1,2,3], space = 2 , name = 'tri3') # triangle eleme
nodeField = FieldOutput()
nodeField.add_data(data = 0., label = 1)
nodeField.add_data(data = 10., label = 2)
nodeField.add_data(data = 20., label = 3)
```

```
elementField = FieldOutput(position='element')
elementField.add_data(label = 1, data =10.)
out = ''
out+=mesh.dump2vtk()
out+=nodeField.dump2vtk('nodeField')
out+=elementField.dump2vtk('elementField')
f = open("FieldOutput-dump2vtk.vtk", "w")
f.write(out)
f.close()
```

Result in Paraview:

Operations and invariants

4.1.2 Vector fields

class abapy.postproc.**VectorFieldOutput**(*data1=None*, *data2=None*, *data3=None*, *position='node'*, *dti='I'*, *dtf='f'*)

3D vector field output. Using this class instead of 3 scalar FieldOutput instances is efficient because labels are stored only since and allows all vector operations like dot, cross, norm.

Parameters

- **data1** (*FieldOutput instance or None*) – x coordinate
- **data2** (*FieldOutput instance or None*) – y coordinate
- **data3** (*FieldOutput instance or None*) – z coordinate
- **position** (*'node' or 'element'*) – position at which data is computed
- **dti** (*'I' for uint32 or 'H' for uint16*) – array.array int data type
- **dtf** (*'f' float32 or 'd' for float64*) – array.array int data type

```
>>> from abapy.postproc import FieldOutput, VectorFieldOutput
>>> data1 = [1,2,3,5,6,0]
>>> data2 = [1. for i in data1]
>>> labels = range(1,len(data1)+1)
>>> fo1, fo2 = FieldOutput(labels = labels, data=data1, position='node' ), FieldOutput(labels =
>>> vector = VectorFieldOutput(data1 = fo1, data2 = fo2 )
>>> vector2 = VectorFieldOutput(data2 = fo2 )
>>> vector # short description
<VectorFieldOutput instance: 6 locations>
>>> print vector # long description
VectorFieldOutput instance
Position = node
Label Data1   Data2   Data3
1     1.0     1.0     0.0
2     2.0     1.0     0.0
3     3.0     1.0     0.0
4     5.0     1.0     0.0
5     6.0     1.0     0.0
6     0.0     1.0     0.0
>>> print vector[6] # Returns a VectorFieldOutput instance
VectorFieldOutput instance
Position = node
Label Data1   Data2   Data3
6     0.0     1.0     1.0
```

```
>>> print vector[1,4,6] # Picking label by label
VectorFieldOutput instance
Position = node
Label Data1   Data2   Data3
1     1.0     1.0     1.0
4     5.0     1.0     1.0
6     0.0     1.0     1.0
>>> print vector[1:6:2] # Slicing
VectorFieldOutput instance
Position = node
Label Data1   Data2   Data3
1     1.0     1.0     1.0
3     3.0     1.0     1.0
5     6.0     1.0     1.0
>>> vector.get_data(6) # Returns 3 floats
(0.0, 1.0, 0.0)
>>> vector.norm() # Returns norm
<FieldOutput instance: 6 locations>
>>> vector.sum() # returns the sum of coords
<FieldOutput instance: 6 locations>
>>> vector * vector2 # Itemwise product (like numpy, unlike matlab)
<VectorFieldOutput instance: 6 locations>
>>> vector.dot(vector2) # Dot/Scalar product
<FieldOutput instance: 6 locations>
>>> vector.cross(vector2) # Cross/Vector product
<VectorFieldOutput instance: 6 locations>
>>> vector + 2 # Itemwise addition
<VectorFieldOutput instance: 6 locations>
>>> vector * 2 # Itemwise multiplication
<VectorFieldOutput instance: 6 locations>
>>> vector / 2 # Itemwise division
<VectorFieldOutput instance: 6 locations>
>>> vector / vector2 # Itemwise division between vectors (numpy way)
Warning: divide by zero encountered in divide
Warning: invalid value encountered in divide
<VectorFieldOutput instance: 6 locations>
>>> abs(vector) # Absolute value
<VectorFieldOutput instance: 6 locations>
>>> vector ** 2 # Power
<VectorFieldOutput instance: 6 locations>
>>> vector ** vector # Itemwize power
<VectorFieldOutput instance: 6 locations>
```

Note:

- data1, data2 and data3 must have same position and label or be None. If one data is None, it is supposed to be zero.
- Storage data dype is the highest standard of all 3 data.
- Numpy is not used in the constructor to allow the creation of instances in Abaqus Python but most other operation require numpy for speed reasons.

Add/remove/get data

VectorFieldOutput.**add_data**(*label*, *data1=0.0*, *data2=0.0*, *data3=0.0*)

Adds one point to a VectorFieldOutput instance. Label must not already exist in the current FieldOutput, if not so, nothing will be changed. Data and label will be inserted in self.data, self.labels in order to keep self.labels sorted.

Parameters

- **label** – labels of the nodes/elements where the field is evaluated.
- **data1** (*float*) – value of the coordinate 1 of the field where it is evaluated.
- **data2** (*float*) – value of the coordinate 2 of the field where it is evaluated.
- **data3** (*float*) – value of the coordinate 3 of the field where it is evaluated.

VectorFieldOutput.**get_data**(*label*)

Returns coordinates at a location with given label.

Parameters **label** (*int > 0*) – location's label.

Return type float, float, float

Note: Requesting data at a label that does not exist in the instance will just lead in a warning but if label is negative or is not int, then an Exception will be raised.

VectorFieldOutput.**get_coord**(*number*)

Returns a coordinate of the vector as a FieldOutput.

Parameters **number** (*1,2 or 3*) – requested coordinate number, 1 is x and so on.

Return type FieldOutput instance

```
>>> v1 = Vec.get_coord(1)
```

VTK Export

VectorFieldOutput.**dump2vtk**(*name='vectorFieldOutput'*, *header=True*)

Converts the VectorFieldOutput instance to VTK format which can be directly read by Mayavi2 or Paraview. This method is very useful to quickly and efficiently plot 3D mesh and fields.

Parameters **name** (*string*) – name used for the field in the output.

Return type string

```
>>> from abapy.postproc import FieldOutput, VectorFieldOutput
>>> from abapy.mesh import RegularQuadMesh
>>> mesh = RegularQuadMesh()
>>> data1 = [2,2,5,10]
>>> data2 = [1. for i in data1]
>>> labels = range(1,len(data1)+1)
>>> fo1, fo2 = FieldOutput(labels = labels, data=data1, position='node' ), FieldOutput(labels =
>>> vector = VectorFieldOutput(data1 = fo1, data2 = fo2 )
>>> out = mesh.dump2vtk() + vector.dump2vtk()
>>> f = open('vector.vtk','w')
>>> f.write(out)
>>> f.close()
```

Operations

`VectorFieldOutput.`**`norm`**`()`
Computes norm of the vector at each location and returns it as a scalar FieldOutput.

```
>>> norm = Vec.norm()
```

Return type FieldOutput instance

`VectorFieldOutput.`**`sum`**`()`
Returns the sum of all coordinates.

Return type FieldOutput instance

`VectorFieldOutput.`**`dot`**`(`*other*`)`
Returns the dot (*i. e.* scalar) product of two vector field outputs.

Parameters **`other`** (`VectorFieldOutput`) – Another vector field

Return type `FieldOutput`

`VectorFieldOutput.`**`cross`**`(`*other*`)`
Returns the cross product of two vector field outputs.

Parameters **`other`** (`VectorFieldOutput`) – Another vector field

Return type `VectorFieldOutput`

4.1.3 Tensor fields

class `abapy.postproc.`**`TensorFieldOutput`**`(`*data11=None*, *data22=None*, *data33=None*, *data12=None*, *data13=None*, *data23=None*, *position='node'*, *dti='I'*, *dtf='f'*`)`
Symmetric tensor field output. Using this class instead of 6 scalar FieldOutput instances is efficient because labels are stored only since and allows all operations like invariants, product, sum...

Parameters

- **`data11`** (*`FieldOutput instance or None`*) – 11 component
- **`data22`** (*`FieldOutput instance or None`*) – 22 component
- **`data33`** (*`FieldOutput instance or None`*) – 33 component
- **`data12`** (*`FieldOutput instance or None`*) – 12 component
- **`data13`** (*`FieldOutput instance or None`*) – 13 component
- **`data23`** (*`FieldOutput instance or None`*) – 23 component
- **`position`** (*`'node' or 'element'`*) – position at which data is computed
- **`dti`** (*`'I' for uint32 or 'H' for uint16`*) – array.array int data type
- **`dtf`** (*`'f' float32 or 'd' for float64`*) – array.array int data type

```
>>> from abapy.postproc import FieldOutput, TensorFieldOutput, VectorFieldOutput
>>> data11 = [1., 1., 1.]
>>> data22 = [2., 4., -1]
>>> data12 = [1., 2., 0.]
>>> labels = range(1,len(data11)+1)
>>> fo11 = FieldOutput(labels = labels, data=data11,position='node')
```

```
    >>> fo22 = FieldOutput(labels = labels, data=data22,position='node')
    >>> fo12 = FieldOutput(labels = labels, data=data12,position='node')
    >>> tensor = TensorFieldOutput(data11 = fo11, data22 = fo22, data12 = fo12 )
    >>> tensor2 = TensorFieldOutput(data11= fo22 )
    >>> tensor
    <TensorFieldOutput instance: 3 locations>
    >>> print tensor
    TensorFieldOutput instance
    Position = node
    Label Data11  Data22  Data33  Data12  Data13  Data23
    1     1.0e+00 2.0e+00 0.0e+00 1.0e+00 0.0e+00 0.0e+00
    2     1.0e+00 4.0e+00 0.0e+00 2.0e+00 0.0e+00 0.0e+00
    3     1.0e+00 -1.0e+00        0.0e+00 0.0e+00 0.0e+00 0.0e+00
    >>> print tensor[1,2]
    TensorFieldOutput instance
    Position = node
    Label Data11  Data22  Data33  Data12  Data13  Data23
    1     1.0e+00 2.0e+00 0.0e+00 1.0e+00 0.0e+00 0.0e+00
    2     1.0e+00 4.0e+00 0.0e+00 2.0e+00 0.0e+00 0.0e+00
    >>> print tensor *2. + 1.
    TensorFieldOutput instance
    Position = node
    Label Data11  Data22  Data33  Data12  Data13  Data23
    1     3.0e+00 5.0e+00 1.0e+00 3.0e+00 1.0e+00 1.0e+00
    2     3.0e+00 9.0e+00 1.0e+00 5.0e+00 1.0e+00 1.0e+00
    3     3.0e+00 -1.0e+00        1.0e+00 1.0e+00 1.0e+00 1.0e+00
    >>> print tensor ** 2 # Piecewise power
    TensorFieldOutput instance
    Position = node
    Label Data11  Data22  Data33  Data12  Data13  Data23
    1     1.0e+00 4.0e+00 0.0e+00 1.0e+00 0.0e+00 0.0e+00
    2     1.0e+00 1.6e+01 0.0e+00 4.0e+00 0.0e+00 0.0e+00
    3     1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00 0.0e+00
    >>> vector = VectorFieldOutput(data1 = fo11)
    >>> print tensor * vector # Matrix product
    VectorFieldOutput instance
    Position = node
    Label Data1   Data2   Data3
    1     1.0     1.0     0.0
    2     1.0     2.0     0.0
    3     1.0     0.0     0.0
    >>> print tensor * tensor2 # Contracted tensor product
    FieldOutput instance
    Position = node
    Label Data
    1     2.0
    2     4.0
    3     -1.0
```

Add/remove/get data

`TensorFieldOutput.`**`add_data`**(*label*, *data11=0.0*, *data22=0.0*, *data33=0.0*, *data12=0.0*, *data13=0.0*, *data23=0.0*)

Adds one point to a VectorFieldOutput instance. Label must not already exist in the current FieldOutput, if not so, nothing will be changed. Data and label will be inserted in self.data, self.labels in order to keep self.labels sorted.

Parameters

- **label** – labels of the nodes/elements where the field is evaluated.
- **data11** – value of the component 11 of the field where it is evaluated.
- **data22** – value of the component 22 of the field where it is evaluated.
- **data33** – value of the component 33 of the field where it is evaluated.
- **data12** – value of the component 12 of the field where it is evaluated.
- **data13** – value of the component 13 of the field where it is evaluated.
- **data23** – value of the component 23 of the field where it is evaluated.

`TensorFieldOutput.`**`get_data`**(*label*)
Returns the components (*11, 22, 33, 12, 13 or 23*) at a location with given label.

Parameters **label** (*int > 0*) – location's label.

Return type float, float, float, float, float, float

Note: Requesting data at a label that does not exist in the instance will just lead in a warning but if label is negative or is not int, then an Exception will be raised.

`TensorFieldOutput.`**`get_component`**(*number*)
Returns a component of the vector as a FieldOutput.

Parameters **number** (*11, 22, 33, 12, 13 or 23*) – requested coordinate number, 1 is x and so on.

Return type FieldOutput instance

```
>>> v1 = Vec.get_coord(1)
```

VTK Export

`TensorFieldOutput.`**`dump2vtk`**(*name='tensorFieldOutput'*, *header=True*)
Converts the TensorFieldOutput instance to VTK format which can be directly read by Mayavi2 or Paraview. This method is very useful to quickly and efficiently plot 3D mesh and fields.

Parameters **name** (*string*) – name used for the field in the output.

Return type string

Operations and invariants

`TensorFieldOutput.`**`sum`**()
Returns the sum of all components of the tensor.

Return type `FieldOutput` instance.

`TensorFieldOutput.`**`trace`**()
Returns the trace of the tensor: $trace(T) = T_{11} + T_{22} + T_{33}$

Return type `FieldOutput` instance.

`TensorFieldOutput.`**`deviatoric`**()
Returns the deviatoric part tensor: $T_d = T - T_s$

Return type TensorFieldOutput instance

TensorFieldOutput.**spheric**()
Returns the spheric part of the tensor: $T_s = \frac{1}{3} trace(T) I_3$

Return type TensorFieldOutput instance

TensorFieldOutput.**i1**()
Returns the first invariant, is equivalent to trace.

Return type FieldOutput instance.

TensorFieldOutput.**i2**()
Returns the second invariant of the tensor defined as: $inv2(T) = trace(T.T)$

Return type FieldOutput instance.

Note: this definition is the most practical one for mechanical engineering but not the only one possible.

TensorFieldOutput.**i3**()
Returns the third invariant of the tensor: $inv3(T) = \det(T)$

Return type FieldOutput instance.

TensorFieldOutput.**j2**()
Returns the second invariant of the deviatoric part of the tensor defined as: $inv2(T) = trace(T_d.T_d)$

Return type FieldOutput instance.

Note: this definition is not the mathematical definition but is the most practical one for mechanical engineering. This should be debated.

TensorFieldOutput.**j3**()
Returns the third invariant of the deviatoric part of the tensor: $inv3(T) = \det(T_d)$

Return type FieldOutput instance.

TensorFieldOutput.**eigen**()
Returns the three eigenvalues with decreasing sorting and the 3 normed respective eigenvectors.

Return type 3 FieldOutput instances and 3 VectorFieldOutput instances.

```
>>> from abapy.postproc import FieldOutput, TensorFieldOutput, VectorFieldOutput, Identity_like
>>> data11 = [0., 0., 1.]
>>> data22 = [0., 0., -1]
>>> data12 = [1., 2., 0.]
>>> labels = range(1,len(data11)+1)
>>> fo11 = FieldOutput(labels = labels, data=data11,position='node')
>>> fo22 = FieldOutput(labels = labels, data=data22,position='node')
>>> fo12 = FieldOutput(labels = labels, data=data12,position='node')
>>> tensor = TensorFieldOutput(data11 = fo11, data22 = fo22, data12 = fo12 )
>>> t1, t2, t3, v1, v2, v3 = tensor.eigen()
>>> print t1
FieldOutput instance
Position = node
Label       Data
1   1.0
2   2.0
3   1.0
```

```
>>> print v1
VectorFieldOutput instance
Position = node
Label       Data1   Data2   Data3
1   0.707106769085  0.707106769085  0.0
2   0.707106769085  0.707106769085  0.0
3   1.0     0.0     0.0
```

`TensorFieldOutput.`**`pressure`**`()`
Returns the pressure.

Return type `FieldOutput` instance.

`TensorFieldOutput.`**`vonmises`**`()`
Returns the von Mises equivalent equivalent *stress* of the tensor: $vonmises(T) = \sqrt{\frac{3}{2}trace(T_d.T_d)}$

Return type `FieldOutput` instance.

`TensorFieldOutput.`**`tresca`**`()`
Returns the tresca equivalent *stress* of the tensor: $tresca(T) = max\left(|t_1 - t_2|, |t_1 - t_3|, |t_2 - t_3|\right)$ where t_i is the i-est eigen value of T.

Return type `FieldOutput` instance.

4.1.4 Getting field outputs from an Abaqus ODB

Scalar fields

`abapy.postproc.`**`GetFieldOutput`**`(`*odb*, *step*, *frame*, *instance*, *position*, *field*, *subField=None*, *labels=None*, *dti='I'*`)`
Retrieves a field output in an Abaqus odb object and stores it in a FieldOutput class instance. Field output that are classically available at integration points must be interpolated at nodes. This can be requested in the Abaqus *inp* file using: *Element Output, position = nodes*.

Parameters

- **`odb`** (*`odb object.`*) – odb object produced by odbAccess.openOdb in abaqus python or abaqus viewer -noGUI
- **`step`** (*`string`*) – step name defined in the abaqus inp file. May be the upper case version of original string name.
- **`frame`** (*`int`*) – requested frame indice in the odb.
- **`instance`** (*`string`*) – instance name defined in the abaqus odb file. May be the upper case version of the original name.
- **`position`** (*`'node', 'element'`*) – position at which the output is to be computed.
- **`field`** (*`string`*) – requested field output ('LE','S','U','AC YIELD',...).
- **`subField`** (*`string or None`*) – requested subfield in the case of non scalar fields, can be a component (U1, S12) or an invariant (mises, tresca, inv3, maxPrincipal). In the case of scalar fields, it has to be None
- **`labels`** (*`list, array.array, numpy.array of unsigned non zero ints or string`*) – if not None, it provides a set of locations (elements/nodes labels or node/element set label) where the field is to be computed. if None, every available location is used and labels are sorted
- **`dti`** (*`'I', 'H'`*) – int data type in `array.array`

Return type `FieldOutput` instance

Note: This function can only be executed in abaqus python or abaqus viewer -noGUI

```
>>> from abapy.postproc import GetFieldOutput
>>> from odbAccess import openOdb
>>> odb = openOdb('indentation.odb')
>>> U2 = GetFieldOutput(odb, step = 'LOADING0', frame = -1, instance ='I_SAMPLE', position =  'n
>>> U1 = GetFieldOutput(odb, step = 'LOADING0', frame = -1, instance ='I_SAMPLE', position =  'n
>>> S11 = GetFieldOutput(odb, step = 'LOADING0', frame = -1, instance ='I_SAMPLE', position =  '
>>> S12 = GetFieldOutput(odb, step = 'LOADING0', frame = -1, instance ='I_SAMPLE', position =  '
```

Note:

- If dti='H' is chosen, labels are stored as unsigned 16 bits ints. If more than 65k labels are stored, an OverFlow error will be raised.
- This function had memory usage problems in its early version, these have been solved by using more widely array.array. It is still a bit slow but with the lack of numpy in Abaqus, no better solutions have been found yet. I'm open to any faster solution even involving the used temporary rpt files procuced by Abaqus

---

`abapy.postproc.`**`MakeFieldOutputReport`**(*odb*, *instance*, *step*, *frame*, *report_name*, *original_position*, *new_position*, *field*, *sub_field=None*, *sub_field_prefix=None*, *sub_set_type=None*, *sub_set=None*)

Writes a field output report using Abaqus. The major interrest of this function is that it is really fast compared to `GetFieldOutput` which tends to get badly slow on odbs containing more than 1500 elements. One other interrest is that it doesn't require to used `position = nodes` option in the INP file to evaluate fields at nodes. It is especially efficient when averaging is necessary (example: computing stress at nodes). The two drawbacks are that it requires `abaqus viewer` (or `cae`) using the `-noGUI` where GetFieldOutput only requires `abaqus python` so it depends on the license server lag (which can be of several seconds). The second drawback is that it requires to write a file in place where you have write permission. This function is made to used in conjunction with `ReadFieldOutputReport`.

Parameters

- **`odb`** (odb instance produced by `odbAccess.openOdb`) – output database to be used.
- **`instance`** (*`string`*) – instance to use.
- **`step`** (*`string or int`*) – step to use, this argument can be either the step number or the step label.
- **`frame`** (*`int`*) – frame number, can be negative for reverse counting.
- **`report_name`** (*`string`*) – name or path+name of the report to written.
- **`original_position`** – position at which the field is expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT'.
- **`new_position`** (*`string`*) – position at which you would like the field to be expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT' or 'ELEMENT_NODAL'. Note that `ReadFieldOutputReport` will be capable of averaging values over elements when 'INTEGRATION_POINT' or 'ELEMENT_NODAL' option is sellected.
- **`field`** (*`string`*) – field to export, example: 'S', 'U', 'EVOL',...

- **sub_field** (*string or int*) – can be a component of an invariant, example: 11, 2, 'Mises', 'Magnitude'. Here the use of 'Mises' instead of 'MISES' can be surprising that's the way abaqus is written..
- **sub_set** (*string*) – set to which the report is restricted, must be the label of an existing node or element set.
- **sub_set_type** (*string*) – type of the sub_set, can be node or element.

All examples below are performed on a small indentation ODB:

```
>>> from odbAccess import openOdb
>>> from abapy.postproc import MakeFieldOutputReport
>>> # Some settings
>>> odb_name = 'indentation.odb'
>>> report_name = 'indentation_core_step0_frame1_S11_nodes.rpt'
>>> step = 0
>>> frame = -1
>>> new_position = 'NODAL'
>>> original_position = 'INTEGRATION_POINT'
>>> field = 'S'
>>> sub_field = 11
>>> instance = 'I_SAMPLE'
>>> sub_set = 'CORE'
>>> sub_set_type = 'element'
>>> # Function testing
>>> odb = openOdb(odb_name)
>>> MakeFieldOutputReport(
...   odb = odb,
...   instance = instance,
...   step = step,
...   frame = frame,
...   report_name = report_name,
...   original_position = original_position,
...   new_position = new_position,
...   field = field,
...   sub_field = sub_field,
...   sub_set_type = sub_set_type,
...   sub_set = sub_set)
>>> new_position = 'INTEGRATION_POINT'
>>> report_name = 'indentation_core_step0_frame1_S11_elements.rpt'
>>> MakeFieldOutputReport(
...   odb = odb,
...   instance = instance,
...   step = step,
...   frame = frame,
...   report_name = report_name,
...   original_position = original_position,
...   new_position = new_position,
...   field = field,
...   sub_field = sub_field,
...   sub_set_type = sub_set_type,
...   sub_set = sub_set)
>>> new_position = 'ELEMENT_NODAL'
>>> report_name = 'indentation_core_step0_frame1_S11_element-nodal.rpt'
>>> MakeFieldOutputReport(
...   odb = odb,
...   instance = instance,
...   step = step,
...   frame = frame,
```

```
...   report_name = report_name,
...   original_position = original_position,
...   new_position = new_position,
...   field = field,
...   sub_field = sub_field,
...   sub_set_type = sub_set_type,
...   sub_set = sub_set)
>>> field = 'U'
>>> sub_field = 'Magnitude'
>>> original_position = 'NODAL'
>>> new_position = 'NODAL'
>>> report_name = 'indentation_core_step0_frame1_U-MAG_nodal.rpt'
>>> MakeFieldOutputReport(
...   odb = odb,
...   instance = instance,
...   step = step,
...   frame = frame,
...   report_name = report_name,
...   original_position = original_position,
...   new_position = new_position,
...   field = field,
...   sub_field = sub_field,
...   sub_set_type = sub_set_type,
...   sub_set = sub_set)
```

Four reports were produced:

- `indentation_core_step0_frame1_S11_nodes.rpt`
- `indentation_core_step0_frame1_S11_elements.rpt`
- `indentation_core_step0_frame1_S11_element-nodal.rpt`
- `indentation_core_step0_frame1_U-MAG_nodal.rpt`

`abapy.postproc.`**`ReadFieldOutputReport`**(*report_name*, *position='node'*, *dti='I'*, *dtf='f'*)

Reads a report file generated by Abaqus (for example using `MakeFieldOutputReport` and converts it in FieldOutputFormat.

Parameters

- **`report_name`** (*`string`*) – report_name or path + name of the report to read.
- **`position`** (*`'node' or 'element'`*) – position where the `FieldOutput` is to be declared. The function will look at the first and the last column of the report. The first will be considered as the label (*i. e.* element or node) and the last the value. In some case, like reports written using 'ELEMENT_NODAL' or 'INTEGRATION_POINT' locations, each label will appear several times. The present function will collect all the corresponding values and average them. At the end, the only possibilities for this parameter should be 'node' or 'element' as described in the doc of `FieldOutput`.
- **`dti`** (*`'I', 'H'`*) – int data type in array.array
- **`dtf`** (*`'f', 'd'`*) – float data type in array.array

Return type `FieldOutput` instance.

Note: This function can be run either in abaqus python, abaqus viewer -noGUI, abaqus cae -noGUI and regular python.

```
>>> from abapy.postproc import ReadFieldOutputReport
>>> report_name = 'indentation_core_step0_frame1_S11_nodes.rpt'
>>> S11 = ReadFieldOutputReport(report_name, position = 'nodes', dti = 'I', dtf = 'f')
```

abapy.postproc.**GetFieldOutput_byRpt**(*odb*, *instance*, *step*, *frame*, *original_position*, *new_position*, *position*, *field*, *sub_field=None*, *sub_field_prefix=None*, *sub_set_type=None*, *sub_set=None*, *report_name='dummy.rpt'*, *dti='I'*, *dtf='f'*, *delete_report=True*)

Wraps `MakeFieldOutputReport` and `ReadFieldOutputReport` in a single function to mimic the behavior `GetFieldOutput`.

Parameters

- **odb** (odb instance produced by `odbAccess.openOdb`) – output database to be used.
- **instance** (*string*) – instance to use.
- **step** (*string or int*) – step to use, this argument can be either the step number or the step label.
- **frame** (*int*) – frame number, can be negative for reverse counting.
- **original_position** – position at which the field is expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT'.
- **new_position** (*string*) – position at which you would like the field to be expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT' or 'ELEMENT_NODAL'. Note that `ReadFieldOutputReport` will be capable of averaging values over elements when 'INTEGRATION_POINT' or 'ELEMENT_NODAL' option is sellected.
- **position** (*'node' or 'element'*) – position where the `FieldOutput` is to be declared. The function will look at the first and the last column of the report. The first will be considered as the label (*i. e.* element or node) and the last the value. In some case, like reports written using 'ELEMENT_NODAL' or 'INTEGRATION_POINT' locations, each label will appear several times. The present function will collect all the corresponding values and average them. At the end, the only possibilities for this parameter should be 'node' or 'element' as described in the doc of `FieldOutput`.
- **field** (*string*) – field to export, example: 'S', 'U', 'EVOL',...
- **sub_field** (*string or int*) – can be a component of an invariant, example: 11, 2, 'Mises', 'Magnitude'.
- **sub_set** (*string*) – set to which the report is restricted, must be the label of an existing node or element set.
- **sub_set_type** (*string*) – type of the sub_set, can be node or element.
- **report_name** (*string*) – name or path+name of the report to written.
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array
- **delete_report** (*boolean*) – if True, report will be deleted, if false, it will remain.

Return type `FieldOutput` instance.

```
>>> from odbAccess import openOdb
>>> from abapy.postproc import GetFieldOutput_byRpt
>>> odb_name = 'indentation.odb'
```

```
>>> odb = openOdb(odb_name)
>>> S11 = GetFieldOutput_byRpt(
...   odb = odb,
...   instance = 'I_SAMPLE',
...   step = 0,
...   frame = -1,
...   original_position = 'INTEGRATION_POINT',
...   new_position = 'NODAL',
...   position = 'node',
...   field = 'S',
...   sub_field = 11,
...   sub_set_type = 'element',
...   sub_set = 'CORE',
...   delete_report = False)
```

Vector fields

abapy.postproc.**GetVectorFieldOutput**(*odb*, *step*, *frame*, *instance*, *position*, *field*, *labels=None*, *dti='I'*)

Returns a VectorFieldOutput from an odb object.

Parameters

- **odb** (*odb object.*) – odb object produced by odbAccess.openOdb in abaqus python or abaqus viewer -noGUI
- **step** (*string*) – step name defined in the abaqus inp file. May be the upper case version of original string name.
- **frame** (*int*) – requested frame indice in the odb.
- **instance** (*string*) – instance name defined in the abaqus odb file. May be the upper case version of the original name.
- **position** (*'node', 'element'*) – position at which the output is to be computed.
- **field** (*string*) – requested vector field output ('U',...).
- **labels** (*list, array.array, numpy.array of unsigned non zero ints or string*) – if not None, it provides a set of locations (elements/nodes labels or node/element set label) where the field is to be computed. if None, every available location is used and labels are sorted
- **dti** (*'I', 'H'*) – int data type in `array.array`

Return type `VectorFieldOutput` instance

Note: This function can only be executed in abaqus python or abaqus viewer -noGUI

```
>>> from abapy.postproc import GetFieldOutput, GetVectorFieldOutput
>>> from odbAccess import openOdb
>>> odb = openOdb('indentation.odb')
>>> U = GetVectorFieldOutput(odb, step = 'LOADING', frame = -1, instance ='I_SAMPLE', position =
>>> odb.close()
```

abapy.postproc.**GetVectorFieldOutput_byRpt**(*odb*, *instance*, *step*, *frame*, *original_position*, *new_position*, *position*, *field*, *sub_field_prefix=None*, *sub_set_type=None*, *sub_set=None*, *report_name='dummy.rpt'*, *dti='I'*, *dtf='f'*, *delete_report=True*)

Uses `GetFieldOutput_byRpt` to produce VectorFieldOutput.

Parameters

- **odb** (odb instance produced by `odbAccess.openOdb`) – output database to be used.
- **instance** (*string*) – instance to use.
- **step** (*string or int*) – step to use, this argument can be either the step number or the step label.
- **frame** (*int*) – frame number, can be negative for reverse counting.
- **original_position** – position at which the field is expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT'.
- **new_position** (*string*) – position at which you would like the field to be expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT' or 'ELEMENT_NODAL'. Note that `ReadFieldOutputReport` will be capable of averaging values over elements when 'INTEGRATION_POINT' or 'ELEMENT_NODAL' option is sellected.
- **position** (*'node' or 'element'*) – position where the `FieldOutput` is to be declared. The function will look at the first and the last column of the report. The first will be considered as the label (*i. e.* element or node) and the last the value. In some case, like reports written using 'ELEMENT_NODAL' or 'INTEGRATION_POINT' locations, each label will appear several times. The present function will collect all the corresponding values and average them. At the end, the only possibilities for this parameter should be 'node' or 'element' as described in the doc of `FieldOutput`.
- **field** (*string*) – field to export, example: 'S', 'U', 'EVOL',...
- **sub_set** (*string*) – set to which the report is restricted, must be the label of an existing node or element set.
- **sub_set_type** (*string*) – type of the sub_set, can be node or element.
- **report_name** (*string*) – name or path+name of the report to written.
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array
- **delete_report** (*boolean*) – if True, report will be deleted, if false, it will remain.

Return type `VectorFieldOutput` instance.

```
>>> from odbAccess import openOdb
>>> from abapy.postproc import GetVectorFieldOutput_byRpt
>>> odb_name = 'indentation.odb'
>>> odb = openOdb(odb_name)
>>> U = GetVectorFieldOutput_byRpt(
...   odb = odb,
...   instance = 'I_SAMPLE',
...   step = 0,
...   frame = -1,
...   original_position = 'NODAL',
...   new_position = 'NODAL',
```

```
...   position = 'node',
...   field = 'U',
...   sub_set_type = 'element',
...   sub_set = 'CORE',
...   delete_report = True)
```

Tensor fields

abapy.postproc.**GetTensorFieldOutput**(*odb*, *step*, *frame*, *instance*, *position*, *field*, *labels=None*, *dti='I'*)

Returns a TensorFieldOutput from an odb object.

Parameters

- **odb** (*odb object.*) – odb object produced by odbAccess.openOdb in abaqus python or abaqus viewer -noGUI
- **step** (*string*) – step name defined in the abaqus inp file. May be the upper case version of original string name.
- **frame** (*int*) – requested frame indice in the odb.
- **instance** (*string*) – instance name defined in the abaqus odb file. May be the upper case version of the original name.
- **position** (*'node', 'element'*) – position at which the output is to be computed.
- **field** (*string*) – requested tensor field output ('LE','S',...).
- **labels** (*list, array.array, numpy.array of unsigned non zero ints or string*) – if not None, it provides a set of locations (elements/nodes labels or node/element set label) where the field is to be computed. if None, every available location is used and labels are sorted
- **dti** (*'I', 'H'*) – int data type in `array.array`

Return type `TensorFieldOutput` instance

Note: This function can only be executed in abaqus python or abaqus viewer -noGUI

```
>>> from abapy.postproc import GetFieldOutput, GetVectorFieldOutput, GetTensorFieldOutput
>>> from odbAccess import openOdb
>>> odb = openOdb('indentation.odb')
>>> S = GetTensorFieldOutput(odb, step = 'LOADING', frame = -1, instance ='I_SAMPLE', position =
>>> odb.close()
```

abapy.postproc.**GetTensorFieldOutput_byRpt**(*odb*, *instance*, *step*, *frame*, *original_position*, *new_position*, *position*, *field*, *sub_field_prefix=None*, *sub_set_type=None*, *sub_set=None*, *report_name='dummy.rpt'*, *dti='I'*, *dtf='f'*, *delete_report=True*)

Uses `GetFieldOutput_byRpt` to produce TensorFieldOutput.

Parameters

- **odb** (odb instance produced by `odbAccess.openOdb`) – output database to be used.
- **instance** (*string*) – instance to use.

- **step** (*string or int*) – step to use, this argument can be either the step number or the step label.
- **frame** (*int*) – frame number, can be negative for reverse counting.
- **original_position** – position at which the field is expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT'.
- **new_position** (*string*) – position at which you would like the field to be expressed. Can be 'NODAL', 'WHOLE_ELEMENT' or 'INTEGRATION_POINT' or 'ELEMENT_NODAL'. Note that `ReadFieldOutputReport` will be capable of averaging values over elements when 'INTEGRATION_POINT' or 'ELEMENT_NODAL' option is sellected.
- **position** (*'node' or 'element'*) – position where the `FieldOutput` is to be declared. The function will look at the first and the last column of the report. The first will be considered as the label (*i. e.* element or node) and the last the value. In some case, like reports written using 'ELEMENT_NODAL' or 'INTEGRATION_POINT' locations, each label will appear several times. The present function will collect all the corresponding values and average them. At the end, the only possibilities for this parameter should be 'node' or 'element' as described in the doc of `FieldOutput`.
- **field** (*string*) – field to export, example: 'S', 'U', 'EVOL',...
- **sub_set** (*string*) – set to which the report is restricted, must be the label of an existing node or element set.
- **sub_set_type** (*string*) – type of the sub_set, can be node or element.
- **report_name** (*string*) – name or path+name of the report to written.
- **dti** (*'I', 'H'*) – int data type in array.array
- **dtf** (*'f', 'd'*) – float data type in array.array
- **delete_report** (*boolean*) – if True, report will be deleted, if false, it will remain.

Return type `TensorFieldOutput` instance.

```
>>> from odbAccess import openOdb
>>> from abapy.postproc import GetTensorFieldOutput_byRpt
>>> odb_name = 'indentation.odb'
>>> odb = openOdb(odb_name)
>>> S = GetTensorFieldOutput_byRpt(
...   odb = odb,
...   instance = 'I_SAMPLE',
...   step = 0,
...   frame = -1,
...   original_position = 'INTEGRATION_POINT',
...   new_position = 'NODAL',
...   position = 'node',
...   field = 'S',
...   sub_set_type = 'element',
...   sub_set = 'CORE',
...   delete_report = True)
```

4.1.5 ZeroFieldOutput_like

`abapy.postproc.`**ZeroFieldOutput_like**(*fo*)

A FieldOutput containing only zeros but with the same position, labels and dtypes as the input.

Parameters **fo** (*FieldOutput instance*) – field output to be used.

Return type FieldOutput instance

Note: uses Numpy.

4.1.6 OneFieldOutput_like

abapy.postproc.**OneFieldOutput_like**(*fo*)

A FieldOutput containing only ones but with the same position, labels and dtypes as the input.

Parameters **fo** (*FieldOutput instance*) – field output to be used.

Return type FieldOutput instance

Note: uses Numpy.

4.1.7 Identity_like

abapy.postproc.**Identity_like**(*fo*)

A TensorFieldOutput containing only identity but with the same position, labels and dtypes as the input.

Parameters **fo** (*TensorFieldOutput instance*) – tensor field output to be used.

Return type TensorFieldOutput instance

```
>>> from abapy.postproc import FieldOutput, TensorFieldOutput, Identity_like
>>> data1 = [1,2,3,5,6,]
>>> data2 = [1. for i in data1]
>>> labels = range(1,len(data1)+1)
>>> fo1, fo2 = FieldOutput(labels = labels, data=data1, position='node' ), FieldOutput(labels =
>>> tensor = TensorFieldOutput(data11 = fo1, data22 = fo2 )
>>> identity = Identity_like(tensor)
>>> print identity
TensorFieldOutput instance
Position = node
Label Data11  Data22  Data33  Data12  Data13  Data23
1     1.0e+00 1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00
2     1.0e+00 1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00
3     1.0e+00 1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00
4     1.0e+00 1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00
5     1.0e+00 1.0e+00 1.0e+00 0.0e+00 0.0e+00 0.0e+00
```

4.2 History Outputs

4.2.1 HistoryOutput class

class abapy.postproc.**HistoryOutput**(*time=[]*, *data=[]*, *dtf='f'*)

Stores history output data from and allows useful operations. The key idea of this class is to allow easy storage of time dependant data without merging steps to allow further separating of each test steps (loading, unloading,...).

The class allows additions, multiplication, ... between class instances and between class instances and int/floats. These operations only affect y data as long as time has no reason to be affected.

Parameters

- **time** (*list of list/array.array containing floats*) – time represented by nested lists, each one corresponding to a step.
- **data** (*list of list/array.array containing floats*) – data (ex: displacement, force, energy ...). It is represented by nested lists, each one corresponding to a step.
- **dtf** (*'f', 'd'*) – float data type used by array.array

```
>>> from abapy.postproc import HistoryOutput
>>> time = [ [1., 2.,3.] , [3.,4.,5.] , [5.,6.,7.] ] # Time separated in 3 steps
>>> data = [ [2.,2.,2.] , [3.,3.,3.] , [4.,4.,4.] ] # Data separated in 3 steps
>>> Data = HistoryOutput(time, data)
>>> print Data
Field output instance: 3 steps
Step 0: 3 points
Time  Data
1.0   2.0
2.0   2.0
3.0   2.0
Step 1: 3 points
Time  Data
3.0   3.0
4.0   3.0
5.0   3.0
Step 2: 3 points
Time  Data
5.0   4.0
6.0   4.0
7.0   4.0
>>> # +, *, **, abs, neg act only on data, not on time
... print Data + Data + 1. # addition
Field output instance: 3 steps
Step 0: 3 points
Time  Data
1.0   5.0
2.0   5.0
3.0   5.0
Step 1: 3 points
Time  Data
3.0   7.0
4.0   7.0
5.0   7.0
Step 2: 3 points
Time  Data
5.0   9.0
6.0   9.0
7.0   9.0
>>> print Data * Data * 2. # multiplication
Field output instance: 3 steps
Step 0: 3 points
Time  Data
1.0   8.0
2.0   8.0
3.0   8.0
```

```
    Step 1: 3 points
    Time  Data
    3.0   18.0
    4.0   18.0
    5.0   18.0
    Step 2: 3 points
    Time  Data
    5.0   32.0
    6.0   32.0
    7.0   32.0
    >>> print ( Data / Data ) / 2. # division
    Field output instance: 3 steps
    Step 0: 3 points
    Time  Data
    1.0   0.5
    2.0   0.5
    3.0   0.5
    Step 1: 3 points
    Time  Data
    3.0   0.5
    4.0   0.5
    5.0   0.5
    Step 2: 3 points
    Time  Data
    5.0   0.5
    6.0   0.5
    7.0   0.5
    >>> print Data ** 2
    Field output instance: 3 steps
    Step 0: 3 points
    Time  Data
    1.0   4.0
    2.0   4.0
    3.0   4.0
    Step 1: 3 points
    Time  Data
    3.0   9.0
    4.0   9.0
    5.0   9.0
    Step 2: 3 points
    Time  Data
    5.0   16.0
    6.0   16.0
    7.0   16.0
    >>> print abs(Data)
    Field output instance: 3 steps
    Step 0: 3 points
    Time  Data
    1.0   2.0
    2.0   2.0
    3.0   2.0
    Step 1: 3 points
    Time  Data
    3.0   3.0
    4.0   3.0
    5.0   3.0
    Step 2: 3 points
    Time  Data
```

```
    5.0   4.0
    6.0   4.0
    7.0   4.0
    >>> print Data[1] # step 1
    Field output instance: 1 steps
    Step 0: 3 points
    Time  Data
    3.0   3.0
    4.0   3.0
    5.0   3.0
    >>> print Data[0:2]
    Field output instance: 2 steps
    Step 0: 3 points
    Time  Data
    1.0   2.0
    2.0   2.0
    3.0   2.0
    Step 1: 3 points
    Time  Data
    3.0   3.0
    4.0   3.0
    5.0   3.0
    >>> print Data[0,2]
    Field output instance: 2 steps
    Step 0: 3 points
    Time  Data
    1.0   2.0
    2.0   2.0
    3.0   2.0
    Step 1: 3 points
    Time  Data
    5.0   4.0
    6.0   4.0
    7.0   4.0
```

Add/get data

HistoryOutput.**add_step**(*time_step*, *data_step*)

Adds data to an HistoryOutput instance.

Parameters

- **time_step** (*list, array.array, np.array containing floats*) – time data to be added.
- **data_step** (*list, array.array, np.array containing floats*) – data to be added.

```
    >>> from abapy.postproc import HistoryOutput
    >>> time = [ [0.,0.5, 1.] , [1., 1.5, 2.] ]
    >>> force = [ [4.,2., 1.] , [1., .5, .2] ] ]
    >>> Force = HistoryOutput(time,force)
    >>> Force.time # time
    [array('f', [0.0, 0.5, 1.0]), array('f', [1.0, 1.5, 2.0])]
    >>> Force.add_step([5.,5.,5.],[4.,4.,4.])
    >>> Force.time
    [array('f', [0.0, 0.5, 1.0]), array('f', [1.0, 1.5, 2.0]), array('f', [5.0, 5.0, 5.0])]
```

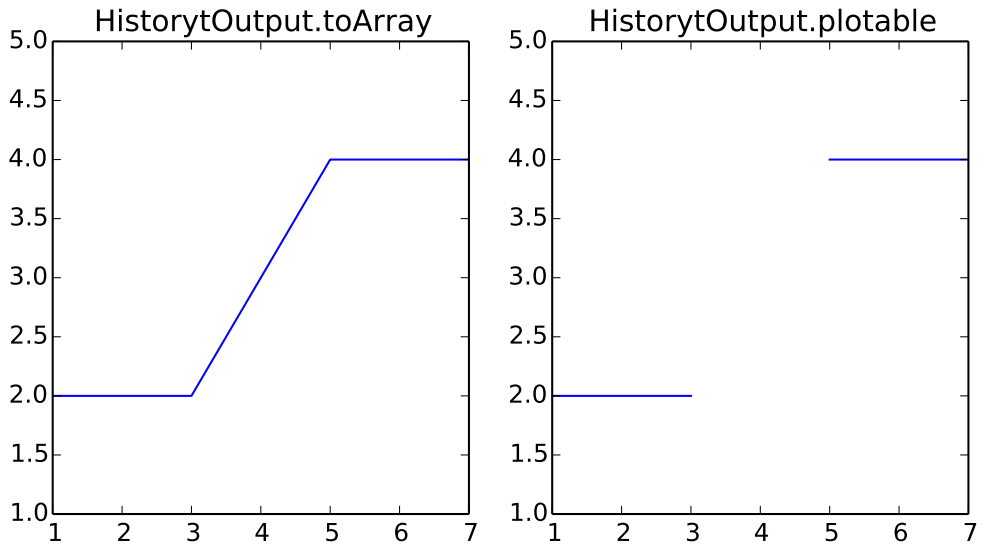
```
>>> Force.data
[array('f', [4.0, 2.0, 1.0]), array('f', [1.0, 0.5, 0.20000000298023224]), array('f', [4.0, 4.0,
```

HistoryOutput.**plotable**()

Gives back plotable version of the history output. Plotable differs from toArray on one point, toArray will concatenate steps into one single array for x and one for y where plotable will add None between steps before concatenation. Adding None allows matplotlib to draw discontinuous lines between steps without requiring ploting several independant arrays. By the way, the None methode is far faster.

Return type 2 lists of floats and None

```
import matplotlib.pyplot as plt
from abapy.postproc import HistoryOutput
time = [ [1., 2.,3.] , [3.,4.,5.] , [5.,6.,7.] ]
force = [ [2.,2.,2.] , [3.,3.,3.] , [4.,4.,4.] ]
Force = HistoryOutput(time, force)
fig = plt.figure(0, figsize=(8,4))
plt.clf()
ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
x,y = Force[[0,2]].toArray()
x2,y2 = Force[[0,2]].plotable()
ax.plot(x,y)
ax2.plot(x2,y2)
ax.set_ylim([1,5])
ax2.set_ylim([1,5])
plt.savefig('HistoryOutput-plotable.png')
ax.set_title('HistorytOutput.toArray')
ax2.set_title('HistorytOutput.plotable')
plt.show()
```



HistoryOutput.**toArray**()

Returns an array.array of concatenated steps for x and y.

Return type array.array

```
>>> from abapy.postproc import HistoryOutput
>>> time = [ [1., 2.,3.] , [3.,4.,5.] , [5.,6.,7.] ]
>>> force = [ [2.,2.,2.] , [3.,3.,3.] , [4.,4.,4.] ]
```

```
>>> Force = HistoryOutput(time, force)
>>> x,y = Force.toArray()
>>> x
array('f', [1.0, 2.0, 3.0, 3.0, 4.0, 5.0, 5.0, 6.0, 7.0])
>>> y
array('f', [2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 4.0, 4.0, 4.0])
```

Utilities

HistoryOutput.**total**()
Returns the total of all data.

Return type float

HistoryOutput.**integral**(*method='trapz'*)
Returns the integral of the history output using the trapezoid or Simpson rule.

Parameters **method** (*string*) – choice between trapezoid rule ('trapz') or Simpson rule ('simps').

Return type float

```
>>> from abapy.postproc import HistoryOutput
>>> time = [ [0., 1.], [3., 4.] ]
>>> data = [ [.5, 1.5], [.5, 1.5] ]
>>> hist = HistoryOutput(time = time, data = data)
>>> hist[0].integral()
1.0
>>> hist[1].integral()
1.0
>>> hist.integral()
2.0
>>> N = 10
>>> from math import sin, pi
>>> time = [pi / 2 * float(i)/N for i in xrange(N+1)]
>>> data = [sin(t) for t in time]
>>> hist = HistoryOutput()
>>> hist.add_step(time_step = time, data_step = data)
>>> trap = hist.integral()
>>> simp = hist.integral(method = 'simps')
>>> trap_error = (trap -1.)
>>> simp_error = (simp -1.)
```

Relative errors:

- Trapezoid rule: -0.21%
- Simpson rule: 0.00033%

Note: uses `scipy`

HistoryOutput.**average**(*method='trapz'*)
Returns the average of all data over time using `integral`. This average is performed step by step to avoid errors due to disconnected steps.

Parameters **method** (*string*) – choice between trapezoid rule ('trapz') or Simpson rule ('simps').

Return type float

```
>>> from abapy.postproc import HistoryOutput
>>> from math import sin, pi
>>> N = 100
>>> hist = HistoryOutput()
>>> time = [pi / 2 * float(i)/N for i in xrange(N+1)]
>>> data = [sin(t) for t in time]
>>> hist.add_step(time_step = time, data_step = data)
>>> time2 = [10., 11.]
>>> data2 = [1., 1.]
>>> hist.add_step(time_step = time2, data_step = data2)
>>> sol = 2. / pi + 1.
>>> print 'Print computed value:', hist.average()
Print computed value: 1.63660673935
>>> print 'Analytic solution:', sol
Analytic solution: 1.63661977237
>>> print 'Relative error: {0:.4}%'.format( (hist.average() - sol)/sol * 100.)
Relative error: -0.0007963%
```

`HistoryOutput.`**`data_min`**`()`
Returns the minimum value of data.

Return type float

`HistoryOutput.`**`data_max`**`()`
Returns the maximum value of data.

Return type float

`HistoryOutput.`**`time_min`**`()`
Returns the minimum value of time.

Return type float

`HistoryOutput.`**`time_max`**`()`
Returns the maximum value of time.

Return type float

`HistoryOutput.`**`duration`**`()`
Returns the duration of the output by computing max(time) - min(time).

Return type float

4.2.2 `GetHistoryOutputByKey` function

`abapy.postproc.`**`GetHistoryOutputByKey`**`(`*odb*, *key*`)`
Retrieves an history output in an odb object using key (U2, EVOL, RF2,...)

Parameters

- **odb** (*odb object*) – Abaqus output database object produced by odbAcces.openOdb.
- **key** (*string*) – name of the requested variable (*i. e.* 'U2', 'COOR1', ...)

Return type dict of HistoryOutput instance where keys are HistoryRegions names (*i. e.* locations)

```
>>> from odbAccess import openOdb
>>> odb = openOdb('mySimulation.odb')
>>> from abapy.postproc import GetHistoryOutputByKey
>>> u_2 = GetHistoryOutputByKey(odb,'U2')
```

4.3 Mesh

4.3.1 GetMesh function

abapy.postproc.**GetMesh**(*odb*, *instance*, *dti='I'*)

Retrieves mesh on an instance in an Abaqus Output Database.

Parameters

- **odb** (*odb object*) – output database
- **instance** (*string*) – instance name declared in the Abaqus inp file.
- **dti** (*'I' or 'H'*) – int data type in array.array

Return type Mesh instance

```
from abapy.postproc import GetMesh
from odbAccess import openOdb
odb = openOdb('myOdb.odb')
mesh = GetMesh(odb,'MYINSTANCE')
```

CHAPTER 5

Indentation

Indentation simulation tools

5.1 Indentation meshes

5.1.1 `ParamInfiniteMesh` function

abapy.indentation.**ParamInfiniteMesh**(*Na=10*, *Nb=10*, *l=1.0*, *core_name='CAX4'*, *add_shell=True*, *shell_name='CINAX4'*, *dti='I'*, *dtf='d'*)

Returns a mesh dedicated to 2D/Axisymmetrical indentation. It is composed of a core of quadrangle elements and a shell of infinite elements. The core is divided into three zones. The center is core_1, the right is core_2 and the bottom is core_3. Core_1 is a Na x Na square mesh whereas core_2 and core_3 are respectively Nb x Na and Na x Nb structured meshes that have been transformed to be connected and guaranty element size progression.

Parameters

- **Na** (*int > 1*) – number of elements per side in core_1
- **Nb** (*int > 0*) – number of radial elements in core_2 and core_3
- **l** (*float > 0*) – core_1 size
- **core_name** (*string*) – element name in core
- **add_shell** (*boolean*) – True if the shell of infinite elements is to be used, False if not.
- **shell_name** (*string*) – element name in shell, should be infinite

5.1.2 `IndentationMesh` function

abapy.indentation.**IndentationMesh**(*Na=8*, *Nb=8*, *Ns=4*, *Nf=2*, *l=1.0*, *name='CAX4'*, *dtf='f'*, *dti='I'*)

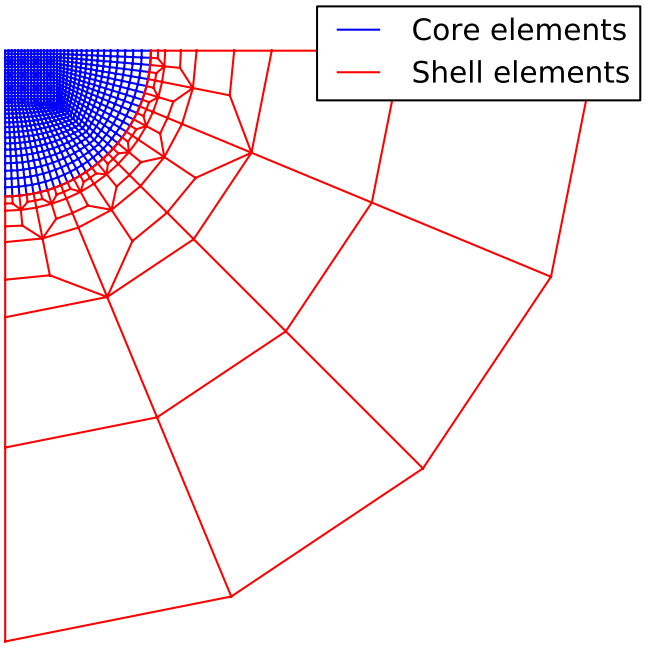
An indentation oriented full quad mesh.

Parameters

- **Na** (*int*) – number of elements along x axis in the finely meshed contact zone. *Must be power of 2*.
- **Nb** (*int*) – number of elements along y axis in the finely meshed contact zone. *Must be power of 2*.

- **Ns** (*int*) – number of radial elements in the shell.
- **Nf** (*int*) – number of orthoradial elements in each half shell. Must be > 0.
- **l** (*float.*) – length of the square zone.
- **name** (*string*) – name of the elements. Note that this mesh if full quad so only one name is required.
- **dtf** (*string*) – float data type in `array.array`, 'd' for float64 or 'f' for float32.
- **dti** (*string*) – int data type in `array.array`, 'I' for unsignedint32 or 'H' for unsignedint16 (dangerous in some cases).

```
from abapy.indentation import IndentationMesh
from matplotlib import pyplot as plt

Na = 16 # Elements along x axis in the square zone
Nb = 16 # Elements along y axis in the square zone
Ns = 2 # Radial number of elements in the shell
Nf = 2 # Minimal number of orthoradial elements in heach half shell
l = 1. # Size of the square zone
name = 'CAX4' # Name of the elements

m = IndentationMesh(Na = Na, Nb = Nb, Ns = Ns, Nf= Nf, l = l, name = name)

m_core = m['core_elements']
m_shell = m['shell_elements']

x_core, y_core, z_core = m_core.get_edges()
x_shell, y_shell, z_shell = m_shell.get_edges()
plt.figure(0)
plt.clf()
plt.axis('off')
plt.grid()
xlim, ylim, zlim = m.nodes.boundingBox()
plt.gca().set_aspect('equal')
plt.xlim(xlim)
plt.ylim(ylim)
plt.plot(x_core,y_core, 'b-', label = 'Core elements')
plt.plot(x_shell,y_shell, 'r-', label = 'Shell elements')
plt.legend()
plt.show()
```



5.2 Indenters

5.2.1 `RigidCone2D` class

class `abapy.indentation.`**`RigidCone2D`**(*half_angle=70.3*, *width=10.0*, *summit_position=(0.0, 0.0)*)
A rigid cone usable in 2D and Axisymmetric simulations.

Parameters

- **`half_angle`** (*`float > 0.`*) – half_angle in DEGREES.
- **`width`** (*`float > 0.`*) – width of the indenter
- **`summit_position`** (*`tuple or list containing two floats.`*) – position of the summit in a 2D space.

`apply_displacement`(*disp*)
Applies a displacement field to the indenter.

Parameters **`disp`** (`abapy.postproc.VectorFieldOutput` instance.) – displacement field (with only one location).

`dump2inp`()
Dumps to Abaqus INP format.

Return type string

get_edges()
Returns a plotable version of the indenter usable directly in `matplotlib.pyplot`.

Return type x and y lists

set_half_angle(*half_angle=70.3*)
Sets the half angle of the indenter. Default is equivalent to modified Berkovich indenter in volume.

Parameters **half_angle** (*float > 0.*) – half_angle in DEGREES.

set_summit_position(*summit_position*)
Sets the position of the indenter.

Parameters **summit_position** (*tuple or list containing two floats.*) – position of the summit in a 2D space.

set_width(*width*)
Sets the width of the indenter.

Parameters **width** (*float > 0.*) – width

5.2.2 DeformableCone2D class

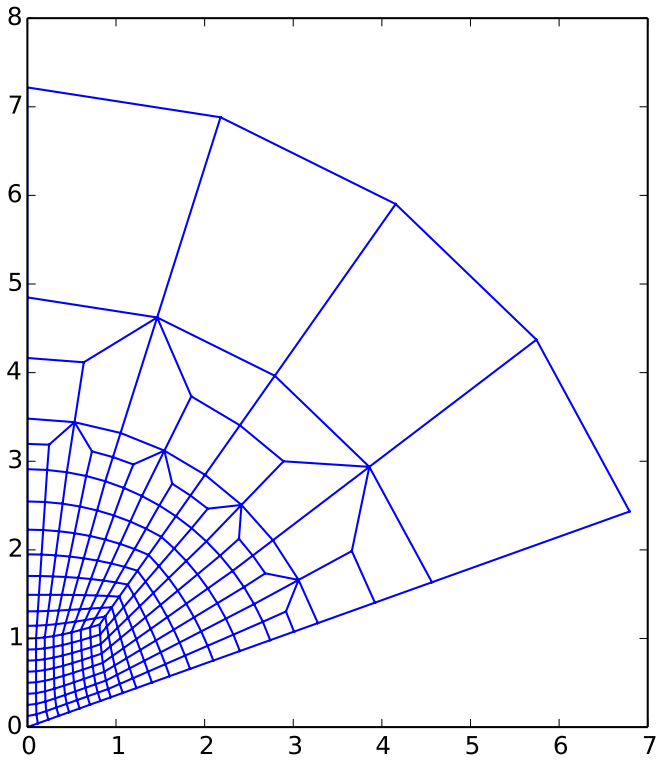
class abapy.indentation.**DeformableCone2D**(*half_angle=70.3*, *Na=4*, *Nb=4*, *Ns=4*, *Nf=2*, *l=1.0*, *mat_label='INDENTER_MAT'*, *summit_position=(0.0, 0.0)*, *rigid=False*)
A deformable cone usable in 2D and Axisymmetric simulations.

Parameters

- **half_angle** (*float > 0.*) – half_angle in DEGREES.
- **Na** (*int*) – number of elements along x axis in the finely meshed contact zone. *Must be power of 2*.
- **Nb** (*int*) – number of elements along y axis in the finely meshed contact zone. *Must be power of 2*.
- **Ns** (*int*) – number of radial elements in the shell.
- **Nf** (*int*) – number of orthoradial elements in each half shell. Must be > 0.
- **l** (*float.*) – length of the square zone.
- **mat_label** (*any material class instance*) – label of the constitutive material of the indenter.
- **summit_position** (*tuple or list containing two floats.*) – position of the summit in a 2D space.
- **rigid** – True if indenter is to be rigid or False if the indenter is to be deformable. If the rigid behavior is chosen, the material label will be necessary but will not influence the results of the simulation.

```
from abapy.indentation import DeformableCone2D
from matplotlib import pyplot as plt
c = DeformableCone2D(Na =8, Nb = 8, Ns = 1)
f = open('DeformableCone2D.inp', 'w')
f.write(c.dump2inp())
f.close()
x,y,z = c.mesh.get_edges()
plt.plot(x,y)
```

```
    plt.gca().set_aspect('equal')
    plt.show()
```



apply_displacement(*disp*)
Applies a displacement field to the indenter.

Parameters **disp** (`abapy.postproc.VectorFieldOutput` instance.) – displacement field (with only one location).

dump2inp()
Dumps to Abaqus INP format.

Return type string

equivalent_half_angle()

get_border(***kwargs*)
Returns a plotable version of the border of the indenter usable directly in `matplotlib.pyplot`.

Return type x and y lists

get_edges(***kwargs*)
Returns a plotable version of the indenter usable directly in `matplotlib.pyplot`.

Return type x and y lists

set_Na(*Na*)
Sets the Na parameter of the indenter (see `IndentationMesh` for explanations).

Parameters **Na** (*int > 1*) – Na

set_Nb(*Nb*)
Sets the Nb parameter of the indenter (see `IndentationMesh` for explanations).

Parameters **Nb** (*int > 1*) – Nb

set_Nf(*Nf*)
Sets the Nf parameter of the indenter (see `IndentationMesh` for explanations).

Parameters **Nf** (*int > 1*) – Nf

set_Ns(*Ns*)
Sets the Ns parameter of the indenter (see `'IndentationMesh` for explanations).

Parameters **Ns** (*int > 1*) – Ns

set_half_angle(*half_angle=70.3*)
Sets the half angle of the indenter. Default is equivalent to modified Berkovich indenter in volume.

Parameters **half_angle** (*float > 0.*) – half_angle in DEGREES.

set_l(*l*)
Sets the l parameter of the indenter (see `ParamInfiniteMesh` for explanations)

Parameters **l** (*float > 0.*) – l

set_mat_label(*mat_label*)
Sets the label of the constitutive material of the indenter.

Parameters **mat_label** (*string*) – mat_label

set_rigid(*rigid*)
Sets the indenter to be rigid (True) or deformable (False).

Parameters **rigid** (*bool*) – True for rigid, False for deformable (default)

set_summit_position(*summit_position*)
Sets the position of the indenter.

Parameters **summit_position** (*tuple or list containing two floats.*) – position of the summit in a 2D space.

5.2.3 `DeformableCone3D` class

class abapy.indentation.**DeformableCone3D**(*half_angle=70.3*, *Na=4*, *Nb=4*, *Ns=4*, *Nf=2*, *l=1.0*, *N=4*, *sweep_angle=45.0*, *mat_label='INDENTER_MAT'*, *summit_position=(0.0, 0.0)*, *rigid=True*, *pyramid=False*)
A deformable cone usable in 3D simulations.

Parameters

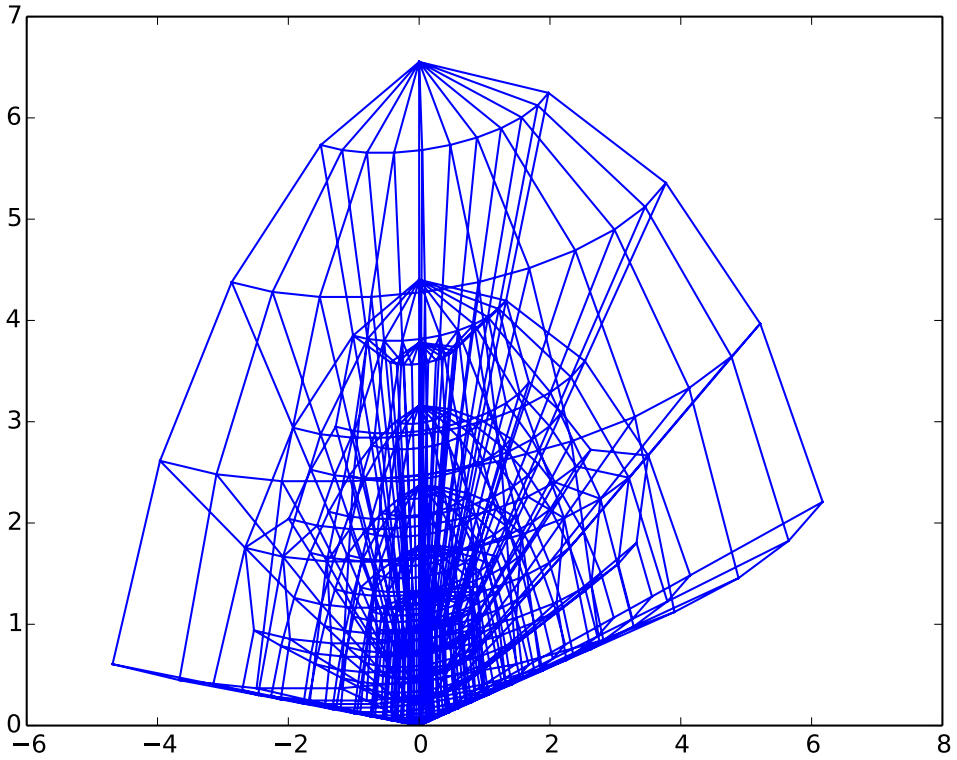
- **half_angle** (*float > 0.*) – half_angle in DEGREES.
- **Na** (*int*) – number of elements along x axis in the finely meshed contact zone. *Must be power of 2*.
- **Nb** (*int*) – number of elements along y axis in the finely meshed contact zone. *Must be power of 2*.
- **Ns** (*int*) – number of radial elements in the shell.
- **Nf** (*int*) – number of orthoradial elements in each half shell. Must be > 0.

- **l** (*float.*) – length of the square zone.
- **N** (*int.*) – Number of sweeped elements.
- **sweep_angle** – sweep angle.
- **mat_label** (*any material class instance*) – label of the constitutive material of the indenter.
- **summit_position** (*tuple or list containing two floats.*) – position of the summit in a 2D space.
- **rigid** – True if indenter is to be rigid or False if the indenter is to be deformable. If the rigid behavior is chosen, the material label will be necessary but will not influence the results of the simulation.
- **pyramid** (*bool*) – Sets the indenter as a revolution cone (False) or a pyramid (True). I the case of the pyramid, the half angle becomes the axis to face angle.

For common 3D indenters, following parameters can be used:

Indenter	half_angle	sweep_angle
Berkovich	65.03	60.00
Modified Berkovich	65.27	60.00
Cube Corner	35.26	60.00
Vickers	68.00	45.00

```
from abapy.indentation import DeformableCone3D
from matplotlib import pyplot as plt
c = DeformableCone3D(Na =4, Nb = 4, Ns = 1, N = 10, sweep_angle = 120.)
c.make_mesh()
f = open('DeformableCone3D.vtk', 'w')
f.write(c.mesh.dump2vtk())
f.close()
x,y,z = c.mesh.get_edges()
# Adding some 3D "home made" perspective:
zx, zy = .3, .3
for i in xrange(len(x)):
  if x[i] != None:
    x[i] += zx * z[i]
    y[i] += zy * z[i]
plt.plot(x,y)
plt.show()
```



apply_displacement(*disp*)
Applies a displacement field to the indenter.

Parameters **disp** (abapy.postproc.VectorFieldOutput instance.) – displacement field (with only one location).

dump2inp()
Dumps to Abaqus INP format.

Return type string

equivalent_half_angle()
Returns the half angle (in degrees) of the equivalent cone in terms of cross area and volume. :retype: float

get_border()
Returns a plotable version of the border of the indenter usable directly in matplotlib.pyplot.

Return type x and y lists

get_edges()
Returns a plotable version of the indenter usable directly in matplotlib.pyplot.

Return type x and y lists

set_N(*N*)
Sets the number of sweeped elements

Parameters **N** (*int > 1*) – N

set_Na(*Na*)
Sets the Na parameter of the indenter (see IndentationMesh for explanations).

Parameters **Na** (*int > 1*) – Na

set_Nb(*Nb*)
Sets the Nb parameter of the indenter (see IndentationMesh for explanations).

Parameters **Nb** (*int > 1*) – Nb

set_Nf(*Nf*)
Sets the Nf parameter of the indenter (see IndentationMesh for explanations).

Parameters **Nf** (*int > 1*) – Nf

set_Ns(*Ns*)
Sets the Ns parameter of the indenter (see 'IndentationMesh for explanations).

Parameters **Ns** (*int > 1*) – Ns

set_half_angle(*half_angle=70.3*)
Sets the half angle of the indenter. Default is equivalent to modified Berkovich indenter in volume.

Parameters **half_angle** (*float > 0.*) – half_angle in DEGREES.

set_l(*l*)
Sets the l parameter of the indenter (see ParamInfiniteMesh for explanations)

Parameters **l** (*float > 0.*) – l

set_mat_label(*mat_label*)
Sets the label of the constitutive material of the indenter.

Parameters **mat_label** (*string*) – mat_label

set_pyramid(*pyramid*)
Sets the indenter as a revolution cone (False) or a pyramid (True). I the case of the pyramid, the half angle becomes the axis to face angle.

Parameters **pyramid** (*bool*) – True for pyramid, False for revolution (default).

set_rigid(*rigid*)
Sets the indenter to be rigid (True) or deformable (False).

Parameters **rigid** (*bool*) – True for rigid, False for deformable (default)

set_summit_position(*summit_position*)
Sets the position of the indenter.

Parameters **summit_position** (*tuple or list containing two floats.*) – position of the summit in a 2D space.

set_sweep_angle(*sweep_angle*)
Sets the sweep angle.

Parameters **sweep_angle** (*int > 1*) – sweep_angle

5.2.4 Indenter miscellaneous

abapy.indentation.**equivalent_half_angle**(*half_angle*, *sweep_angle*)
Returns the half angle (in degrees) of the equivalent cone in terms of cross area and volume of a pyramidal indenter. :param half_angle: indenter half angle in degrees :type half_angle: float :param sweep_angle: sweep angle in degrees :type sweep_angle: float :rtype: float

5.3 Simulation tools

5.3.1 Steps definition

class abapy.indentation.**Step**(*name, disp=1.0, nlgeom=True, nframes=100, fieldOutputFreq=999999, boundaries_3D=False, full_3D=False, rigid_indenter_3D=True, nodeFieldOutput=['COORD', 'U'], elemFieldOutput=['LE', 'EE', 'PE', 'PEEQ', 'S'], mode='bulk'*)

Builds a typical indentation step.

Parameters

- **name** (*string*) – step name.
- **disp** (*float > 0.*) – displacement.
- **nframes** (*int*) – frame number.
- **nlgeom** (*boolean*) – nlgeom state.
- **fieldOutputFreq** (*int*) – field output frequency
- **boundaries_3D** (*boolean*) – 3D or 2D boundary conditions. If 3D is True, then boundary conditions will be applied to the node sets `front` and `back`.
- **full_3D** (*boolean*) – set to True if the model is a complete 3D model without symmetries and then does not need side boundaries.
- **rigid_indenter_3D** (*boolean*) – Set to True if a 3D indenter is rigid
- **nodeFieldOutput** (*string or list of strings*) – node outputs.
- **elemFieldOutput** (*string or list of strings*) – node outputs.

Step.**set_name**(*name*)

Sets step name.

Parameters **name** (*string*) – step name.

Step.**set_displacement**(*disp*)

Sets the displacement.

Parameters **disp** (*float > 0.*) – displacement.

Step.**set_nframes**(*nframes*)

Sets the number of frames.

Parameters **nframes** (*int*) – frame number.

Step.**set_nlgeom**(*nlgeom*)

Sets NLGEOM on or off.

Parameters **nlgeom** (*boolean*) – nlgeom state.

Step.**set_fieldOutputFreq**(*freq*)

Sets the field output period.

Parameters **freq** (*int*) – field output frequency

Step.**set_nodeFieldOutput**(*nodeOutput*)

Sets the node field output to be recorded.

Parameters **nodeOutput** (*string or list of strings*) – node outputs.

Step.**set_elemFieldOutput**(*elemOutput*)
Sets the element field output to be recorded.

Parameters **elemOutput** (*string or list of strings*) – node outputs.

Step.**dump2inp**()
Dumps the step to Abaqus INP format.

5.3.2 Inp builder

abapy.indentation.**MakeInp**(*sample_mesh=None*, *indenter=None*, *sample_mat=None*, *indenter_mat=None*, *friction=0.0*, *steps=None*, *is_3D=False*, *heading='Abapy Indentation Simulation'*)
Builds a complete indentation INP for Abaqus and returns it as a string.

Parameters

- **sample_mesh** (abapy.mesh.Mesh instance or None) – mesh to use for the sample. If None, default ParamInfiniteMesh will be used.
- **indenter** (*any indenter instance or None*) – indenter to use. If None, default RigidCone2D will be used.
- **sample_mat** (*any material instance or None*) – sample material to use. If None, default abapy.materials.VonMises will be used.
- **indenter_mat** (*any material instance or None*) – indenter material to use. If None, a default elastic material will be used. If a rigid indenter is chosen, this material will not interfer with the simulation results.
- **friction** (*float >= 0.*) – friction coefficient between indenter and sample.
- **steps** (list of Step instances or None) – steps to used during the test.

Return type string

```
    #-----------------------
    # PACKAGES
    from abapy.indentation import MakeInp, DeformableCone2D, DeformableCone3D, IndentationMesh, Step
    from abapy.materials import Elastic, VonMises
    from math import radians, tan
    #-----------------------

    #-----------------------
    # FIRST EXEMPLE: AXISYMMETRIC INDENTATION
    # Model parameters
    half_angle = 70.29      # Indenter half angle, 70.3 degrees is equivalent to Berkovich indenter
    Na, Nb, Ns, Nf, l = 8, 8, 16, 2, 1.
    E, nu = 1., 0.3         # E is the Young's modulus and nu is the Poisson's ratio.
    ey = 0.01 * E
    max_disp = l/3.*tan(radians(70.3))/tan(radians(half_angle)) # Maximum displacement

    # Building model
    indenter = DeformableCone2D(half_angle = half_angle, rigid = True, Na = Na, Nb = Nb, Ns = Ns, Nf
    sample_mesh = IndentationMesh(Na = Na, Nb = Nb, Ns = Ns, Nf = Nf, l=l)# Sample Mesh
    #sample_mat = Elastic(labels = 'SAMPLE_MAT', E = E, nu=  nu)     # Sample material
    sample_mat = VonMises(labels = 'SAMPLE_MAT', E = E, nu=  nu, sy = E * ey)     # Sample material
    indenter_mat = Elastic(labels = 'INDENTER_MAT')     # Indenter material
    steps = [                                                   # Steps
      Step(name='preloading', nframes = 50, disp = max_disp / 2.),
```

```
    Step(name='loading',    nframes = 50, disp = max_disp ),
    Step(name='unloading',  nframes = 50, disp = 0.), ]

# Making INP file
inp = MakeInp(indenter = indenter,
sample_mesh = sample_mesh,
sample_mat = sample_mat,
indenter_mat = indenter_mat,
steps = steps)
f = open('workdir/indentation_axi.inp', 'w')
f.write(inp)
f.close()
#-----------------------

#-----------------------
# SECOND EXAMPLE: 3D BERKOVICH INDENTATION
# Model Parameters
half_angle = 65.27     # Indenter half angle, 65.27 leads to a modified Berkovich geometry, see
Na, Nb, Ns, Nf, l = 8, 8, 8, 2, 1. # with 4, 4, 4, 2, 1., simulation is very fast, the mesh is c
sweep_angle, N = 60., 8
E, nu = 1., 0.3         # E is the Young's modulus and nu is the Poisson's ratio.
ey = 0.01 # yield strain
max_disp = l/3.*tan(radians(70.3))/tan(radians(half_angle)) # Maximum displacement
pyramid = True

# Building model
indenter = DeformableCone3D(half_angle = half_angle, rigid = True, Na = Na, Nb = Nb, Ns = Ns, Nf
sample_mesh = IndentationMesh(Na = Na, Nb = Nb, Ns = Ns, Nf = Nf, l=l).sweep(sweep_angle = sweep
#sample_mat = Elastic(labels = 'SAMPLE_MAT', E = E, nu=  nu)     # Sample material
sample_mat = VonMises(labels = 'SAMPLE_MAT', E = E, nu=  nu, sy = E * ey)     # Sample material
indenter_mat = Elastic(labels = 'INDENTER_MAT')     # Indenter material
steps = [                                                    # Steps
  Step(name='preloading',   nframes = 100, disp = max_disp / 2., boundaries_3D=True),
  Step(name='loading',      nframes = 100, disp = max_disp,      boundaries_3D=True),
  Step(name='unloading',    nframes = 200, disp = 0.,            boundaries_3D=True)]

# Making INP file.
inp = MakeInp(indenter = indenter,
sample_mesh = sample_mesh,
sample_mat = sample_mat,
indenter_mat = indenter_mat,
steps = steps, is_3D = True)
f = open('workdir/indentation_berko.inp', 'w')
f.write(inp)
f.close()
#-----------------------

# Simulation can be launched using abaqus job=indentation and can be roughly post processed usin
```

Returns:

`indentation_axi.inp`

`indentation_berko.inp`

5.3.3 Simulation manager

class abapy.indentation.**Manager**(*workdir=''*, *abqlauncher='abaqus'*, *samplemesh=None*, *indenter=None*, *samplemat=None*, *indentermat=None*, *friction=0.0*, *steps=None*, *is_3D=False*, *simname='indentation'*, *files2delete=['sta', 'sim', 'prt', 'odb', 'msg', 'log', 'dat', 'com', 'inp', 'lck', 'pckl'], abqpostproc='abqpostproc.py'*)

The spirit of the simulation manager is to allow you to work at a higher level. Using it allows you to define once and for all where you work, where Abaqus is, it fill manage subprocesses (Abaqus, Abaqus python) automatically. It is particularly interresting to perform parametric simulation because you can modify one parameter (material property, inenter property, ...) and keep all other parameters fixed and rerun directly the simulation process without making any mistake.

Note: This class is still under developpement, important changes may then happen.

Parameters

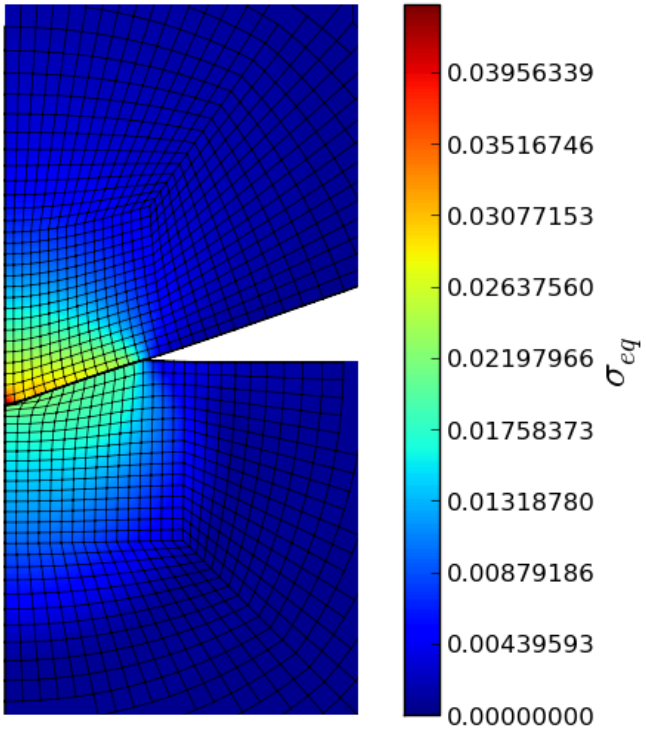
- **workdir** (*string*) – work directory where simulation is to be run.
- **abqlauncher** (*string*) – abaqus launcher or path to it. Take care about aliases under linux because they often don't work under non interactive shells. A direct path to the launcher may be a good idea.
- **samplemesh** (abapy.mesh.Mesh instance or None) – mesh to be used by MakeInp, None will let MakeInp use it's own default.
- **indenter** (*indenter instance or None*) – indenter to be used by MakeInp, None will let MakeInp use it's own default.
- **samplemat** (*material instance or None*) – sample material to be used by MakeInp, None will let MakeInp use it's own default.
- **indentermat** (*material instance or None*) – indenter material to be used by MakeInp, None will let MakeInp use it's own default.
- **steps** (list of Step instances.) – steps to use during the test.
- **is_3D** (*Bool*) – has to be True if the simulation is 3D, else must be False.
- **simname** (*string*) – simulation name used for all files.
- **files2delete** (*list of strings.*) – file types to delete when cleaning work directory.

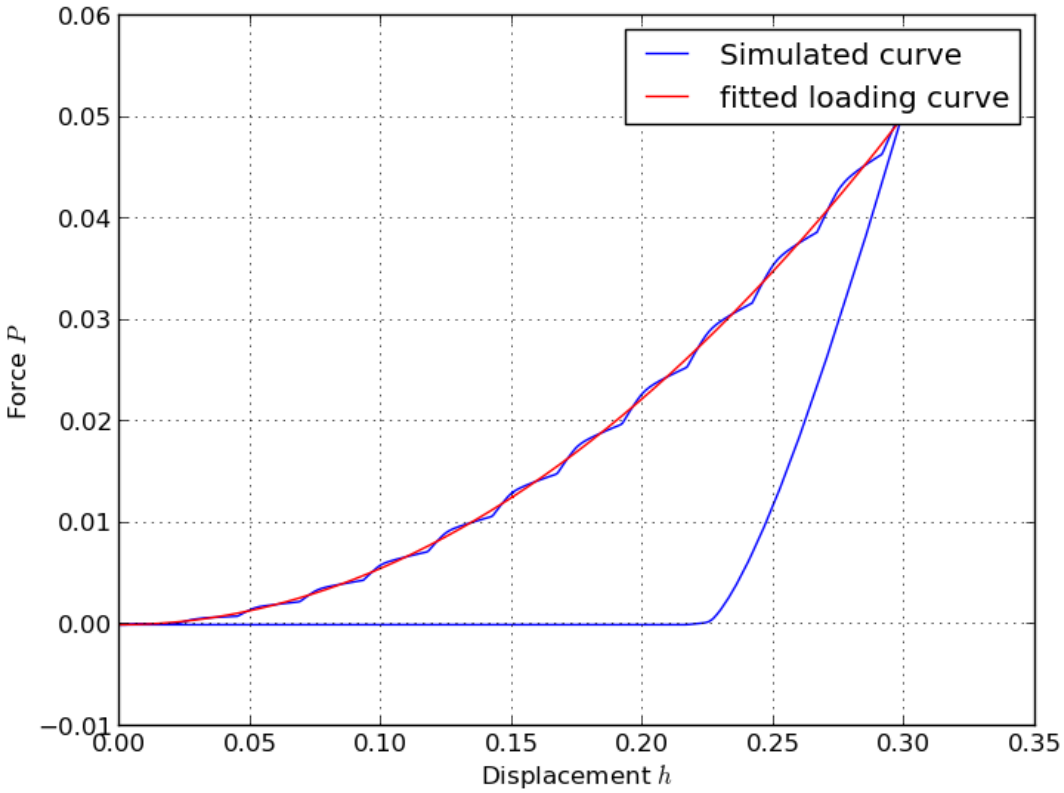
```
from abapy.indentation import Manager, IndentationMesh, Step, RigidCone2D, DeformableCone2D
from abapy.materials import VonMises, Elastic
from math import radians, tan
import numpy as np
import matplotlib.pyplot as plt
#---------------------------------------
# Python post processing function:
# Role: data extraction from odb is performed in Abaqus python but nothing more. A regular Pytho
def pypostproc(data):
  if data['completed']:
    return data
  else:
    print '<Warning: Simulation aborted, check .msg file for explanations.>'
    return data
```

```
#---------------------------------------
# Defining test parameters:
Na, Nb, Ns, Nf = 16,16, 16, 2
half_angle = 70.29
rigid_indenter = False # Sets the indenter rigid of deformable
mesh = IndentationMesh(Na = Na, Nb = Nb, Ns = Ns, Nf = Nf)                # Chosing sample mesh
#indenter = RigidCone2D(half_angle = 70.3)                 # Chosing indenter
indenter = DeformableCone2D(half_angle = half_angle, Na = Na, Nb = Nb, Ns=Ns, Nf=Nf, rigid = rig
E = 1.                                                    # Young's modulus
sy = E * .01                                              # Yield stress
samplemat = VonMises(labels = 'SAMPLE_MAT', E = E, sy = sy)    # Sample material
indentermat = Elastic(labels = 'INDENTER_MAT', E = E)
max_disp = .3 * tan(radians(70.3))/tan(radians(half_angle))
nframes = 200
steps = [                                                 # Steps
  Step(name='loading0', nframes = nframes, disp = max_disp/2.),
  Step(name='loading1', nframes = nframes, disp = max_disp),
  Step(name = 'unloading', nframes = nframes, disp = 0.)]
#---------------------------------------
# Directories: absolute pathes seems more secure to me since we are running some 'rm'.
workdir = 'workdir/'
abqlauncher = '/opt/Abaqus/6.9/Commands/abaqus'
simname = 'indentation'
abqpostproc = 'abqpostproc.py'
#---------------------------------------
# Setting simulation manager
m = Manager()
m.set_abqlauncher(abqlauncher)
m.set_workdir(workdir)
m.set_simname(simname)
m.set_abqpostproc(abqpostproc)
m.set_samplemesh(mesh)
m.set_samplemat(samplemat)
m.set_indentermat(indentermat)
m.set_steps(steps)
m.set_indenter(indenter)
m.set_pypostprocfunc(pypostproc)
#---------------------------------------
# Running simulation and post processing
#m.erase_files() # Workdir cleaning
m.make_inp() # INP creation
#m.run_sim() # Running the simulation
#m.run_abqpostproc() # First round of post processing in Abaqus
data = m.run_pypostproc() # Second round of custom post processing in regular Python

#---------------------------------------

if data['completed']:
  # Ploting results
  step2plot = 0
  Nlevels = 200
  plt.figure(0)
  # Ploting load vs. disp curve
  plt.clf()
  ho = data['history']
  F = -ho['force']
  h = -ho['disp']
  C = (F[1]/h[1]**2).average()
```

```
    F_fit = C * h **2
    plt.plot(h.plotable()[1], F.plotable()[1], 'b-',label = 'Simulated curve', linewidth = 1.)
    plt.plot(h[0,1].plotable()[1], F_fit[0,1].plotable()[1],'r-', label = 'fitted loading curve',
    plt.xlabel('Displacement $h$')
    plt.ylabel('Force $P$')
    plt.legend()
    plt.grid()
    plt.savefig(workdir + simname + '_load-disp.png')
    # Ploting deformed shape

    plt.clf()
    plt.gca().set_aspect('equal')
    plt.axis('off')
    fo = data['field']
    #stress = fo['S'][step2plot].vonmises()
    stress = fo['S'][step2plot].pressure()
    if 'Sind' in fo.keys():
      #ind_stress = fo['Sind'][step2plot].vonmises()
      ind_stress = fo['Sind'][step2plot].pressure()
    smax= max( max(stress.data), max(ind_stress.data))
    smin= min( min(stress.data), min(ind_stress.data))
    #levels = [(n+1)/float(Nlevels)*smax for n in xrange(Nlevels)]
    levels = np.linspace(smin, smax, Nlevels)
    field_flag = r'$\sigma_{eq}$'
    disp = fo['U'][step2plot]
    ind_disp = fo['Uind'][step2plot]
    indenter.apply_displacement(ind_disp) # Applies the displacement to the indenter.

    #plt.plot(xbi,ybi,'k-')
    mesh.nodes.apply_displacement(disp) # This Nodes class method allows to deform a Nodes instanc
    xlim, ylim, zlim = mesh.nodes.boundingBox() # This little method allows nicer ploting producin
    xmin, xmax = 0., 2.
    ymin, ymax = -2., 2.
    plt.xlim([xmin, xmax])
    plt.ylim([ymin, ymax])
    x, y, z, tri = mesh.dump2triplot() # This method translates the whole mesh in matplotlib.pyplo
    xi, yi, zi, trii = indenter.mesh.dump2triplot()
    xe, ye, ze = mesh.get_edges(xmin = xmin, xmax = xmax, ymin = ymin, ymax = ymax)
    xb, yb, zb = mesh.get_border(xmin = xmin, xmax = xmax, ymin = ymin, ymax = ymax)
    xei, yei, zei = indenter.mesh.get_edges(xmin = xmin, xmax = xmax, ymin = ymin, ymax = ymax) #
    xbi, ybi, zbi = indenter.mesh.get_border(xmin = xmin, xmax = xmax, ymin = ymin, ymax = ymax) #
    plt.plot(xe,ye,'-k', linewidth = 0.5) # Mesh ploting.
    plt.plot(xb,yb,'-k', linewidth = 1.) # Sample border ploting.
    plt.plot(xei,yei,'-k', linewidth = 0.5) # Mesh ploting.
    plt.plot(xbi,ybi,'-k', linewidth = 1.) # Sample border ploting.
    grad = plt.tricontourf(x, y, tri, stress.data, levels) # Gradiant plot, Nlevels specifies the
    plt.tricontourf(xi,yi,trii, ind_stress.data, levels)
    #plt.tricontour(xi,yi,trii, ind_stress.data, levels,  colors = 'black')
    cbar = plt.colorbar(grad)
    cbar.ax.set_ylabel(field_flag, fontsize=20)
    #plt.tricontour(x, y, tri, stress.data, levels,  colors = 'black') # Isovalue plot which make
    #plt.show()
    plt.savefig(workdir + simname + '_field.png')
```

Gives:


0.03956339
0.03516746
0.03077153
0.02637560
0.02197966
0.01758373
0.01318780
0.00879186
0.00439593
0.00000000
σ_{eq}



Note: In order to used abaqus Python, you have to build a post processing script that is executed in `abaqus python`. Here is an example `abqpostproc.py`:

```
# ABQPOSTPROC.PY
# Warning: executable only in abaqus abaqus viewer -noGUI,... not regular python.
import sys
from abapy.postproc import GetFieldOutput_byRpt as gfo
from abapy.postproc import GetVectorFieldOutput_byRpt as gvfo
from abapy.postproc import GetTensorFieldOutput_byRpt as gtfo
from abapy.postproc import GetHistoryOutputByKey as gho
from abapy.indentation import Get_ContactData
from abapy.misc import dump
from odbAccess import openOdb
from abaqusConstants import JOB_STATUS_COMPLETED_SUCCESSFULLY



# Odb opening
file_name = 'indentation'
odb = openOdb(file_name + '.odb')
data = {}

# Check job status:
job_status = odb.diagnosticData.jobStatus

if job_status == JOB_STATUS_COMPLETED_SUCCESSFULLY:
```

```
        data['completed'] = True
        # Field Outputs
        data['field'] = {}
        fo = data['field']
        fo['Uind'] = [
          gvfo(odb = odb,
            instance = 'I_INDENTER',
            step = 1,
            frame = -1,
            original_position = 'NODAL',
            new_position = 'NODAL',
            position = 'node',
            field = 'U',
            delete_report = True),
          gvfo(odb = odb,
            instance = 'I_INDENTER',
            step = 2,
            frame = -1,
            original_position = 'NODAL',
            new_position = 'NODAL',
            position = 'node',
            field = 'U',
            delete_report = True)]

        fo['U'] = [
          gvfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 1,
            frame = -1,
            original_position = 'NODAL',
            new_position = 'NODAL',
            position = 'node',
            field = 'U',
            sub_set_type = 'element',
            delete_report = True),
          gvfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 2,
            frame = -1,
            original_position = 'NODAL',
            new_position = 'NODAL',
            position = 'node',
            field = 'U',
            sub_set_type = 'element',
            delete_report = True)]

        fo['S'] = [
          gtfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 1,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'S',
            sub_set_type = 'element',
            delete_report = True),
          gtfo(odb = odb,
```

```
          instance = 'I_SAMPLE',
          step = 2,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
          new_position = 'NODAL',
          position = 'node',
          field = 'S',
          delete_report = True)]

      fo['Sind'] = [
        gtfo(odb = odb,
          instance = 'I_INDENTER',
          step = 1,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
          new_position = 'NODAL',
          position = 'node',
          field = 'S',
          sub_set_type = 'element',
          delete_report = True),
        gtfo(odb = odb,
          instance = 'I_SAMPLE',
          step = 2,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
          new_position = 'NODAL',
          position = 'node',
          field = 'S',
          delete_report = True)]

      fo['LE'] = [
        gtfo(odb = odb,
          instance = 'I_SAMPLE',
          step = 1,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
          new_position = 'NODAL',
          position = 'node',
          field = 'LE',
          sub_set_type = 'element',
          delete_report = True),
        gtfo(odb = odb,
          instance = 'I_SAMPLE',
          step = 2,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
          new_position = 'NODAL',
          position = 'node',
          field = 'LE',
          sub_set_type = 'element',
          delete_report = True)]

      fo['EE'] = [
        gtfo(odb = odb,
          instance = 'I_SAMPLE',
          step = 1,
          frame = -1,
          original_position = 'INTEGRATION_POINT',
```

```
            new_position = 'NODAL',
            position = 'node',
            field = 'EE',
            sub_set_type = 'element',
            delete_report = True),
          gtfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 2,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'EE',
            sub_set_type = 'element',
            delete_report = True)]

        fo['PE'] = [
          gtfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 1,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'PE',
            sub_set_type = 'element',
            delete_report = True),
          gtfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 2,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'PE',
            sub_set_type = 'element',
            delete_report = True)]

        fo['PEEQ'] = [
          gfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 1,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'PEEQ',
            sub_set_type = 'element',
            delete_report = True),
          gfo(odb = odb,
            instance = 'I_SAMPLE',
            step = 2,
            frame = -1,
            original_position = 'INTEGRATION_POINT',
            new_position = 'NODAL',
            position = 'node',
            field = 'PEEQ',
            sub_set_type = 'element',
```

```
            delete_report = True)]
        # History Outputs
        data['history'] = {}
        ho = data['history']
        ref_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['REF_NODE'].nodes[0].label
        ho['force'] =  gho(odb,'RF2')['Node I_INDENTER.'+str(ref_node)] # GetFieldOutputByKey returns
        ho['disp'] =   gho(odb,'U2')['Node I_INDENTER.'+str(ref_node)]
        tip_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['TIP_NODE'].nodes[0].label
        ho['tip_penetration'] =   gho(odb,'U2')['Node I_INDENTER.'+str(tip_node)]
        ho['allse'] =   gho(odb,'ALLSE').values()[0]
        ho['allpd'] =   gho(odb,'ALLPD').values()[0]
        ho['allfd'] =   gho(odb,'ALLFD').values()[0]
        ho['allwk'] =   gho(odb,'ALLWK').values()[0]
        #ho['carea'] =  gho(odb,'CAREA    ASSEMBLY_I_SAMPLE_SURFACE_FACES/ASSEMBLY_I_INDENTER_SURFACE_

        # CONTACT DATA PROCESSING
        ho['contact'] = Get_ContactData(odb = odb, instance = 'I_SAMPLE', node_set = 'TOP_NODES')

      else:
        data['completed'] = False
      # Closing and dumping
      odb.close()
      dump(data, file_name+'.pckl')
```

Settings

Manager.**set_simname**(*simname*)
Sets simname.

Parameters **simname** (*string*) – simulation name that is used to name simulation files.

Manager.**set_workdir**(*workdir*)
Sets work directory

Parameters **workdir** (*string*) – relative or absolute path to workdir where simulations are run.

Manager.**set_abqlauncher**(*abqlauncher*)
Sets Abaqus launcher :param abqlauncher: alias, relative path or absolute path to abaqus launcher. :type abqlaucher: string

Manager.**set_samplemesh**(*samplemesh*)
Sets sample mesh.

Parameters **samplemesh** (abapy.mesh.Mesh instance) – sample mesh.

Manager.**set_indenter**(*indenter*)
Sets indenter.

Parameters **indenter** (*instance of any indenter class*) – indenter to be used.

Manager.**set_samplemat**(*samplemat*)
Sets sample material.

Parameters **samplemat** (*instance of any material class*) – core material.

Manager.**set_steps**(*steps*)
Sets steps

Parameters **steps** (list of Steps instances) – description of steps.

Manager.**set_files2delete**(*files2delete*)
Sets files to delete when cleaning.

Parameters **files2delete** (*list of strings.*) – files types to be deleted when cleaning workdir.

Manager.**set_abqpostproc**(*abqpostproc*)
Sets the path to the abaqus post-processing python script, this path must be absolute or relative to workdir.

Parameters **abqpostproc** (*string*) – link to the abaqus post processing script.

Manager.**set_pypostprocfunc**(*func*)
Sets the Python post processing function.

Parameters **func** (*function*) – post processing function of the data produced by abaqus post processing.

Launchers

Manager.**erase_files**()
Erases all files with types declared in files2delete in the work directory with the name *simname*.

Manager.**make_inp**()
Builds the INP file using MakeInp and stores it in workdir as "simname.inp".

Manager.**run_sim**()
Runs the simulation.

Manager.**run_abqpostproc**()
Runs the first pass of post processing inside Abaqus Python.

Manager.**run_pypostproc**()
Runs the Python post processing function.

Return type data returned by pypostprocfunc

5.3.4 Indentation post-processing

Contact Data

class abapy.indentation.**ContactData**(*repeat=3*, *is_3D=False*, *dtf='f'*)

ContactData class aims to store and proceed all contact related data:

•Position of nodes involved in the contact relationship.

•Contact pressure on these nodes.

This class can be used to perform various tasks:

•Find contact shape.

•Compute contact area.

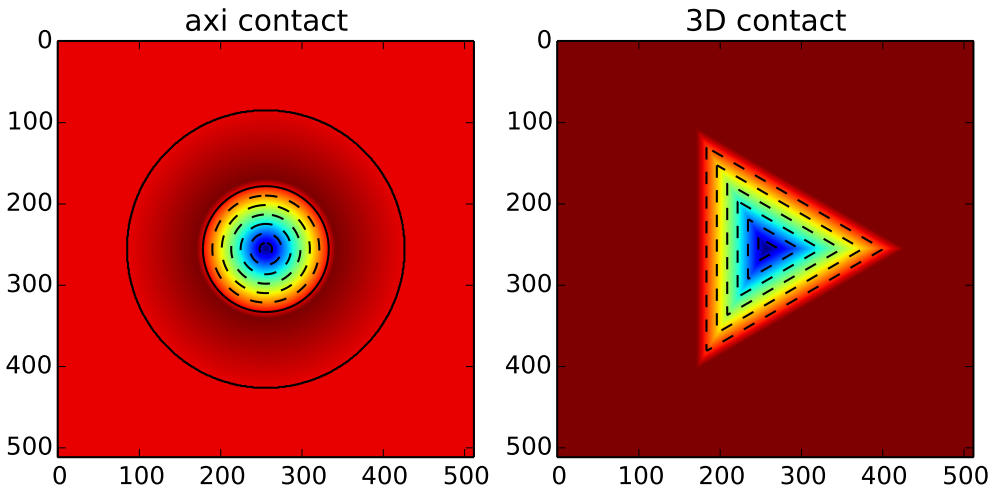
•Find contact contour.

•Produce matrix (AFM-like) images using the SPYM module.

Parameters

- **repeat** (*int > 0*) – this parameter is only used in the case of 3D contact. Due to symmetries, only a portion of the problem is computed. To get back to complete problem, a symmetry has to be performed and then a given number of copies of the result, this number is repeat. For example, to simulate a Vickers 4 sided indenter indentation, you will compute only 1 / 8 of the problem. So after a symmetry, you will need 4 copies of the result, the repeat = 4. To summarize, repeat is the number of faces of the indenter.
- **is_3D** (*bool*) – True for 3D contact, False for axisymmetric contact.
- **dtf** (*string*) – `array.array` data type for floats, 'f' for 32 bit float, 'd' for 64 float.

```
import numpy as np
from abapy.indentation import ContactData

X = np.linspace(-3., 3., 512)
Y = np.linspace(-3., 3., 512)
X, Y = np.meshgrid(X, Y)
# Axi
cd = ContactData()
x = [0, 1, 2, 10]
alt = [-1,.1, 0, 0]
press = [1, 0, 0, 0]
cd.add_data(x, altitude = alt, pressure = press)
Alt_axi, Press_axi = cd.interpolate(X, Y, method ='linear')
area = cd.contact_area()


# 3D
cd = ContactData(repeat = 3, is_3D = True)
k = np.cos(np.radians(60))
p = np.sin(np.radians(60))
x = [0, 4, 10, k*4, k*10]
y = [0, 0, 0,  p*4, p*10]
alt = [-1, 0, 0, 0, 0]
cd.add_data(x, altitude = alt)
Alt_3D, Press_3D = cd.interpolate(X, Y, method ='linear')


from matplotlib import pyplot as plt

fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(121)
grad = plt.imshow(Alt_axi)
plt.contour(Alt_axi, colors= 'black')
plt.title('axi contact')
ax1 = fig.add_subplot(122)
grad = plt.imshow(Alt_3D)
plt.contour(Alt_3D, colors= 'black')
plt.title('3D contact')
plt.show()
```



add_data (*coor1*, *coor2=0.0*, *altitude=0.0*, *pressure=0.0*)
Adds data to a ContactData instance.

Parameters

- **coor1** (*float or list like*) – radial position in the axisymmetric case or in plane position first coordinate in the 3D case.
- **coor2** (*float of list like*) – orthoradial position in the axisymmetric case of second in plane coordinate in the 3D case.
- **altitude** (*float or list like*) – out of plane position.
- **pressure** (*float or list like*) – normal contact pressure.

contact_area (*delaunay_disp=None*)
Returns the cross area of contact using the contact pressure field. The contact area is computed using a Delaunay triangulation of the available data points. This triangulation can be oriented using the delaunay_disp option (use very carefuly).

contact_contour (*delaunay_disp=None*)
Returns the contour of the contact zone.

export2spym (*lx*, *ly*, *xc=0.0*, *yc=0.0*, *nx=256*, *ny=256*, *xy_unit='m'*, *alt_unit='m'*, *press_unit='Pa'*, *axi_repeat=100*, *delaunay_disp=None*, *method='linear'*)
Exports data to `spym.generic.Spm_image` format.

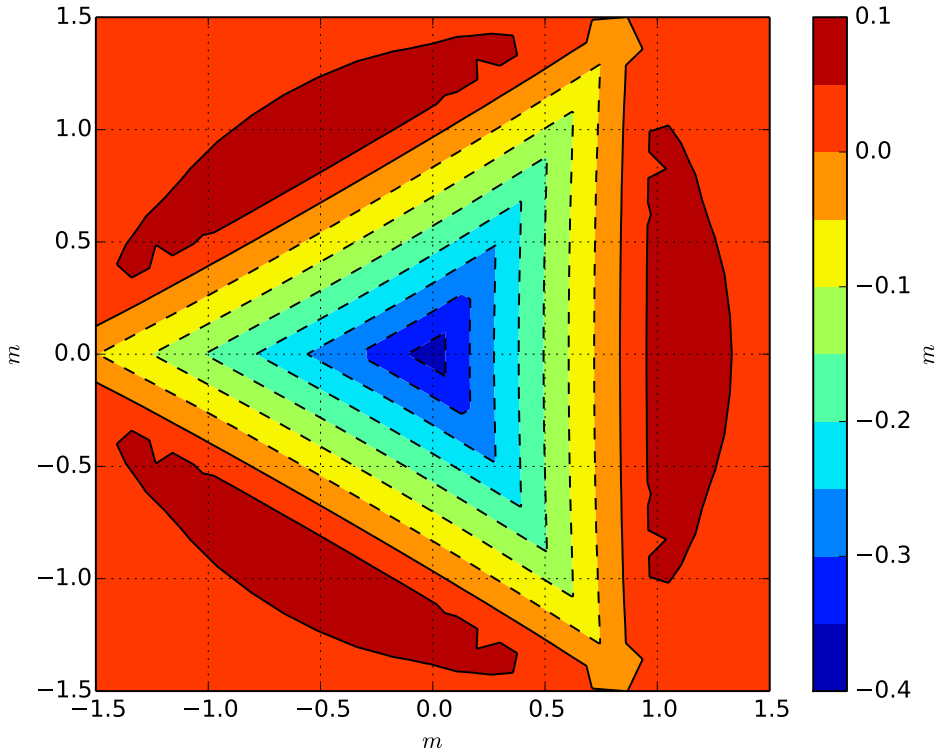
Parameters

- **lx** (*float*) – length on x axis

- **ly** (*float*) – length on y axis
- **xc** (*float*) – position of the center of the image on the x axis
- **yc** (*float*) – position of the center of the image on the y axis
- **nx** (*uint*) – x resolution
- **ny** (*uint*) – y resolution
- **xy_unit** (*str*) – xy unit
- **alt_unit** (*str*) – altitude unit
- **press_unit** (*str*) – contact pressure unit

See get_3D_data for other params.

```
from abapy.misc import load
import numpy as np
import matplotlib.pyplot as plt

# In this case, a 3D FEM simulation has beed performed and the results are stored in the fil


out = load('ContactData_berk.pckl')
cdl = out[1][-1] # loading
cdu = out[2][-1] # unloading
hmax = -cdl.min_altitude()
l = 7. * hmax
alt, press = cdu.export2spym(lx = l, ly = l, nx = 512, ny = 512)
alt.dump2gsf('image.gsf')
xlabel, ylabel, zlabel = alt.get_labels()
X,Y,Z = alt.get_xyz()
plt.figure()
plt.clf()
plt.gca().set_aspect('equal')
plt.grid()
plt.contourf(X, Y, Z, 10)
cbar = plt.colorbar()
cbar.set_label(zlabel)
plt.contour(X, Y, Z, 10, colors = 'black')
plt.xlabel(xlabel)
plt.ylabel(ylabel)
plt.show()
```



This script also produces a GSF image file, readable by both `spym` and Gwyddion: `image.gsf`

get_3D_data(*axi_repeat=100*, *delaunay_disp=None*, *crit_dist=1e-05*)

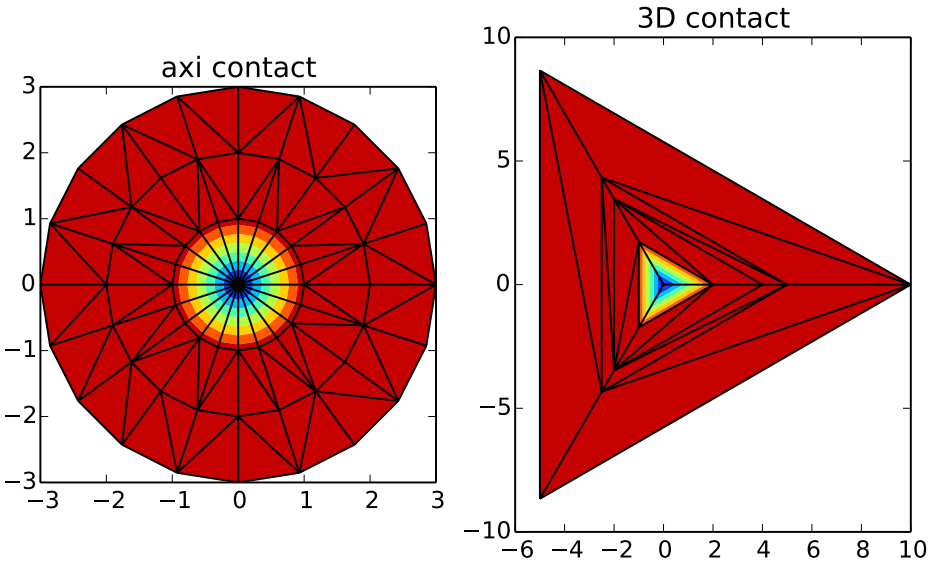
Returns full 3D data usable for interpolation or for ploting. This method performs all the copy and paste needed to get the complete contact (due to symmetries) and also producec a triangular mesh of the surface using the Delaunay algorithm (via `scipy`).

Parameters **axi_repeat** (*int > 0*) – number of times axisymmetric profile has to be pasted orthoradially.

Return type 3 arrays points, alt, press and conn

```
from abapy.indentation import ContactData
from matplotlib import pyplot as plt


# Axi contact
cd = ContactData()
x = [0, 1, 2, 3]
alt = [-1,.1, 0, 0]
press = [1, 0, 0, 0]
cd.add_data(x, altitude = alt, pressure = press)
points_axi, alt_axi, press_axi, conn_axi = cd.get_3D_data(axi_repeat = 20)



# 3D contact
cd = ContactData(repeat = 3, is_3D = True)
```

```
k = np.cos(np.radians(60))
p = np.sin(np.radians(60))
x = [0, 4, 10, k*4, k*10]
y = [0, 0, 0,  p*4, p*10]
alt = [-1, 0, 0, 0, 0]
cd.add_data(x, altitude = alt)
points_3D, alt_3D, press_3D, conn_3D = cd.get_3D_data()



fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(121)
ax1.set_aspect('equal')
plt.tricontourf(points_axi[:,0], points_axi[:,1], conn_axi, alt_axi)
plt.triplot(points_axi[:,0], points_axi[:,1], conn_axi)

plt.title('axi contact')
ax1 = fig.add_subplot(122)
ax1.set_aspect('equal')
plt.tricontourf(points_3D[:,0], points_3D[:,1], conn_3D, alt_3D)
plt.triplot(points_3D[:,0], points_3D[:,1], conn_3D)
plt.title('3D contact')
plt.show()
```



interpolate(*coor1*, *coor2*, *axi_repeat=100*, *delaunay_disp=None*, *method='linear'*)

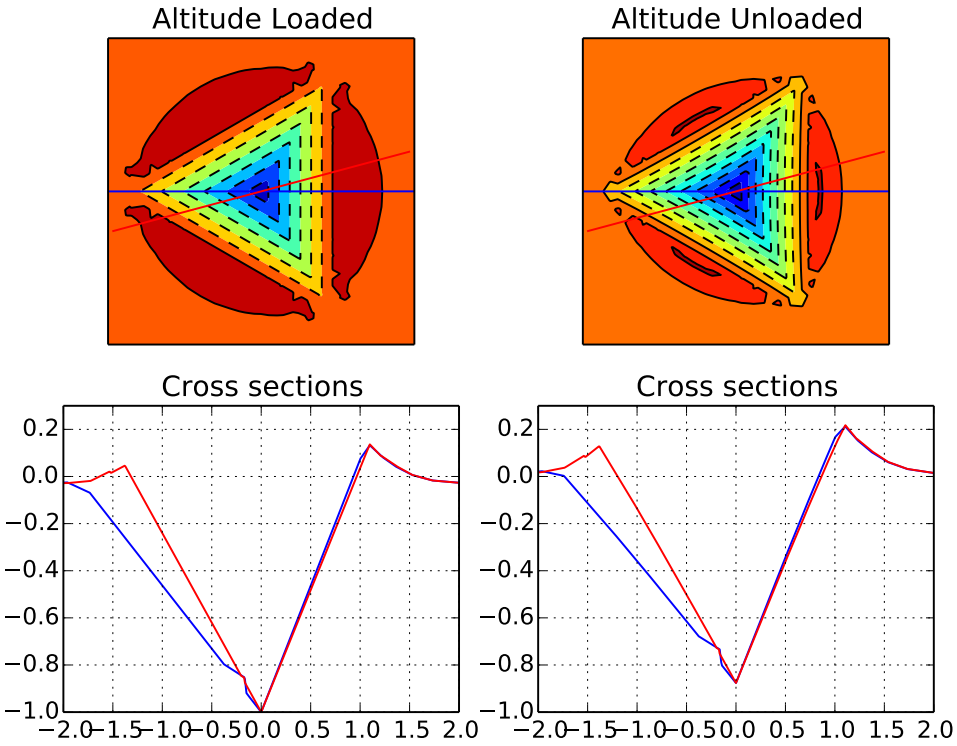
Allows general interpolation on the a contact data instance.

Parameters

- **coor1** (*any list/array of floats*) – radial position in the axisymmetric case or in plane position first coordinate in the 3D case.
- **coor2** (*any list/array of floats*) – orthoradial position in the axisymmetric case of second in plane coordinate in the 3D case.
- **axi_repeat** (*int > 0*) – number of times axisymmetric profile has to be pasted orthoradially.

```
from abapy.misc import load
import numpy as np
from matplotlib import pyplot as plt


# In this case, a 3D FEM simulation has beed performed and the results are stored in the fil


out = load('ContactData_berk.pckl')
cd0 = out[1][-1] # First step data: loading
cd1 = out[2][-1] # Second step data: unloading
hmax = -cd0.min_altitude()

# First let's map altitude and pressure on cartesian grids.
x = np.linspace(-2., 2., 256)
X, Y = np.meshgrid(x, x)

Alt0, Press0 = cd0.interpolate(X, Y, method ='linear')
Alt1, Press1 = cd1.interpolate(X, Y, method ='linear')
Alt0 = Alt0 / hmax
Alt1 = Alt1 / hmax

# Now we wan to get some sections of the imprint
s = np.linspace(0., 2., 256)
s = np.append((-s)[::-1], s)
theta0 = np.radians(0.01)
theta1 = np.radians(15.)
xs0 = np.cos(theta0) * s
ys0 = np.sin(theta0) * s
xs1 = np.cos(theta1) * s
ys1 = np.sin(theta1) * s
# Sections under full load
Alt0_sec_l, Press0_sec_l = cd0.interpolate(xs0, ys0, method ='linear')
Alt1_sec_l, Press1_sec_l = cd0.interpolate(xs1, ys1, method ='linear')
Alt0_sec_l = Alt0_sec_l / hmax
Alt1_sec_l = Alt1_sec_l / hmax
# Sections after unloading
Alt0_sec_u, Press0_sec_u = cd1.interpolate(xs0, ys0, method ='linear')
Alt1_sec_u, Press1_sec_u = cd1.interpolate(xs1, ys1, method ='linear')
Alt0_sec_u = Alt0_sec_u / hmax
Alt1_sec_u = Alt1_sec_u / hmax




fig = plt.figure()
plt.clf()
ax1 = fig.add_subplot(221)
ax1.set_xticks([])
```

```
        ax1.set_yticks([])
        grad = plt.contourf(X, Y, Alt0, 10)
        plt.contour(X, Y, Alt0, 10, colors = 'black')
        plt.grid()
        plt.plot(xs0, ys0, 'b-')
        plt.plot(xs1, ys1, 'r-')
        ax1.set_aspect('equal')
        plt.title('Altitude Loaded')
        ax2 = fig.add_subplot(222)
        ax2.set_xticks([])
        ax2.set_yticks([])
        grad = plt.contourf(X, Y, Alt1, 10)
        plt.contour(X, Y, Alt1, 10, colors = 'black')
        plt.grid()
        plt.plot(xs0, ys0, 'b-')
        plt.plot(xs1, ys1, 'r-')
        ax2.set_aspect('equal')
        plt.title('Altitude Unloaded')
        ax3 = fig.add_subplot(223)
        ax3.set_ylim([-1,0.3])
        plt.plot(s, Alt0_sec_l, 'b-')
        plt.plot(s, Alt1_sec_l, 'r-')
        plt.title('Cross sections')
        plt.grid()
        ax4 = fig.add_subplot(224)
        ax4.set_ylim([-1,0.3])
        plt.plot(s, Alt0_sec_u, 'b-')
        plt.plot(s, Alt1_sec_u, 'r-')
        plt.title('Cross sections')
        plt.grid()
        plt.show()
```



max_altitude()
Returns the maximum altitude.

max_pressure()
Returns the maximum pressure.

min_altitude()
Returns the minimum altitude.

min_pressure()
Returns the minimum pressure.

Get Contact Data

abapy.indentation.**Get_ContactData**(*odb*, *instance*, *node_set*)
Finds and reformulate contact data on a given node set and a give instance. This function aims to read in Abaqus odb files and is then only usable in `abaqus python` and `abaqus viewer -noGUI`. Following conventions are used:

- The normal to the initial surface must be the y axis.
- In axisymmetrical simulations, this is nearly automatic. In 3D simulations, the initial plane surface must be in parallel to the (x,z) plane.
- As a consequence, coor1 will be x, coor2 will be z and the altitude is y.

Parameters

- **odb** (odb instance obtained using `odbAccess.openOdb`.) – the odb instance where needed data is.

- **instance** (*string*) – name of an instance in contact.
- **node_set** (*string*) – name of a node set belonging to the instance.

5.4 Elasticity

5.4.1 Hertz

class abapy.indentation.**Hertz**(*F=1.0*, *a=None*, *h=None*, *R=1.0*, *E=1.0*, *nu=0.3*)

Hertz spherical indentation model.

```
from abapy.indentation import Hertz
from abapy.mesh import Mesh, Nodes, RegularQuadMesh
from matplotlib import pyplot as plt
import numpy as np

"""
===========
Hertz model
===========
"""

H = Hertz(F = 1., E=1., nu = 0.1)
Ne = 50

mesh = RegularQuadMesh(N1 = Ne, N2 = Ne, l1 = H.a * 2., l2 = H.a * 2., dtf = 'd')
mesh.nodes.translate(H.a/20., H.a/20.)
S = mesh.nodes.eval_tensorFunction(H.sigma)
R,Z,T,tri = mesh.dump2triplot()
R, Z = np.array(R), np.array(Z)
# Some fields
srr = S.get_component(11)
szz = S.get_component(22)
stt = S.get_component(33)
srz = S.get_component(12)
smises = S.vonmises()
s1, s2, s3, v1, v2, v3 = S.eigen() # Eigenvalues and eigenvectors
data = smises.data

N = 20
levels = np.linspace(0., max(data), N)
a = H.a
plt.figure()
plt.tricontourf(R/a, Z/a, tri, data, levels)
plt.colorbar()
plt.tricontour(R/a, Z/a, tri, data, levels, colors = 'black')
plt.xlabel('$r/a$', fontsize = 14.)
plt.ylabel('$z/a$', fontsize = 14.)
#plt.quiver(R, Z, v1.data1, v1.data2)
plt.grid()
plt.show()
```

Eeq

Eeq: equivalent modulus (GPa)

get_Eeq()

Eeq: equivalent modulus (GPa)

sigma(*r*, *z*, *t=0.0*, *labels=None*)
To do.

5.4.2 Hanson

class abapy.indentation.**Hanson**(*F=None*, *a=None*, *h=None*, *half_angle=70.29*, *E=1.0*, *nu=0.3*)
Hanson conical indentation model.

```
from abapy.indentation import Hanson
from abapy.mesh import Mesh, Nodes, RegularQuadMesh
from matplotlib import pyplot as plt
import numpy as np

"""
===========
Hanson model for conical indentation
===========
"""

H = Hanson(F = 1., E=1., nu = 0.3, half_angle = 70.29)
Ne = 20

mesh = RegularQuadMesh(N1 = Ne, N2 = Ne, l1 = H.a * 2., l2 = H.a * 2., dtf = 'd')
mesh.nodes.translate(H.a/20., H.a/20.)
S = mesh.nodes.eval_tensorFunction(H.sigma)
R,Z,T,tri = mesh.dump2triplot()
R, Z = np.array(R), np.array(Z)
# Some fields
srr = S.get_component(11)
szz = S.get_component(22)
stt = S.get_component(33)
srz = S.get_component(12)
smises = S.vonmises()
s1, s2, s3, v1, v2, v3 = S.eigen() # Eigenvalues and eigenvectors
data = smises.data

N = 20
levels = np.linspace(0., max(data), N)
a = H.a
plt.figure()
plt.tricontourf(R/a, Z/a, tri, data, levels)
plt.colorbar()
plt.tricontour(R/a, Z/a, tri, data, levels, colors = 'black')
plt.xlabel('$r/a$', fontsize = 14.)
plt.ylabel('$z/a$', fontsize = 14.)
#plt.quiver(R, Z, v1.data1, v1.data2)
plt.grid()
plt.show()
```

Eeq
Eeq: equivalent modulus (GPa)

get_Eeq()
Eeq: equivalent modulus (GPa)

sigma(*r*, *z*, *t=0.0*, *labels=None*)
To do.

CHAPTER 6

Miscellaneous

abapy.misc.**dump**(*data*, *name*, *protocol=2*)

Dumps an object to file using `pickle`.

Parameters

- **data** (*any*) – object to dump.
- **name** (*string*) – file name or path to file.

Note: This function allows clean array pickling whereas standard `pickle.dump` will raise an error if `array.array` are in the pickled object (which is the case of all objects in Abapy).

abapy.misc.**load**(*name*)

Loads back a pickled object.

Parameters **name** (*string*) – file name or path to file.

Return type unpickled object

Note: This function allows clean array unpickling whereas standard `pickle.load` will raise an error if `array.array` are in the pickled object (which is the case of all objects in Abapy).

abapy.misc.**read_file**(*path*, *ncol=2*, *separator=None*)

Read a tabular data file and returns a *numpy.array* containing the data. Header lines must begin with a #.

CHAPTER 7

Advanced Examples

Here we show some fancier examples using the tools available in abapy all together.

7.1 Indentation

7.1.1 2D / 3D , multi material indentation

All files used in this example are available in `doc/advanced_examples/indentation/simulations`

In this example, we focus on the indentation behavior of an elastic-plastic (von Mises) sample indented by an axisymmetric cone. To build this example, we first need to create 2 classes: `Simulation` and `Database_Manager` as follows in the file `classes.py`:

```
# VON MISES CLASSES
# Parametric indentation tool

#-------------------------------------------------------------------------------------------------------------
# IMPORTS
from sqlalchemy import create_engine, Column, Integer, String, Float, Boolean, PickleType, UniqueCons
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
#-------------------------------------------------------------------------------------------------------------


Base = declarative_base()




#-------------------------------------------------------------------------------------------------------------
# SIMULATION CLASS
# Declaration of the Simulation class which stores all simulation data so that each simulation is an
class Simulation(Base):
  __tablename__ = 'simulations'
  id = Column(Integer, primary_key=True)
  # Inputs
  three_dimensional            = Column(Boolean, default = False, nullable = False)
  sweep_angle         = Column(Float, nullable = False, default = 60.)
  rigid_indenter               = Column(Boolean, default = True, nullable = False)
  indenter_half_angle          = Column(Float, nullable = False, default = 70.3)
  indenter_pyramid             = Column(Boolean, default = True, nullable = False)
```

```
  indenter_mat_type            = Column(String, nullable = False, default = 'elastic')
  indenter_mat_args            = Column(PickleType,
    nullable = False,
    default = {'young_modulus': 1., 'poisson_ratio': 0.3})
  sample_mat_type              = Column(String, nullable = False, default = 'vonmises')
  sample_mat_args              = Column(PickleType,
    nullable = False,
    default = {'young_modulus': 1., 'poisson_ratio': 0.3, 'yield_stress': 0.01})
  friction                     = Column(Float, nullable = False, default = 0.)
  mesh_Na                      = Column(Integer, default = 4,  nullable = False)
  mesh_Nb                      = Column(Integer, default = 4,  nullable = False)
  mesh_Ns                      = Column(Integer, default = 16, nullable = False)
  mesh_Nf                      = Column(Integer, default = 2,  nullable = False)
  mesh_Nsweep                  = Column(Integer, default = 8,  nullable = False)
  indenter_mesh_Na             = Column(Integer, default = 0,  nullable = False)
  indenter_mesh_Nb             = Column(Integer, default = 0,  nullable = False)
  indenter_mesh_Ns             = Column(Integer, default = 0,  nullable = False)
  indenter_mesh_Nf             = Column(Integer, default = 0,  nullable = False)
  indenter_mesh_Nsweep         = Column(Integer, default = 0,  nullable = False)
  mesh_l                       = Column(Float,   default = 1., nullable = False)
  max_disp                     = Column(Float,   default = 1., nullable = False)
  sample_mesh_disp             = Column(PickleType, nullable = False, default = False )
  # Internal parameters
  frames                       = Column(Integer, default = 30, nullable = False)
  completed                    = Column(Boolean, default = False, nullable = False)
  priority                     = Column(Integer, default = 1, nullable = False)
  # Preprocess
  mesh                         = Column(PickleType)
  indenter                     = Column(PickleType)
  sample_mat                   = Column(PickleType)
  indenter_mat                 = Column(PickleType)
  steps                        = Column(PickleType)
  # Time histories
  force_hist                   = Column(PickleType)
  disp_hist                    = Column(PickleType)
  tip_penetration_hist         = Column(PickleType)
  elastic_work_hist            = Column(PickleType)
  plastic_work_hist            = Column(PickleType)
  friction_work_hist           = Column(PickleType)
  total_work_hist              = Column(PickleType)
  # Contact data
  contact_data                 =  Column(PickleType)
  # Fields
  stress_field                 = Column(PickleType)
  disp_field                   = Column(PickleType)
  total_strain_field           = Column(PickleType)
  plastic_strain_field         = Column(PickleType)
  elastic_strain_field         = Column(PickleType)
  equivalent_plastic_strain_field = Column(PickleType)
  disp_field                   = Column(PickleType)
  ind_disp_field               = Column(PickleType)

  # Table args
  __table_args__ = (
    UniqueConstraint(
    'three_dimensional',
    'indenter_pyramid',
    'rigid_indenter',
```

```
    'sample_mat_type',
    'sample_mat_args',
    'indenter_mat_type',
    'indenter_mat_args',
    'indenter_half_angle',
    'sweep_angle',
    'friction',
    'mesh_Na',
    'mesh_Nb',
    'mesh_Ns',
    'mesh_Nf',
    'mesh_l',
    'mesh_Nsweep',
    'indenter_mesh_Na',
    'indenter_mesh_Nb',
    'indenter_mesh_Ns',
    'indenter_mesh_Nf',
    'indenter_mesh_Nsweep',
    'max_disp',
    'sample_mesh_disp'),
    {})

  # Post processing script
  def abqpostproc_byRpt(self):
    if self.three_dimensional: # 3D post processing script
      out = """# ABQPOSTPROC.PY
# Warning: executable only in abaqus abaqus viewer -noGUI,... not regular python.
import sys
from abapy.postproc import GetFieldOutput_byRpt as gfo
from abapy.postproc import GetVectorFieldOutput_byRpt as gvfo
from abapy.postproc import GetTensorFieldOutput_byRpt as gtfo
from abapy.postproc import GetHistoryOutputByKey as gho
from abapy.indentation import Get_ContactData
from abapy.misc import dump
from odbAccess import openOdb
from abaqusConstants import JOB_STATUS_COMPLETED_SUCCESSFULLY



# Odb opening
file_name = '#FILE_NAME'
odb = openOdb(file_name + '.odb')
data = {}

# Check job status:
job_status = odb.diagnosticData.jobStatus

if job_status == JOB_STATUS_COMPLETED_SUCCESSFULLY:
  data['completed'] = True
  # Field Outputs
  data['field'] = {}
  fo = data['field']
  fo['Uind'] = [
    gvfo(odb = odb,
      instance = 'I_INDENTER',
      step = 1,
      frame = -1,
      original_position = 'NODAL',
```

```
    new_position = 'NODAL',
    position = 'node',
    field = 'U',
    delete_report = True),
  gvfo(odb = odb,
    instance = 'I_INDENTER',
    step = 2,
    frame = -1,
    original_position = 'NODAL',
    new_position = 'NODAL',
    position = 'node',
    field = 'U',
    delete_report = True)]

fo['U'] = [
  gvfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'NODAL',
    new_position = 'NODAL',
    position = 'node',
    field = 'U',
    sub_set_type = 'element',
    delete_report = True),
  gvfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'NODAL',
    new_position = 'NODAL',
    position = 'node',
    field = 'U',
    sub_set_type = 'element',
    delete_report = True)]

fo['S'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'S',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'S',
    sub_set_type = 'element',
    delete_report = True)]
```

```
fo['LE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'LE',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'LE',
    sub_set_type = 'element',
    delete_report = True)]

fo['EE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'EE',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'EE',
    sub_set_type = 'element',
    delete_report = True)]

fo['PE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'PE',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
```

```
      frame = -1,
      original_position = 'INTEGRATION_POINT',
      new_position = 'NODAL',
      position = 'node',
      field = 'PE',
      sub_set_type = 'element',
      delete_report = True)]

  fo['PEEQ'] = [
    gfo(odb = odb,
      instance = 'I_SAMPLE',
      step = 1,
      frame = -1,
      original_position = 'INTEGRATION_POINT',
      new_position = 'NODAL',
      position = 'node',
      field = 'PEEQ',
      sub_set_type = 'element',
      delete_report = True),
    gfo(odb = odb,
      instance = 'I_SAMPLE',
      step = 2,
      frame = -1,
      original_position = 'INTEGRATION_POINT',
      new_position = 'NODAL',
      position = 'node',
      field = 'PEEQ',
      sub_set_type = 'element',
      delete_report = True)]
  # History Outputs
  data['history'] = {}
  ho = data['history']
  ref_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['REF_NODE'].nodes[0].label
  ho['force'] =  gho(odb,'RF2')['Node I_INDENTER.'+str(ref_node)] # GetFieldOutputByKey returns all t
  ho['disp'] =   gho(odb,'U2')['Node I_INDENTER.'+str(ref_node)]
  tip_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['TIP_NODE'].nodes[0].label
  ho['tip_penetration'] =   gho(odb,'U2')['Node I_INDENTER.'+str(tip_node)]
  ho['allse'] =   gho(odb,'ALLSE').values()[0]
  ho['allpd'] =   gho(odb,'ALLPD').values()[0]
  ho['allfd'] =   gho(odb,'ALLFD').values()[0]
  ho['allwk'] =   gho(odb,'ALLWK').values()[0]
  #ho['carea'] =  gho(odb,'CAREA    ASSEMBLY_I_SAMPLE_SURFACE_FACES/ASSEMBLY_I_INDENTER_SURFACE_FACES

  # CONTACT DATA PROCESSING
  ho['contact'] = Get_ContactData(odb = odb, instance = 'I_SAMPLE', node_set = 'TOP_NODES')

else:
  data['completed'] = False
# Closing and dumping
odb.close()
dump(data, file_name+'.pckl')"""
    else:
      out = """# ABQPOSTPROC.PY
# Warning: executable only in abaqus abaqus viewer -noGUI,... not regular python.
import sys
from abapy.postproc import GetFieldOutput_byRpt as gfo
from abapy.postproc import GetVectorFieldOutput_byRpt as gvfo
from abapy.postproc import GetTensorFieldOutput_byRpt as gtfo
```

```
from abapy.postproc import GetHistoryOutputByKey as gho
from abapy.indentation import Get_ContactData
from abapy.misc import dump
from odbAccess import openOdb
from abaqusConstants import JOB_STATUS_COMPLETED_SUCCESSFULLY



# Odb opening
file_name = '#FILE_NAME'
odb = openOdb(file_name + '.odb')
data = {}

# Check job status:
job_status = odb.diagnosticData.jobStatus

if job_status == JOB_STATUS_COMPLETED_SUCCESSFULLY:
  data['completed'] = True
  # Field Outputs
  data['field'] = {}
  fo = data['field']
  fo['Uind'] = [
    gvfo(odb = odb,
      instance = 'I_INDENTER',
      step = 1,
      frame = -1,
      original_position = 'NODAL',
      new_position = 'NODAL',
      position = 'node',
      field = 'U',
      delete_report = True),
    gvfo(odb = odb,
      instance = 'I_INDENTER',
      step = 2,
      frame = -1,
      original_position = 'NODAL',
      new_position = 'NODAL',
      position = 'node',
      field = 'U',
      delete_report = True)]

  fo['U'] = [
    gvfo(odb = odb,
      instance = 'I_SAMPLE',
      step = 1,
      frame = -1,
      original_position = 'NODAL',
      new_position = 'NODAL',
      position = 'node',
      field = 'U',
      sub_set_type = 'element',
      delete_report = True),
    gvfo(odb = odb,
      instance = 'I_SAMPLE',
      step = 2,
      frame = -1,
      original_position = 'NODAL',
      new_position = 'NODAL',
```

```
    position = 'node',
    field = 'U',
    sub_set_type = 'element',
    delete_report = True)]

fo['S'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'S',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'S',
    delete_report = True)]

fo['LE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'LE',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'LE',
    sub_set_type = 'element',
    delete_report = True)]

fo['EE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'EE',
    sub_set_type = 'element',
```

```
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'EE',
    sub_set_type = 'element',
    delete_report = True)]

fo['PE'] = [
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'PE',
    sub_set_type = 'element',
    delete_report = True),
  gtfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'PE',
    sub_set_type = 'element',
    delete_report = True)]

fo['PEEQ'] = [
  gfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 1,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'PEEQ',
    sub_set_type = 'element',
    delete_report = True),
  gfo(odb = odb,
    instance = 'I_SAMPLE',
    step = 2,
    frame = -1,
    original_position = 'INTEGRATION_POINT',
    new_position = 'NODAL',
    position = 'node',
    field = 'PEEQ',
    sub_set_type = 'element',
    delete_report = True)]
# History Outputs
data['history'] = {}
ho = data['history']
```

```
  ref_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['REF_NODE'].nodes[0].label
  ho['force'] =  gho(odb,'RF2')['Node I_INDENTER.'+str(ref_node)] # GetFieldOutputByKey returns all t
  ho['disp'] =   gho(odb,'U2')['Node I_INDENTER.'+str(ref_node)]
  tip_node = odb.rootAssembly.instances['I_INDENTER'].nodeSets['TIP_NODE'].nodes[0].label
  ho['tip_penetration'] =   gho(odb,'U2')['Node I_INDENTER.'+str(tip_node)]
  ho['allse'] =   gho(odb,'ALLSE').values()[0]
  ho['allpd'] =   gho(odb,'ALLPD').values()[0]
  ho['allfd'] =   gho(odb,'ALLFD').values()[0]
  ho['allwk'] =   gho(odb,'ALLWK').values()[0]
  #ho['carea'] =  gho(odb,'CAREA    ASSEMBLY_I_SAMPLE_SURFACE_FACES/ASSEMBLY_I_INDENTER_SURFACE_FACES

  # CONTACT DATA PROCESSING
  ho['contact'] = Get_ContactData(odb = odb, instance = 'I_SAMPLE', node_set = 'TOP_NODES')

else:
  data['completed'] = False
# Closing and dumping
odb.close()
dump(data, file_name+'.pckl')"""
    return out
  # Scalar Outputs
  # Load Prefactor
  load_prefactor = Column(Float)
  def Load_prefactor(self, update = False):
    '''
    Defined during loading phase by force = c * penetration **2 where c is the load prefactor.
    '''
    load_prefactor = (self.force_hist[1] / self.disp_hist[1]**2).average(method = 'simps')
    if update:
      self.load_prefactor = load_prefactor
    else:
      return load_prefactor

  # Irreversible work ratio:
  irreversible_work_ratio = Column(Float)
  def Irreversible_work_ratio(self, update = False):
    '''
    Irreversible work divided by the total work
    '''
    unload = self.total_work_hist[2]
    irreversible_work_ratio = 3* unload.data_min() / self.load_prefactor
    if update:
      self.irreversible_work_ratio = irreversible_work_ratio
    else:
      return irreversible_work_ratio

  # Plastic work ratio:
  plastic_work_ratio = Column(Float)
  def Plastic_work_ratio(self, update = False):
    '''
    Plastic work divided by the total work
    '''
    plastic_work_ratio = (self.plastic_work_hist[1] / self.total_work_hist[1]).average()
    if update:
      self.plastic_work_ratio = plastic_work_ratio
    else:
      return plastic_work_ratio
```

```
  # Unloading fit
  contact_stiffness = Column(Float)
  final_displacement = Column(Float)
  unload_exponent = Column(Float)
  def Unloading_fit(self):
    '''
    Computes the contact stiffness and the final displacement using a power fit of the unloading curv
    '''
    import numpy as np
    from scipy.optimize import leastsq
    unload = 2
    disp = np.array(self.disp_hist[unload].data[0])
    force = np.array(self.force_hist[unload].data[0])
    max_force = force.max()
    max_disp = disp.max()
    loc = np.where(force >= max_force * .1)
    disp = disp[loc] /max_disp
    force = force[loc] / max_force
    func = lambda k, x: ( (x - k[0]) / (1. - k[0] ) )**k[1]
    err = lambda v, x, y: (func(v,x)-y)
    k0 = [0., 1.]
    k, success = leastsq(err, k0, args=(disp,force), maxfev=10000)
    self.final_displacement = k[0] / max_disp
    self.contact_stiffness = k[1] / (1. - k[0])
    self.unload_exponent = k[1]

  # Contact area:
  contact_area = Column(Float)
  def Contact_area(self, update = False):
    '''
    Cross contact areat under load divided by the square of the indenter displacement.
    '''
    import numpy as np
    from abapy.postproc import HistoryOutput
    contact_step = self.contact_data[1] # we only look at the loading 1 here
    ca= np.array([contact_step[i].contact_area() for i in xrange(len(contact_step))])
    disp = np.array(self.disp_hist[1].data)
    ca = ca / (disp**2)
    contact_area = ca.mean()
    if update:
      self.contact_area = contact_area
    else:
      return contact_area


  # Tip penetration
  tip_penetration = Column(PickleType)
  def Tip_penetration(self, update = True):
    '''
    Tip penetration under load divided by the displacement.
    '''
    tip_pen = self.tip_penetration_hist[1]
    disp = self.disp_hist[1]
    tip_pen = (-tip_pen/disp).average()
    if update:
      self.tip_penetration = tip_pen
    else:
      return tip_penetration
```

```
def __repr__(self):
  return '<Simulation: id={0}>'.format(self.id)

def difficulty(self):
  if self.sample_mat_type == 'elastic': mat_diff = 1.
  if self.sample_mat_type == 'vonmises':
    args = self.sample_mat_args
    mat_diff = args['young_modulus'] / args['yield_stress']
  if self.sample_mat_type == 'druckerprager':
    args = self.sample_mat_args
    mat_diff = args['young_modulus'] / args['yield_stress']
  if self.sample_mat_type == 'hollomon':
    args = self.sample_mat_args
    mat_diff = args['young_modulus'] / args['yield_stress']
  mesh_diff = self.max_disp /self.mesh_l * (self.mesh_Na + self.mesh_Nb) / 2.
  return mat_diff * mesh_diff

def preprocess(self):
  from abapy.indentation import IndentationMesh, Step, DeformableCone2D, DeformableCone3D
  from abapy.materials import VonMises, Elastic, DruckerPrager, Hollomon
  from math import tan, radians
  mesh_l = 2 * max( self.max_disp , tan(radians(self.indenter_half_angle)) ) # Adjusting mesh size
  if self.three_dimensional:
    self.mesh = IndentationMesh(
      Na = self.mesh_Na,
      Nb = self.mesh_Nb,
      Ns = self.mesh_Ns,
      Nf = self.mesh_Nf,
      l = mesh_l).sweep(
      sweep_angle = self.sweep_angle,
      N = self.mesh_Nsweep)
    if self.sample_mesh_disp != False:
      field = self.mesh.nodes.eval_vectorFunction(self.sample_mesh_disp)
      self.mesh.nodes.apply_displacement(field)
    if self.indenter_mesh_Nf == 0: Nf_i = self.mesh_Nf
    self.indenter = DeformableCone3D(
      half_angle = self.indenter_half_angle,
      sweep_angle = self.sweep_angle,
      pyramid = self.indenter_pyramid,
      l = mesh_l,
      Na = self.mesh_Na * (self.indenter_mesh_Na == 0) + self.indenter_mesh_Na * (self.indenter_mes
      Nb = self.mesh_Nb * (self.indenter_mesh_Nb == 0) + self.indenter_mesh_Nb * (self.indenter_mes
      Ns = self.mesh_Ns * (self.indenter_mesh_Ns == 0) + self.indenter_mesh_Ns * (self.indenter_mes
      Nf = self.mesh_Nf * (self.indenter_mesh_Nf == 0) + self.indenter_mesh_Nf * (self.indenter_mes
      N =  self.mesh_Nsweep * (self.indenter_mesh_Nsweep == 0) + self.indenter_mesh_Nsweep * (self
      rigid = self.rigid_indenter)

  else:
    self.mesh = IndentationMesh(
      Na = self.mesh_Na,
      Nb = self.mesh_Nb,
      Ns = self.mesh_Ns,
      Nf = self.mesh_Nf,
      l = mesh_l)
    self.indenter = DeformableCone2D(
```

```
        half_angle = self.indenter_half_angle,
        l = mesh_l,
        Na = self.mesh_Na * (self.indenter_mesh_Na == 0) + self.indenter_mesh_Na * (self.indenter_mes
        Nb = self.mesh_Nb * (self.indenter_mesh_Nb == 0) + self.indenter_mesh_Nb * (self.indenter_mes
        Ns = self.mesh_Ns * (self.indenter_mesh_Ns == 0) + self.indenter_mesh_Ns * (self.indenter_mes
        Nf = self.mesh_Nf * (self.indenter_mesh_Nf == 0) + self.indenter_mesh_Nf * (self.indenter_mes
        rigid = self.rigid_indenter)
    self.steps = [
      Step(name='loading0',
        nframes = self.frames,
        disp = self.max_disp/2.,
        boundaries_3D= self.three_dimensional),
      Step(name='loading1',
        nframes = self.frames,
        disp = self.max_disp,
        boundaries_3D= self.three_dimensional),
      Step(name = 'unloading',
        nframes = self.frames,
        disp = 0.,
        boundaries_3D= self.three_dimensional)]
    if self.sample_mat_type == 'hollomon':
      self.sample_mat = Hollomon(
        labels = 'SAMPLE_MAT',
        E =  self.sample_mat_args['young_modulus'],
        nu = self.sample_mat_args['poisson_ratio'],
        sy = self.sample_mat_args['yield_stress'],
        n = self.sample_mat_args['hardening'])
    if self.sample_mat_type == 'druckerprager':
      self.sample_mat = DruckerPrager(
        labels = 'SAMPLE_MAT',
        E =  self.sample_mat_args['young_modulus'],
        nu = self.sample_mat_args['poisson_ratio'],
        sy = self.sample_mat_args['yield_stress'],
        beta = self.sample_mat_args['beta'],
        psi = self.sample_mat_args['psi'],
        k = self.sample_mat_args['k'])
    if self.sample_mat_type == 'vonmises':
      self.sample_mat = VonMises(
        labels = 'SAMPLE_MAT',
        E =  self.sample_mat_args['young_modulus'],
        nu = self.sample_mat_args['poisson_ratio'],
        sy = self.sample_mat_args['yield_stress'])
    if self.sample_mat_type == 'elastic':
      self.sample_mat = Elastic(
        labels = 'SAMPLE_MAT',
        E = self.sample_mat_args['young_modulus'],
        nu = self.sample_mat_args['poisson_ratio'])
    if self.indenter_mat_type == 'elastic':
      self.indenter_mat = Elastic(
        labels = 'INDENTER_MAT',
        E = self.indenter_mat_args['young_modulus'],
        nu = self.indenter_mat_args['poisson_ratio'])



  def run(self, work_dir = 'workdir/', abqlauncher = '/opt/Abaqus/6.9/Commands/abaqus'):
    print '# Running {0}: id={1}, frames = {2}'.format(self.__class__.__name__, self.id, self.frames)
    self.preprocess()
```

```
    from abapy.indentation import Manager
    import numpy as np
    from copy import copy
    simname = self.__class__.__name__ + '_{0}'.format(self.id)
    # Creating abq postproc script
    f = open('{0}{1}_abqpostproc.py'.format(work_dir,self.__class__.__name__),'w')
    name = self.__class__.__name__ + '_' + str(self.id)
    out = self.abqpostproc_byRpt().replace('#FILE_NAME', name)
    f.write(out)
    f.close()
    abqpostproc = '{0}_abqpostproc.py'.format(self.__class__.__name__)
    #----------------------------------------
    # Setting simulation manager
    m = Manager()
    m.set_abqlauncher('/opt/Abaqus/6.9/Commands/abaqus')
    m.set_workdir(work_dir)
    m.set_simname(self.__class__.__name__ + '_{0}'.format(self.id))
    m.set_abqpostproc(abqpostproc)
    m.set_samplemesh(self.mesh)
    m.set_samplemat(self.sample_mat)
    m.set_indentermat(self.indenter_mat)
    m.set_friction(self.friction)
    m.set_steps(self.steps)
    m.set_indenter(self.indenter)
    m.set_is_3D(self.three_dimensional)
    m.set_pypostprocfunc(lambda data: data) # Here we just want to get back data
    #----------------------------------------
    # Running simulation and post processing
    m.erase_files()           # Workdir cleaning
    m.make_inp()              # INP creation
    m.run_sim()               # Running the simulation
    m.run_abqpostproc()       # First round of post processing in Abaqus
    data = m.run_pypostproc() # Second round of custom post processing in regular Python
    #m.erase_files()           # Workdir cleaning
    #----------------------------------------
    if data['completed']:
      print '# Simulation completed.'
      # Storing raw data
      self.completed = True
      sweep_factor = 1. # This factor aims to modify values that are affected by the fact that only a
      if self.three_dimensional: sweep_factor = 360. / self.sweep_angle
      self.force_hist = - sweep_factor * data['history']['force']
      self.disp_hist = -data['history']['disp']
      self.elastic_work_hist = sweep_factor * data['history']['allse']
      self.plastic_work_hist = sweep_factor * data['history']['allpd']
      self.friction_work_hist = sweep_factor * data['history']['allfd']
      self.total_work_hist = sweep_factor * data['history']['allwk']
      self.tip_penetration_hist = data['history']['tip_penetration']
      self.disp_field = data['field']['U']
      self.ind_disp_field = data['field']['Uind']
      self.stress_field = data['field']['S']
      self.total_strain_field = data['field']['LE']
      self.elastic_strain_field = data['field']['EE']
      self.plastic_strain_field = data['field']['PE']
      self.equivalent_plastic_strain_field = data['field']['PEEQ']
      self.contact_data = data['history']['contact']
      # Updating data
      self.Load_prefactor(update = True)
```

```
      self.Irreversible_work_ratio(update = True)
      self.Plastic_work_ratio(update = True)
      self.Tip_penetration(update = True)
      self.Contact_area(update = True)
      self.Unloading_fit()
    #---------------------------------------
    else:
      print '# Simulation not completed.'
#----------------------------------------------------------------------------------------------------



#----------------------------------------------------------------------------------------------------
# DATABASE MANAGEMENT CLASS
#
class Database_Manager:
  def __init__(self, database_dir, database_name, cls , work_dir, abqlauncher):
    db = 'sqlite:///{0}{1}.db'.format(database_dir, database_name)
    engine = create_engine(db,echo=False)
    Base.metadata.create_all(engine)
    Session = sessionmaker(bind=engine)
    self.session = Session()
    self.cls = cls
    self.work_dir = work_dir
    self.abqlauncher = abqlauncher

  def add_simulation(self, simulation):
    '''
    Adds a simulation to the database.

    Args:
    * simulation: Simulation class instance
    '''
    if isinstance(simulation, self.cls) == False: raise Exception, 'simulation must be Simulation ins
    try:
      self.session.add(simulation)
      self.session.commit()
    except:
      print 'simulation already exists or has not been declared corectly, nothing changed'
      self.session.rollback()

  def get_next(self):
    '''
    Returns the next simulation to do regarding to difficulty criterias defined under.
    '''
    simus = self.session.query(self.cls).filter(self.cls.completed == False)
    if simus.all() != []:
      # finding max priority
      max_priority = self.session.query(self.cls).filter(self.cls.completed == False).order_by(desc(s
      # finding less difficult simulation with max priority
      simus = simus.filter(self.cls.priority == max_priority)
      simus = sorted(simus, key=lambda simu: simu.difficulty())
      simu = simus[0]
      # Adjusting number of requested frames
      diff = simu.difficulty()
      completed_simus = self.session.query(self.cls).filter(self.cls.completed == True)
      for csim in completed_simus:
        if csim.difficulty() <= diff:
```

```
          if csim.frames > simu.frames: simu.frames = csim.frames
      completed_simus = [c_simu for c_simu in completed_simus if c_simu.difficulty <= diff]
      self.session.commit()
    else:
      simu = None
    return simu

  def run_next(self):
    '''
    Runs the next simulation.
    '''
    simu = self.get_next()
    if simu != None:
      while True:
        simu.run(work_dir = self.work_dir, abqlauncher = self.abqlauncher)
        if simu.completed: break
        simu.frames = int(simu.frames * 1.5)
        self.session.commit()
        print '# Number of frames changed to {0}.'.format(simu.frames)
      self.session.commit()
    else:
      print '# No more simulations to run'

  def run_all(self):
    '''
    Runs all the simulation in the right order until they all have been completed.
    '''
    while True:
      left_sims = self.session.query(self.cls).filter(self.cls.completed == False).count()
      if left_sims == 0:
        print '# All simulations have been run.'
        break
      print '# {0} simulations left to run.'.format(left_sims)
      self.run_next()

  def query(self):
    '''
    Shortcut for database queries.
    '''
    return self.session.query(self.cls)
#-------------------------------------------------------------------------------------------------
```

Then we need to define some basic settings here `settings.py`:

```
from classes import Simulation, Database_Manager

#-------------------------------------------------------------------------------------------------
# SETTINGS
work_dir      = 'workdir/'
plot_dir      = 'plots/'
database_dir  = 'database/'
database_name = 'database'
abqlauncher   = '/opt/Abaqus/6.9/Commands/abaqus'
cls           = Simulation
#-------------------------------------------------------------------------------------------------

#-------------------------------------------------------------------------------------------------
# Starting Database Manager
db_manager =  Database_Manager(
```

```
  work_dir = work_dir,
  database_dir = database_dir,
  database_name = database_name,
  abqlauncher = abqlauncher,
  cls = cls)
#-------------------------------------------------------------------------------
```

Then we can load simulations request in the SQLite database `settings.py`:

```
# LOADER: loads simulations to do in the database
import numpy as np
from copy import copy
from abapy.indentation import equivalent_half_angle

# Setting up the database
execfile('settings.py')

# creating some shortcuts
d = db_manager      # database manager
c = db_manager.cls # useful to call Simulation attributs


#--------------------------
# FIXED PARAMETERS
#--------------------------

# Fixed Parameters
Na_berk, Nb_berk     = 8, 8  # Mesh parameters
Ns_berk, Nf_berk     = 16, 2 # Mesh parameters
Na_cone, Nb_cone     = 16, 16  # Mesh parameters
Ns_cone, Nf_cone     = 16, 2 # Mesh parameters
Nsweep, sweep_angle = 8, 60. # Mesh sweep parameters
E_s         = 1.   # Sample's Young's modulus
nu          = 0.2   # Poisson's ratio
half_angle = 65.27  # Indenter half angle of the modified Berkovich
frames      = 50    # Number frames per step


#--------------------------
# DRUCKER PRAGER SIMULATIONS
#--------------------------

ey  = [0.01, 0.015,0.02, 0.025,0.03, 0.035,0.04, 0.045, 0.05] # Yield strain
beta = [0., 5., 10., 15., 20., 25., 30.]

print 'LOADING DRUCKER PRAGER SIMULATIONS'

for i in xrange(len(ey)):
  print '* epsilon_y = ', ey[i]
  for j in xrange(len(beta)):
    print 'beta= ', beta[j]
    print '* Conical indenter'
    simu = Simulation(
      rigid_indenter= True,
      indenter_pyramid = True,
      three_dimensional = False,
      sweep_angle = sweep_angle,
      sample_mat_type = 'druckerprager',
      sample_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3, 'yield_stress': ey[i] * E_s, 'bet
```

```
      indenter_mat_type = 'elastic',
      indenter_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3},
      mesh_Na = Na_cone,
      mesh_Nb = Nb_cone,
      mesh_Ns = Ns_cone,
      mesh_Nf = Nf_cone,
      indenter_mesh_Na = 2,
      indenter_mesh_Nb = 2,
      indenter_mesh_Ns = 1,
      indenter_mesh_Nf = 1,
      indenter_mesh_Nsweep = 2,
      mesh_Nsweep = Nsweep,
      indenter_half_angle = equivalent_half_angle(half_angle, sweep_angle),
      frames = frames )
    db_manager.add_simulation(simu)

    '''
    print '* Berkovich indenter'
    simu = Simulation(
      rigid_indenter= True,
      indenter_pyramid = True,
      three_dimensional = True,
      sweep_angle = sweep_angle,
      sample_mat_type = 'druckerprager',
      sample_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3, 'yield_stress': ey[i] * E_s, 'bet
      indenter_mat_type = 'elastic',
      indenter_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3},
      mesh_Na = Na_berk,
      mesh_Nb = Nb_berk,
      mesh_Ns = Ns_berk,
      mesh_Nf = Nf_berk,
      mesh_Nsweep = Nsweep,
      indenter_mesh_Na = 2,
      indenter_mesh_Nb = 2,
      indenter_mesh_Ns = 1,
      indenter_mesh_Nf = 2,
      indenter_mesh_Nsweep = 2,
      indenter_half_angle = half_angle,
      sample_mesh_disp = False,
      frames = frames )
    db_manager.add_simulation(simu)
    '''

#--------------------------
# HOLLOMON SIMULATIONS
#--------------------------
#ey  = [0.001, 0.002, 0.003, 0.004, 0.005, 0.006, 0.007, 0.008, 0.009, 0.01] # Yield strain
ey = [0.001]
#n = [0., .1, .2, .3, .4]
n = [.3, .4]

print 'LOADING HOLLOMON SIMULATIONS'

for i in xrange(len(ey)):
  print '* epsilon_y = ', ey[i]
  for j in xrange(len(n)):
    print '* n = ', n[j]
    print '* Conical indenter'
```

```
    simu = Simulation(
      rigid_indenter= True,
      indenter_pyramid = True,
      three_dimensional = False,
      sweep_angle = sweep_angle,
      sample_mat_type = 'hollomon',
      sample_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3, 'yield_stress': ey[i] * E_s, 'ha
      indenter_mat_type = 'elastic',
      indenter_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3},
      mesh_Na = Na_cone,
      mesh_Nb = Nb_cone,
      mesh_Ns = Ns_cone,
      mesh_Nf = Nf_cone,
      indenter_mesh_Na = 2,
      indenter_mesh_Nb = 2,
      indenter_mesh_Ns = 1,
      indenter_mesh_Nf = 2,
      indenter_mesh_Nsweep = 2,
      mesh_Nsweep = Nsweep,
      indenter_half_angle = equivalent_half_angle(half_angle, sweep_angle),
      frames = frames )
    db_manager.add_simulation(simu)

  '''
    print '* Berkovich indenter'
    simu = Simulation(
      rigid_indenter= True,
      indenter_pyramid = True,
      three_dimensional = True,
      sweep_angle = sweep_angle,
      sample_mat_type = 'hollomon',
      sample_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3, 'yield_stress': ey[i] * E_s, 'ha
      indenter_mat_type = 'elastic',
      indenter_mat_args = {'young_modulus': 1., 'poisson_ratio': 0.3},
      mesh_Na = Na_berk,
      mesh_Nb = Nb_berk,
      mesh_Ns = Ns_berk,
      mesh_Nf = Nf_berk,
      mesh_Nsweep = Nsweep,
      indenter_mesh_Na = 2,
      indenter_mesh_Nb = 2,
      indenter_mesh_Ns = 1,
      indenter_mesh_Nf = 2,
      indenter_mesh_Nsweep = 2,
      indenter_half_angle = half_angle,
      sample_mesh_disp = False,
      frames = frames )
    db_manager.add_simulation(simu)
   '''
```

After executing loader, we see that all simulation have been entered in the database:

```
>>> execfile('loader.py')
```

Then we just launch the simulations using `launcher.py`:

```
# Setting up the database
execfile('settings.py')

# Running all simulations
db_manager.run_all()
#db_manager.run_next()
```

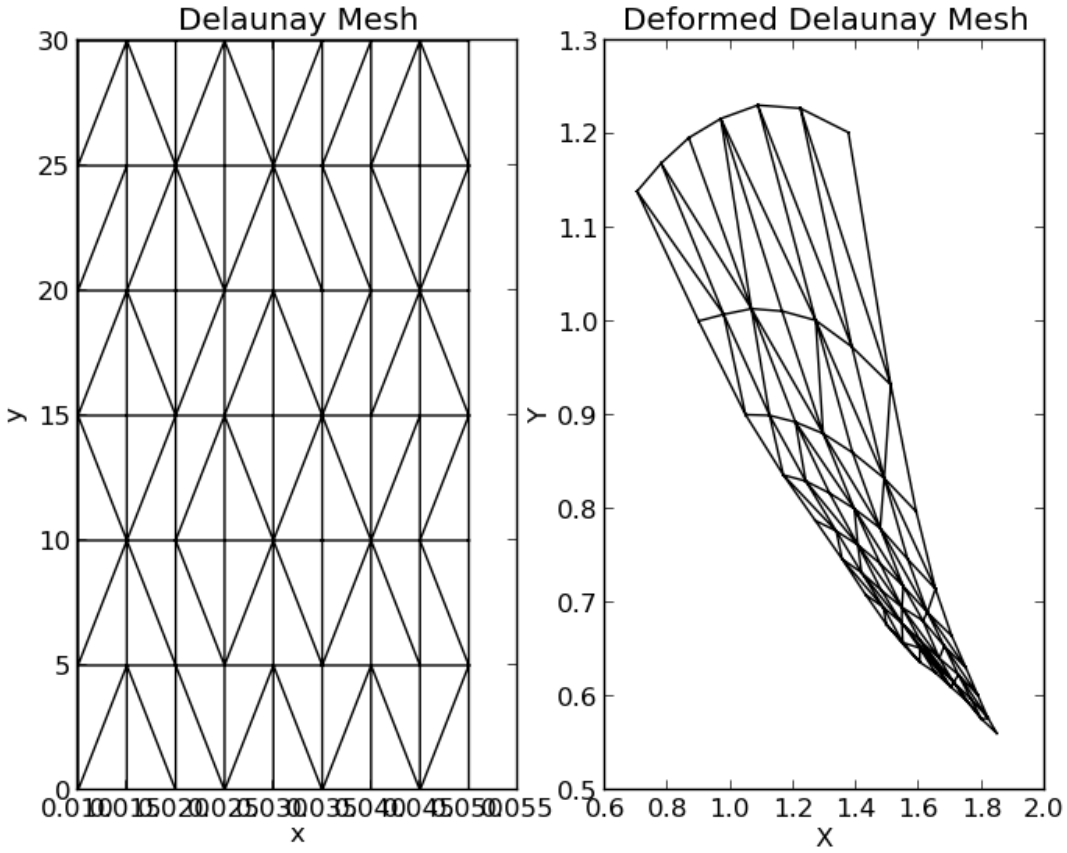
```
>>> execfile('launcher.py')
```

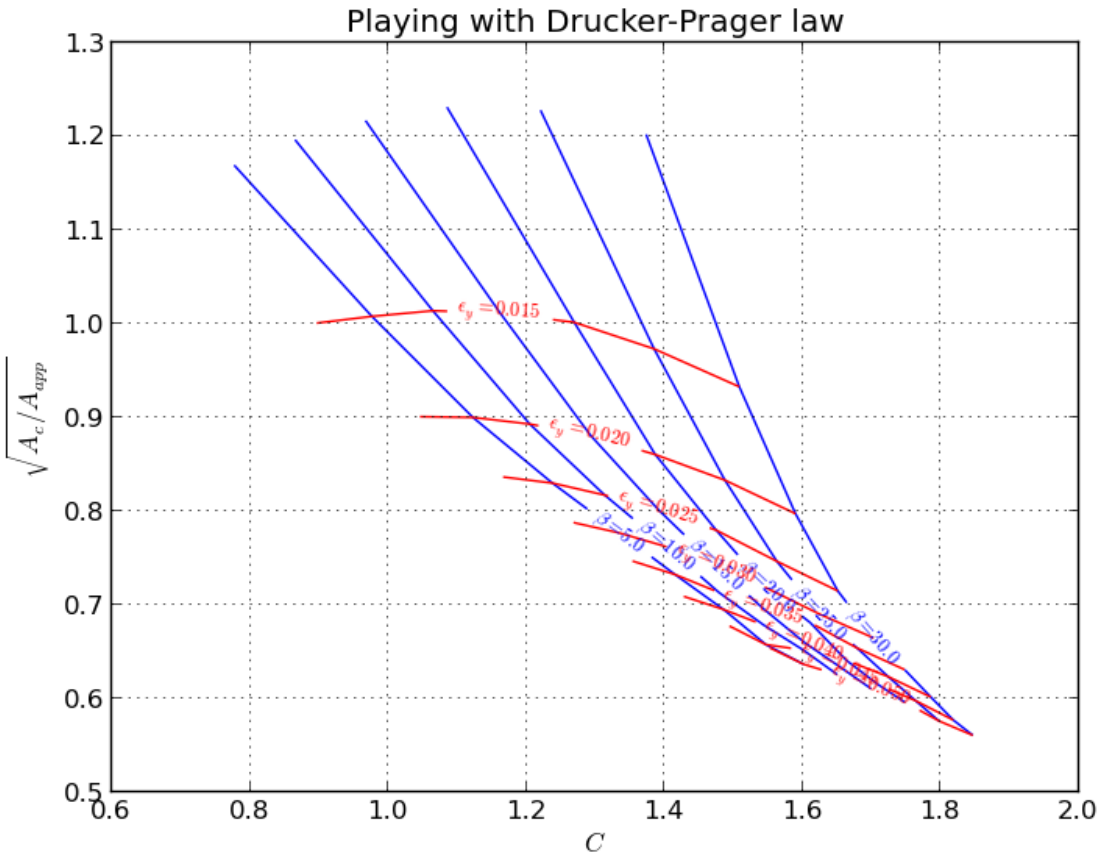
And now after these fast simulations it's time to collect some results or perform some reverse analysis. Here is a very brief example of ploting `basic_plot_DP.py`:

```
# Importing packages
from matplotlib import pyplot as plt
import numpy as np
from matplotlib import cm
from scipy.spatial import Delaunay

# Setting up the database
execfile('settings.py')


# creating some shortcuts
d = db_manager     # database manager
c = db_manager.cls # useful to call Simulation attributs

# Load simulations
simus = d.query().filter(c.completed == True).filter(c.sample_mat_type == 'druckerprager').all()

#--------------------------
# PLOTING CONSTRAINT FACTOR
#--------------------------

# Getting primary data
sy = np.array([s.sample_mat_args['yield_stress'] for s in simus])
E = np.array([s.sample_mat_args['young_modulus'] for s in simus])
beta = np.array([s.sample_mat_args['beta'] for s in simus])
C = np.array([s.load_prefactor for s in simus])
Ac = np.array([s.contact_area for s in simus])
hmax = np.array([s.max_disp for s in simus])
phi = np.array([s.indenter_half_angle for s in simus])
wirr_wtot = np.array([s.irreversible_work_ratio for s in simus])

# Building secondary data
ey = sy/E
H = C/Ac
hch = Ac / (np.pi*np.tan(np.radians(phi))**2)

# Building Delaunay triangular connectivity
x = ey
y = beta
points = np.array([x,y]).transpose()
conn = Delaunay(points).vertices

# For checking purpose, let's plot the generated mesh. You may see that the mesh is self improving ha


# Ploting stuff
```

```
title = 'Playing with Drucker-Prager law'

X, xlabel = C, r'$C$'
#X, xlabel = ey, r'$\sigma_{yc}/E$'
#X, xlabel = wirr_wtot, r'$W_{irr}/W_{tot}$'
Y, ylabel = hch, '$\sqrt{A_c / A_{app}}$'
#Y, ylabel = C, '$C/E$'

Z, zlabel = beta, r'$\beta = %1.1f$'
Zlevels = list(set(Z))
Z2, z2label = ey, r'$\epsilon_y = %1.3f$'
Z2levels = list(set(Z2))

# For checking purpose, let's plot the generated mesh. You may see that the mesh is self improving ha

fig = plt.figure(0)
plt.clf()
fig.add_subplot(121)
plt.triplot(x, y, conn)
plt.title('Delaunay Mesh')
plt.xlabel('x')
plt.ylabel('y')
fig.add_subplot(122)
plt.triplot(X, Y, conn)
plt.title('Deformed Delaunay Mesh')
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('plots/basic_plot_mesh_DP.png')

plt.figure(0)
plt.clf()
plt.title(title)
plt.grid()
plt.xlabel(xlabel)
plt.ylabel(ylabel)
#plt.triplot(X,Y,conn)
#plt.tricontourf(X,Y,conn,Z, Zlevels)
cont = plt.tricontour(X,Y,conn,Z, Zlevels, colors = 'blue')
plt.clabel(cont, fmt = zlabel, fontsize=9, inline=1)
cont = plt.tricontour(X,Y,conn,Z2, Z2levels, colors = 'red')
plt.clabel(cont, fmt = z2label, fontsize=9, inline=1)
plt.savefig('plots/basic_plot_DP.png')

'''
#--------------------------
# PLOTING SECTIONS
#--------------------------
plt.figure(0)
plt.clf()
plt.gca().set_aspect('equal')
rmax = 6.
x = np.linspace(0., rmax, 128)
y = np.zeros_like(x)
ey = np.array(list(set(ey)))
ey.sort()
for s in simus:
  cd = s.contact_data[2][-1]
  alt, press = cd.interpolate(x, y, method = 'linear')
```

```
  ey_s = s.sample_mat_args['yield_stress']
  loc = np.where(ey==ey_s)[0][0]
  alt += -loc
  plt.plot(x, alt)
plt.show()
'''
```




Playing with Drucker-Prager law
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
C
$\sqrt{A_c/A_{app}}$
$\epsilon_y = 0.015$
$\epsilon_y = 0.020$
$\epsilon_y = 0.025$
$\beta = 5.0$
$\beta = 10.0$
$\beta = 15.0$
$\beta = 20.0$
$\beta = 25.0$
$\beta = 30.0$

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

a

P

R

S

T

U

V

Z