PyFEHM Documentation

Release 1.1.0

David Dempsey

January 23, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

Python is an open-source, object-oriented scripting language with extensive functionality, a large, active development community and loads of online resources for troubleshooting. PyFEHM is a set of classes and methods to enable use of FEHM and auxiliary tasks within the Python scripting environment. Briefly, PyFEHM functionality includes

1. Unstructured/structure grid parsing, structured grid generation and manipulation (see Chapter *2*).
2. FEHM input file construction. Support for many macros, restart files, stress and carbon dioxide modules (see Chapter *3*).
3. Post-processing of output (see Chapter *4*), including visualisation in Paraview.
4. Scripting tools that supports Python's built-in multi-processing capabilities for batch simulations.

Introductory *tutorials* to PyFEHM are also provided.

1.1 Installation

1.1.1 Python

PyFEHM is supported on Python 2.6 and 2.7, but NOT 3.x. Instructions for downloading and installing Python can be found at www.python.org

PyFEHM requires the following Python modules to be installed: NumPy, SciPy, Matplotlib. For windows users, 32- and 64-bit installers (for several Python versions) of these modules can be obtained from http://www.lfd.uci.edu/~gohlke/pythonlibs/

1.1.2 PyFEHM

A download link for the latest release version of PyFEHM can be found at pyfehm.lanl.gov

To install, download and extract the zip file, and run the setup script at the command line:

```
python setup.py install
```

1.1.3 FEHM

The purpose of PyFEHM is construction of input and grid files for running FEHM simulations. Therefore, if the user does not have a copy of FEHM, their experience with PyFEHM is likely to be short and unsatisfactory.

Downloading PyFEHM does not provide the user with an FEHM executable. FEHM is free, but controlled, software and must be obtained by registering as a user at fehm.lanl.gov

1.1.4 Paraview

Paraview is a parallel, open-source visualisation software. PyFEHM supports a bridge with Paraview; information about the grid, model, and output data can be automatically exported and viewed in this program.

Instructions for downloading and installing Paraview can be found at www.paraview.org

After installation, make sure to include the path to paraview.exe in you PATH environment variable. This will ensure PyFEHM can find Paraview and make full use of its capabilities.

1.1.5 LaGriT

LaGriT is a mesh generation package, developed at Los Alamos, with a variety of tools for creating and modifying tetrahedral meshes. PyFEHM requires an executable of LaGriT to perform some specialised tasks. For more information on obtaining a LaGriT executable, visit lagrit.lanl.gov

1.2 Using PyFEHM

PyFEHM consists of several Python modules. To access their functionality, the user must include the following line at the top of any Python script

```
from fdata import*
```

A configuration file, pyfehmrc, can be used to customise your installation of PyFEHM. For instance, default paths to FEHM or Paraview can be defined in the config file to avoid having to write these out in scripts. Other default behaviour in PyFEHM can also be modified by making changes to the config file.

A sample configuration file, pyfehmrc_example, is included with the PyFEHM download. Copy this file to the site-packages folder in your Python installation, rename it to 'pyfehmrc' or '.pyfehmrc', and modify according to the instructions inside.

1.3 Using this manual

This manual comprises sections for each of the important PyFEHM modules: *fgrid*, *fdata*, *fpost*, and *fvars*. In these, the important classes and their methods are documented, and example usage provided.

One way to get acquainted with PyFEHM is to work through the *tutorials* provided. These provide an introduction to the basic workflow of creating, running and visualising an FEHM simulation. They also document some of the more specialised capabilities of PyFEHM.

CHAPTER

TWO

FGRID: FEHM GRID MANIPULATION

This module contains classes and methods for the manipulation of FEHM grid files. Typically, usage will be limited to reading and writing of grid files, and the use of the spatial and connectivity information provided.

PyFEHM can parse unstructured or structured grids in FEHM format. PyFEHM can construct orthogonal grids of arbitrary complexity using the `fmake` class or by calling `fgrid.make()`.

For the purposes of this manual, the variable `geo` will be assumed to refer to a previously defined instance of the `fgrid` class.

2.1 Nodes

The smallest quantum of the finite element grid. Node objects and associated connectivity information are automatically created when a grid file is parsed (`fgrid.read()`).

In FEHM, a node is associated with a position in space and a control volume - the region of space uniquely associated with the node. It is connected to other nodes, forming elements and the finite element grid.

When a grid is loaded and associated with an FEHM input file, material properties and zone information are mapped back onto the nodes. For example, if the **ROCK** macro has been assigned in an input file, then density information for a given node is accessed through its density attribute, `fnode.density`.

class `fgrid.`**`fnode`**(*index*, *position*)
FEHM grid node object.

2.1.1 Geometry attributes

`fnode.`**`index`**
(*int*) Integer number denoting the node.

`fnode.`**`position`**
(*lst[fl64]*) List of the node's coordinates in 2- or 3-D space.

`fnode.`**`vol`**
(*fl64*) Control volume associated with the node. This information only available if volumes() method called from grid attribute.

`fnode.`**`connected_nodes`**
(*lst[fnode]*) List of node objects connected to this node. This information only available if full_connectivity=True passed to fgrid.read()

`fnode.`**`connections`**
(*lst[fconn]*) List of connection objects of which the node is a member.

fnode.**elements**
(*lst[felem]*) List of element objects of which the node is a member.

2.1.2 Material property attributes

fnode.**permeability**
(*list*) permeability values at node.

fnode.**conductivity**
(*list*) conductivity values at node.

fnode.**density**
(*fl64*) density at node.

fnode.**specific_heat**
(*fl64*) specific heat at node.

fnode.**porosity**
(*fl64*) porosity at node.

fnode.**youngs_modulus**
(*fl64*) Youngs modulus at node.

fnode.**poissons_ratio**
(*fl64*) Poissons ratio at node.

fnode.**thermal_expansion**
(*fl64*) Coefficient of thermal expansion at node.

fnode.**pressure_coupling**
(*fl64*) Biot pressure coupling coefficient at node.

fnode.**rlpmodel**
(*int*) index of relative permeability model assigned to node

fnode.**permmodel**
(*int*) index of stress-permeability model assigned to node.

fnode.**pormodel**
(*int*) index of variable porosity model assigned to node.

fnode.**condmodel**
(*int*) index of variable conductivity model assigned to node.

2.1.3 State attributes

fnode.**Pi**
(*fl64*) initial pressure at node.

fnode.**Ti**
(*fl64*) initial temperature at node.

fnode.**Si**
(*fl64*) initial water saturation at node.

fnode.**S_co2gi**
(*fl64*) initial gaseous CO2 saturation at node.

fnode.**S_co2li**
(*fl64*) initial liquid CO2 saturation at node.

fnode.**co2aqi**
(*fl64*) initial dissolved CO2 concentration at node.

fnode.**strsi**
(*fl64*) initial stresses at node.

fnode.**dispi**
(*fl64*) initial displacements at node.

fnode.**P**
(*fl64*) pressure at node during a simulation.

fnode.**T**
(*fl64*) temperature at node.

fnode.**S**
(*fl64*) water saturation at node.

fnode.**S_co2g**
(*fl64*) gaseous CO2 saturation at node.

fnode.**S_co2l**
(*fl64*) liquid CO2 saturation at node.

fnode.**co2aq**
(*fl64*) dissolved CO2 concentration at node.

fnode.**strs**
(*list*) stresses at node ([xx,yy,xy] for 2D, [xx,yy,zz,xy,yz,xz] for 3D).

fnode.**disp**
(*list*) displacements at node ([x,y] for 2D, [x,y,z] for 3D).

2.1.4 Other attributes

fnode.**zone**
(*dict*) Dictionary of zones to which the node belongs.

fnode.**zonelist**
(*lst[fzone]*) List of zones of which the node is a member

fnode.**generator**
(*dict*) Dictionary of generator properties associated with node.

2.1.5 Methods

fnode.**what**
Print to screen information about the node.

2.2 Connections

class fgrid.**fconn**(*nodes*)
Connection object, comprising two connected nodes, separated by some distance.

A connection is associated with a distance between the two nodes.

2.2.1 Attributes

`fconn.`**`nodes`**
(*lst[fnode]*) List of node objects (fnode()) that define the connection.

`fconn.`**`distance`**
(*fl64*) Distance between the two connected nodes.

2.3 Elements

class `fgrid.`**`felem`**(*index=None*, *nodes=*[ ])
Finite element object, comprising a set of connected nodes.

A finite element is associated with an element centre and an element volume.

2.3.1 Attributes

`felem.`**`index`**
(*int*) Integer number denoting the element.

`felem.`**`nodes`**
(*lst[fnode]*) List of node objects that define the element.

`felem.`**`centre`**
(*ndarray*) Coordinates of the element centroid.

2.3.2 Methods

`felem.`**`what`**
Print to screen information about the element.

2.4 Grids

The `fgrid` object contains all information about the finite element grid.

The grid object corresponds to an FEHM grid file and comprises an assembly of `fnode`, `fconn` and `felem` objects. This assembly is constructed by reading an existing FEHM grid files (`fgrid.read()`) or by creating an empty `fgrid` object and creating a new mesh using the `fgrid.make()` command. Read and write support for FEHM stor files is supported.

class `fgrid.`**`fgrid`**(*full_connectivity=True*)
FEHM grid object.

2.4.1 Attributes: object lists

`fgrid.`**`node`**
(*dict[fnode]*) Dictionary of grid nodes, indexed by node integer.

`fgrid.`**`nodelist`**
(*lst[fnode]*) List of all node objects in the grid.

fgrid.**conn**
(*dict[fconn]*) Dictionary of connections, indexed by a two element tuple of the member node integers.

fgrid.**connlist**
(*lst[fconn]*) List of all connection objects in the grid.

fgrid.**elem**
(*dict[felem]*) Dictionary of elements, indexed by element integer.

fgrid.**elemlist**
(*lst[felem]*) List of all element objects in the grid.

2.4.2 Attributes: grid properties

fgrid.**xmin**
Minimum x-coordinate for all nodes.

fgrid.**xmax**
Maximum x-coordinate for all nodes.

fgrid.**ymin**
Minimum y-coordinate for all nodes.

fgrid.**ymax**
Maximum y-coordinate for all nodes.

fgrid.**zmin**
Minimum z-coordinate for all nodes.

fgrid.**zmax**
Maximum z-coordinate for all nodes.

fgrid.**dimensions**
(*int*) Dimensions of the grid.

fgrid.**number_nodes**
Number of nodes in grid.

fgrid.**number_elems**
Number of elements in grid.

2.4.3 Methods

fgrid.**read**(*gridfilename*, *full_connectivity=True*, *octree=False*, *storfilename=None*)
Read data from an FEHM or AVS grid file. If an AVS grid is specified, PyFEHM will write out the corresponding FEHM grid file.

Parameters

- **gridfilename** (*str*) – name of grid file, including path specification.
- **full_connectivity** (*bool*) – read element and connection data and construct corresponding objects. Defaults to False. Use if access to connectivity information will be useful.
- **octree** (*bool*) – flag to use octree search algorithm for finding node locations (default = False).
- **storfilename** (*bool*) – name of optional stor file, including path specification.

fgrid.**write**(*filename=None*, *format='fehm'*, *remove_duplicates=True*, *recalculate_coefficients=True*, *compress_eps=1e-05*)

Write grid object to a grid file (FEHM, AVS STOR file formats supported). Stor file support only for orthogonal hexahedral grids.

Parameters

- **filename** (*str*) – name of FEHM grid file to write to, including path specification, e.g. c:pathfile_out.inp
- **format** (*str*) – FEHM grid file format ('fehm','avs','stor'). Defaults to 'fehm' unless filename is passed with extension '*.avs' or* '.stor'.
- **remove_duplicates** (*bool*) – Remove duplicate entries in the stor file. Improves simulation run time, particularly for structured grids.
- **compress_eps** (*fl64*) – Float used to determine geometric coefficients to remove (Coefficients less than max(geom_coef)*compress_eps will be removed)
- **keepcons** (*lst(connection keys)*) – List of connections to keep in stor file no matter what

fgrid.**node_nearest_point**(*pos=[]*)

Return node object nearest to position in space. Method uses octree structure for speed up if available.

Parameters **pos** (*list*) – Coordinates, e.g. [2300., -134.8, 0.].

Returns fnode() – node object closest to position.

fgrid.**plot**(*save='', angle=[45, 45], color='k', connections=False, equal_axes=True, xlabel='x / m', ylabel='y / m', zlabel='z / m', title='', font_size='small', cutaway=[], zones=[]*)

Generates and saves a 3-D plot of the grid.

Parameters

- **save** (*str*) – Name of saved zone image.
- **angle** (*[fl64,fl64], str*) – View angle of zone. First number is tilt angle in degrees, second number is azimuth. Alternatively, if angle is 'x', 'y', 'z', view is aligned along the corresponding axis.
- **color** (*str, [fl64,fl64,fl64]*) – Colour of zone.
- **connections** (*bool*) – Plot connections. If `True` all connections plotted. If between 0 and 1, random proportion plotted. If greater than 1, specified number plotted.
- **equal_axes** (*bool*) – Force plotting with equal aspect ratios for all axes.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **zlabel** (*str*) – Label on z-axis.
- **title** (*str*) – Title of plot.
- **font_size** (*str, int*) – Size of text on plot.
- **cutaway** (*[fl64,fl64,fl64], str*) – Coordinate from which cutaway begins. Alternatively, specifying 'middle','centre' with choose the centre of the grid as the cutaway point.
- **zones** (*int, str*) – List of zone indices or names. If these are defined then nodes contained in those zones are highlighted in the output plot. Maximum of six zones.

fgrid.**make**(*gridfilename*, *x*, *y*, *z*, *full_connectivity=True*, *octree=False*)

Generates an orthogonal mesh for input node positions.

The mesh is constructed using the `fgrid.`**fmake** object and an FEHM grid file is written for the mesh.

Parameters

- **gridfilename** (*str*) – Name to which to save the grid file.
- **x** (*list[fl64]*) – Unique set of x-coordinates.
- **y** (*list[fl64]*) – Unique set of y-coordinates.
- **z** (*list[fl64]*) – Unique set of z-coordinates.
- **full_connectivity** (*bool*) – read element and connection data and construct corresponding objects. Defaults to False. Use if access to connectivity information will be useful.

fgrid.**lagrit_stor**(*grid=None*, *stor=None*, *exe='c:\path\to\lagrit\lagrit.exe'*, *overwrite=False*)
Uses LaGriT to create a stor file for the simulation, this will be used in subsequent runs. To create the stor file, LaGriT will convert a mesh comprised of hexahedral elements into one comprising only tetrahedrals. Therefore, a new FEHM grid file will be created and parsed, reflecting the modified element structure.

Parameters

- **grid** (*str*) – Name of grid file to be created. Destination directory supported.
- **stor** (*str*) – Name of stor file to be created. Destination directory supported.
- **exe** (*str*) – Path to lagrit executable (default, fdflt.lagrit_path, in environment file 'fdflt.py').
- **overwrite** (*bool*) – Flag to request that the new FEHM grid file overwrites the old one.

fgrid.**volumes**(*volumefilename*)
Reads a lagrit generated file containing control volume information.

Parameters **volumefilename** (*str*) – Name of lagrit output file containing control volume information.

fgrid.**remove_zeros**(*tolerance=1e-08*, *keepcons=[]*)
Removes node connections with geometric coefficients smaller than a supplied tolerance.

Parameters

- **tolerance** (*fl64*) – Relative tolerance below which connections will be removed (default = 1.e-8).
- **keepcons** (*lst(connection keys) or lst(connections)*) – List of connections to keep in stor file ignoring tolerance.

fgrid.**what**
Print to screen information about the grid.

2.4.4 Examples

1. Create an `fgrid` object and read an existing FEHM grid file.

   ```
   geo=fgrid()
   geo.read('c:\\path\\to\\old_GRID.inp')
   ```

   or

   ```
   geo=fgrid('c:\\path\\to\\old_GRID.inp')
   ```

2. Plot a view of the grid looking down the x-axis, with gridlines coloured blue.

   ```
   geo.plot('FEHMgrid1.png',color='r',angle='x')
   ```

3. Plot a view of the grid looking along the axis x=y=z, gridlines coloured red, with a cutaway beginning at the centre.

```
geo.plot('FEHMgrid2.png',color='b',angle=[45,45],cutaway='middle')
```

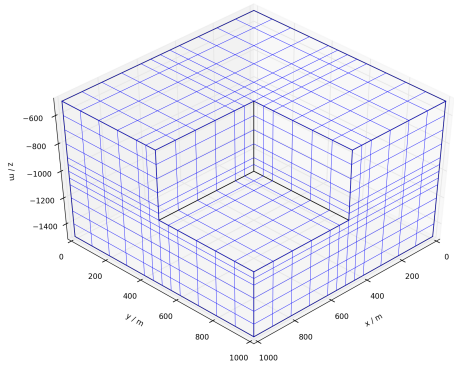


Figure 2.1: Image produced by example 3.

4. Make changes to an `fgrid` object and write out the changes to a new FEHM grid file.

   ```
   for nd in geo.nodelist:  nd.position[2]+=1000.
   geo.write('new_GRID.inp')
   ```

5. Find the node nearest a given location in space.

   ```
   nd=geo.node_nearest_point([225,1600,-356])
   ```

6. Create a new grid using the `fgrid.make()` command.

   ```
   geo.make('mygrid.inp', x=[0,1,2,3], y=[-10,-20,-30], z=[0.1,0.3,0.8])
   ```

7. For an existing grid, use `lagrit_stor()` method to instruct LaGriT to create a stor file.

   ```
   geo.lagrit_stor(overwrite=True, stor='stor\\geo.stor',
   exe='c:\\bin\\lagrit.exe')
   ```

CHAPTER

THREE

FDATA: FEHM INPUT FILE MANIPULATION

This module contains classes and methods for the reading, writing, construction and manipulation of FEHM input files.

An FEHM input file corresponds in PyFEHM to a collection of zones (`fzone`), macros (`fmacro`), boundary conditions (`fboun`), initial conditions (`fincon`) and other objects. These are all linked together, along with an FEHM grid object (`fgrid`), to the central `fdata` structure. Within this framework, zones are linked to macros, nodes are linked to zones and initial conditions, etc. The user is permitted to, say, choose a node and establish which zones and macros it is linked to.

For the purposes of this manual, the variable `dat` will be assumed to refer to a previously defined `fdata` object.

3.1 Zones

Zone object, a tool for defining sets of nodes to which specific properties can be assigned via various macros. (see macro FEHM user manual macro **ZONE** or **ZONN**). For the purposes of this manual, the variable `zn` will be assumed to refer to a previously defined `fzone` object.

There are several ways to define a zone in FEHM. The default definition (here, assigned `fzone.type = 'rect'`) is that of a box, edges aligned along the coordinate axes, that contains all nodes desired to be in that zone. Alternative definitions include listing those nodes to be included in the zone (`type = 'nnum'`) or the coordinates of those nodes (`type = 'list'`).

Zones are created and associated with an `fdata` object using `fdata.add()`. All zones added to the `fdata` object appear in `fdata.zonelist` and `fdata.zone` attributes. Zones in `fdata.zone` are accessed by either zone index or zone name, e.g., `fdata.zone['xmin']` or `fdata.zone[12]`.

class `fdata.`**`fzone`**(*index=None*, *type=''*, *points=[]*, *nodelist=[]*, *file=''*, *name=''*)
FEHM Zone object.

3.1.1 Attributes

`fzone.`**`index`**
(*int*) Integer number denoting the zone.

`fzone.`**`type`**
(*str*) String denoting the zone type. Default is 'rect', alternatives are 'list', 'nnum'

`fzone.`**`name`**
(*str*) Name of the zone. Will appear commented beside the zone definition in the input file. Can be used to index the `fdata.zone` attribute.

`fzone.`**`node`**
(*dict[fnode]*) Dictionary of nodes contained within the zone, indexed by node number.

`fzone.`**`nodelist`**
(*lst[fnode]*) List of nodes contained within the zone.

`fzone.`**`file`**
(*str*) File name if zone data is or is to be contained in a separate file. If file does not exist, it will be created and written to when the FEHM input file is being written out.

3.1.2 Material attributes

In assigning these attributes, macros corresponding to the zone object will be automatically created. For example, assigning a value to the `permeability` attribute will create the corresponding permeability macro.

`fzone.`**`permeability`**
(*fl64*,*lst*) Permeability properties of zone.

`fzone.`**`conductivity`**
(*fl64*,*lst*) Conductivity properties of zone.

`fzone.`**`density`**
(*fl64*) Density of zone.

`fzone.`**`specific_heat`**
(*fl64*) Specific heat of zone.

`fzone.`**`porosity`**
(*fl64*) Porosity of zone.

`fzone.`**`youngs_modulus`**
(*fl64*) Young's modulus of zone.

`fzone.`**`poissons_ratio`**
(*fl64*) Poisson's ratio of zone.

`fzone.`**`thermal_expansion`**
(*fl64*) Coefficient of thermal expansion of zone.

`fzone.`**`pressure_coupling`**
(*fl64*) Pressure coupling term of zone.

`fzone.`**`Pi`**
(*fl64*) Initial pressure in zone.

`fzone.`**`Ti`**
(*fl64*) Initial temperature in zone.

`fzone.`**`Si`**
(*fl64*) Initial saturation in zone.

3.1.3 Methods

`fzone.`**`rect`**(*p1*, *p2*)
Create a rectangular zone corresponding to the bounding box delimited by p1 and p2.

Parameters

- **p1** (*ndarray*) – coordinate of first corner of the bounding box.
- **p2** (*ndarray*) – coordinate of the second corner of the bounding box.

`fzone.`**`fix_temperature`**(*T*, *multiplier=10000000000.0*, *file=None*)

Fixes temperatures at nodes within this zone. Temperatures fixed by adding an HFLX macro with high heat flow multiplier.

Parameters

- **T** (*fl64*) – Temperature to fix.
- **multiplier** (*fl64*) – Multiplier for HFLX macro (default = 1.e10)
- **file** (*str*) – Name of auxiliary file to save macro.

`fzone.`**`fix_pressure`**(*P=0*, *T=30.0*, *impedance=1000000.0*, *file=None*)

Fixes pressures at nodes within this zone. Pressures fixed by adding a FLOW macro with high impedance.

Parameters

- **P** (*fl64*) – Pressure to fix. Default is 0, corresponding to fixing initial pressure.
- **T** (*fl64*) – Temperature to fix (default = 30 degC).
- **impedance** (*fl64*) – Impedance for FLOW macro (default = 1.e6)
- **file** (*str*) – Name of auxiliary file to save macro.

`fzone.`**`plot`**(*save=''*, *angle=[45, 45]*, *color='k'*, *connections=False*, *equal_axes=True*, *xlabel='x / m'*, *ylabel='y / m'*, *zlabel='z / m'*, *title=''*, *font_size='small'*)

Generates and saves a 3-D plot of the zone.

Parameters

- **save** (*str*) – Name of saved zone image.
- **angle** (*[fl64,fl64], str*) – View angle of zone. First number is azimuth angle in degrees, second number is tilt. Alternatively, if angle is 'x', 'y', 'z', view is aligned along the corresponding axis.
- **color** (*str, [fl64,fl64,fl64]*) – Colour of zone.
- **connections** (*bool*) – Plot connections. If `True` all connections plotted. If between 0 and 1, random proportion plotted. If greater than 1, specified number plotted.
- **equal_axes** (*bool*) – Force plotting with equal aspect ratios for all axes.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **zlabel** (*str*) – Label on z-axis.
- **title** (*str*) – Title of plot.
- **font_size** (*str, int*) – Size of text on plot.

Example:

```
zn.plot(save='myzone.png', angle = [45,45], xlabel = 'x / m', font_size =
'small', color = 'r')
```

fzone.**topo**(*save=''*, *cbar=True*, *equal_axes=True*, *method='nearest'*, *divisions=[30, 30]*, *xlims=[]*, *ylims=[]*, *clims=[]*, *levels=10*, *clabel=''*, *xlabel='x / m'*, *ylabel='y / m'*, *zlabel='z / m'*, *title=''*, *font_size='small'*)

Returns a contour plot of the top surface of the zone.

Parameters

- **divisions** (*[int,int]*) – Resolution to supply mesh data.
- **method** (*str*) – Method of interpolation, options are 'nearest', 'linear'.
- **levels** (*lst[fl64], int*) – Contour levels to plot. Can specify specific levels in list form, or a single integer indicating automatic assignment of levels.
- **cbar** – Add colour bar to plot.
- **type** – bool
- **xlims** (*[fl64, fl64]*) – Plot limits on x-axis.
- **ylims** (*[fl64, fl64]*) – Plot limits on y-axis.
- **clims** (*[fl64,fl64]*) – Colour limits.
- **save** (*str*) – Name to save plot. Format specified extension (default .png if none give). Supported extensions: .png, .eps, .pdf.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **title** (*str*) – Plot title.
- **font_size** (*str, int*) – Specify text size, either as an integer or string, e.g., 10, 'small', 'x-large'.
- **equal_axes** (*bool*) – Specify equal scales on axes.

Example:

```
dat.zone[2].topo('zoneDEMtopo.png',method = 'linear')
```

fzone.**what**

Print to screen information about the zone.

3.1.4 Examples

1. Create a rectangular zone that encompasses all nodes within a horizontal reservoir. Bounding box limits are used as inputs for `rect()`.

```
zn = fzone( type ='rect', index = 5, name = 'res')

zn.rect([0.1,0.1,2200.],[5000.1,5000.1,2400.])

dat.add(zn)
```

2. Create a zone named *injectors* with index 10 and comprising two previously identified nodes. The zone information will be written out to the auxiliary file *zones.macro*. The zone is created and added to the `dat` in a single step.

```
nd1 = geo.node_nearest_point([100,200,-1500])

nd2 = geo.node_nearest_point([-500,400,-1500])

dat.add(fzone(index=10,          type='nnum',          name='injectors',
nodelist=[nd1.index, nd2], file='zones.macro'))
```

3. For the zone named `zmax`, fix its temperature to 90degC.

   ```
   dat.zone['zmax'].fix_temperature(T=90.)
   ```

4. Assign anisotropic permeability to the zone `caprock`

   ```
   dat.zone['caprock'].permeability=[1.e-19,1.e-19,1.e-21]
   ```

3.2 Macros

The macro object, this is how the majority of FEHM's text inputs are defined.

A macro is a way of telling FEHM to do something specific. It may be as simple as assigning a standard permeability value to every node in the grid. It may be as specific as assigning a stress boundary condition to one edge of the model, a source or sink to one or many nodes, or elastic and rock properties to a particular zone.

The attribute `type` is set when a new macro is created. Macro names in PyFEHM and FEHM are identical. Thus, to generate a macro object for permeability properties, one supplies the command `fmacro('perm')`.

Macros are applied to a specific spatial domain. This may be the entire model, a previously defined zone, or a set of nodes. This spatial assignment occurs is defined in `fmacro.zone`.

The macro must be supplied with several pieces of information, informing its operation. E.g., the perm macro requires permeabilities, the rock macro requires density, porosity and specific heat, and the grad macro requires information about an initial gradient in some variable. These properties are set in `fmacro.param`, a dictionary with keys corresponding to the required pieces of information.

class `fdata.`**`fmacro`**(*type=''*, *zone=[]*, *param=[]*, *subtype=''*, *file=None*, *write_one_macro=False*)
FEHM macro object.

3.2.1 Attributes

`fmacro.`**`type`**
(*str*) Name of the macro. Macro names are identical to those invoked in FEHM.

`fmacro.`**`zone`**
(*fzone, lst[fzone], tuple[int,int,int], zone key*) The zone, zones or nodes to which the macro is assigned. Note, only permmodel and rlp can be assigned lists of zones. Optionally, a key (index or string) may be passed, in which case the zone will be retrieved when the macro is added to the model.

`fmacro.`**`param`**
(*dict[fl64]*) A dictionary of values defining the operation of the macro. See table below for macro-specific dictionary keys.

`fmacro.`**`subtype`**
(*str*) Macro subtype, required for **STRESSBOUN** macro.

`fmacro.`**`file`**
(*str*) File string where information about the macro is stored. If file does not currently exist, it will be created and written to when the FEHM input file is written.

3.2.2 Methods

`fmacro.`**`what`**
Return a summary of the macros function.

3.2.3 Macro parameter lists

The table below shows the parameter names available to be set for each macro. Parameters are accessed/set via the `param` attribute, e.g., to set the *density* property for a **ROCK** macro write

```
mcr = fmacro('rock')
mcr.param['density']=2500
```

Warnings will be printed if a parameter is not set when the FEHM data file is written.

Macro	Parameters (Units)	FEHM variable	Notes
pres	pressure (*MPa*)	PHRD	
	temperature (*degC*)	TIND	
	saturation	IEOSD	
perm	kx (*m^2, log10(m^2)*)	PNXD	
	ky	PNYD	
	kz	PNZD	
cond	cond_x (*W/m/K*)	THXD	
	cond_y	THYD	
	cond_z	THZD	
flow	rate (*kg/s, MPa*)	SKD	Fixed rate or pressure depending on AIPED.
	energy (*MJ/kg, degC*)	EFLOW	>0 = enthalpy, <0 = temperature.
	impedance	AIPED	Determine generator type.
rock	density (*kg/m^3*)	DENRD	
	specific_heat (*J/kg/K*)	CPRD	
	porosity	PSD	Set negative to remove node from simulation.
grad	reference_coord (*m*)	CORDG	
	direction	IDRG	1,2,3 = x,y,z
	variable	IGRADF	1,2,3,9 = P,T,S,P_co2 / 4,5,6,10 = fixed boun P,T,S,P_co2
	reference_value (*unit*)	VAR0	
	gradient (*unit/m*)	GRAD1	
hflx	heat_flow (*MW*)	QFLUX	
	multiplier	QFLXM	
biot	thermal_expansion (*/K*)	ALPHA	
	pressure_coupling	PP_FAC	
elastic	youngs_modulus (*MPa*)	ELASIC_MOD	
	poissons_ratio	POISSON	
co2frac	water_rich_sat	FW	
	co2_rich_sat	FL	Set non-zero at co2 injectors.
	co2_mass_frac	YC	
	init_salt_conc (*ppm*)	CSALT	
	override_flag	INICO2FLG	
co2flow	rate (*kg/s, MPa*)	SKTMP	
	energy (*MJ/kg, degC*)	ESKTMP	
	impedance	AIPED	Similar operation to *flow*.
	bc_flag	IFLG_FLOWMAC	
co2diff	diffusion	DIFF	
	tortuosity	TORTCO2	
co2pres	pressure (*MPa*)	PHICO2	
	temperature (*degC*)	TCO2	
	phase	ICES	
stressboun	value (*m, MPa, MPa/m*)	BOUNVAL	Fixed displacement, force or stress gradient.

Continued on next page

Table 3.1 – continued from previous page

Macro	Parameters (Units)	FEHM variable	Notes
	direction	KQ	1,2,3 = x,y,z displacement, -1,-2,-3 = x,y,z force.
	sdepth (*m*)	SDEPTH	Only used for `subtype='lithograd'`.
	gdepth (*m*)	GDEPTH	Only used for `subtype='lithograd'`.

3.2.4 Examples

1. An initial conditions macro, **PRES**, is created, and assigned to the pre-existing zone 10. The macro is assigned parameters, found by looking up *pres* in the table above, and added to `dat`.

```
mcr=fmacro('pres')
mcr.zone=dat.zone[10]
mcr.param['pressure'] = 1.
mcr.param['temperature'] = 80.
mcr.param['saturation'] = 1.
dat.add(mcr)
```

2. A rock properties macro, **ROCK**, is created, assigned zone 20 and properties, and added to `dat`.

```
mcr=fmacro('rock', zone=20, param=(('density',2500), ('porosity',0.1),
('specific_heat',800)))
dat.add(mcr)
```

3. A mass source macro, **FLOW**, is created, assigned to the *injection* zone, assigned properties, and added to `dat` in a single command.

```
dat.add(fmacro('flow', zone='injection', param=(('rate',-2),
('energy',-50), ('impedance',100))))
```

4. Creation of a lithograd stress boundary condition, **STRESSBOUN**.

```
mcr=fmacro('stressboun', zone='xmin', subtype='lithograd')
mcr.param['sdepth'] = 0.
mcr.param['gdepth'] = 0.
mcr.param['direction'] = 1
mcr.param['value'] = 0.03
```

3.3 Models

The model object, this is how the some of FEHM's text inputs are defined.

Models are similar to macros in function but are more flexible in their definition of physical behaviour. For example, the definition of thermal conductivity through the COND macro is rather narrow; constant values are supplied. However, by invoking the VCON macro, the user can choose between multiple models for thermal conductivity behaviour. The user can specify a model index - say 2, for square root variation of thermal conductivity with liquid saturation - and then pass the necessary parameters for the model.

Additional models exist to define relative permeability in terms of saturation (RLP), porosity in terms of fluid pressure (PPOR), or permeability in terms of stress (PERMMODEL).

class fdata.**fmodel**(*type=''*, *zonelist=[ ]*, *param=[ ]*, *index=None*, *file=None*)
Model object, used in a variety of macro definitions.

Model objects should have:

- a type for the macro
- a list of zones to which the model is assigned.
- an index for the specific model.
- a dictionary of parameters for the model.

3.3.1 Attributes

fmodel.**type**
(*str*) Name of the macro for this model. Macro names are identical to those invoked in FEHM.

fmodel.**zonelist**
(*list*) A list of zones to which the model is applied.

fmodel.**param**
(*dict[fl64]*) A dictionary of values defining the operation of the macro. See table below for macro-specific dictionary keys.

fmodel.**index**
(*int*) Index of the model to be invoked.

fmodel.**file**
(**)

3.3.2 Available models

Currently, models are available for thermal conductivity, permeability, porosity and relative permeability. Information about the various models and their parameters is available through the fdata.help attribute.

3.3.3 Examples

1. Adding a model in which thermal conductivity depends linearly upon temperature.

```
vcon=fmodel(type='vcon',index=1)
vcon.zonelist=dat.zone['salt']
vcon.param['T_ref']=30.
vcon.param['cond_ref']=2.
vcon.param['dcond_dT']=0.2
dat.add(bc)
```

2. Creation of a relative permeability model number 2, **RLP**, for the entire model (default when zone is not specified). See appendix for model-specific *rlp* parameter names.

```
rlp=fmodel('rlp',index = 2,              param=(('resid_liq_sat',0.2),
('resid_vap_sat',0.), ('cap_pres_zero_sat',1.e4),
('sat_zero_cap_pres',1.))
```

3. Accessing help on all permeability models.

   ```
   dat.help.permmodel()
   ```

4. Accessing help on variable porosity model -1.

   ```
   dat.help.pormodel(-1)
   ```

3.4 Boundary conditions

Boundary conditions are applied to zones, and involve fixing some variable according to supplied time-series data. In contrast to other macros, one boundary condition may be applied to multiple zones.

The `fboun` object is supplied a vector of times, corresponding to values of one or more variables. This defines the time evolution of the variable, with step-changes (`fboun.type='ti'`) or linear interpolation (`type='ti_linear'`) between elements in `fboun.times`.

The `fboun.variable` attribute comprises a list of variable vectors. Each variable vector first contains the FEHM boun variable keyword followed by the vector of values corresponding to `fboun.times`. For example, a fixed temperature sequence would be supplied `variable= [['t',100,80,90],]` (note the nesting of the lists).

class `fdata.`**`fboun`**(*zone=*[ ], *type='ti'*, *times=*[ ], *variable=*[ ], *file=None*)
Boundary condition object.

3.4.1 Attributes

`fboun.`**`type`**
(*str*) Boundary condition type, 'ti' = step changes, 'ti_linear' = linear changes.

`fboun.`**`zone`**
(*lst[fzone]*) List of zones to which the boundary condition is to be applied.

`fboun.`**`variable`**
(*lst[lst[str,fl64,...]]*) List of lists of boundary data. Each list begins with a string denoting the boundary condition variable, and is followed by the time-series data, e.g., ['sw',0.2,0.5,0.8].

`fboun.`**`times`**
(*lst[fl64]*) Vector of times.

`fboun.`**`n_times`**
(*int*) Length of time-series.

3.4.2 Methods

`fboun.`**`what`**
Print information about the boundary condition object.

3.4.3 Examples

1. Create a boundary condition for a fixed temperature sequence and add it to `dat`.

```
bc=fboun(type='ti')
bc.zone=dat.zone['base']
bc.times=[0,10,100,1000]
bc.variable= [['t',20,30,50,80]]
dat.add(bc)
```

2. Create a boundary condition for both temperature and pressure. Use linear interpolation between times.

```
bc=fboun(type='ti_linear', zone=2, times=[0,10,20],
variable=[['t',100,80,100], ['pw',15,10,15]])
```

3.5 History output

History output object, makes request for history data to be output (see macro **HIST**).

This data type outputs specified variables (e.g., temperature, permeability) for particular nodes and zones at specified times (one file = one variable). The data can be output in several formats (all of which are readable by PyFEHM) and at specified times.

Output requests for specific nodes and zones is supplied through the **NODE** and **FLXZ** macros. PyFEHM writes takes care of these macros within the scope of the `fhist` object.

class `fdata.`**`fhist`** (*format='tec'*, *timestep_interval=1*, *time_interval=1e+30*, *variables*=[ ], *nodelist*=[ ], *zonelist*=[ ], *zoneflux*=[ ])
FEHM history output object.

3.5.1 Attributes

`fhist.`**`format`**
(*str*) File format for contour output: 'tecplot', 'csv', 'surfer'

`fhist.`**`time_interval`**
(*flt*) Time interval to output data.

`fhist.`**`timestep_interval`**
(*int*) Time step interval to output data.

`fhist.`**`variables`**
(*lst[str]*)List of variables to write contour data for, e.g., ['temperature','pressure']

`fhist.`**`node`**
(*dict[fnode]*) Dictionary of nodes, indexed by node number, for which history output requested.

`fhist.`**`zone`**
(*dict[fzone]*) Dictionary of zones, indexed by number and name, for which history output is required.

`fhist.`**`zoneflux`**
(*lst[fzone,int]*) List of zone objects for which zone flux output required.

`fhist.`**`nodelist`**
(*list[fnode]*) list of node objects for which history output requested.

`fhist.`**`zonelist`**
(*lst[fzone]*) List of zone objects for which history output required.

3.5.2 Methods

`fhist.`**`options`**
Print out eligible variables for output.

`fhist.`**`what`**
Print out information about the attribute.

3.5.3 Examples

1. Request temperature, pressure and flow history output for specific nodes at every time step.

```
dat.hist.variables.append(['temperature','pressure','flow'])
dat.hist.timestep_interval=1
dat.hist.nodelist.append([10, 15, 20])
nd = dat.grid.node_nearest_point([200,300,400])
dat.hist.nodelist.append(nd)
```

2. Request CO2 and water source/sink history output for specific zones every 10 days.

```
dat.hist.variables.append(['cflz','zfl'])
dat.hist.time_interval=10
dat.hist.zoneflux.append(10)
dat.hist.nodelist.append(dat.zone['injection'])
```

3.6 Contour ouput

class `fdata.`**`fcont`**(*format='surf'*, *timestep_interval=1000*, *time_interval=1e+30*, *time_flag=True*, *variables=[]*, *zones=[]*)
Contour output object, makes request for contour data to be output (see macro **CONT**).

This data type outputs specified variables (e.g., temperature, permeability) for x,y,z or node locations at fixed times (one file = one time). The data can be output in several formats (all of which are readable by PyFEHM) and at specified times.

3.6.1 Attributes

`fcont.`**`format`**
(*str*) File format for contour output: 'tec', 'surf', 'avs', 'avsx'

`fcont.`**`time_interval`**
(*flt*) Time interval to output data.

`fcont.`**`timestep_interval`**
(*int*) Time step interval to output data.

`fcont.`**`variables`**
(*lst[str]*)List of variables to write contour data for, e.g., ['temperature','pressure']

`fcont.`**`time_flag`**
(*bool*) Set to True to include in output title.

fcont.**zones**
(*lst[fzone]*) List of zone objects for which contour data is to be written - **NOT FULLY SUPPORTED**.

3.6.2 Methods

fcont.**options**
Print out eligible variables for output.

fcont.**what**
Print out information about the attribute.

3.6.3 Examples

1. Request contour data to be output every 10 time steps, every 100 days, in surfer format.

```
dat.cont.timestep_interval=10
dat.cont.time_interval=100
dat.cont.format='surf'
dat.cont.variables.append(['temperature','permeability','displacement','co2'])
dat.cont.time_flag=True
```

2. The user can force contour output to be printed at specific times by setting the :attr: *.fdata.output_times* attribute to a vector of times

```
dat.output_times=powspace(1.e1,1.e4,15,2.)
```

3.7 Data files

The fdata class is the central structure for collating and organising all information about a simulation. Grid information, zone definitions, boundary conditions and other macros are collected here. An fdata object is automatically created and populated when an existing input file is parsed using fdata.read(). Alternatively, an empty fdata object can be created and populated with an fgrid object, some fzone, fmacro and other objects to create a new simulation.

Just as each FEHM input file is associated with a grid file, each PyFEHM fdata object must be associated with an fgrid object. Failure to specify a grid file may lead to problems specifying other macros.

FEHM simulations can be initialised within the PyFEHM environment by writing out the *fehmn.files* control file and supplying a command line path to an FEHM executable. Information contained in the control file is accessed via the files attribute. The control file is automatically written out when invoking fdata.run().

PyFEHM also allows for simulation restarts from initial condition files dumped by FEHM, referred to here as *incon* files. An incon file is associated with the incon attribute and read in by fincon.read(). The user is permitted to read in incon files, make changes within PyFEHM, and write out a new incon file for a future simulation.

PyFEHM supports the use of FEHM's stress and carbon dioxide modules via the strs and carb attributes, respectively.

class fdata.**fdata**(*filename=''*, *gridfilename=''*, *inconfilename=''*, *sticky_zones=True*, *associate=True*, *work_dir=None*, *full_connectivity=True*, *skip=[]*, *keep_unknown=True*)
Class for FEHM data file.

fdata.**filename**
(*str*) File name for reading and writing FEHM input text file.

`fdata.`**`gridfilename`**
(*str*) File name of FEHM grid file.

`fdata.`**`inconfilename`**
(*str*) File name of FEHM restart file.

3.7.1 Attributes: daughter structures

`fdata.`**`grid`**
(*fgrid*) Grid object associated with the model.

`fdata.`**`cont`**
(*fcont*) Contour output for the model.

`fdata.`**`hist`**
(*fhist*) History output for the model.

`fdata.`**`incon`**
(*fincon*) Initial conditions (restart file) associated with the model.

`fdata.`**`files`**
(*files*) Simulation execution object.

`fdata.`**`strs`**
(*fstrs*) Stress module object.

`fdata.`**`carb`**
(*fcarb*) CO2 module object.

`fdata.`**`trac`**
(*ftrac*) Species transport module object.

`fdata.`**`bounlist`**
(*lst[fzone]*) List of boundary condition objects in the model.

`fdata.`**`zone`**
(*dict[fzone]*) Dictionary of zone objects, indexed by zone number or name.

`fdata.`**`zonelist`**
(*lst[fzone]*) List of zone objects in the model.

`fdata.`**`help`**
(*fhelp*) Module for interactive assistance.

3.7.2 Attributes: macro lists

An FEHM simulation may contain multiple instances of the same macro, which allows assignment of differing material properties between regions, multiple regions of inflow and outflow via **FLOW** and **BOUN**, etc. Each set of macros is collected and made accessible via a list and a dictionary, denoted by `fdata.`*macro_namelist* and `fdata.`*macro_name* respectively. The list contains macros in which they were added to the `fdata` object or read from an input file. The dictionary is indexed by the zone number of name associated with each macro.

For example, to loop through all macros of the **PERM** type, one would use `fdata.permlist`

```
for perm in dat.permlist:  perm.param['kx'] = 1.e-16
```

To access a specific **ROCK** macro assigned to zone 20, use the `fdata.`**rock** dictionary

```
dat.rock[20].param['porosity'] = -1
```

To access **FLOW** generators assigned to the previously defined 'injection' zone, use the `fdata.flow` dictionary

```
dat.flow['injection'].param['impedance'] = 100
```

For reference, the available macro lists and dictionaries are given below.

fdata.**perm**

fdata.**permlist**

fdata.**pres**

fdata.**preslist**

fdata.**rock**

fdata.**rocklist**

fdata.**cond**

fdata.**condlist**

fdata.**grad**

fdata.**gradlist**

fdata.**flow**

fdata.**flowlist**

fdata.**hflx**

fdata.**hflxlist**

fdata.**biot**

fdata.**biotlist**

fdata.**elastic**

fdata.**elasticlist**

fdata.**co2frac**

fdata.**co2fraclist**

fdata.**co2flow**

fdata.**co2flowlist**

fdata.**co2pres**

fdata.**co2preslist**

fdata.**co2diff**

fdata.**co2difflist**

fdata.**stressboun**

fdata.**stressbounlist**

fdata.**permmodel**

fdata.**permmodellist**

fdata.**rlp**

fdata.**rlplist**

fdata.**rlpm**

fdata.**rlpmlist**

3.7.3 Attributes: parameter dictionaries

FEHM provides the user with control over a variety of time stepping and solver parameters. In PyFEHM, these parameters are accessible via four dictionaries, which are automatically generated and populated with defaults when a new `fdata` object is created. A table of parameter names and default values is given with each of the four dictionaries below.

`fdata.time`
(*dict[fl64,int]*) Time stepping parameters (see macro **TIME**).

PyFEHM parameter	FEHM equivalent	Default value
initial_timestep_DAY	DAY	1 (*days*)
max_time_TIMS	TIMS	365 (*days*)
max_timestep_NSTEP	NSTEP	200
print_interval_IPRTOUT	IPRTOUT	1
initial_year_YEAR	YEAR	None
initial_month_MONTH	MONTH	None
initial_day_INITTIME	INITTIME	None

`fdata.ctrl`
(*dict[fl64,int]*) Control parameters (see macro **CTRL**).

PyFEHM parameter	FEHM equivalent	Default value
max_newton_iterations_MAXIT	MAXIT	10
newton_cycle_tolerance_EPM	EPM	1e-5
number_orthogonalizations_NORTH	NORTH	8
max_solver_iterations_MAXSOLVE	MAXSOLVE	24
acceleration_method_ACCM	ACCM	‘gmre’
JA	JA	1
JB	JB	0
JC	JC	0
order_gauss_elim_NAR	NAR	2
implicitness_factor_AAW	AAW	1
gravity_direction_AGRAV	AGRAV	3
upstream_weighting_UPWGT	UPWGT	1.0
max_multiply_iterations_IAMM	IAMM	7
timestep_multiplier_AIAA	AIAA	1.5
min_timestep_DAYMIN	DAYMIN	1e-5 (*days*)
max_timestep_DAYMAX	DAYMAX	30 (*days*)
geometry_ICNL	ICNL	0
stor_file_LDA	LDA	0

`fdata.iter`
(*dict[fl64,int]*) Iteration parameters (see macro **ITER**).

PyFEHM parameter	FEHM equivalent	Default value
linear_converge_NRmult_G1	G1	1e-5
quadratic_converge_NRmult_G2	G2	1e-5
stop_criteria_NRmult_G3	G3	1e-3
machine_tolerance_TMCH	TMCH	-1e-5
overrelaxation_factor_OVERF	OVERF	1.1
reduced_dof_IRDOF	IRDOF	0
reordering_param_ISLORD	ISLORD	0
IRDOF_param_IBACK	IBACK	0
number_SOR_iterations_ICOUPL	ICOUPL	0
max_machine_time_RNMAX	RNMAX	3600

`fdata.`**`sol`**
(*dict[fl64,int]*) Solution parameters (see macro **SOL**).

PyFEHM parameter	FEHM equivalent	Default value
coupling_NTT	NTT	1
element_integration_INTG	INTG	-1

3.7.4 Attributes: time stepping shortcuts

Assigning final simulation time, maximum number of time steps, etc., via the dictionaries in `ctrl` and `time` involves cumbersome lookup of parameter names and then assignment. Therefore, some shortcut attributes are made available to bypass this process

`fdata.`**`ti`**
(*fl64*) Initial simulation time (shortcut), defaults to zero.

`fdata.`**`tf`**
(*fl64*) Final simulation time (shortcut).

`fdata.`**`dti`**
(*fl64*) Initial time step size (shortcut).

`fdata.`**`dtmin`**
(*fl64*) Minimum time step size (shortcut).

`fdata.`**`dtmax`**
(*fl64*) Maximum time step size (shortcut).

`fdata.`**`dtn`**
(*int*) Maximum number of time steps (shortcut).

`fdata.`**`dtx`**
(*fl64*) Time step multiplier, acceleration (shortcut).

`fdata.`**`output_times`**
(*lst*) List of times at which FEHM should produce output.

For example, writing

```
dat.tf=365.25*10.
dat.dti=1.
dat.dtn=500
dat.dtmax=365.25
```

will request the simulation to run for 10 years, with an initial time step of 1 day, a maximum time step of 1 year and to stop if it reaches 500 time steps.

The attribute `output_times` can be used to request output data at specific times during the simulation. Use of this attribute is incompatible with the `change_timestepping` method.

3.7.5 Attributes: flags

`fdata.`**`work_dir`**
(*str*) Directory in which to store files and run simulation.

`fdata.`**`verbose`**
(*bool*) Boolean signalling if simulation output to be printed to screen.

fdata.**sticky_zones**
(*bool*) If `True` zone definitions will be written to the input file immediately before they are used inside a macro.

fdata.**nobr**
(*int*) Boolean integer calling for no breaking of connections between boundary condition nodes.

fdata.**nfinv**
(*int*) Boolean integer calling for generation of finite element coefficients (not recommended, see macro **NFINV**).

3.7.6 Methods

fdata.**read**(*filename=''*, *gridfilename=''*, *inconfilename=''*, *full_connectivity=True*, *skip=[]*)
Read FEHM input file and construct fdata object.

Parameters

- **filename** (*str*) – Name of FEHM input file. Alternatively, supplying 'fehmn.files' will cause PyFEHM to query this file for input, grid and restart file names if they are available.
- **gridfilename** (*str*) – Name of FEHM grid file.
- **inconfilename** (*str*) – Name of FEHM restart file.
- **skip** (*list*) – List of macro strings to ignore when reading an input file.

fdata.**write**(*filename=''*, *writeSubFiles=True*)
Write fdata object to FEHM input file and fehmn.files file.

Parameters

- **filename** (*str*) – Name of FEHM input file to write to.
- **writeSubFiles** (*bool*) – Boolean indicating whether macro and zone information, designated as contained within other input files, should be written out, regardless of its existence. Non-existant files will always be written out.

fdata.**run**(*input=''*, *grid=''*, *incon=''*, *exe='/home/ddempsey/fehm-open/xfehm_v3.1.1lin.06Nov12'*, *files=['outp', 'hist', 'check']*, *verbose=None*, *until=None*, *autorestart=0*, *use_paths=False*, *write_files_only=False*, *diagnostic=False*)
Run an fehm simulation. This command first writes out the input file, *fehmn.files* and this incon file if changes have been made. A command line call is then made to the FEHM executable at the specified path (defaults to *fehm.exe* in the working directory if not specified).

Parameters

- **input** (*str*) – Name of input file. This will be written out.
- **grid** (*str*) – Name of grid file. This will be written out.
- **incon** (*str*) – Name of restart file.
- **exe** (*str*) – Path to FEHM executable.
- **files** (*lst[str]*) – List of additional files to output. Options include 'check', 'hist' and 'outp'.
- **until** (*func*) – Name of a function defined inside the script. The function returns a boolean indicating the simulation should be halted. See tutorial 4 for usage.
- **autorestart** (*int*) – Number of times FEHM should restart itself in attempting to find a solution.
- **use_paths** (*bool*) – Flag to indicate that PyFEHM should favour full paths in fehmn.files rather than duplication of source files.

- **write_files_only** (*bool*) – Flag to indicate the PyFEHM should write out input, incon, grid, fehmn.files, etc. but should not execute a simulation.
- **diagnostic** (*bool*) – Flag to indicate PyFEHM should flash up a diagnostic window to monitor the simulation.

fdata.**paraview**(*exe='paraview.exe'*, *filename='temp.vtk'*, *contour=None*, *show='kx'*, *zones='user'*, *diff=True*, *zscale=1.0*, *spatial_derivatives=False*, *time_derivatives=False*)

Exports the model object to VTK and loads in paraview.

Parameters

- **exe** (*str*) – Path to Paraview executable.
- **filename** (*str*) – Name of VTK file to be output.
- **contour** (*fcontour*) – Contout output data object loaded using fcontour().
- **show** (*str*) – Variable to show when Paraview starts up (default = 'kx').
- **zones** (*str*) – Zones to plot: 'user' = user-defined zones (default), 'all' = all zones except zone[0].
- **diff** (*bool*) – Flag to request PyFEHM to also plot differences of contour variables (from initial state) with time.
- **zscale** (*fl64*) – Factor by which to scale z-axis. Useful for visualising laterally extensive flow systems.
- **time_derivatives** (*bool*) – Calculate new fields for time derivatives of contour data. For precision reasons, derivatives are calculated with units of 'per day'.

fdata.**add**(*obj*, *overwrite=False*)

Attach a zone, boundary condition or macro object to the data file.

Parameters

- **obj** (*fzone, fmacro, fmodel, fboun*) – Object to be added to the data file.
- **overwrite** (*bool*) – Flag to overwrite macro if already exists for a particular zone.

fdata.**delete**(*obj*)

Delete a zone, boundary condition or macro object from the data file.

Parameters **obj** (*fzone, fmacro, fmodel, fboun, list*) – Object to be deleted from the data file. Can be a list of objects.

fdata.**change_timestepping**(*at_time*, *new_dti=None*, *new_dtmax=None*, *new_dtx=None*, *new_implicitness=None*, *new_print_out=None*)

Change timestepping during a simulation. Note, if time stepping arguments are omitted, FEHM will force output to be written at the change time. The default for all optional arguments is no change.

Parameters

- **at_time** (*fl64*) – Simulation time to change time stepping behaviour.
- **new_dti** (*fl64*) – Initial time step at change time.
- **new_dtmax** (*fl64*) – New maximum time step after change time.
- **new_dtx** (*fl64*) – New time step multiplier at change time.
- **new_implicitness** (*fl64*) – New implicitness factor at change time.
- **new_print_out** (*int*) – New time step interval at which to print information.

fdata.**new_zone**(*index=None*, *name=None*, *rect=None*, *nodelist=None*, *file=None*, *from_file=None*, *permeability=None*, *conductivity=None*, *density=None*, *specific_heat=None*, *porosity=None*, *youngs_modulus=None*, *poissons_ratio=None*, *thermal_expansion=None*, *pressure_coupling=None*, *Pi=None*, *Ti=None*, *Si=None*, *overwrite=False*)

Create and assign a new zone. Material properties are optionally specified, new macros will be created if required.

Parameters

- **index** (*int*) – Zone index.
- **name** (*str*) – Zone name.
- **rect** (*lst*) – Two item list. Each item is itself a three item (or two for 2D) list containing [x,y,z] coordinates of zone bounding box.
- **nodelist** (*lst*) – List of node objects or indices of zone. Only one of rect or nodelist should be specified (rect will be taken if both given).
- **file** (*str*) – Name of auxiliary file for zone
- **from_file** (*str*) – Name of auxiliary file in which to find zone information.
- **permeability** (*fl64, list*) – Permeability of zone. One float for isotropic, three item list [x,y,z] for anisotropic.
- **conductivity** (*fl64, list*) – Conductivity of zone. One float for isotropic, three item list [x,y,z] for anisotropic.
- **density** (*fl64*) – Density of zone. If not specified, defaults will be used for specific heat and porosity.
- **specific_heat** (*fl64*) – Specific heat of zone. If not specified, defaults will be used for density and porosity.
- **porosity** (*fl64*) – Porosity of zone. If not specified, defaults will be used for density and specific heat.
- **youngs_modulus** (*fl64*) – Young's modulus of zone. If not specified, default will be used for Poisson's ratio.
- **poissons_ratio** (*fl64*) – Poisson's ratio of zone. If not specified, default will be used for Young's modulus.
- **thermal_expansion** (*fl64*) – Coefficient of thermal expansion for zone. If not specified, default will be used for pressure coupling term.
- **pressure_coupling** (*fl64*) – Pressure coupling term for zone. If not specified, default will be used for coefficient of thermal expansion.
- **Pi** (*fl64*) – Initial pressure in the zone. If not specified, default will be used for initial temperature and saturation calculated.
- **Ti** (*fl64*) – Initial temperature in the zone. If not specified, default will be used for initial pressure and saturation calculated.
- **Si** (*fl64*) – Initial saturation in the zone. If not specified, default will be used for initial pressure and the saturation temperature calculated.
- **overwrite** (*bool*) – If zone already exists, delete it and create the new one.

fdata.**temperature_gradient**(*filename*, *offset=0.0*, *first_zone=100*, *auxiliary_file=None*, *hydrostatic=0*)

Assign initial temperature distribution to model based on supplied temperature profile.

Parameters

- **filename** (*str*) – Name of a file containing temperature gradient data. File should be two columns, comma or space separated, with elevation in the first column and temperature (degC) in the second.
- **offset** (*fl64*) – Vertical offset added to the elevation in temperature gradient data. Useful if model limits don't correspond to data.
- **first_zone** (*int*) – Index of first zone created. Zone index will be incremented by 1 thereafter.
- **auxiliary_file** (*str*) – Name of auxiliary file in which to store **PRES** macros.
- **hydrostatic** (*fl64*) – Pressure at top of well profile. PyFEHM will calculate hydrostatic pressures consistent with the temperature profile. If left blank, default pressures will be used.

`fdata.`**`what`**

`fdata.`**`print_ctrl`**`()`
Display contents of CTRL macro.

`fdata.`**`print_iter`**`()`
Display contents of ITER macro.

`fdata.`**`print_time`**`()`
Display contents of TIME macro.

3.7.7 Examples

Examples given here are limited to the usage of `fdata` methods. More in depth explanations of FEHM simulation within the PyFEHM framework is given in the *tutorial* section of this guide.

1. Read an FEHM input file into PyFEHM.

```
dat=fdata(filename='myInput.dat', gridfilename='myGrid.inp')
```

2. Read in an FEHM input file with initial conditions.

```
dat=fdata('myInput.dat', 'myGrid.inp', 'myIncon.ini')
```

3. Write out FEHM input deck.

```
dat.write('myInput2.dat')
```

4. Delete a previously defined flow macro, assigned to the zone 'injection, from the input file.

```
dat.delete(dat.fdata.flow['injection'])
```

5. Run an FEHM simulation through PyFEHM.

```
dat.run(input='myInput.dat', grid='myGrid.inp', exe='C:\\path\\to\\FEHM\\source',
files = ['hist', 'check', 'outp'])
```

6. Create a new zone called `reservoir` and assign it permeability and elastic properties.

```
dat.zone['zmax'].new_zone(index=10, name='reservoir',
rect=[[-0.1,-0.1,800.], [10000.1,10000.1,900.]], permeability=1.e-14,
youngs_modulus=2.5e4)
```

3.8 FEHM control files

Information written to the *fehmn.files* control file is collated and modified within this object.

The control file can be written independently using `files.write()`, but in general should be written automatically when a PyFEHM simulation is run using `fdata.run()`.

Some attributes of `files` will be written automatically when certain actions are taken within PyFEHM. For example, reading a new grid using `fdata.grid.read('myGrid.inp')` will cause `fdata.files.grid='myGrid.inp'`.

class `fdata.`**`files`**(*root=''*, *input=''*, *grid=''*, *incon=''*, *rsto=''*, *outp=''*, *check=''*, *hist=''*, *co2in=''*, *stor=''*, *exe='fehm.exe'*, *co2_inj_time=None*)
Class containing information necessary to write out fehmn.files.

3.8.1 Attributes

`files.`**`input`**
(*str*) Name of input file. This is set automatically when reading an input file in PyFEHM or when running a simulation.

`files.`**`grid`**
(*str*) Name of grid file. This is set automatically when reading an grid file in PyFEHM.

`files.`**`incon`**
(*str*) Name of restart file to read in (initial condition). This is set automatically when reading an incon file in PyFEHM.

`files.`**`root`**
(*str*) Default file name string. If not already specified, this is set automatically when running a simulation.

`files.`**`rsto`**
(*str*) Name of restart file to write out.

`files.`**`outp`**
(*str*) Name of output file.

`files.`**`check`**
(*str*) Name of check file.

`files.`**`co2in`**
(*str*) Name or path to co2 properties file

`files.`**`hist`**
(*str*) Name of history file.

`files.`**`stor`**
(*str*) Name of store file.

`files.`**`exe`**
(*str*) Path to FEHM executable. Default is 'fehm.exe'.

3.8.2 Methods

`files.`**`write`**(*use_paths=False*)
Write out *fehmn.files*.

3.9 Initial conditions/simulation restart

Initial conditions, or restart, files can be parsed, modified and rewritten in PyFEHM.

Reading of incon files occurs during model initialisation, e.g., `fdata('myInput.dat','myGrid.inp','myIncon.ini')`, or later by invoking `fincon.read()`.

If changes are made to any of the restart variables, care should be taken to rewrite in the incon file using `fincon.write()` before running a simulation.

Tools are available for user modification of restart files. At present, these include the construction of critically-stressed lithostatic stress gradients using `fincon.stressgrad()` and `fincon.critical_stress()`.

class `fdata.`**`fincon`**(*inconfilename=''*)
Initial conditions object. Also called a restart file.

Reading one of these files associates the temperature, pressure, saturation etc. data with grid nodes and sets up fehmn.files to use the file for restarting.

3.9.1 Attributes: general

`fincon.`**`filename`**
(*str*) Name of restart file (initial conditions)

`fincon.`**`source`**
(*str*) Name of input file that generated the restart.

`fincon.`**`time`**
(*fl64*) Time stamp of initial conditions file (end time of simulation that produced this file).

3.9.2 Attributes: variables

When an incon file is parsed in PyFEHM, that information is made available through variable list attributes. Changes can be made to these variables and a new incon file written out using the `write()` method. For obvious reasons, some variables are only available if the relevant modules are being used, e.g., `fincon.disp_x` returns an empty list unless the FEHM stress module is being used.

Note that, because python indexing begins at 0, the variable associated with a node is accessed at the list position one less than the node index, e.g., the pressure at node 100 is accessed via `fincon.P[99]`.

`fincon.`**`P`**
(*lst[fl64]*) Initial node pressures, ordered by node index.

`fincon.`**`T`**
(*lst[fl64]*) Initial node temperatures, ordered by node index.

`fincon.`**`S`**
(*lst[fl64]*) Initial node water saturations, ordered by node index.

`fincon.`**`S_co2l`**
(*lst[fl64]*) Initial node co2 liquid/super-critical saturations, ordered by node index.

`fincon.`**`S_co2g`**
(*lst[fl64]*) Initial node co2 gas saturations, ordered by node index.

`fincon.`**`co2aq`**
(*lst[fl64]*) Initial node dissolved co2 concentrations, ordered by node index.

fincon.**eos**
(*lst[fl64]*) Initial node water equation of state indices, ordered by node index.

fincon.**co2_eos**
(*lst[fl64]*) Initial node co2 equation of state indices, ordered by node index.

fincon.**co2_eos**
(*lst[fl64]*) Initial node co2 equation of state indices, ordered by node index.

fincon.**disp_x**
(*lst[fl64]*) Initial node x displacement, ordered by node index

fincon.**disp_y**
(*lst[fl64]*) Initial node y displacement, ordered by node index

fincon.**disp_z**
(*lst[fl64]*) Initial node z displacement, ordered by node index

fincon.**strs_xx**
(*lst[fl64]*) Initial node x stress, ordered by node index

fincon.**strs_yy**
(*lst[fl64]*) Initial node y stress, ordered by node index

fincon.**strs_zz**
(*lst[fl64]*) Initial node z stress, ordered by node index

fincon.**strs_xy**
(*lst[fl64]*) Initial node xy stress, ordered by node index

fincon.**strs_xz**
(*lst[fl64]*) Initial node xz stress, ordered by node index

fincon.**strs_yz**
(*lst[fl64]*) Initial node yz stress, ordered by node index

3.9.3 Methods

fincon.**read**(*inconfilename=''*, *if_new=False*)
Parse a restart file for variable information.

Parameters **inconfilename** (*str*) – Name of restart file.

fincon.**write**(*inconfilename=''*)
Write out a restart file.

Parameters **inconfilename** (*str*) – Name of restart file to write out.

fincon.**stressgrad**(*xgrad*, *ygrad*, *zgrad*, *xygrad=0.0*, *xzgrad=0.0*, *yzgrad=0.0*, *calculate_vertical=False*, *vertical_fraction=False*)
Construct initial stress state with vertical stress gradients.

Parameters

- **xgrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in x stress (MPa/m), assumed intercept at [0,0]. If a two element list is given, the first value is interpreted as the gradient, and the second value as the elevation where stress is zero (i.e., the intercept with the z-axis).
- **ygrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in y stress (MPa/m), format as for **xgrad**.
- **zgrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in z stress (MPa/m), format as for **xgrad**.

- **xygrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in xy stress (MPa/m), default is 0, format as for **xgrad**.
- **xzgrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in xz stress (MPa/m), default is 0, format as for **xgrad**.
- **yzgrad** (*fl64, list[fl64,fl64]*) – Vertical gradient in yz stress (MPa/m), default is 0, format as for **xgrad**.
- **calculate_vertical** (*bool*) – Vertical stress calculated by integrating density. If true, then zgrad specifies the overburden.
- **vertical_fraction** (*bool*) – Horizontal stresses calculated as a fraction of the vertical. If true, xgrad and ygrad are interpreted as fractions.

fincon.**critical_stress**(*regime=1*, *horiz_stress='x'*, *mu=0.6*, *cohesion=0*, *proximity=0.0*, *overburden=0.0*)

Construct initial stress state near Mohr-Coulomb failure. The vertical stress is calculated using the assigned density. Minimum or maximum horizontal stress calculated using the specified friction coefficient. Intermediate principal stress is assumed to be the average of the other two.

Parameters

- **regime** (*bool int*) – Stress regime, 0 = compression, 1 = extension (default).
- **horiz_stress** (*str*) – Horizontal coordinate direction ('x' or 'y') to assign the minimum or maximum principal stress (depending on stress regime).
- **mu** (*fl64*) – Friction coefficient for Mohr-Coulomb failure (default = 0.6).
- **cohesion** (*fl64*) – Cohesion for Mohr-Coulomb failure (default = 0).
- **proximity** (*fl64*) – A negative quantity indicating how close the minimum principal stress is to Mohr-Coulomb failure (MPa, default = 0).
- **overburden** (*fl64*) – Overburden at top of model (MPa, default = 0).

3.9.4 Examples

1. Read in an incon file, increase all temperatures by 10degC and write out the new incon file.

```
dat.incon.read('myIncon.ini')
for i in len(range(dat.incon.T)): dat.incon.T[i] += 10
dat.incon.write('myNewIncon.ini')
```

2. Read in an incon file and create vertical gradients in the stress field.

```
dat.incon.read('noStress.ini')
dat.incon.stressgrad(zgrad=0.02, xgrad = [0.015,-100], ygrad =
[0.018, -100])
dat.incon.write('withStress.ini')
```

3. Read in an incon file and create a critical stress state.

```
dat.incon.read('noStress.ini')
dat.incon.critical_stress(regime=1, horiz_stress = 'x', mu = 0.8,
cohesion = 1, proximity = 0.5, overburden = 10)
dat.incon.write('withStress.ini')
```

3.10 Relative permeability

Relative permeability models in FEHM can be defined through either the **RLP** or **RLPM** macro. In PyFEHM, **RLP** macros are defined via the ordinary `fmodel('rlp')` object, whereas **RLPM** is defined within the specialised `frlpm` class detailed here.

For each phase present in the model, the user defines a relperm model. The user can additionally define capillary pressure models for phase pairs. The following phases can have relative permeability and capillary pressure models defined: *water, air, co2_liquid, co2_gas, vapor.*

Parameters for the available relative permeability models are given in the table below.

Relative permeability model	Parameter name	Default
constant	None	N/A
linear	minimum_saturation	0
	maximum_saturation	1
exponential	minimum_saturation	0
	maximum_saturation	1
	exponent	1
	maximum_relperm	1
corey	minimum_saturation	0
	maximum_saturation	1
brooks-corey	minimum_saturation	0
	maximum_saturation	1
	exponent	1
vg	maximum_saturation	0
	maximum_saturation	1
	air_entry_head	1
	exponent	1

Parameters for the available capillary pressure models are given in the table below.

Capillary pressure model	Parameter name	Default
linear_cap	cap_at_zero_sat	None
	sat_at_zero_cap	None
brooks-corey_cap	minimum_saturation	0
	maximum_saturation	1
	exponent	1
	capillary_entry_presure	0.01
	low_saturation_fitting	None
	cutoff_saturation	None
vg_cap	minimum_saturation	0
	maximum_saturation	1
	air_entry_head	1
	exponent	1
	low_saturation_fitting	None
	cutoff_saturation	None

class `fdata.`**`frlpm`**(*zone*=[ ], *group*=*None*, *relperm*=[ ], *capillary*=[ ])

Relative permeability model.

Relative permeability models are applied to zones. Each model assigns the relative permeability characteristics for a particular phase (in attribute relperm) and capillary pressure relationships between phases (in attribute capillary). In contrast to other macros, one relative permeability model may be applied to multiple zones.

3.10.1 Attributes

frlpm.**group**
(*int*) Group assignment for the relative permeability model.

frlpm.**zone**
(*fzone*) Zone or list of zones to which the relative permeability model is assigned.

frlpm.**relperm**
(*dict*) Dictionary of relative permeability models for each phase, indexed by phase name, e.g., 'water'. Each relperm model has a model type, and set of parameters for that type.

frlpm.**capillary**
(*dict*) Dictionary of capillary pressure models, indexed by a tuple phase name pair, e.g., ('water/air'). Each capillary pressure model has a model type, and set of parameters for that type.

3.10.2 Methods

frlpm.**add_relperm**(*phase*, *type*, *param*=[ ])
Add a new relative permeability model for a given phase.

Parameters

- **phase** (*str*) – Phase for which the relperm model is being defined, e.g., 'air','water','co2_liquid','co2_gas','vapor'.
- **type** (*str*) – Type of model being assigned for that phase, e.g., 'constant','linear','exponential','corey','brooks-corey','vg' (Van Genuchten).
- **param** (*list*) – List of parameters for the specified model. See table above for a list of parameter names for each model.

frlpm.**add_capillary**(*phase*, *type*, *param*=[ ])
Add a new capillary pressure relationship between two phases.

Parameters

- **phase** (*list, tuple*) – List or tuple of two phases for which the relationship is defined, e.g., ['air','water'].
- **type** (*str*) – Type of model being assigned for that phase pair, e.g., 'linear_cap','vg_cap','brooks-corey_cap'.
- **param** (*list*) – List of parameters for the specified model. See table above for a list of parameter names for each model.

frlpm.**delete**(*model*)
Delete a previously defined relative permeability or capillary pressure model.

3.10.3 Examples

In the following example, a relative permeability and capillary pressure model is added for a water/co2 mixture.

```
rlpm=frlpm(group=1,zone=dat.zone[0])
rlpm.add_relperm ('water','exponential',[0.2,1.,3.1,1.])
rlpm.add_relperm ('co2_liquid','exponential',[0.2,1.,3.1,0.8])
rlpm.add_capillary (('water','co2_liquid'),'vg_cap',[0,0.87,.0015,3.5,7,0.])
```

```
dat.add(rlpm)
```

3.11 Stress module

FEHM contains an additional module that allows for coupled thermo-hydro-mechanical modelling. In PyFEHM, the stress module is invoked via the `fstrs` class.

For a well formed coupled flow-stress problem, the user will need to specify elastic material properties via `fmacro('elastic')` and some fixed displacement stress boundary conditions to prevent the model block from flying off into space (`fmacro('stressboun')`).

Additional complexity can be included, e.g.:

1. gravitational bodyforces (`fstrs.bodyforce`)
2. stress-permeability models (`fmodel('permmodel')`)
3. thermal and poroelastic coupling between flow and stress (`fmacro('biot')`)
4. stress restarts and critical stress states (`fincon.critical_stress()`)

The stress module is *off* by default when a new data file is created. It is turned on via the `fdata.strs.on()` method.

class `fdata.`**`fstrs`**(*initcalc=True*, *fem=True*, *bodyforce=True*, *tolerance=-0.01*, *param={}*, *parent=None*)
Stress module object, sets properties for execution of FEHM stress module (see macro **STRS**).

3.11.1 Attributes

`fstrs.`**`param`**
(*dict[flt]*) Dictionary of stress parameters: 'IHMS' - coupling parameter, 'ISTRS' - type of stress solution.

`fstrs.`**`bodyforce`**
(*bool*) Boolean calling for body force calculations (gravity). Default is True.

`fstrs.`**`initcalc`**
(*bool*) Boolean signalling if initial stress calculations should be performed. Default is True.

`fstrs.`**`fem`**
(*bool*) Boolean signalling use of finite element modules for calculating stress and displacement. Default is True.

`fstrs.`**`tolerance`**
(*flt*) Tolerance of stress calculations.

`fstrs.`**`excess_she`**
Dictionary of excess shear parameters:

3.11.2 Methods

`fstrs.`**`on`**`()`
Set parameters to turn stress calculations ON.

`fstrs.`**`off`**`()`
Set param to turn stress calculations OFF.

3.11.3 Examples

Switch on the stress module, turn off bodyforce calculations (important if doing a restart from a simulation that previously contained gravity).

```
dat.strs.on()

dat.strs.bodyforce=False
```

Add some elastic material properties and a stress boundary condition.

```
dat.add       (fmacro('elastic',         param=(('youngs_modulus',1e4),
('poissons_ratio',0.25))))

dat.add     (fmacro('stressboun',      zone='xmin',      subtype='fixed',
param=(('direction',1),('value',0))))
```

3.12 Carbon dioxide module

Mixtures of water and gaseous, liquid and super-critical carbon dioxide are modelled in FEHM using the **CARB** macro. In PyFEHM this module is loaded using the `fcarb` class.

As with water, a CO2 simulation should contain CO2 sinks or sources (`fmacro('co2flow')`) and may include conditions for any CO2 initially present (`fmacro('co2pres')`) and `fmacro('co2frac')`).

When using the CO2 module, a relative permeability model for water/CO2 mixtures should be specified, either via `fmodel('rlp')` or `frlpm`.

The CO2 module is *off* by default when a new data file is created. It is turned on via `fcarb.on()`.

class `fdata.`**fcarb**(*iprtype=1*, *brine=False*, *parent=None*)
CO2 module object, sets properties for execution of FEHM CO2 module (see macro **CARB**).

3.12.1 Attributes

`fcarb.`**iprtype**
(*int*) Integer indicating type of simulation.

`fcarb.`**brine**
(*bool*) Boolean signalling calculation of brine in simulation.

3.12.2 Methods

`fcarb.`**on**(*iprtype=3*)
Set parameters to turn CO2 calculations ON.

Parameters **iprtype** (*int*) – Integer denoting type of simulation (1 = water only, 2 = CO2 only, 3 = water+CO2, no solubility, 4 = water+CO2, with solubility, 5 = water+CO2+air, with solubility)

`fcarb.`**off**()
Set parameters to turn CO2 calculations OFF.

`fcarb.`**what**
Return a summary of the CO2 module.

3.12.3 Examples

Switch on the CO2 module and specify that CO2 dissolution in water should be modelled.

```
dat.carb.on(iprtype = 4)
```

Add a CO2 injection source and assign the injection block to initially have non-zero CO2 fraction. This is important, as relative permeability effects may not permit the source to flow.

```
dat.add  (fmacro('co2flow',  zone='injection',  param=(('rate',10),
('energy',-40), ('impedance',1.e-2), ('bc_flag',1))))
```

```
dat.add(fmacro('co2frac', zone='injection', param=(('water_rich_sat',0.5),
('co2_rich_sat',0.5), ('co2_mass_frac',0.), ('init_salt_conc',0.),
('override_flag',1))))
```

3.13 Species transport module

Support of FEHM's trac macro is currently at 'stupid' level. The user supplies a list of zones required by the macro and a file path to an auxiliary text file which contains the macro definition verbatim.

Full support is in development.

class `fdata.`**`ftrac`**(*parent=None*, *ldsp=False*)
Chemistry module object, sets properties for execution of FEHM chemistry module (see macro **TRAC**).

3.13.1 ftrac attributes

`ftrac.`**`zonelist`**
(*lst[fzone]*) A list of zones to be written before the trac macro, for the situation that trac is used in 'stupid' model, i.e., the file attribute has been set to an auxiliary file.

`ftrac.`**`file`**
(*str*) Path of auxiliary file containing trac information. PyFEHM will copy the contents of this file into the input file verbatim. Setting this variable will cause PyFEHM to use trac in 'stupid' mode.

CHAPTER

FOUR

FPOST: FEHM OUTPUT FILE MANIPULATION

Processing simulation output data is as important as the generation of the data itself. PyFEHM contains a range of basic plotting tools to visualise FEHM output, all based on the matplotlib python library.

FEHM can output both time history data for specified variables at specified nodes () and contour data for specified variables at specified times. PyFEHM can read in any output data format, e.g., *surfer, tecplot*, so the user is not required to output in a specific format (although *surfer* format is likely to be the least buggy).

4.1 Contour output

Data are read from a list of FEHM output files corresponding to a particular time during a simulation. Depending on input specifications in `fdata.cont`, for each node, x, y, z data, pressure, temperature, permeability information is available.

Setting the target filename using wildcards will cause PyFEHM to read in multiple input files for different times. This format will accommodate FEHM's distinction of scalar, vector and concentration output, e.g., information from run.0001_con_node.dat and run.0001_sca_node.data will be read into a single object.

This format also supports material property files, generally denoted with 'mat_node' in the title. This information, when parsed, is made available through the `material` attribute.

class `fpost.`**`fcontour`**(*filename=None*, *latest=False*, *first=False*, *nearest=None*)
Contour output information object.

4.1.1 Attributes

`fcontour.`**`times`**
(*lst[fl64]*) List of times (in seconds) for which output data are available.

`fcontour.`**`variables`**
(*lst[str]*) List of variables for which output data are available.

`fcontour.`**`user_variables`**
(*lst[str]*) List of user-defined variables for which output data are available.

`fcontour.`**`material`**
(*dict[str]*) Dictionary of material properties, keyed by property name, items indexed by node_number - 1. This attribute is empty if no material property file supplied.

fcontour.**format**
(*str*) Format of output file, options are 'tec', 'surf', 'avs' and 'avsx'.

fcontour.**filename**
(*str*) Name of FEHM contour output file. Wildcards can be used to define multiple input files.

4.1.2 Grid attributes

fcontour.**x**
(*lst[fl64]*) Unique list of nodal x-coordinates for grid.

fcontour.**y**
(*lst[fl64]*) Unique list of nodal y-coordinates for grid.

fcontour.**z**
(*lst[fl64]*) Unique list of nodal z-coordinates for grid.

fcontour.**xmin**
(*fl64*) Minimum nodal x-coordinate for grid.

fcontour.**xmax**
(*fl64*) Maximum nodal x-coordinate for grid.

fcontour.**ymin**
(*fl64*) Minimum nodal y-coordinate for grid.

fcontour.**ymax**
(*fl64*) Maximum nodal y-coordinate for grid.

fcontour.**zmin**
(*fl64*) Minimum nodal z-coordinate for grid.

fcontour.**zmax**
(*fl64*) Maximum nodal z-coordinate for grid.

4.1.3 Methods

fcontour.**read**(*filename*, *latest=False*, *first=False*, *nearest=[]*)
Read in FEHM contour output information.

Parameters

- **filename** (*str*) – File name for output data, can include wildcards to define multiple output files.
- **latest** (*bool*) – Boolean indicating PyFEHM should read the latest entry in a wildcard search.
- **first** (*bool*) – Boolean indicating PyFEHM should read the first entry in a wildcard search.
- **nearest** (*fl64,list*) – Read in the file with date closest to the day supplied. List input will parse multiple output files.

fcontour.**node**(*node*, *time=None*, *variable=None*)
Returns all information for a specific node.

If time and variable not specified, a dictionary of time series is returned with variables as the dictionary keys.

If only time is specified, a dictionary of variable values at that time is returned, with variables as dictionary keys.

If only variable is specified, a time series vector is returned for that variable.

If both time and variable are specified, a single value is returned, corresponding to the variable value at that time, at that node.

Parameters

- **node** (*int*) – Node index for which variable information required.
- **time** (*fl64*) – Time at which variable information required. If not specified, all output.
- **variable** (*str*) – Variable for which information requested. If not specified, all output.

`fcontour.`**`new_variable`**(*name*, *time*, *data*)

Creates a new variable, which is some combination of the available variables.

Parameters

- **name** (*str*) – Name for the variable.
- **time** (*fl64*) – Time key which the variable should be associated with. Must be one of the existing keys, i.e., an item in fcontour.times.
- **data** (*lst[fl64]*) – Variable data, most likely some combination of the available parameters, e.g., pressure*temperature, pressure[t=10] - pressure[t=5]

`fcontour.`**`paraview`**(*grid*, *stor=None*, *exe='paraview.exe'*, *filename='temp.vtk'*, *show=None*, *diff=False*, *zscale=1.0*, *time_derivatives=False*)

Launches an instance of Paraview that displays the contour object.

Parameters

- **grid** (*str*) – Path to grid file associated with FEHM simulation that produced the contour output.
- **stor** (*str*) – Path to grid file associated with FEHM simulation that produced the contour output.
- **exe** (*str*) – Path to Paraview executable.
- **filename** (*str*) – Name of VTK file to be output.
- **show** (*str*) – Variable to show when Paraview starts up (default = first available variable in contour object).
- **diff** (*bool*) – Flag to request PyFEHM to also plot differences of contour variables (from initial state) with time.
- **zscale** (*fl64*) – Factor by which to scale z-axis. Useful for visualising laterally extensive flow systems.
- **time_derivatives** (*bool*) – Calculate new fields for time derivatives of contour data. For precision reasons, derivatives are calculated with units of 'per day'.

`fcontour.`**`what`**

(*str*) Print out information about the fcontour object.

4.1.4 Examples

4.1.5 Slice plots

`fcontour.`**`slice`**(*variable*, *slice*, *divisions*, *time=None*, *method='nearest'*)

Returns mesh data for a specified slice orientation from 3-D contour output data.

Parameters

- **variable** (*str*) – Output data variable, for example 'P' = pressure. Alternatively, variable can be a five element list, first element 'cfs', remaining elements fault azimuth (relative to x), dip, friction coefficient and cohesion. Will return coulomb failure stress.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data.
- **slice** (*lst[str,fl64]*) – List specifying orientation of output slice, e.g., ['x',200.] is a vertical slice at `x = 200`, ['z',-500.] is a horizontal slice at `z = -500.`, [point1, point2] is a fixed limit vertical or horizontal domain corresponding to the bounding box defined by point1 and point2.
- **divisions** (*[int,int]*) – Resolution to supply mesh data.
- **method** (*str:*) – Method of interpolation, options are 'nearest', 'linear'.

Returns X – x-coordinates of mesh data.

`fcontour.`**`slice_plot`**(*variable=None, time=None, slice=''', divisions=[20, 20], levels=10, cbar=False, xlims=[], ylims=[], colors='k', linestyle='-', save='', xlabel='x / m', ylabel='y / m', title='', font_size='medium', method='nearest', equal_axes=True, mesh_lines=None, perm_contrasts=None, scale=1.0*)

Returns a filled plot of contour data. Invokes the `slice()` method to interpolate slice data for plotting.

Parameters

- **variable** (*str*) – Output data variable, for example 'P' = pressure.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data. If a list of two times is passed, the difference between the first and last is plotted.
- **slice** (*lst[str,fl64]*) – List specifying orientation of output slice, e.g., ['x',200.] is a vertical slice at `x = 200`, ['z',-500.] is a horizontal slice at `z = -500.`, [point1, point2] is a fixed limit vertical or horizontal domain corresponding to the bounding box defined by point1 and point2.
- **divisions** (*[int,int]*) – Resolution to supply mesh data.
- **method** (*str*) – Method of interpolation, options are 'nearest', 'linear'.
- **levels** (*lst[fl64], int*) – Contour levels to plot. Can specify specific levels in list form, or a single integer indicating automatic assignment of levels.
- **cbar** (*bool*) – Add colour bar to plot.
- **xlims** (*[fl64, fl64]*) – Plot limits on x-axis.
- **ylims** (*[fl64, fl64]*) – Plot limits on y-axis.
- **colors** (*lst[str]*) – Specify colour string for contour levels.
- **linestyle** (*str*) – Style of contour lines, e.g., 'k-' = solid black line, 'r:' red dotted line.
- **save** (*str*) – Name to save plot. Format specified extension (default .png if none give). Supported extensions: .png, .eps, .pdf.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **title** (*str*) – Plot title.
- **font_size** (*str, int*) – Specify text size, either as an integer or string, e.g., 10, 'small', 'x-large'.

- **equal_axes** (*bool*) – Specify equal scales on axes.
- **mesh_lines** (*bool*) – Superimpose mesh on the plot (line intersections correspond to node positions) according to specified linestyle, e.g., 'k:' is a dotted black line.
- **perm_contrasts** (*bool*) – Superimpose permeability contours on the plot according to specified linestyle, e.g., 'k:' is a dotted black line. A gradient method is used to pick out sharp changes in permeability.

fcontour.**cutaway_plot**(*variable=None, time=None, divisions=[20, 20, 20], levels=10, cbar=False, angle=[45, 45], xlims=[], method='nearest', ylims=[], zlims=[], colors='k', linestyle='-', save='', xlabel='x / m', ylabel='y / m', zlabel='z / m', title='', font_size='medium', equal_axes=True, grid_lines=None*)

Returns a filled plot of contour data on each of 3 planes in a cutaway plot. Invokes the `slice()` method to interpolate slice data for plotting.

Parameters

- **variable** (*str*) – Output data variable, for example 'P' = pressure.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data. If a list of two times is passed, the difference between the first and last is plotted.
- **divisions** (*[int,int,int]*) – Resolution to supply mesh data in [x,y,z] coordinates.
- **levels** (*lst[fl64], int*) – Contour levels to plot. Can specify specific levels in list form, or a single integer indicating automatic assignment of levels.
- **cbar** (*bool*) – Include colour bar.
- **angle** (*[fl64,fl64], str*) – View angle of zone. First number is tilt angle in degrees, second number is azimuth. Alternatively, if angle is 'x', 'y', 'z', view is aligned along the corresponding axis.
- **method** (*str*) – Method of interpolation, options are 'nearest', 'linear'.
- **xlims** (*[fl64, fl64]*) – Plot limits on x-axis.
- **ylims** (*[fl64, fl64]*) – Plot limits on y-axis.
- **zlims** (*[fl64, fl64]*) – Plot limits on z-axis.
- **colors** (*lst[str]*) – Specify colour string for contour levels.
- **linestyle** (*str*) – Style of contour lines, e.g., 'k-' = solid black line, 'r:' red dotted line.
- **save** (*str*) – Name to save plot. Format specified extension (default .png if none give). Supported extensions: .png, .eps, .pdf.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **zlabel** (*str*) – Label on z-axis.
- **title** (*str*) – Plot title.
- **font_size** (*str, int*) – Specify text size, either as an integer or string, e.g., 10, 'small', 'x-large'.
- **equal_axes** (*bool*) – Force plotting with equal aspect ratios for all axes.
- **grid_lines** (*bool*) – Extend tick lines across plot according to specified linestyle, e.g., 'k:' is a dotted black line.

fcontour.**slice_plot_line**(*variable=None, time=None, slice='', divisions=[20, 20], labels=False, label_size=10.0, levels=10, xlims=[], ylims=[], colors='k', linestyle='-', save='', xlabel='x / m', ylabel='y / m', title='', font_size='medium', method='nearest', equal_axes=True*)

Returns a line plot of contour data. Invokes the `slice()` method to interpolate slice data for plotting.

Parameters

- **variable** (*str*) – Output data variable, for example 'P' = pressure.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data. If a list of two times is passed, the difference between the first and last is plotted.
- **slice** (*lst[str,fl64]*) – List specifying orientation of output slice, e.g., ['x',200.] is a vertical slice at `x = 200`, ['z',-500.] is a horizontal slice at `z = -500.`, [point1, point2] is a fixed limit vertical or horizontal domain corresponding to the bounding box defined by point1 and point2.
- **divisions** (*[int,int]*) – Resolution to supply mesh data.
- **method** (*str*) – Method of interpolation, options are 'nearest', 'linear'.
- **labels** (*bool*) – Specify whether labels should be added to contour plot.
- **label_size** (*str, int*) – Specify text size of labels on contour plot, either as an integer or string, e.g., 10, 'small', 'x-large'.
- **levels** (*lst[fl64], int*) – Contour levels to plot. Can specify specific levels in list form, or a single integer indicating automatic assignment of levels.
- **xlims** (*[fl64, fl64]*) – Plot limits on x-axis.
- **ylims** (*[fl64, fl64]*) – Plot limits on y-axis.
- **linestyle** (*str*) – Style of contour lines, e.g., 'k-' = solid black line, 'r:' red dotted line.
- **save** (*str*) – Name to save plot. Format specified extension (default .png if none give). Supported extensions: .png, .eps, .pdf.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **title** (*str*) – Plot title.
- **font_size** (*str, int*) – Specify text size, either as an integer or string, e.g., 10, 'small', 'x-large'.
- **equal_axes** (*bool*) – Specify equal scales on axes.

4.1.6 Profile plots

fcontour.**profile**(*variable*, *profile*, *time=None*, *divisions=30*, *method='nearest'*)

Return variable data along the specified line in 3-D space. If only two points are supplied, the profile is assumed to be a straight line between them.

Parameters

- **variable** (*str, lst[str]*) – Output data variable, for example 'P' = pressure. Can specify multiple variables with a list.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data.

- **profile** (*ndarray*) – Three column array with each row corresponding to a point in the profile.
- **divisions** (*int*) – Number of points in profile. Only relevant if straight line profile being constructed from two points.
- **method** (*str*) – Interpolation method, options are 'nearest' (default) and 'linear'.

Returns Multi-column array. Columns are in order x, y and z coordinates of profile, followed by requested variables.

`fcontour.`**`profile_plot`**(*variable=None*, *time=None*, *profile=[]*, *divisions=30*, *xlims=[]*, *ylims=[]*, *color='k'*, *marker='x-'*, *save=''*, *xlabel='distance / m'*, *ylabel=''*, *title=''*, *font_size='medium'*, *method='nearest'*, *verticalPlot=False*, *elevationPlot=False*)

Return a plot of the given variable along a specified profile. If the profile comprises two points, these are interpreted as the start and end points of a straight line profile.

Parameters

- **variable** (*str, lst[str]*) – Output data variable, for example 'P' = pressure. Can specify multiple variables with a list.
- **time** (*fl64*) – Time for which output data is requested. Can be supplied via `fcontour.times` list. Default is most recently available data. If a list of two times is passed, the difference between the first and last is plotted.
- **profile** (*ndarray*) – Three column array with each row corresponding to a point in the profile.
- **divisions** (*int*) – Number of points in profile. Only relevant if straight line profile being constructed from two points.
- **method** (*str*) – Interpolation method, options are 'nearest' (default) and 'linear'.
- **xlims** (*[fl64, fl64]*) – Plot limits on x-axis.
- **ylims** (*[fl64, fl64]*) – Plot limits on y-axis.
- **color** (*str*) – Colour of profile.
- **marker** (*str*) – Style of line, e.g., 'x-' = solid line with crosses, 'o:' dotted line with circles.
- **save** (*str*) – Name to save plot. Format specified extension (default .png if none give). Supported extensions: .png, .eps, .pdf.
- **xlabel** (*str*) – Label on x-axis.
- **ylabel** (*str*) – Label on y-axis.
- **title** (*str*) – Plot title.
- **font_size** (*str, int*) – Specify text size, either as an integer or string, e.g., 10, 'small', 'x-large'.
- **verticalPlot** (*bool*) – Flag to plot variable against profile distance on the y-axis.
- **elevationPlot** (*bool*) – Flag to plot variable against elevation on the y-axis.

4.2 History output

Data are read from a list of FEHM output files corresponding to a particular variable during a simulation. Depending on input specifications in `fdata.hist`, for each requested node, time series data is available.

Setting the target filename using wildcards will cause PyFEHM to read in multiple input files for different times.

class fpost.**fhistory**(*filename=None*, *verbose=True*)
History output information object.

4.2.1 Zone fluxes

Zone flux information is parsed using the fzoneflux class, which is a derived class of fhistory. Where identical, methods and attributes are shown only for fhistory.

class fpost.**fzoneflux**(*filename=None*, *verbose=True*)
Zone flux history output information object.

4.2.2 Attributes

fhistory.**times**
(*lst[fl64]*) List of times (in seconds) for which output data are available.

fhistory.**variables**
(*lst[str]*) List of variables for which output data are available.

fhistory.**nodes**
(*lst[fl64]*) List of node indices for which output data are available.

fzoneflux.**zones**
(*lst[int]*) List of zone indices for which output data are available.

fhistory.**format**
(*str*) Format of output file, options are 'tec', 'surf', 'avs' and 'avsx'.

fhistory.**filename**
(*str*) Name of FEHM contour output file. Wildcards can be used to define multiple input files.

4.2.3 Methods

fhistory.**read**(*filename*)
Read in FEHM history output information.

Parameters **filename** (*str*) – File name for output data, can include wildcards to define multiple output files.

4.2.4 Time series plots

fhistory.**time_plot**(*variable=None*, *node=0*, *t_lim=[ ]*, *var_lim=[ ]*, *marker='x-'*, *color='k'*, *save=''*, *xlabel=''*, *ylabel=''*, *title=''*, *font_size='medium'*, *scale=1.0*, *scale_t=1.0*)
Generate and save a time series plot of the history data.

Parameters

- **variable** (*str*) – Variable to plot.
- **node** (*int*) – Node number to plot.
- **t_lim** (*lst[fl64,fl64]*) – Time limits on x axis.
- **var_lim** (*lst[fl64,fl64]*) – Variable limits on y axis.
- **marker** (*str*) – String denoting marker and linetype, e.g., ':s', 'o–'. Default is 'x-' (solid line with crosses).

- **color** (*str*) – String denoting colour. Default is 'k' (black).
- **save** (*str*) – Name to save plot.
- **xlabel** (*str*) – Label on x axis.
- **ylabel** (*str*) – Label on y axis.
- **title** (*str*) – Title of plot.
- **font_size** (*str*) – Font size for axis labels.
- **scale** (*fl64*) – If a single number is given, then the output variable will be multiplied by this number. If a two element list is supplied then the output variable will be transformed according to y = scale[0]*x+scale[1]. Useful for transforming between coordinate systems.
- **scale_t** (*fl64*) – As for scale but applied to the time axis.

4.2.5 Node fluxes

Inter node flux information is parsed using the `fnodeflux` class. Time plotting functionality is not currently available for this class.

class `fpost.`**`fnodeflux`**(*filename=None*)
Internode flux information.

Can read either water or CO2 internode flux files.

The fnodeflux object is indexed first by node pair - represented as a tuple of node indices - and then by either the string 'liquid' or 'vapor'. Data values are in time order, as given in the 'times' attribute.

4.2.6 Attributes

`fnodeflux.`**`times`**
(*lst*) times for which node flux information is reported.

`fnodeflux.`**`nodepairs`**
(*lst*) node pairs for which node flux information is available. Each node pair is represented as a two item tuple of node indices.

`fnodeflux.`**`filename`**
(*str*) filename target for internode flux file.

4.2.7 Methods

`fnodeflux.`**`read`**(*filename*)
Read in FEHM contour output information.

Parameters **filename** (*str*) – File name for output data, can include wildcards to define multiple output files.

4.3 Multi document pdf

class `fpost.`**`multi_pdf`**(*combineString='gswin64'*, *save='multi_plot.pdf'*, *files=[]*, *delete_files=True*)
Tool for making a single pdf document from multiple eps files.

4.3.1 Attributes

multi_pdf.**save**
(*str*) Name of the final pdf to output.

multi_pdf.**files**
(*lst[str]*) List of eps files to be assembled into pdf.

multi_pdf.**combineString**
(*str*) Command line command, with options, generate pdf from multiple eps files. See manual for further instructions.

4.3.2 Methods

multi_pdf.**add**(*filename*, *pagenum=None*)
Add a new page. If a page number is specified, the page will replace the current. Otherwise it will be appended to the end of the document.

Parameters

- **filename** (*str*) – Name of .eps file to be added.
- **pagenum** (*int*) – Page number of file to be added.

multi_pdf.**insert**(*filename*, *pagenum*)
Insert a new page at the given page number.

Parameters

- **filename** (*str*) – Name of .eps file to be inserted.
- **pagenum** (*int*) – Page number of file to be inserted.

multi_pdf.**make**()
Construct the pdf.

4.3.3 Examples

CHAPTER

FIVE

FVARS: FEHM THERMODYNAMIC CALCULATIONS

It is occasionally useful to be able to calculate thermodynamic properties of the fluids that FEHM simulates. For convenience, the functions that calculate these properties, for two-phase water and CO2, are supplied here. Density (`dens()`), enthalpy (`enth()`) and viscosity (`visc()`), and their corresponding derivatives with respect to pressure and temperature, can be calculated for given pressure and temperature conditions. Furthermore, the position and derivatives of the saturation line for water can be calculated for a given temperature or pressure (`sat()` and `tsat()`).

Note, these functions are vectorised; that is, the user can supply vectors of temperature and pressure to output a vector of the corresponding variable.

5.1 Functions

`fvars.`**`dens`**(*P*, *T*, *derivative=''*)

Return liquid water, vapor water and CO2 density, or derivatives with respect to temperature or pressure, for specified temperature and pressure.

Parameters

- **P** (*fl64*) – Pressure (MPa).
- **T** (*str*) – Temperature (degC)
- **derivative** – Supply 'T' or 'temperature' for derivatives with respect to temperature, or 'P' or 'pressure' for derivatives with respect to pressure.

Returns Three element tuple containing (liquid, vapor, CO2) density or derivatives if requested.

`fvars.`**`enth`**(*P*, *T*, *derivative=''*)

Return liquid water, vapor water and CO2 enthalpy, or derivatives with respect to temperature or pressure, for specified temperature and pressure.

Parameters

- **P** (*fl64*) – Pressure (MPa).
- **T** (*str*) – Temperature (degC)
- **derivative** – Supply 'T' or 'temperature' for derivatives with respect to temperature, or 'P' or 'pressure' for derivatives with respect to pressure.

Returns Three element tuple containing (liquid, vapor, CO2) enthalpy or derivatives if requested.

fvars.**visc**(*P*, *T*, *derivative=''*)
Return liquid water, vapor water and CO2 viscosity, or derivatives with respect to temperature or pressure, for specified temperature and pressure.

Parameters

- **P** (*fl64*) – Pressure (MPa).
- **T** (*str*) – Temperature (degC)
- **derivative** – Supply 'T' or 'temperature' for derivatives with respect to temperature, or 'P' or 'pressure' for derivatives with respect to pressure.

Returns Three element tuple containing (liquid, vapor, CO2) viscosity or derivatives if requested.

fvars.**sat**(*T*)
Return saturation pressure and first derivative for given temperature.

Parameters **T** (*fl64*) – Temperature (degC)

Returns Two element tuple containing (saturation pressure, derivative).

fvars.**tsat**(*P*)
Return saturation temperature and first derivative for given pressure.

Parameters **P** (*fl64*) – Pressure (degC)

Returns Two element tuple containing (saturation temperature, derivative).

fvars.**fluid_column**(*z*, *Tgrad*, *Tsurf*, *Psurf*, *iterations=3*)
Calculate thermodynamic properties of a column of fluid.

Parameters

- **z** (*ndarray*) – Vector of depths at which to return properties. If z does not begin at 0, this will be prepended.
- **Tgrad** (*fl64*) – Temperature gradient in the column (degC / m).
- **Tsurf** (*fl64*) – Surface temperature (degC).
- **Psurf** – Surface pressure (MPa).
- **iterations** (*int*) – Number of times to recalculate column pressure based on updated density.

Returns Three element tuple containing (liquid, vapor, CO2) properties. Each contains a three column array corresponding to pressure, temperature, density, enthalpy and viscosity of the fluid.

5.2 Examples

1. Calculate the density of liquid water at 10 MPa and 150degC

   ```
   rho=dens(10,150)[0]
   ```

2. Calculate the derivative of water vapor enthalpy with respect to temperature, at 2 MPa and 200 - 250 degC.

   ```
   dhdt=enth(2,np.linspace(200,250,10),'T')[1]
   ```

3. Calculate the pressure at which water turns two-phase, for a temperature of 200degC.

   ```
   Psat=sat(200)[0]
   ```

4. Calculate the viscosity of CO2 at 40degC and 9 MPa.

```
mu=visc(9,40)[2]
```

5. Calculate the pressure of a column of CO2, with a surface pressure of 8 MPa, and a geothermal gradient of 25degC / km.

```
z=np.linspace(0,4e3,200)
```

```
Pco2=fluid_column(z,0.025,20,8)[2][:,0]
```

CHAPTER

SIX

TUTORIALS

These tutorials build script files from scratch that, when executed, construct, run and process the results of, an FEHM simulation. One way to enjoy this tutorial, is create a new script file and copy these commands across as you go through them. The script can be executed at any stage within a python terminal using `execfile('myScript.py')` so that you can inspect the variables and objects, or just generally mess around in python.

Within a tutorial, any code commands given are assumed to be appended to the end of a single PyFEHM script for that tutorial. A complete python script for each tutorial should have been received with PyFEHM.

6.1 Cube with fixed pressures/temperatures

This first tutorial describes the set up, simulation and post-processing of a simple cube model. The model contains

1. Three zones of contrasting permeability.
2. An initially homogeneous temperature and pressure distribution.
3. Fixed temperature and pressure boundary conditions.

Generation of a simple grid and construction of two contour plots of temperature and pressure are also demonstrated.

6.1.1 Getting started

Because this is a new script that builds a model and does some post-processing, we will need access to a few modules. So python can find the PyFEHM library files, first write

```
import os,sys
sys.path.append('c:\\python\\pyfehm')
```

Note, if you have created a PYTHONPATH environment variable that points to *c:\python\pyfehm* you can skip this step.

If your PyFEHM library files (*fdata.py, fgrid.py*, etc) are in a different location, then change the path to that location. If this path is contained in the PYTHONPATH environment variable, then you can skip this step.

Get access to PyFEHM grid generation, model construction and post-processing utilities.

```
from fdata import*
from fpost import*
```

As a first action, let us create an empty model object.

```
dat = fdata()
```

Define a root label for use as the default naming convention

```
root = 'tut1'
```

6.1.2 Grid generation

We will create a simple grid with 11 nodes on each side, spaced 1 m apart. First use `np.linspace()` to create a vector of these positions. The use the `fgrid.make()` command to create the grid, and the `fgrid.plot()` command to generate a visualisation (see Figure *7.1*).

```
x=np.linspace(0,10,11)
```

```
dat.grid.make(root+'_GRID.inp',x=x,y=x,z=x)
```

```
dat.grid.plot(root+'_GRID.png',color='r',angle=[45,45])
```

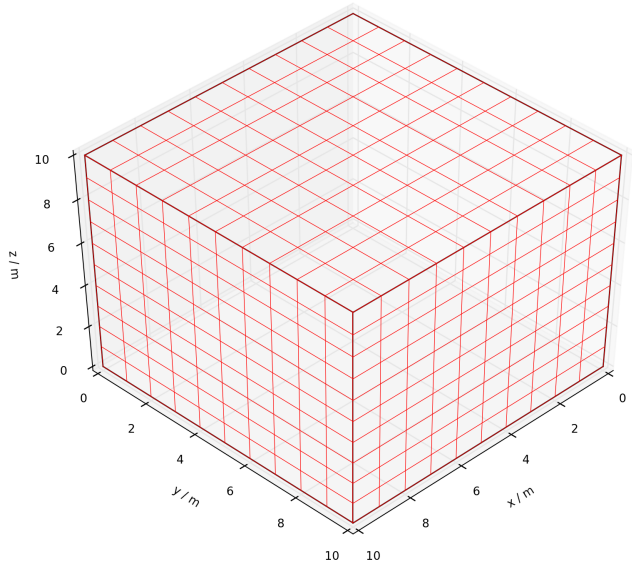


Figure 6.1: View of grid, as produced by `fgrid.plot()`.

6.1.3 Zone creation

We need to create zones corresponding to three layers with different material properties. For zones that are bounded by a rectangle, the easiest method for zone generation is using the `fzone.rect()` command. Below we add a zone with index = 1, named `'lower'` that includes all the zones with z-coordinates between -0.1 and 3.1.

First, create the zone object with index and name.

```
zn=fzone(index=1,name='lower')
```

Next, use the `fzone.rect()` command to define its limits.

```
zn.rect([-0.,-0.,-0.1],[10.1,10.1,3.1])
```

Finally, attach the zone to the `fdata` object.

```
dat.add(zn)
```

This three step process can be simplified by calling the `fdata.new_zone()` method, which takes as input information about the zone (index, name, limits) and automates object creation and attachment processes. The code below is equivalent to the three commands above

```
dat.new_zone(index=1, name='lower', rect=[[-0.,-0.,-0.1], [10.1,10.1,3.1]])
```

Two more zones, called 'middle' and 'upper' are created in the same way.

```
dat.new_zone(index=2, name='middle', rect=[[-0.,-0.,-3.1], [10.1,10.1,6.1]])
```

```
dat.new_zone(index=3, name='upper', rect=[[-0.,-0.,-6.1], [10.1,10.1,10.1]])
```

In the next section, we will look at assigning differing permeability properties to these zones. However, if we know that this zone is being created for the sole purpose of assigning it some permeability, then why not do that during zone creation? This is achieved by passing additional material property arguments to the `new_zone()` method. For example, to assign anisotropic permeability to the `'upper'` zone we should have written

```
dat.new_zone(index=3, name='upper', rect=[[-0.,-0.,-6.1], [10.1,10.1,10.1]],
permeability=[1.e-14,1.e-14,1.e-15])
```

This single command will both create the zone and create a PERM macro for that zone with the specified permeability properties. If isotropic permeability is desired, the argument can be passed as a single float, i.e., `permeability=1.e-14`.

6.1.4 Adding macros

Now that the zones have been established, we can begin assigning material properties, either to specific zones or to the entire model. For example, adding rock properties to the entire model.

First, create a new `'rock'` `fmacro` object with a tuple of parameter name/value pairings. See Table *3.1* for parameter names. Note, by leaving the zone argument blank, the macro properties are assigned to all nodes in the model.

```
rm=fmacro('rock',param=(('density',2500),('specific_heat',1000),('porosity',0.1)))
```

Finally, attach the macro to the `fdata` object.

```
dat.add(rm)
```

Note that, if ROCK properties are omitted, FEHM does not generally behave well. Therefore, if PyFEHM notes that no global rock properties have been assigned, it will add its own default values before running the simulation (and print a warning to notify the user). This applies for thermal conductivity properties assigned through COND.

Now we assign different permeability properties to each of the three zones defined in the previous section.

First, create a new 'perm' `fmacro` object with a tuple of parameter name/value pairings. This time, the index of the first zone, 1, is assigned to the zone argument.

```
pm=fmacro('perm',zone=1,param=(('kx',1.e-15),('ky',1.e-15),('kz',1.e-16)))
```

Attach the macro to the `fdata` object.

```
dat.add(pm)
```

The process can be shortened by assigning permeability properties directly to the zone. In this case, PyFEHM will create the corresponding macro object (if it needs to, one may already exist) and attach it to the fdata structure.

```
dat.zone['lower'].permeability=[1.e-15,1.e-15,1.e-16]
```

In this variant, permeability is assigned to the variable `kmid` ahead of time, then called during assignment.

```
kmid=1.e-20
```

```
dat.zone[2].permeability=kmid
```

Finally, permeability properties for the 'upper' zone were assigned during zone creation, no further action is required.

6.1.5 Initial and boundary conditions

We wish to assign initial pressures and temperatures within the cube. The PRES macro is used for this

Create a new 'pres' fmacro object with a tuple of parameter name/value pairings. The zone argument is left blank so that initial conditions are assigned to all nodes.

```
pres=fmacro('pres',param=(('pressure',5.),('temperature',60.),('saturation',1)))
```

```
dat.add(pres)
```

Again, this process is made a little simpler by assigning to the zone directly

```
dat.zone.Pi=1.
```

```
dat.zone.Ti=60.
```

We wish to set the temperature at the top and bottom boundaries to 30 and 80 degC, respectively. To do this, we will take advantage of boundary zones automatically created by PyFEHM during grid generation. These zones have indices 999, 998, 997, 996, 995, 994 corresponding to names 'XMIN','XMAX','YMIN','YMAX','ZMIN','ZMAX'.

Create a new 'hflx' fmacro object with a tuple of parameter name/value pairings. Setting the 'multiplier' to a large value will ensure that the temperature is maintained very near to 'heat_flow'. 'ZMAX' is specified in the zone argument.

```
hflx=fmacro('hflx',zone='ZMAX',param=(('heat_flow',30),('multiplier',1.e10)))
```

```
dat.add(hflx)
```

This process can be streamlined by calling the fzone.fix_temperature() command from a zone associated with the fdata object.

```
dat.zone['ZMIN'].fix_temperature(80)
```

Now set pressures at two lateral boundaries, 'YMIN' and 'YMAX', to 4 and 6 MPa, respectively. This can be achieved using the **FLOW** macro

```
flow=fmacro('flow',zone='YMIN',param=(('rate',6.),('energy',-60.),('impedance',1.e6)))
```

```
dat.add(flow)
```

Once again, this process is streamlined by calling the fzone.fix_pressure() command from a zone associated with the fdata object.

```
dat.zone['YMAX'].fix_pressure(P=4.,T=60.)
```

6.1.6 Running the simulation

Before running the simulation, we want to specify variables to be printed out - in this case temperature and pressure - and request that this occurs at every time step.

```
dat.cont.variables.append(['xyz','temperature','pressure'])
```

```
dat.cont.timestep_interval=1
```

Now set the end of the simulation to be 10 days using the shortcut attribute.

```
dat.tf=10.
```

An alternative way to set the final time is directly through the time attribute

```
dat.time['max_time_TIMS']=10.
```

However, using the shortcut attribute `tf` is simpler.

Assign the root label to be used for all files created by FEHM during its execution.

```
dat.files.root=root
```

Run the simulation, assuming 'fehm.exe' is sitting in the current directory.

```
dat.run(root+'_INPUT.dat',exe='fehm.exe',files=['outp'])
```

6.1.7 Visualisation

Finally, we produce a couple of contour plots of vertical slices through the model, to verify that pressures and temperatures are behaving, at least qualitatively, as we expect.

First, load in the contour information, printed to '.csv' files. Wildcards (*) are used to load multiple files.

```
c=fcontour(root+'*.csv')
```

Now, call the slice plotting methods. Slice plots are shown in Figure 7.2.

```
c.slice_plot(save='Tslice.png', time=[c.times[-1], c.times[-2]], cbar=True,
levels=11, slice=['x',5], variable='T', method='linear', title='temperature /
degC', xlabel='y / m', ylabel='z / m')
```

```
c.slice_plot(save='Pslice.png', cbar=True, levels=np.linspace(4,6,9),
slice=['x',5], variable='P', method='linear', title='pressure / MPa',
xlabel='y / m', ylabel='z / m')
```

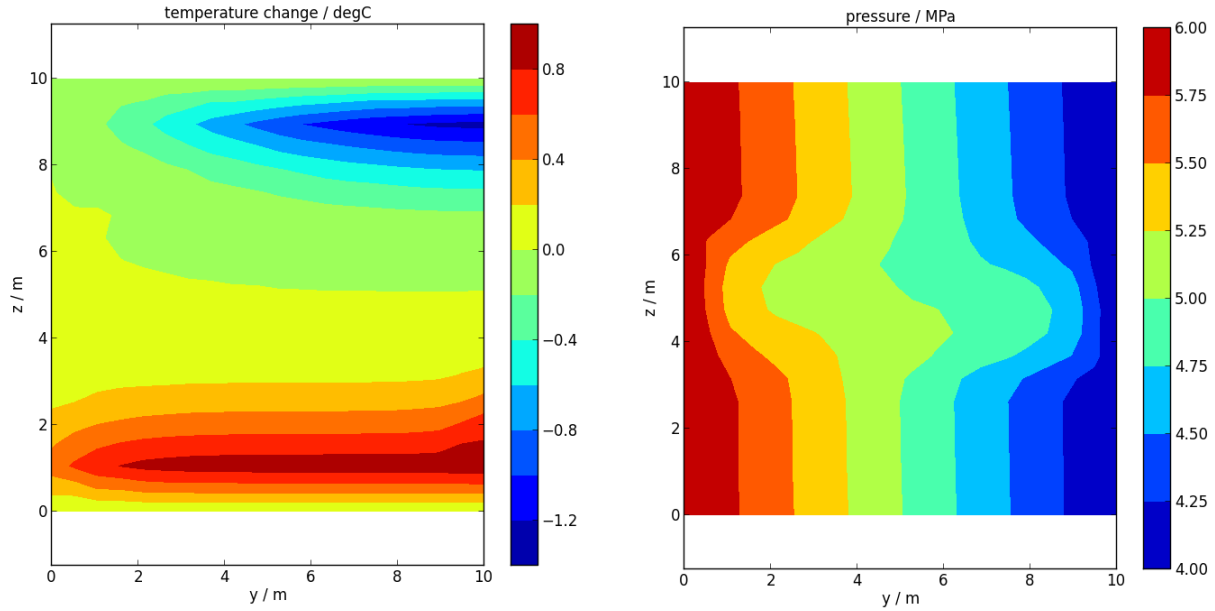


Figure 6.2: Left: Contour plot of a vertical slice of temperature change during the final time step, as produced by `slice_plot()`. Right: Contour plot of a vertical slice of pressure.

6.2 Five-spot injection and production

This second tutorial sets up a five-spot injection/production simulation for an EGS style system. The model considers cold-water injection into a hot reservoir with attendant effects on stress. The model grid is generated using built-in PyFEHM tools and is a quarter-reduced domain.

6.2.1 First steps

As in the previous tutorial, we will need access to a few modules. If the PYTHONPATH environment variable is not set, follow the instructions in tutorial 1 for adding a path to your PyFEHM library files.

Get access to PyFEHM grid generation, model construction and post-processing utilities.

```
from fdata import*
```

```
from fpost import*
```

As a first action, let us create an empty model object and assign the working directory to be tutorial2.

```
dat = fdata(work_dir='tutorial2')
```

6.2.2 Grid generation

Because we will model injection and production against fixed pressures, it is crucial that pressure gradients in the vicinity of the well be fully-resolved. To this end, it will be useful to have a mesh with variable resolution: `fgrid.make()` is useful for generating such meshes.

First, let's define some dimensions for the mesh. We want it to extend 1 km in each of the horizontal dimensions, and span between -500 and -1500 m depth (assuming `z = 0` corresponds to the surface).

```
X0,X1 = 0,1.e3
```

```
Z0,Z1 = -1.5e3,-0.5e3
```

We need to know the position of the injection and production wells so the mesh can be refined in the vicinity. The injection well is in the centre (the corner of the quarter spot) and the production well is at `(300,300)`

```
injX,injY = 0.,0.
```

```
proX,proY = 300.,300.
```

A power-law scaled node spacing will generate closer spacing nearer the injection and production locations.

```
base = 3
```

```
dx = proX/2.
```

```
x = dx**(1-base)*np.linspace(0,dx,8)**base
```

```
dx2 = X1 - proX
```

```
x2 = dx2**(1-base)*np.linspace(0,dx2,10)**base
```

```
X = np.sort(list(x) + list(2*dx-x)[:-1] + list(2*dx+x2)[1:])
```

Injection and production will be hosted within an aquifer, confined above and below by a caprock. We need to define the extent of these formations.

```
Za_base = -1.1e3
```

```
Za_top = -800.
```

As all the action will be going on in the aquifer, we will want comparatively more nodes in there.

```
Z = list(np.linspace(Z0,Za_base,5)) + list(np.linspace(Za_base,Za_top,11))[1:]
+ list(np.linspace(Za_top,Z1,5))[1:]
```

Now that the node positions have been defined, we can create the mesh using the `fgrid.make()` command.

```
dat.grid.make('quarterGrid.inp', x = X, y = X, z = Z)
```

Plot a picture of the grid to check it is as expected (see Figure *7.3*).

```
dat.grid.plot('quarterGrid.png', angle = [45,45], color = 'b', cutaway =
[proX, proY, -1000.])
```

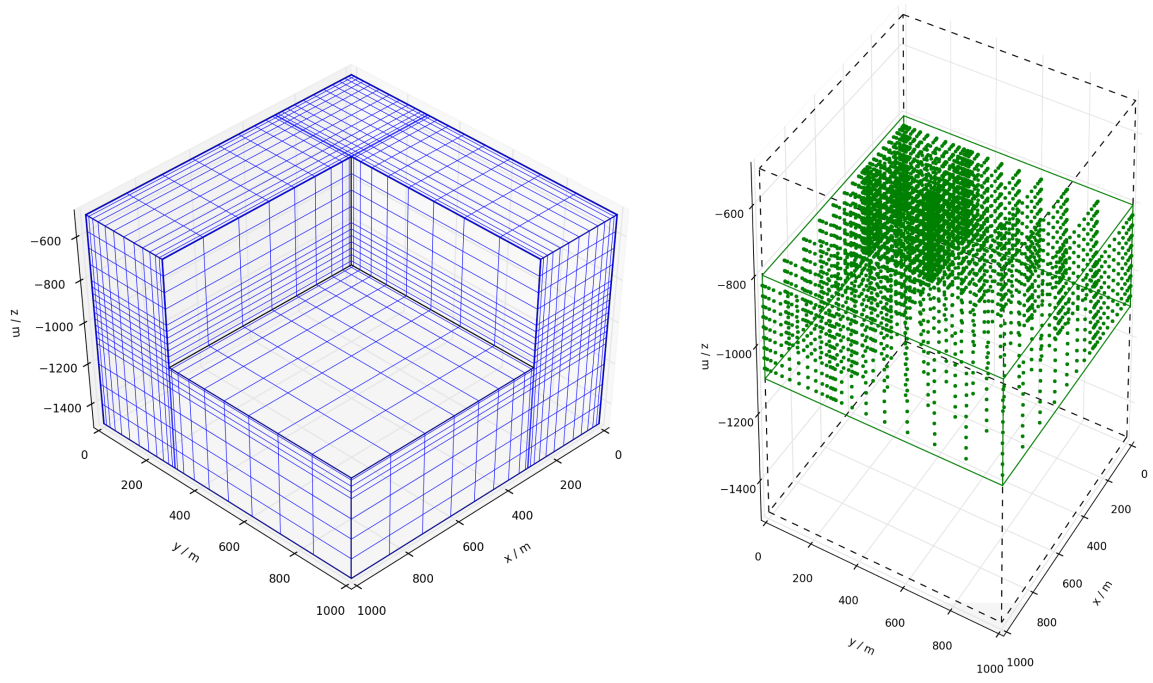


Figure 6.3: Left: Cutaway view of quarterGrid.inp, as produced by `fgrid.plot()`. Right: Nodes contained in `dat.zone['reservoir']`, as produced by `fzone.rect()` and illustrated by `fzone.plot()`.

Note that the files *quarterGrid.inp* and *quarterGrid.png* should now exist in the python working directory. Grid information is now available within `dat`, e.g., `dat.grid.node` should be populated. This concludes grid generation.

6.2.3 Zone creation

This problem has three defined zones, a reservoir (denoted `res`) and upper and lower confining formations (denoted `con`). For simplicity, we will not assign different material properties to the two confining layers.

Before we can assign material properties via the **PERM**, **ROCK** and **COND** macros, we need zones to which these macros can be assigned. As these zones are rectangular, we will define them using the `new_zone()` method, passing it the rect argument for a bounding box defined by two corner points. First the reservoir zone.

dat.‘‘`new_zone(10, name='reservoir', rect=[[X0-0.1,X0-0.1,Za_base+0.1],`
`[X1+0.1,X1+0.1,Za_top-0.1]])`

We can plot which nodes are contained in the reservoir zone to verify we have made the correct selection (see Figure *7.1*).

```
dat.zone['reservoir'].plot('reservoirZone.png', color='g', angle = [30,30])
```

The two confining zones are generated similarly

```
dat.new_zone(20, name='confining_lower', rect=[[X0-0.1,X0-0.1,Z0-0.1],
[X1+0.1,X1+0.1,Za_base+0.1]])
```

```
dat.new_zone(21, name='confining_upper', rect=[[X0-0.1,X0-0.1,Za_top-0.1],
[X1+0.1,X1+0.1,Z1+0.1]])
```

Now that the zones have been defined, we can begin assigning material properties.

6.2.4 Material property assignment

First, declare some parameters: permeability (`perm`), density (`rho`), porosity (`phi`), thermal conductivity (`cond`), and specific heat (`H`).

```
perm_res, perm_con = 1.e-14, 1.e-16
```

```
rho_res, rho_con = 2300., 2500.
```

```
phi_res, phi_con = 0.1, 0.01
```

```
cond = 2.5
```

```
H = 1.e3
```

Now, create a new `fmacro` object to which to assign reservoir permeability. For this first time, we will take an unnecessary number of steps to demonstrate macro assignment

```
perm=fmacro('perm')
```

```
perm.zone='reservoir'
```

```
perm.param['kx']=perm_res
```

```
perm.param['ky']=perm_res
```

```
perm.param['kz']=perm_res
```

```
dat.add(perm)
```

In general, this process can be streamlined to a few or even a single step, e.g., assigning upper confining formation permeability

```
perm=fmacro('perm',zone=21,           param=(('kx',perm_con), ('ky',perm_con),
('kz',perm_con)))
```

```
dat.add(perm)
```

Permeability can also be assigned to a zone through its `permeability` attribute. Behind the scenes, PyFEHM takes care of the macro definition and zone association required by FEHM to assign this permeability.

```
dat.fdata.zone['confining_lower'].permeability=perm_con
```

Similarly, rock properties are defined either through a macro object, or by zone attributes, e.g., through a macro object

```
dat.add(fmacro('rock',zone=dat.zone['reservoir'],  param=(('density',rho_res),
('specific_heat',H), ('porosity',phi_res))))
```

```
dat.add(fmacro('rock',zone=dat.zone['confining_lower'],param=(('density',rho_con),
('specific_heat',H), ('porosity',phi_con))))
```

through zone attributes

```
dat.fdata.zone['confining_upper'].density=rho_con
```

```
dat.fdata.zone['confining_upper'].specific_heat=H
```

```
dat.fdata.zone['confining_upper'].porosity=phi_con
```

Thermal conductivity (**COND**) properties are to be the same everywhere, so we will use the conductivity attribute of zone 0.

```
dat.fdata.zone[0].conductivity=cond
```

6.2.5 Injectors and producers

The EGS problem requires both injection and production wells. To demonstrate some of FEHM's flexibility, we will include a source that injects cold fluid at a fixed rate, and a production well that operates against a specified production pressure.

First we require a zone for each of the production and injection wells. For the injection well, we will consider fluid exiting the wellbore over an open-hole length, i.e., sources at multiple nodes. For the production well, we will consider a single feed-zone at a fixed depth, i.e., a single node sink.

First define the open hole injection nodes. The aquifer extends from -1100 to -800 m depth; we will choose a 100 m open hole section between -1000 and -900 m. Nodes contained in this zone are well-defined by the bounding box approach of `rect()`.

```
dat.new_zone(30, name='injection', rect=[[injX-0.1,injY-0.1,-1000.1],[injX+0.1,injY+0.1,-8
```

Note that, in general, it is good to enlarge the bounding box by some nominal amount (in this case, 0.1 m) to insure that the nodes are in fact 'bounded' by the box.

Now to define the production feed-zone; let's suppose that it is at the known depth of -950 m. As we want to find the node closest to this location, we will use the `node_nearest_point()` command.

```
pro_node=dat.grid.node_nearest_point([proX,proY,-950])
```

```
dat.new_zone(40, 'production', nodelist=pro_node)
```

Now that the zones have been defined, we will assign mass flow generators via the **FLOW** and **BOUN** macros. First the production well, which is simply production against a fixed pressure, let's say 6 MPa. We will use the `fmacro('flow')` object with a non-zero impedance parameter indicating production against a fixed pressure.

```
flow=fmacro('flow',zone='production',          param=(('rate',6), ('energy',30),
('impedance',1.)))
```

```
dat.add(flow)
```

There are multiple nodes in the injection zone, but we wish to specify a single mass injection rate. The **BOUN** macro is useful for distributing a fixed source across multiple nodes. Recall that, in contrast to **FLOW**, **BOUN** has its own macro object, `fboun`.

```
injRate=4.
```

```
injTemp=60.
```

```
boun=fboun(zone=['injection'],times=[0,1e10],variable=[['dsw',-injRate,-injRate],
['ft',injTemp,injTemp]])
```

```
dat.add(boun)
```

This creates a source of 60degC water, injecting at a rate of 2 kg/s, distributed evenly across all nodes in the 'injection' zone ('dsw' = distributed source water). By assigning a large value in `boun.times` we ensure that the source will continue to operate for the entire simulation.

6.2.6 Initial conditions

Before running the simulation we need to set up initial conditions for temperature and pressure. For simplicity, we will assume that gradients in both are linear from the surface, although more complex configurations can of course be accommodated.

For the temperature field, we will use the **GRAD** macro, with a 70degC / km temperature gradient, 25degC surface temperature corresponding to `z = 0`. Again, by omitting the zone parameter when creating the `fmacro` object, the macro will automatically be applied to all nodes. More complex initial temperature distributions can be created by passing measured temperature profile data to the `temperature_gradient()` command.

```
dat.add(fmacro('grad', param=(('reference_coord',0.), ('direction',3),
('variable',2), ('reference_value',25.), ('gradient',-0.06))))
```

For the pressure distribution, we will assume this is initially hydrostatic, although FEHM will recalculate pressures based on the temperature dependent fluid density. Specifying the pressure distribution requires two macros: (i) a **GRAD** for the pressure gradient, and (ii) a fixed pressure, implemented by the `fix_pressure()` method, at the top surface representing the submerged upper surface of the model. First add the pressure gradient

```
dat.add(fmacro('grad', param=(('reference_coord',0.), ('direction',3),
('variable',1), ('reference_value',0.1), ('gradient',-9.81*1e3/1e6))))
```

Upon grid initialisation, PyFEHM already created the zone for the top surface, assigned the key `'ZMAX'`. We use the `fix_pressure()` method to assign surface pressure conditions

```
dat.zone['ZMAX'].fix_pressure(P=0.1+Z1*-9.81*1e3/1e6,T=25.+Z1*-0.06)
```

6.2.7 Setting up stresses

Set up for a stress solution in FEHM requires (i) specification of material parameters relevant to mechanical deformation, e.g., Young's modulus, thermal expansion coefficient, (ii) boundary conditions, either displacement or force, and (iii) optionally an initial stress state or (iv) a stress-permeability model. In this example we will include the first three features.

Initial stress states are calculated and loaded in via FEHM's restart or INCON file. This file also contains information on the restart of temperature and pressure; therefore, to perform a stress restart we first require the temperature and pressure restart information. The easiest way to obtain this is to run one time step of the model (without the stress solution) and request it to output a restart file at the end of the time step. We do this by setting the `dat.files.rsto` attribute to the name of the restart file.

```
dat.dtn=1
```

```
dat.files.rsto='EGS_INCON.ini'
```

```
dat.run('EGS_flow_INPUT.dat')
```

Note that, because we have not specified an FEHM executable in the *exe* argument of `run`, PyFEHM will automatically search for *fehm.exe* in the current working directory.

Now that the model has run a single time step, the output restart file (containing only temperature and pressure data) can be read as an initial conditions file (`fincon`).

```
dat.incon.read('EGS_INCON.ini')
```

Vertical gradients in the three principal stresses can be calculated using the `fincon.stressgrad()` command. In this case we will request that PyFEHM calculates the vertical load by integrating the variable density information supplied in `fmacro['rock']`, and the horizontal stresses as fractions of the vertical.

```
dat.incon.stressgrad(xgrad=0.6, ygrad=0.8, zgrad=2500*abs(Z1)*9.81/1e6,
calculate_vertical=True, vertical_fraction=True)
```

Now to turn the stress solution on and assign material parameters to the various zones in the model. We will assign different deformation parameters (**ELASTIC**) to the reservoir and confining units, but assume that stress-flow coupling parameters (**BIOT**) are the same throughout. These are material properties and thus can be assigned using appropriate zone attributes

```
dat.strs.on()
E_res,E_con = 2e3,2e4
nu_res,nu_con = 0.15,0.35
dat.fdata.zone['reservoir'].youngs_modulus=E_res
dat.fdata.zone['confining_lower'].youngs_modulus=E_con
dat.fdata.zone['confining_upper'].youngs_modulus=E_con
dat.fdata.zone['reservoir'].poissons_ratio=nu_res
dat.fdata.zone['confining_lower'].poissons_ratio=nu_con
dat.fdata.zone['confining_upper'].poissons_ratio=nu_con
dat.fdata.zone[0].thermal_expansion=3.e-5
dat.fdata.zone[0].pressure_coupling=1.
```

Because we have prescribed the initial stress state, boundary conditions and body forces to not need to reflect gravitational or tectonic loading. For this reason, we should turn body force calculations off, and assign fixed displacement boundary conditions to prevent model drift (roller boundary conditions on the `x=0`, `y=0` and `z=0` planes). Note that these zones already exist, automatically created by PyFEHM with the names `'XMIN'`, `'YMIN'` and `'ZMIN'`. Furthermore, it is advisable when defining boundary conditions to pass the `write_one_macro=True` argument - this improves stability of the stress solution.

```
dat.strs.fem=1.
dat.strs.bodyforce=0.
dat.sol['element_integration_INTG']=-1
dat.add(fmacro('stressboun', zone=60, subtype='fixed', param=(('direction',1),
('value',0))))
dat.add(fmacro('stressboun', zone=61, subtype='fixed', param=(('direction',2),
('value',0))))
dat.add(fmacro('stressboun', zone=62, subtype='fixed', param=(('direction',3),
('value',0))))
```

Note that, for a well-behaved model I have set the `dat.strs.fem` and `dat.sol['element_integration_INTG']` attributes to particular values - mess with these at your peril.

6.2.8 Running the model

Before running the final model, we need to tell FEHM to output variable information at particular locations and times. This is done through the `dat.cont` and `dat.hist` attributes.

First, to the contour output, we will request information on pressure, temperature, stress and other variables at the end of the simulation, and every six months.

```
dat.cont.variables.append(['xyz', 'pressure', 'liquid', 'temperature',
'stress', 'displacement', 'permeability'])
```

```
dat.cont.format='surf'
```

```
dat.cont.timestep_interval=1000.
```

```
dat.cont.time_interval=365.25/2.
```

For history output, we will request information on temperature, pressure and flow, at each time step for the single production node and the total injection zone.

```
dat.hist.variables.append(['temperature', 'pressure', 'flow', 'zfl'])
```

```
dat.hist.format='surf'
```

```
dat.hist.nodeflux.append(dat.zone['production'].nodelist[0])
```

```
dat.hist.zoneflux.append(dat.zone['injection'])
```

Finally, we need to change the time step count (as this was originally set to 1), and request the simulation to continue for 10 years with a maximum time step of 1 year. Shortcut attributes are utilised for these three requests.

```
dat.dtn=1000
```

```
dat.tf=365.25*10.
```

```
dat.dtmax=365.25
```

Now to run the simulation. Post-processing of the simulation output will be detailed in the next section.

```
dat.files.root='EGS'
```

```
dat.run('EGS_stress_INPUT.dat',files=['hist','outp','check'])
```

6.2.9 Post-processing

Let's first produce some slice plots of the final temperature, pressure and stress distributions. Contour output is read in by the `fcontour` class. Although we have only requested a single contour output file (at the end of the simulation), the code below is general enough to search through *all* contour output, but take only the latest generated file (presumably corresponding to the end of the simulation).

```
cont=fcontour(dat.files.root+'*.csv',latest=True)
```

There are several options for generating data plots, and we will demonstrate some of PyFEHM's plotting capabilities here.

1. A horizontal slice plot of temperature (see Figure *7.4*).

   ```
   cont.slice_plot(save='temperature_slice.png', cbar=True, levels
   = 10, slice = ['z',-1000], divisions=[100,100], variable='T',
   xlims=[0,500], ylims=[0,500], title='final temperature distribution')
   ```

2. A profile plot of pressure between the injection and production wells (see Figure *7.4*).

   ```
   cont.profile_plot(save='pressure_profile.png', profile=np.array([[0,0,-1000],
   [400,400,-1000]]), variable='P', ylabel='pressure / MPa',
   title='pressure with distance from injector', color='g',
   marker='o--', method='linear')
   ```

3. A vertical profile plot of stress at `x,y = [600,600]` (see Figure *7.5*).

   ```
   cont.profile_plot(save='stress_profile.png', profile=np.array([[600,600,-500],
   [600,600,-1500]]), variable='strs_xx', xlabel='depth / m',
   ylabel='pressure / MPa', title='horizontal stress with depth',
   color='k', marker='-', method='linear', elevationPlot=True)
   ```

4. A cutway plot of temperature (see Figure *7.5*).

```
cont.cutaway_plot(variable='T', save='temperature_cutaway.png',
xlims=[0,400], ylims=[0,400], zlims=[-1000,-800], cbar=True,
levels=np.linspace(60,90,11), grid_lines='k:', title='temperature
contours / $^o$C')
```

Now let's look at some of the time-series plotting capability. First we need to read in the history output for the flow data at the production well.

```
hist=fhistory(dat.files.root+'_flow_.his.dat')
```

Now produce a time series plot of the production data (see Figure *7.6*).

```
hist.time_plot(variable='flow', save='extraction_plot.png',
node=hist.nodes[0], scale_t=1./365.25, xlabel='time / years', ylabel='flow
kg s$^{-1}$')
```

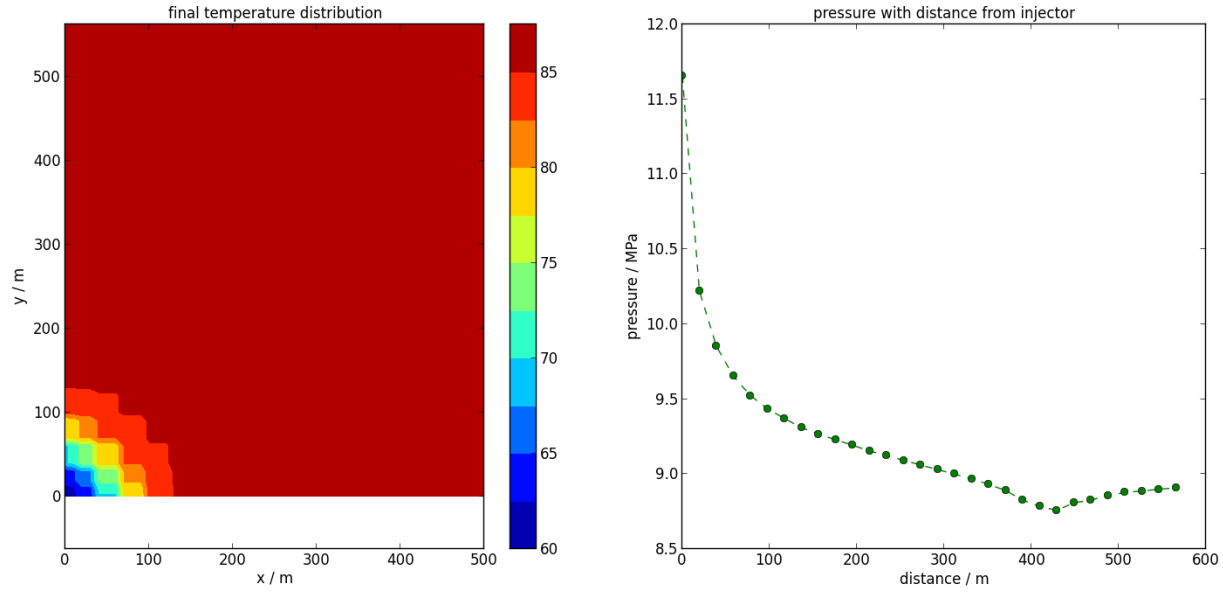


Figure 6.4: Left: Slice plot of final temperature distribution at `z=-1000`, as produced by `slice_plot()`. Right: Horizontal profile plot of pressure with distance from the injector, as produced by `profile_plot()`.

6.3 Multiprocessing and batch job submission

Having constructed and performed an FEHM simulation, often it is necessary to rerun the model for different parameter values. This could be for either calibration or parameter sensitivity purposes. In these situations, the models can be run as batches - with no requirement for information transfer between simulations, they can be run asynchronously in parallel.

Using Python's multiprocessing module, we can generalise a script that creates, runs and postprocesses an FEHM simulation for one specific parameter set so as to distribute multiple simulations across a specified number of cores. As an example, we shall extend the first tutorial (Section *7.1*).

6.3.1 Setting up a batch of simulations

First, import the multiprocessing module.

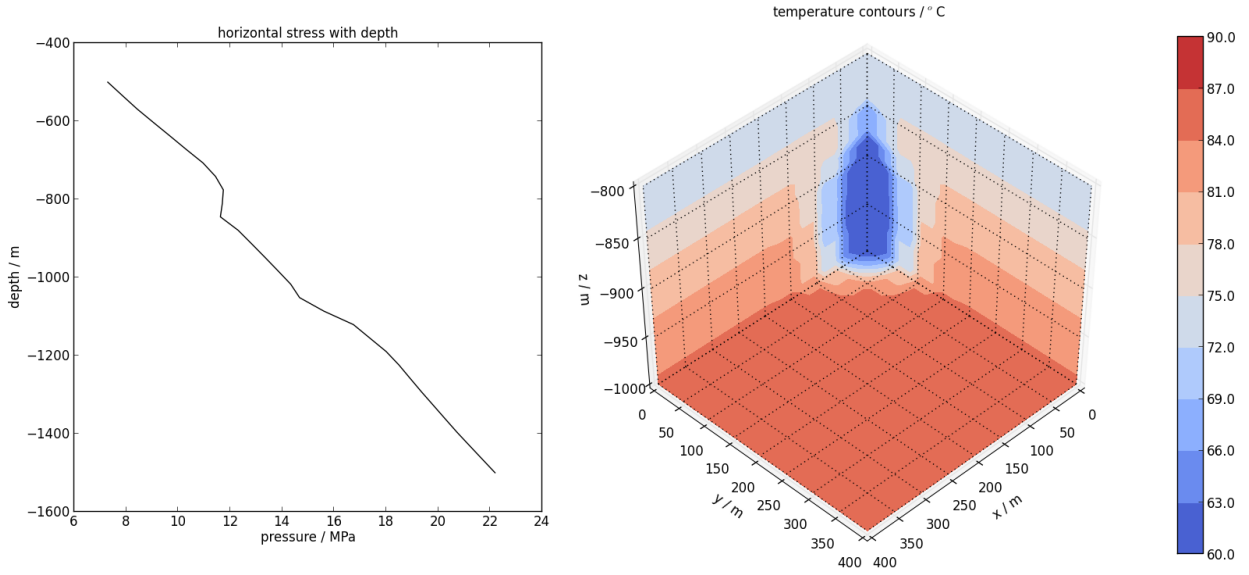


Figure 6.5: Left: Vertical profile plot of horizontal stress, as produced by `profile_plot()`. Right: Cutaway plot of temperature, as produced by `cutaway_plot()`.

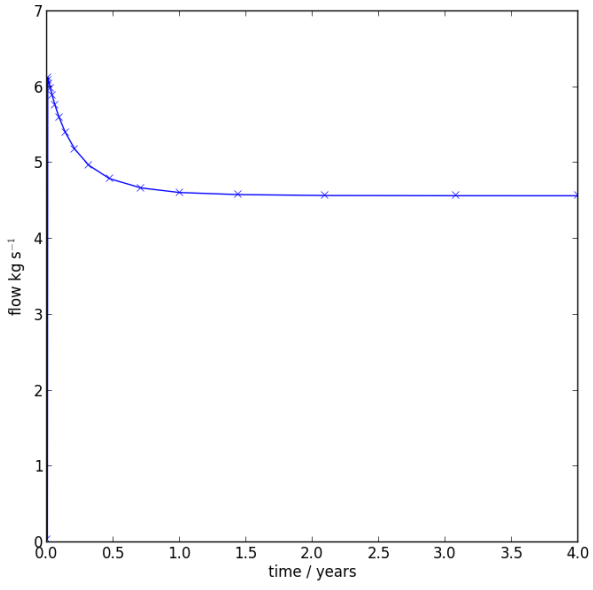


Figure 6.6: Time series plot of production flow rate, as produced by `time_plot()`.

```
import multiprocessing
```

We will use indices to keep track of models. In this case, we will run the model four times with different parameters and so require indices 1 through 4, given by the `range()` command. Additionally, we will choose to distribute these jobs to two or our machine's processors.

```
models = range(1,5)
processors = 2
```

Now we need to define the parameter sets for each of the four simulations. In this case, we are looking at two different permeabilities for the centre layer, and two different temperatures for the boundary condition at the base of the model. Each parameter set is a list, and all parameter sets are contained in a single list, i.e., a nested list structure. The formatting used below is simply a visual aide to keep track of simulations.

```
params = [
#[kmid,Tbase], # legend
[1.e-20,80], # 1
[1.e-20,120], # 2
[1.e-16,80], # 3
[1.e-16,120], # 4
]
```

6.3.2 Generalising to a function

In tutorial 1 (Section *7.1*), we wrote a script to construct, run and post-process a simple FEHM simulation. As a first step, this script must be generalised as a callable function - this is relatively easy.

The corresponding function is defined by writing

```
def simulation(j,param):
```

and then indenting the rest of the script below this line (excluding import commands which should remain outside the function call). The name of the function is `simulation` and it receives two input arguments: `j`, an integer index indicating which model we are running, and `param` a list of parameters to be assigned within the function call. For example, in the case describe here `param` might contain `[1.e-20,120]`, i.e., the second parameter set in the list of all parameter sets, `params`, defined in the previous section.

6.3.3 Parameters as function arguments

In tutorial 1, the parameters to be varied are defined explicitly in the script with the lines

```
kmid=1.e-20
dat.zone['ZMIN'].fix_temperature(80)
```

These lines need to be altered to reflect the general parameters supplied as the argument param. First, we will unpack the list `param` into separate variables by writing

```
kmid, Tbase = param
```

or equivalently

```
kmid=param[0]
Tbase=param[1]
```

Now, `kmid` is clearly defined, and we can simply delete the line reading

```
kmid=1.e=20
```

If left in, this would override the value of `kmid` passed as an argument to `simulation`.

In the case of `Tbase`, we need to replace the argument of `fix_temperature`‘‘ so that it reads

```
dat.zone['ZMIN'].fix_temperature(Tbase)
```

6.3.4 Assigning the simulation a target directory

As we are performing four simulations, for the sake of tidiness, it makes sense that all files pertaining to a simulation should be placed in a separate simulation directory. This is necessary, as calling fehm.exe multiple times simultaneously in the same directory can lead to sharing issues.

Performing simulations in target directories is relatively simple with PyFEHM. The first step is to generate a different root label for each simulation, using the model index (passed to `simulation` as the argument `j`) to distinguish between each. We replace

```
root = 'tut1'
```

with

```
root = 'tut3_'+str(j)
```

which will yield root labels ‘tut3_1’, ‘tut3_2’, ‘tut3_3’ and ‘tut3_4’ for the four simulations.

Second, when the `fdata` object is created, we pass the root label to the input argument `work_dir`.

```
dat = fdata(work_dir=root)
```

This tells PyFEHM that all grid, input, incon and output files should be created in the new folder `root` inside the current work directory. Furthermore, the simulation itself is performed inside `root` and not the current directory.

Two additional changes are required in the post-processing section so that the correct files are targeted. These are

```
c=fcontour(root+'\\*.csv',latest=True)
```

and

```
c.slice_plot(save=root+'\\Tslice.png', cbar=True, levels=11, slice=['x',5],
variable='T', method='linear', title='temperature / degC', xlabel='y / m',
ylabel='z / m')
```

```
c.slice_plot(save=root+'\\Pslice.png', cbar=True, levels=np.linspace(4,6,9),
slice=['x',5], variable='P', method='linear', title='pressure / MPa',
xlabel='y / m', ylabel='z / m')
```

6.3.5 Assembling the processing pool and distributing the simulations

First, we write a simple function that, for a given model index, executes the simulation function (`simulation`) one time, passing it the correct input arguments. The name of this function will be `execute()` and its definition is given below.

```
def execute(j):

    simulation(j,params[j-1])
```

Now there are simply two commands to submit the batch job. The first, `multiprocessing.Pool` assembles a pool of available processors that will be used to run simulations. As a simulation is complete, its processor is returned to the pool and is then available to run a new simulation. The second command maps the simulation function, as

accessed through `execution()` with the set of models to be run `models`, defined in the first section. These two tasks are coded below

```
if __name__ == '__main__':
    p=multiprocessing.Pool(processes = processors)
    p.map(execute,models)
```

The `if` conditional preceding these commands is very important. Omitting this can lead to an infinite loop that spawns a new python process each time - this can rapidly overwhelm the system and is generally counterproductive.

6.3.6 Basic operation of the parallel batch script

When this modified script is executed, the following actions are executed

1. A pool of two available processors is assembled (`multiprocessing.Pool`).
2. Four indices (1,2,3, and 4) corresponding to four models (`params[0]`, `params[1]`, etc.) are submitted to the function `execute`. Because there are two available processors, `execute` proceeds with `j=1` on the first and `j=2` on the second.
3. `execute` passes both `j` and the parameter set `params[j-1]` to the function `simulation()`. This function
   (a) Creates a new directory numbered by `j`.
   (b) Creates a grid in that directory.
   (c) Constructs and runs an FEHM model with the specific parameters in that directory.
   (d) Post-processes output results in that directory.
4. When `simulation` and `execution` complete, the processor is returned to the pool. Model indices `j=3` and `j=4` are still waiting and will take these returned processors as they become available, executing step 3 themselves.
5. When all model indices have been input to and returned by `execute`, the batch simulation is complete.

6.4 Dynamic simulation monitoring

In some instances, the final simulation time for a model is specified as some arbitrarily large value. This is to ensure that the model doesn't finish before the behaviour we are trying to model has. However, if we could observe the simulation as it proceeded, we could monitor for an arbitrary finish criterion and terminate the process when it is satisfied.

In this tutorial, we will demonstrate how to specify an arbitrary kill condition for a simulation - in this case, when the temperature at a particular node drops below a given value.

As in the previous tutorial, we will use as a basis the model developed in tutorial 1.

6.4.1 Defining a kill condition

First, the simulation time is doubled to 20 days.

```
dat.tf=20.
```

The kill condition is defined as a function with particular input and return conditions. The function must accept a single input, an `fdata` object, and return True or False depending on whether the condition has been satisfied.

This function is passed to the `until` argument in the `run()` method when the simulation is executed. In operation, the function is passed the `fdata` instance from which `run()` is called. PyFEHM monitors for output files containing information for a restart simulation that are periodically written by FEHM; these are parsed and the information made available at the nodal level. For example, nodal temperatures are available through the `fnode.T` attribute inside the function.

First define the function

```
def kill_condition(dat):
```

We want to set a kill condition when the node at position [7,7,9] has a temperature less than 50degC. First we need to locate that node

```
    nd=dat.grid.node_nearest_point([7.,7.,9.])
```

This node's state attributes are updated at each time step during the simulation. Thus, all that is required is for us to test the kill condition

```
    return nd.T<50
```

Now run the simulation with the modified `run()` method

```
dat.run(root+'_INPUT.dat',until=kill_condition)
```

This simulation now finishes after 5 time steps or approximately 13 days, instead of the 20 days specified as the end of the simulation.

6.5 Paraview visualisation

PyFEHM supports 3-D visualisation of model and output data in the software Paraview (www.paraview.org). Paraview must be installed ahead of time before For instance, the user can visualise various information on the model grid, e.g., material properties like permeability or porosity, zone information, or output data such as temperatures and pressures. Once loaded in Paraview, these data can be sliced, contoured and otherwise manipulated using Paraview's built-in tools.

An instance of Paraview is initialised by calling the `paraview()` method, either from an `fdata` or `fcontour` object. Behind the scenes, PyFEHM formats and writes out grid, model and output data in the VTK format, and then subsequently loads these files into Paraview.

If a sequence of contour output data files are loaded in, then the user can optionally request two derived variables types. By passing the `paraview()` method `diff=True`, variable differences are created, e.g., the new variable `T_diff` will plot the change in temperature from its initial simulation value. By passing `time_derivative=True`, time derivatives for each variable will be calculated.

PyFEHM will assume that `paraview.exe` is available for command line execution. The user can ensure this is the case by including the folder containing the paraview executable in their PATH environment variable. Alternatively, the user can pass `exe='/path/to/paraview.exe'` directly to the `paraview()` method, or set this path in the config file.

6.5.1 Using the Paraview method

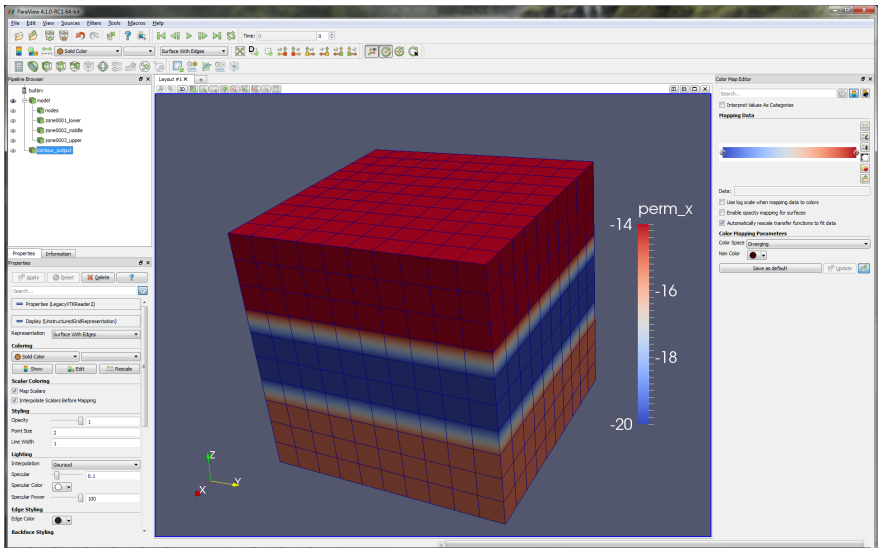
To get started, in tutorial1.py, uncomment the paraview command near the end of the script

```
    dat.paraview(contour=c, diff=True, time_derivatives=True)
```

Note that we have omitted the `exe` argument as it is expected that `paraview.exe` is available at the command line or otherwise defined in the config file. Because `paraview` is a method of `fdata`, it has access to all the zone, permeability, etc. information defined in that object. However, it has no knowledge of simulation output that is

encapsulated in the `fcontour` object `c` defined in the previous command. Therefore, this object has to be explicitly passed to `paraview` via `contour=c`.

Running the script tutorial1.py, you should now see an instance of Paraview opened automatically (once the simulation has concluded). There is a short period of activity after Paraview has been opened that it is busy loading files, creating layers etc. - after this has finished you should see something like the figure below.



6.5.2 Model information

In the centre panel, a visualisation of the model is displayed, in this case contouring `x` permeability. A low permeability zone in the centre of the model, defined on the line

```
dat.zone[2].permeability=kmid
```

is evident, as well as the regular grid spacing, defined on the lines

```
x=np.linspace(0,10,11)
```

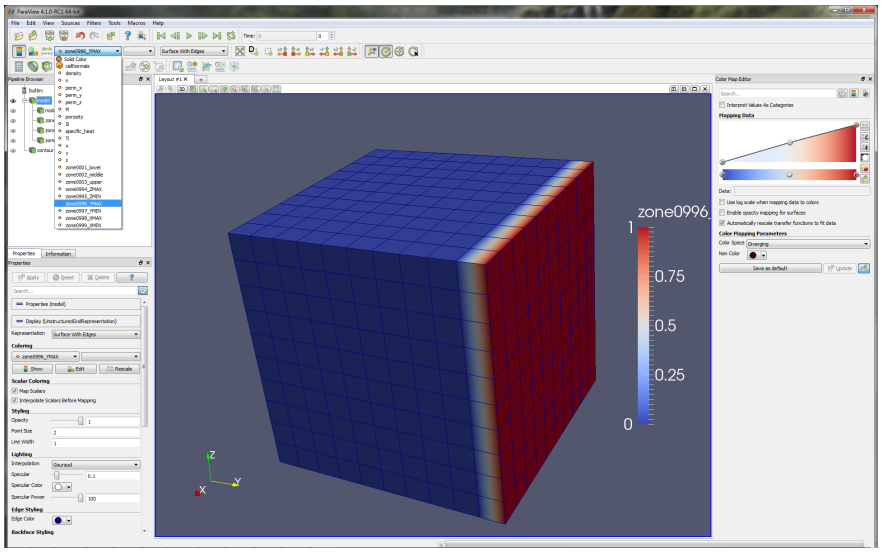
```
dat.grid.make(root+'_GRID.inp',x=x,y=x,z=x)
```

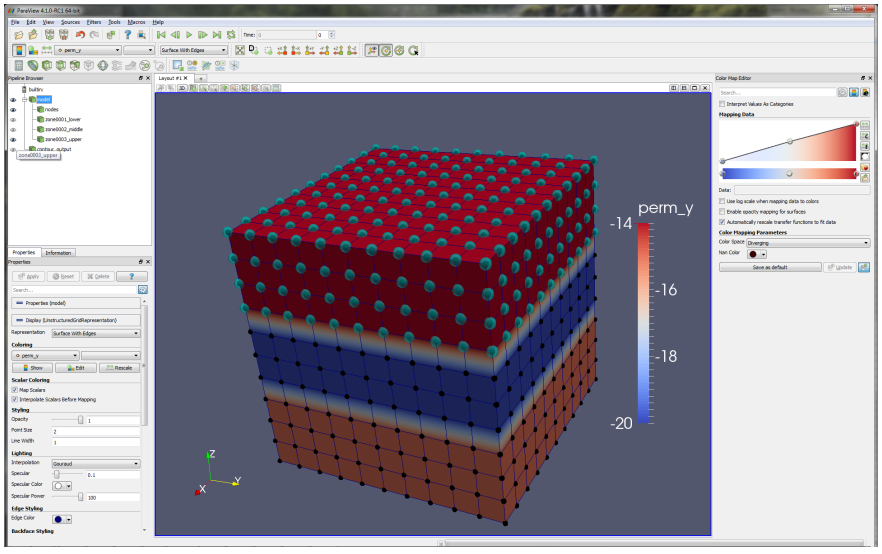
The top left panel contains a branching structure with members `model` and `contour_output`; these allow us to switch in and out various model information. For instance, with `model` selected use the drop-down menu to select `zone_996_YMAX` and note how the colouration changes to highlight the location of this zone.

This type of visualisation can be a quick means of checking that zones a user has defined in a PyFEHM script (and are consequently represented in FEHM) actually correspond to the region of the model they have in mind. The `paraview` method does not require a simulation to have actually been executed before the model can be visualised; the method can be called on any 'half-finished' `fdata` object.

Within the `model` branch, you can choose to switch on node visualisation (by toggling the 'eye' symbol next to `nodes`) or just certain zones. By default, only user-defined zones are included here; however, visualisations for the boundary zones can be obtained by passing `zones='all'` when calling the `paraview` method.

Below we show the `upper` zone and nodes superimposed on a distribution of `y` permeability. In this way, it is evident that permeability is in some way linked to the definition of the `upper` zone (well obviously!) - in more complex simulations, this can be a good way to keep track of what material properties are being assigned where.

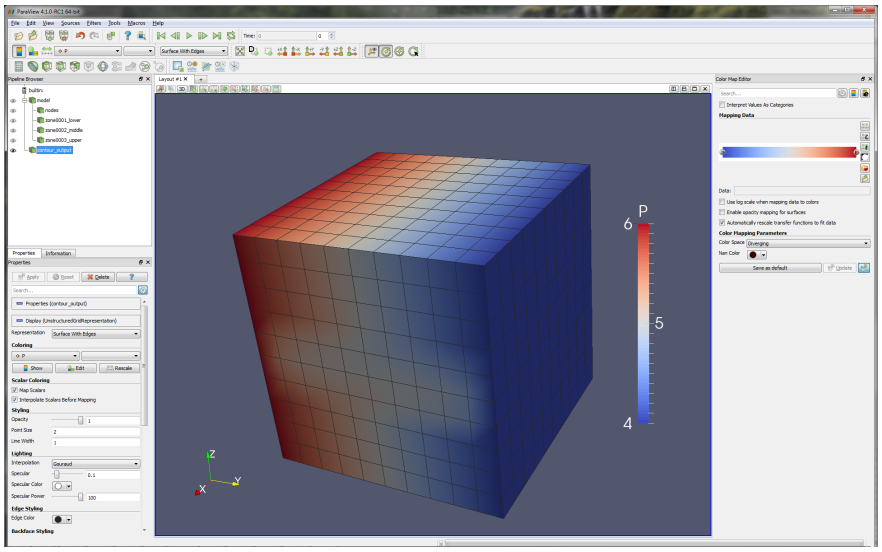

zone0996
1
0.75
0.5
0.25
0


perm_y
-14
-16
-18
-20

6.5.3 Contour output

The branch `contour_output` contains information from the contour object that we passed to `paraview`. This includes differenced or time derivative variables that may have been created by passing the relevant arguments to `paraview`. The `model` and `contour_output` branches are not mutually exclusive. For example, one could superimpose a zone or node visualisation from `model` on, say, temperature distributions in `contour_output` to get a sense of any correspondence between these items.

Turning off the `model` branch, and turning on `contour_output` (again using the 'eye' toggles), we are presented with a view of model pressures at the end of the simulation.



In this view, it is clear that the horizontal pressure gradient has been perturbed by the presence of the low permeability zone in the centre of the model.

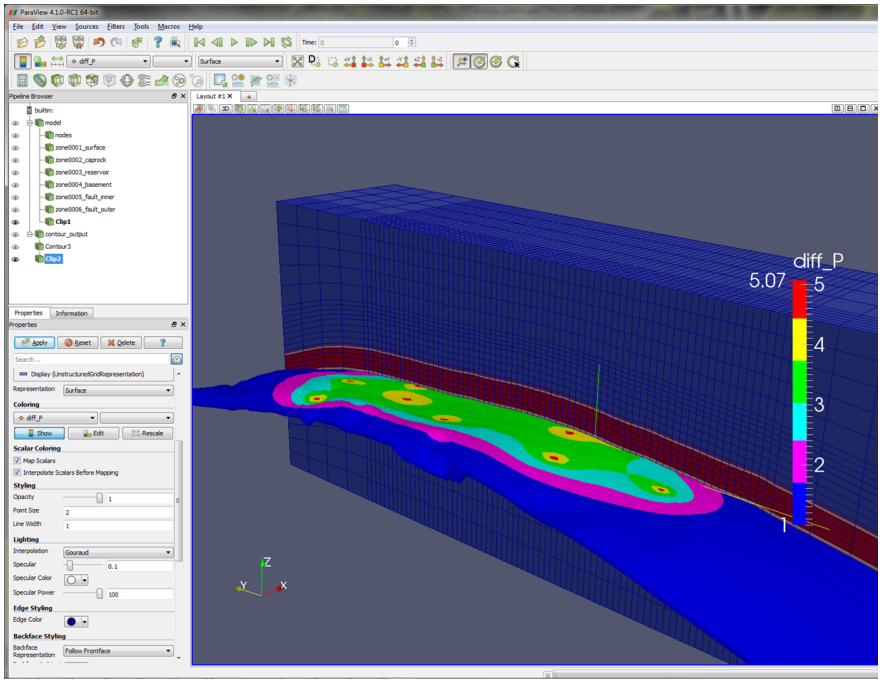
We can look at other variables by selecting the `model` branch and using the same drop-down menu we earlier used to inspect material properties. Note that, in this menu, are the additional differenced and time derivative variables we requested to be defined.

6.5.4 Some additional Paraview tools

Here we briefly list several of Paraview's useful visualisation features.

1. Slices can be taken through a model to view material properties or output variables in the interior. Access this feature through `Filters > Common > Clip` or `Slice`.
2. Isosurfaces can be defined that follow a specific zone: `Filters > Common > Contour`
3. If multiple contour output files have been imported (corresponding to different output times), an animation can be played using the green animation buttons at the top of the screen.
4. Save a visualisation via `File > Save Screenshot`.

For example, the figure below shows a vertical slice that clips away half the model. Grid lines are superimposed upon the solid section that remains, with the colouration corresponding to permeability. An isosurface has been constructed for the base of the reservoir zone and contours of pressure have been plotted on this isosurface.



6.5.5 Paraview script

Installation of PyFEHM provides access to the python script 'fehm_paraview.py'. This script operates as an executable that accepts command line arguments and opens a Paraview visualisation of a particular FEHM simulation.

fehm_paraview.py resides in the /Scripts folder of your Python installation. If /Scripts is included in the PATH environment variable, and the default program for *.py files is Python (on Windows), then this script can be called as an executable.

In its simplest use, fehm_paraview.py can be called without arguments in any directory containing an 'fehmn.files' FEHM control file. For example, in running the script 'tutorial1.py', we created the subdirectory /tut1, which contains various simulation files, including 'fehmn.files' for that simulation. Navigate to this directory and at the command line execute

```
..\\pyfehm\\tutorials\\tut1>fehm_paraview.py
```

this will cause an instance of Paraview to be opened and initialised with all model, grid and contour output data. For more options on command line use of fehm_paraview.py, call this script with the `--help` argument

```
fehm_paraview.py --help
```

6.6 Diagnostic window

FEHM supplies a stream of information to the terminal when a simulation is running. For instance, time step size, number, elapsed time, mass and energy balances, residuals and some nodal information can all be reported. Often it may be useful to track and plot these information in time, particularly for long simulations with many time steps.

PyFEHM provides a diagnostic tool that both plots these data in real time and dumps the output information in column format to several auxiliary *.dgs files. Additional information is provided about 'Largest N-R' corrections that occur before time step cutbacks and provide and indication of which regions of the model FEHM is struggling to achieve convergence.

As for the previous section, the PyFEHM diagnostic tool will be demonstrated by modifying and running tutorial1.py. In this script, uncomment two lines before the `run` method

```
dat.tf=100.
```

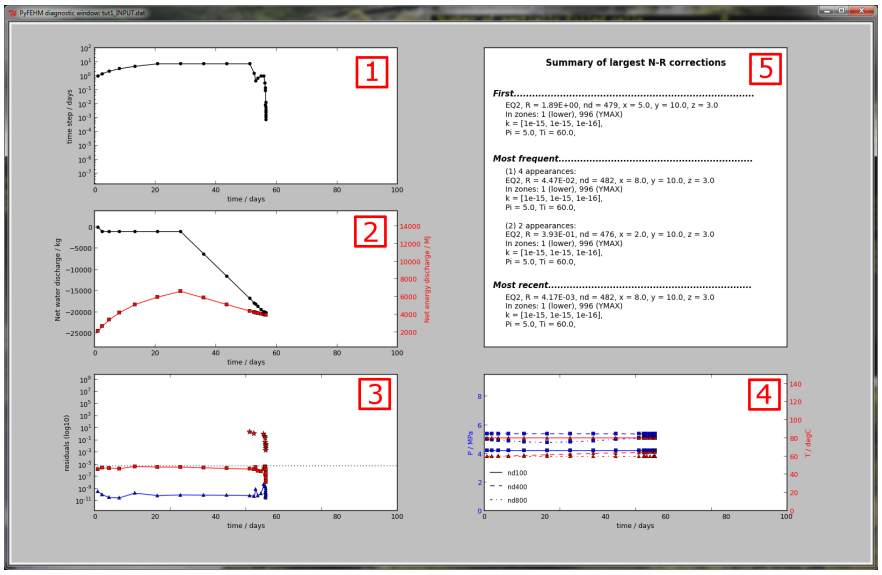
```
dat.iter['machine_tolerance_TMCH']=-0.5e-5
```

The increased run time provides a better demo and the lower tolerance allows us to demonstrate time step cutbacks.

As in tutorial 1, the simulation is performed by calling the `run` method, but this time modified to pass `diagnostic=True`

```
dat.run(root+'_INPUT.dat',diagnostic=True)
```

As the simulation executes, a dialogue window will open with several empty ploting axes. These are populated as the simulation proceeds with information taken from the output stream.



In the figure above, there are five boxes, numbered 1 through 5, containing different information. Each of these is explained below:

1. Time step size on a log-plot. Both the x and y axes will readjust dynamically depending on the range of data. If this curve starts trending down, it is an indication of time step cutbacks and convergence issues.
2. Net water (black) and energy (red) discharge. Appropriate y-scale is in the corresponding color.
3. Residuals on a log-scale for the each governing equation: mass (blue) and energy (red) balance, and, if running with the stress or CO2 modules, the static-force or CO2 balance (black). A dotted line shows the residual tolerance; if the residual exceeds this line, FEHM will cutback the time step to achieve better convergence.

   Where Largest N-R corrections occur and the time-step is cutback, star symbols in the appropriate color are plotted, e.g., in the figure above, cutbacks occurred when the energy balance (red) residual exceeded the tolerance, hence red stars are plotted.
4. Node properties. FEHM outputs temperature, pressure and flow information to the screen for history nodes in `dat.hist.nodelist`. Up to the first four of these are plotted using different line styles (see legend). The first two variables defined in `dat.hist.variables`are plotted on the two axes.
5. When Largest N-R corrections occur, it is useful to know what node they occurred at, what its material properties are, what zone it belongs in etc. This information is provided in Box 5, including (i) the first node to produce a Largest N-R cutback, (ii) the two most-frequent nodes that produced cutbacks, and (iii) the most-recent node to produce a cutback. The final two update as the simulation proceeds.

The diagnostic window has several interactive properties. It can be maximised/minimised, resized and moved around the screen. It can also be closed without interrupting the simulation. However, `run` method that produces the diagnostic window will not return until the window is closed; scripts will pause at the end of a simulation until the user intervenes.

6.6.1 Output files

In addition to visualising this information in the diagnostic window, PyFEHM writes the information in delimited format to three output files, named '*_convergence.dgs', '*_NR.dgs' and '*_node.dgs'. Alternatively, these output files can be produced by reading an FEHM '*.outp' file (which contains the same terminal stream) using the `process_output` function.

`fdata.`**`process_output`**(*filename*, *input=None*, *grid=None*, *hide=False*)

Runs an FEHM *.outp file through the diagnostic tool. Writes output files containing simulation balance, convergence, time stepping information.

Parameters

- **filename** (*str*) – Path to the *.outp file.
- **input** (*str*) – Path to corresponding FEHM input file .
- **grid** (*str*) – Path to corresponding FEHM grid file.
- **hide** (*bool*) – If True, suppresses the diagnostic window and just produces the output files.

6.6.2 Diagnostic script

Similar to Paraview visualisation, a script is supplied on installation for command line execution of the diagnostic tool. This script should precede an FEHM executable call that would otherwise execute an FEHM simulation. For example, if the user has prepared a directory containing the control file 'fehmn.files', with directions to an FEHM input and grid file, then an FEHM simulation would be started by typing

```
tutorials\tut1>fehm.exe
```

at the command line (presuming that 'fehm.exe' is in the PATH). To perform the same simulation with a diagnostic window, simply precede the executable with 'diagnose.py'

```
tutorials\tut1>diagnose.py fehm.exe
```

This will cause the simulation to be run through PyFEHM and a diagnostic window to be opened.

Additional instructions on making installed Python scripts available at the command line can be found in the previous section on Paraview visualisation.

INDEX

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z