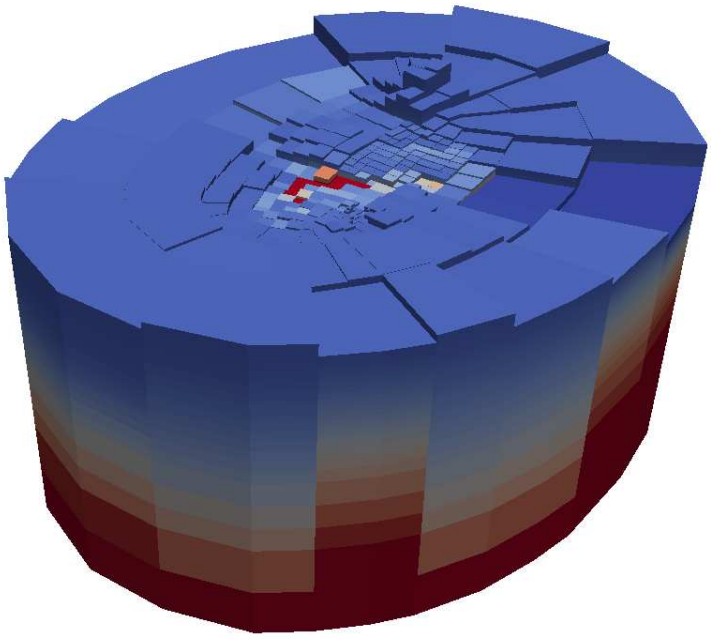
PyTOUGH user's guide



Dr. Adrian Croucher
Department of Engineering Science
University of Auckland
Auckland, New Zealand

Version 1.3.6
February 2013

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

Contents

List of Tables

Chapter 1

Introduction

1.1 What is PyTOUGH?

PyTOUGH (**Py**thon **TOUGH**) is a set of Python software routines for making it easier to use the TOUGH2 geothermal reservoir simulator. Using PyTOUGH, it is possible to automate the creation and editing of TOUGH2 model grids and data files, and the analysis and display of model simulation results.

1.2 What are TOUGH2 and AUTOUGH2?

TOUGH2 (Pruess et al., 1999) is a general-purpose simulator for modelling subsurface fluid and heat flow, often used for simulating geothermal reservoirs.

AUTOUGH2 is the University of Auckland version of TOUGH2. The main differences between AUTOUGH2 and TOUGH2 are:

- **EOS handling**: AUTOUGH2 includes all different equations of state (EOSes) in a single executable program, whereas TOUGH2 uses different executables for each EOS. As a result, the main input data file for an AUTOUGH2 simulation also includes extra data blocks to specify which EOS is to be used.
- **Generator types**: AUTOUGH2 includes a variety of extra generator types developed for geothermal reservoir simulation (e.g. makeup and reinjection wells).

TOUGH2_MP (Zhang et al., 2008) is a multi-processor version of TOUGH2. TOUGH+ is a redeveloped version of TOUGH2, with a more modular code structure implemented in Fortran-95.

1.2.1 TOUGH2 data files

TOUGH2 takes its main input from a **data file**, which contains information about the model grid, simulation parameters, time stepping, sources of heat and mass etc. The data file formats for TOUGH2 and AUTOUGH2 are almost identical, with minor differences. TOUGH2_MP can read TOUGH2 data files, but also supports some extensions (e.g. for 8-character instead of 5-character block names) to this format. PyTOUGH does not currently support the TOUGH2_MP extensions. TOUGH+ data files can also have some extensions, which PyTOUGH does not support as yet.

Because TOUGH2 uses a finite volume formulation, the only model grid data it needs are the volumes of the grid blocks and the distances and areas associated with the connections between blocks. Hence, the TOUGH2 data file need not contain any information about the specific locations of the blocks in space, and it contains no information about the locations of the vertices or edges of the blocks. This makes it easy to use TOUGH2 to simulate one-, two- or three-dimensional models, all with the same format of data file. However, this lack of reference to any coordinate system also makes it more difficult to generate model grids, and to visualise simulation results in space.

1.2.2 MULgraph geometry files

For this reason, a separate **geometry file** can be used to create grids for TOUGH2 simulations and visualise simulation results. The geometry file contains information about the locations of the grid block vertices. The geometry file can be used to visualise results using the **MULgraph** graphical post-processor for TOUGH2 and AUTOUGH2 (O'Sullivan and Bullivant, 1995), developed at the University of Auckland in the 1990s.

The MULgraph geometry file assumes the grid has a layered structure, with blocks arranged in layers and columns, and the same arrangement of columns on each layer. (At the top of the model grid, blocks in some columns may be missing, to allow the grid to follow the surface topography.)

1.2.3 TOUGH2 listing files

The output from TOUGH2 is written to a **listing file**, which is a text file containing tables of results for each time step (or only selected time steps, if preferred). At each time step there is an 'element table', containing results for block properties (e.g. pressure, temperature etc.). There may also be a 'connection table', containing results for flows between blocks, and a 'generation table', containing results (e.g. flow rates) at the generators in the model (e.g. wells).

The formats of the listing files produced by TOUGH2, AUTOUGH2, TOUGH2_MP and TOUGH+ are all slightly different, and also vary depending on the EOS used. However, PyTOUGH attempts to detect and read all of these formats.

1.3 What is Python?

Python is a general-purpose programming language. It is free and open-source, and runs on many different computer operating systems (Linux, Windows, Mac OS X and others). Python can be downloaded from the Python website (`http://www.python.org`), which also contains detailed reference material about the Python language. If you are using Linux you probably already have Python, as it is included in most Linux distributions.

PyTOUGH should run on Python version 2.4 or newer (though version 2.5 or newer is recommended).

If you are unfamiliar with Python (even if you have used another programming language before), it is highly recommended that you do one of the many Python tutorials available online, e.g.

- `http://docs.python.org/tutorial/`
- `http://wiki.python.org/moin/BeginnersGuide`

1.3.1 Python basics

Objects

Python is what is known as an **object-oriented** language, which means that it is possible to create special customised data types, or 'classes', to encapsulate all the properties and behaviour of the things (objects) we are dealing with in a program. This is a very useful way of simplifying complex programs. (In fact, in Python, everything is treated as an object, even simple things like integers and strings.)

For example, in a TOUGH2 model grid we have collections of grid blocks, and we need to store the names of these blocks and their volumes and rock types. In a non-object-oriented language, these could be stored in three separate arrays: a string array for the names, a real (or 'float') array for the volumes and another string array for the rock types. In an object-oriented language like Python, we can define a new data type (or 'class') for blocks, which holds the name, volume and rock type of the block. If we declare an object called `blk` of this block class, we can access or edit its volume by referring to `blk.volume`. In this way, we can store our blocks in one single array of block objects. When we add or delete blocks from our grid, we can just add or delete block objects from the array, rather than having to keep track of three separate arrays.

In general, an object not only has **properties** (like `blk.volume`) but also **methods**, which are functions the object can carry out. For example, if we wanted to rotate a MULgraph geometry file by 30°, we could do this in PyTOUGH by declaring a MULgraph geometry file object called `geo`, and calling its `rotate` method: `geo.rotate(30)`. The methods of an object are accessed in the same way that its properties are accessed: by adding a dot (.) after the object's name and then adding the name of the property or method. Any arguments of the method (e.g. the angle in the `rotate` function above) are added in parentheses afterwards.

Lists, dictionaries, tuples and sets

Most programming languages have simple data types built in, e.g. float, double precision or integer numbers, strings, and arrays of these. Python has some other data types which are very useful and are used a lot.

The first of these is the **list**. A list can contain any ordered collection of objects, of any type, or even of different types, and is delimited by square brackets. So for example we can declare a list `things = [1,'two',3.0]` containing an integer, a string and a float. We can access the list's elements in much the same way as we access the elements of an array, for example `things[1]` would return the value `'two'` (note that in Python, as in most other languages besides Fortran, the indices of arrays and lists start at 0, not 1). Additional elements can be added to a list at any time, without having to re-declare the size of the list: for example, `things.append('IV')` would add an extra element to the end of the list, giving it the value `[1,'two',3.0,'IV']`. It is also possible to remove elements from a list, e.g. `things.remove(3.0)`, which would give our list the value `[1,'two','IV']`.

Another useful Python data type is the **dictionary**. Dictionaries are mainly used to store collections of objects (again, of any type or of different types) that are referenced by name rather than by index (as in an array or list). A dictionary is delimited by curly brackets. So for example we can declare a dictionary `phone={'Eric':8155,'Fred':2350,'Wilma':4667}` and then find Fred's phone number from `phone['Fred']`, which would return `2350`. For TOUGH2 models, blocks, generators, rock types and other objects are often referred to by name rather than index, so dictionaries are an appropriate way to store them.

A third Python data type, similar to a list, is the **tuple**. A tuple is essentially a list that cannot be changed, and is often used just for grouping objects together. A tuple is delimited by parentheses. For example, `things=(1,'two',3.0)` declares a tuple with three elements. We can still refer to the elements of a tuple using e.g. `a[1]`, but we cannot assign new values to these elements or add or remove elements from the tuple once it has been declared.

Python also has a **set** data type, which represents a mathematical set- an unordered collection of objects. One of the useful aspects of sets is that they cannot contain duplicate items. As a result, for example, duplicate items can be removed from a list `x` simply by converting it to a set, and then back to a list: `x=list(set(x))`.

1.3.2 How to run Python

Python can be run either interactively or via scripts.

Running Python interactively

The simplest way to run Python interactively is just by typing `python` at the command line. (On Windows the directory that Python was installed into may have to be added to your `PATH` environment variable first.) The command line then becomes an interactive Python environment in which you can type Python commands at the Python command prompt `>>>`, e.g. in Windows:

```
C:\>python
Python 2.6.4 (r264:75708, Oct 26 2009, 08:23:19) [MSC v.1500 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> things = [1,'two',3.0]
>>> print things[1]
two
>>> exit()

C:\>
```

In the interactive Python environment, you can view help on the properties and methods of any Python object by typing `help(`*`objectname`*`)`, where *`objectname`* is the name of an object that has been declared. This will list the properties and methods of the object and a description of each one.

You can exit the interactive Python environment by typing `exit()` or `Ctrl-Z` on Windows, or `Ctrl-D` on Linux.

Python scripts

The real power of Python, however, lies in using it to write **scripts** to automate repetitive or complex tasks. You can just type Python commands into a text file, save it with the file extension `.py`, and execute it by typing `python` *`filename.py`*, where *`filename.py`* is the name of the file. (Once again, on Windows the directory that Python was installed into may have to be added to your `PATH` environment variable first.)

You can also debug a Python script using the 'pdb' command-line debugger. Typing `python -m pdb` *`filename.py`* will start debugging the script *filename.py*.

It is also possible to run a Python script from within the interactive Python environment. From the Python environment command line, typing `execfile('filename.py')` will execute the script *filename.py*.

1.3.3 Python libraries

Python comes with a large number of features already built in, but for specialised tasks, additional **libraries** of Python software can be imported into Python as you need them. PyTOUGH itself is a set of such libraries, and it in turn makes use of some other Python libraries.

Numerical Python

The most important of these is Numerical Python ('numpy'), which you will need to have installed on your computer before you can use PyTOUGH at all [1]. Numerical Python adds a special `numpy.array` class for fast multi-dimensional arrays, which PyTOUGH makes heavy use of, and a whole range of other features, e.g. linear algebra routines, Fourier transforms and statistics.

Numerical Python can be freely downloaded from `http://numpy.scipy.org/` (or simply installed via your package manager on Linux). The latest versions may not yet be available for 64-bit Windows from this site. However, you can find unofficial builds at `http://www.lfd.uci.edu/~gohlke/pythonlibs/`.

Other libraries

Some parts of PyTOUGH use other Python libraries. You do not need these libraries unless you are using the parts of PyTOUGH that depend on them.

- **matplotlib** (`http://matplotlib.sourceforge.net/`), a library of graphical plotting routines
- **Scientific Python** (`http://www.scipy.org/`), a library of advanced mathematical functions (e.g. interpolation, calculus, optimisation)
- **VTK** (`http://www.vtk.org/`), the Visualization Tool Kit (a library for 3D visualisation of data via VTK itself, or software such as ParaView, Mayavi etc.) and **VTK Python**, the Python interface for VTK.

On Linux systems you can download and install these libraries via your package manager (e.g. on Debian or Debian-based distributions like Ubuntu, `aptitude install python-matplotlib python-scipy libvtk5.4 python-vtk`).

On Windows you will need to download and install them yourself from the internet (see links above) or elsewhere. At the time of writing there is not yet an official 64-bit build of Scientific Python for Windows. However an unofficial build can be found at `http://www.lfd.uci.edu/~gohlke/pythonlibs/`. Use of this version of Scientific Python requires that their version of Numerical Python is also used.

[1] PyTOUGH will run using Numeric, a now-obsolete predecessor of Numerical Python, though the PyTOUGH plotting functions will not work. In general it is recommended to use Numerical Python if possible.

Installing VTK-Python on Windows

There does not appear to be an official Windows build of VTK Python, but you can instead download an unofficial Python-enabled Windows version of VTK (i.e. VTK with VTK Python included) from `http://www.lfd.uci.edu/~gohlke/pythonlibs/`.

Make sure you download and install the right version, corresponding to the version of Python you are using. You will need to add a few things to the environment variables on your system to get it to work. (If you aren't sure how to edit your environment variables, see section 1.4 below.)

In the directory where VTK is installed to, there should be sub-directories called something like `VTK\bin` and `VTK\lib\site-packages`. You need to add the full path to the `bin` directory to the PATH environment variable on your system, and the paths to both the `bin` and `lib\site-packages` directories to your PYTHONPATH environment variable.

Alternatively, there are third-party repackaged distributions of Python which come bundled with some or all of these additional libraries (see `http://www.python.org/download/`).

Importing libraries

To use any Python library, you just need to **import** it first. For example, once you have installed Numerical Python, you can make it available (in the interactive Python environment or in a Python script) by typing the command `import numpy`, or alternatively `from numpy import *`. This imports all classes and commands from Numerical Python and makes them available for use. (You can also import only parts of a library rather than the whole thing, e.g. `from numpy import linalg` just imports the linear algebra routines from Numerical Python.)

When you import a library, you can also change its name. For example, PyTOUGH imports Numerical Python using the command `import numpy as np`, which renames `numpy` as the abbreviated `np`. This means it can, for example, access the Numerical Python `numpy.array` data type as `np.array`. It also means you have access to Numerical Python as `np` in your own scripts and in the interactive Python environment, without having to import it yourself.

1.4 Installing and accessing PyTOUGH

First, make sure you have Python, and the Numerical Python library (see section 1.3.3) installed. On Windows, you may have to add the directory where Python has installed (e.g. `C:\Python26` or similar, depending on which version you have) to your PATH environment variable, before you can access Python from the command line.

On the PyTOUGH website, click the 'Download ZIP' button at the upper left:

```
https://github.com/acroucher/PyTOUGH/archive/master.zip
```

to download PyTOUGH as a .zip file. Unzip this to any directory on your computer. This will create a directory containing a file called `setup.py`. At the command line type:

```
python setup.py install
```

You should now be able to import the PyTOUGH libraries into the Python interactive environment or your Python scripts, from any directory on your computer. For example, you can import the MULgraph geometry library using `from mulgrids import *` (see chapter 2).

1.5 Licensing

PyTOUGH is free software, distributed under the GNU Lesser General Public License (LGPL). For more information, see `http://www.gnu.org/licenses/`.

Chapter 2

MULgraph geometry files

2.1 Introduction

The `mulgrids` library in PyTOUGH contains classes and routines for creating, editing and saving MULgraph geometry files. It can be imported using the command:

```
from mulgrids import *
```

2.2 `mulgrid` objects

The `mulgrids` library defines a `mulgrid` class, used for representing MULgraph geometry files.

Example:

```
geo=mulgrid()
```

creates an empty `mulgrid` object called `geo`.

```
geo=mulgrid('geom.dat')
```

creates a `mulgrid` object called `geo` and reads its contents from a file named `'geom.dat'`.

Printing a `mulgrid` object (e.g. `print geo`) displays a summary of information about the grid: how many nodes, columns, layers, blocks and wells it contains, as well as its naming convention and atmosphere type.

2.2.1 Properties

The main properties of a `mulgrid` object are listed in Table 2.3. Some of these properties are 'header' information, corresponding to the data at the start of a MULgraph geometry file (`type`, `convention`, `atmosphere_type`, `atmosphere_volume`, `atmosphere_connection` and `unit_type`).

The most important properties of a `mulgrid` object are `node`, `column`, `connection`, `layer` and `well`, which are dictionaries of the grid nodes, columns, connections, layers and wells, accessed by name. For example, grid layer 'AA' of a `mulgrid` object `geo` can be accessed by `geo.layer['AA']`. (The `nodelist`, `columnlist`, `connectionlist`, `layerlist`

0	3 characters for column followed by 2 digits for layer
1	3 characters for layer followed by 2 digits for column
2	2 characters for layer followed by 3 digits for column

Table 2.1: Naming conventions of a `mulgrid` object

0	A single atmosphere block
1	One atmosphere block over each column
2	No atmosphere blocks

Table 2.2: Atmosphere types of a `mulgrid` object

and `welllist` properties offer access to the nodes, columns, connections, layers and wells by index, which is sometimes useful e.g. for looping over all columns in the grid.)

Connections are slightly different from nodes, columns etc. in that they are not named individually. However, they can be accessed by the names of the columns connected by the connection. For example, the connection between columns ' 10' and ' 11' in a `mulgrid` called `geo` is given by `geo.connection[' 10',' 11']`.

The elements of these lists and dictionaries are of type `node`, `column`, `connection`, `layer` and `well` respectively. These are additional object classes to represent nodes, columns, connections, layers and wells, defined in the `mulgrids` library (see section 2.3).

2.2.2 Naming conventions and atmosphere types

The grid block naming convention and atmosphere type used in the `mulgrid` object, specified by the `convention` and `atmosphere_type` parameters, are both integers which can be given the value 0, 1 or 2. The meanings of these values are shown in Table 2.1 and 2.2.

Grid diagnostics

A `mulgrid` object has some properties (and methods) for evaluating its integrity. The property `column_angle_ratio` returns an `np.array` of the 'angle ratio' for each column (the ratio of largest to smallest interior angles – see section 2.3.2), a measure of skewness. The `column_side_ratio` returns an `np.array` of the 'side ratio' for each column (the ratio of largest to smallest side length), a measure of elongation. These array properties can be plotted using the `layer_plot` method (see section 2.2.3) for a graphical overview of grid quality.

There is also a `connection_angle_cosine` property, which returns an `np.array` of the angle cosine for each connection (the cosine of the angle between a line joining the nodes in the connection and a line joining the centres of the blocks in the connection). In general it is desirable for these lines to be as close to perpendicular as possible, making the cosines close to zero.

The `bad_columns`, `bad_layers`, `missing_connections`, `extra_connections` and `orphans` properties return actual problems with the grid which should be fixed. A summary of all these problems is given by the `check` method (see section 2.2.3).

Blocks at the ground surface that have very small vertical thickness can sometimes cause problems. The `min_surface_block_thickness` property gives a tuple containing the

minimum surface block thickness and the name of the column in which it occurs. Thin surface blocks of this type can be eliminated using the `snap_columns_to_layers()` method.

Property	Type	Description
`area`	float	total horizontal area covered by the grid
`atmosphere_connection`	float	connection distance to atmosphere blocks
`atmosphere_type`	integer	type of atmosphere
`atmosphere_volume`	float	volume of atmosphere blocks
`bad_columns`	set	columns that do not contain their own centres
`bad_layers`	set	layers that do not contain their own centres
`block_name_index`	dictionary	indices of blocks (by name)
`block_name_list`	list	names of blocks (by index)
`boundary_columns`	set	set of columns on the outer boundary of the grid
`boundary_nodes`	list	ordered list of nodes on the outer boundary of the grid
`boundary_polygon`	list	list of points representing grid boundary (extra colinear points removed)
`bounds`	list	[bottom left, top right] horizontal bounds of grid
`centre`	`np.array`	position of horizontal centre of the grid
`columnlist`	list	columns (by index, e.g. `columnlist[23]`)
`column_angle_ratio`	`np.array`	angle ratio for each column
`column_side_ratio`	`np.array`	side ratio for each column
`column`	dictionary	columns (by name, e.g. `column['AA']`)
`connection_list`	list	connections between columns (by index)
`connection_angle_cosine`	`np.array`	angle cosines for all connections
`convention`	integer	naming convention for columns and layers
`default_surface`	Boolean	`True` if all columns have default surface elevation
`extra_connections`	set	connections defined between columns that are not against each other
`filename`	string	file name on disk
`node_kdtree`	`cKDTree`	tree structure for fast searching for nodes
`layerlist`	list	layers (by index)
`layer`	dictionary	layers (by name)
`min_surface_block_thickness`	(float, string)	thickness of thinnest surface block (and associated column name)
`missing_connections`	set	missing connections between columns
`nodelist`	list	nodes (by index)
`node`	dictionary	nodes (by name)
`num_atmosphere_blocks`	integer	number of atmosphere blocks
`num_blocks`	integer	total number of blocks in the grid
`num_columns`	integer	number of columns
`num_connections`	integer	number of connections between columns
`num_layers`	integer	number of layers
`num_nodes`	integer	number of nodes
`num_underground_blocks`	integer	number of non-atmosphere blocks
`num_wells`	integer	number of wells
`orphans`	set	orphaned nodes (nodes not belonging to any column)

permeability_angle	float	rotation angle (degrees anticlockwise) of first horizontal permeability direction
type	string	type of geometry (currently only 'GENER' supported)
unit_type	string	distance unit (blank for metres, 'FEET' for ft)
welllist	list	wells (by index)
well	dictionary	wells (by name)

Table 2.3: Properties of a mulgrid object

2.2.3 Methods

The main methods of a mulgrid object are listed in Table 2.4. Details of these methods are given below.

Method	Type	Description
add_column	–	adds a column to the grid
add_connection	–	adds a connection to the grid
add_layer	–	adds a layer to the grid
add_node	–	adds a node to the grid
add_well	–	adds a well to the grid
block_contains_point	Boolean	whether a block contains a 3D point
block_mapping	dictionary	mapping from the blocks of another mulgrid object
block_name	string	name of block at given layer and column
block_name_containing_point	string	name of block containing specified point
check	Boolean	checks grid for errors (and optionally fixes them)
column_boundary_nodes	list	nodes around the outer boundary of a group of columns
column_bounds	list	bounding rectangle around a list of columns
column_containing_point	column	column containing specified horizontal point
column_mapping	dictionary	mapping from the columns of another mulgrid object
column_name	string	column name of a block name
column_neighbour_groups	list	groups connected columns
column_quadtree	quadtree	quadtree structure for searching columns
column_surface_layer	column	surface layer for a specified column
column_values	tuple	values of a variable down a column
columns_in_polygon	list	columns inside a specified polygon (or rectangle)
connects	Boolean	whether the grid has a connection between two specified columns
copy_layers_from	–	copies layer structure from another geometry
copy_wells_from	–	copies wells from another geometry
delete_column	–	deletes a column from the grid
delete_connection	–	deletes a connection from the grid

delete_layer	–	deletes a layer from the grid
delete_node	–	deletes a node from the grid
delete_orphans	–	deletes any orphaned nodes from the grid
delete_orphan_wells	–	deletes any orphaned wells from the grid
delete_well	–	deletes a well from the grid
empty	–	empties contents of grid
export_surfer	–	exports to various files on disk for visualization in Surfer
fit_surface	–	fits column surface elevations from data
from_gmsh	mulgrid	imports a grid from a gmsh mesh
layer_containing_elevation	layer	layer containing specified vertical elevation
layer_mapping	dictionary	mapping from the layers of another mulgrid object
layer_name	string	layer name of a block name
layer_plot	–	plots a variable over a layer of the grid
line_values	tuple	values of a variable along an arbitrary line through the grid
line_plot	–	plots a variable along an arbitrary line through the grid
nodes_in_columns	list	nodes in a specified list of columns
nodes_in_polygon	list	nodes inside a specified polygon (or rectangle)
node_nearest_to	node	node nearest to a specified point
optimize	–	adjusts node positions to optimize grid quality
polyline_values	tuple	values of a variable along an arbitrary polyline through the grid
read	mulgrid	reads geometry file from disk
rectangular	mulgrid	creates rectangular grid
reduce	–	reduces a grid to contain only specified columns
refine	–	refines specified columns in the grid
refine_layers	–	refines specified layers in the grid
rename_column	Boolean	renames a column
rename_layer	Boolean	renames a layer
rotate	–	rotates a grid in the horizontal plane
rotate	–	rotates a grid in the horizontal plane
slice_plot	–	plots a variable over a vertical slice through the grid
snap_columns_to_layers	–	snaps column surfaces to layer bottoms
split_column	Boolean	splits a quadrilateral column into two triangles
translate	–	moves a grid by simple translation in 3D
well_values	tuple	values of a variable down a well
write	–	writes to geometry file on disk
write_bna	–	writes to Atlas BNA file on disk
write_vtk	–	writes to VTK file on disk

Table 2.4: Methods of a mulgrid object

`add_column(col)`

Adds a `column` object `col` to the grid.

`add_connection(con)`

Adds a `connection` object `con` to the grid.

`add_layer(lay)`

Adds a `layer` object `lay` to the grid.

`add_node(n)`

Adds a `node` object `n` to the grid.

`add_well(w)`

Adds a `well` object `w` to the grid.

`block_contains_point(blockname, pos)`

Returns `True` if the grid block with the given name contains the 3D point `pos`.

Parameters:

- **blockname**: string
  The name of the block.
- **pos**: `np.array`
  3-element array representing the 3D point.

`block_mapping(geo, column_mapping=False)`

Returns a dictionary mapping each block name in the `mulgrid` object `geo` to the name of the nearest block in the object's own geometry. Can optionally also return the associated column mapping.

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` object to create a block mapping from.
- **column_mapping**: Boolean
  If `True`, the column mapping will also be returned (i.e. the function will return a tuple containing the block mapping and the column mapping). Default value is `False`.

`block_name(layer_name, column_name)`

Gives the name of the block corresponding to the specified layer and column names, according to the naming convention of the grid.

Parameters:

- **layer_name**, **column_name**: string
  Name of layer and column (the widths of these strings are determined by the grid's naming convention).

`block_name_containing_point(`*pos*`, `*qtree*`=None)`

Gives the name of the block containing a specified 3-D position in the grid (returns `None` if the point lies outside the grid).

Parameters:

- **pos**: `np.array`
  Position of point in 3-D
- **qtree**: `quadtree`
  Quadtree object for fast searching of grid columns (can be constructed using the `column_quadtree()` method).

`check(`*fix*`=False,`*silent*`=False)`

Checks a grid for errors and optionally fixes them. Errors checked for are: missing connections, extra connections, orphaned nodes, and columns and layers that do not contain their own centres. Returns `True` if no errors were found, and `False` otherwise. If `fix` is `True`, any identified problems will be fixed. If `silent` is `True`, there is no printout (only really useful if `fix` is `True`).

Parameters:

- **fix**: Boolean
  Whether to fix any problems identified.
- **silent**: Boolean
  Whether to print out feedback or not.

`column_boundary_nodes(`*columns*`)`

Returns the nodes around the outer boundary of a list of columns. The list is ordered, in a counter-clockwise direction.

Parameters:

- **columns**: list
  The list of columns for which the boundary is required.

`column_bounds(`*columns*`)`

Returns a bounding rectangle around a list of columns.

Parameters:

- **columns**: list
  The list of columns for which the bounds are required.

`column_containing_point(`*pos*`, `*columns*`=None, `*guess*`=None, `*bounds*`=None, `*qtree*`=None)`

Returns the grid column containing the specified horizontal point. If `columns` is specified, only columns in the given list will be searched. An initial `guess` column can optionally be specified. If `bounds` is specified, points outside the given polygon will always return `None`. A quadtree structure can also be specified to speed up searching.

Parameters:

- **pos**: `np.array`
  Horizontal position (x, y)

- **columns**: list of `column` (or `None`)
  List of columns to search. If `None`, the entire grid will be searched.

- **guess**: `column` (or `None`)
  Guess of required column. If specified, this column will be tested first, followed (if necessary) by its neighbours; only if none of these contain the point will the remaining columns be searched. This can speed up the process if data follow a sequential pattern in space, e.g. a grid or lines.

- **bounds**: list of `np.array` (or `None`)
  Polygon or rectangle representing e.g. the boundary of the grid: points outside this polygon will always return `None`. If the polygon has only two points, it will be interpreted as a rectangle [bottom left, top right].

- **qtree**: `quadtree`
  A quadtree object for searching the columns of the grid. If many points are to be located, this option can speed up the search. The quadtree can be constructed before searching using e.g. `qtree = geo.column_quadtree(columns)`, where `columns` is the list of `column` objects to be searched.

`column_mapping(`*geo*`)`

Returns a dictionary mapping each column name in the `mulgrid` object `geo` to the name of the nearest column in the object's own geometry. If the SciPy library is available, a KDTree structure is used to speed searching.

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` object to create a column mapping from.

`column_name(`*block_name*`)`

Gives the name of the column corresponding to the specified block name, according to the naming convention of the grid.

Parameters:

- **block_name**: string
  Block name.

`column_neighbour_groups(`*columns*`)`

From the given list or set of columns, finds sets of columns that are connected together, and returns a list of them.

Parameters:

- **columns**: list or set
  List or set of columns to group.

`column_quadtree(`*`columns`*`=None)`

Returns a quadtree structure for fast searching of grid columns, to find which column a given point lies in. This can then be passed into various other `mulgrid` methods that do such searching, e.g. `block_name_containing_point()` or `well_values`, to speed them up (useful for large grids).

Parameters:

- **columns**: list (or `None`)
  A list of columns in the grid, specifying the search area. This parameter can be used to further speed searching if it is only necessary to search columns in a defined area. If `None`, the search area is the whole grid (all columns).

`column_surface_layer(`*`col`*`)`

Returns the layer containing the surface elevation of a specified column.

Parameters:

- **col**: `column`
  The column for which the surface layer is to be found.

`column_values(`*`col, variable, depth`* `= False)`

Returns values of a specified variable down a specified column. The variable can be a list or `np.array` containing a value for every block in the grid.

The routine returns a tuple of two arrays (`d`,`v`), the first (`d`) containing the elevation (or depth from surface if the `depth` parameter is set to `True`), and the second (`v`) containing the value of the variable at each block in the column.

Parameters:

- **col**: `column` or string
  The column for which values are to be found.

- **variable**: list (or `np.array`)
  Values of variable, of length equal to the number of blocks in the grid.

- **depth**: Boolean
  Set to `True` to give depths from surface, instead of elevations, as the first returned array.

`columns_in_polygon(`*`polygon`*`)`

Returns a list of all columns with centres inside the specified polygon or rectangle.

Parameters:

- **polygon**: list (of `np.array`)
  List of points defining the polygon (each point is a two-element `np.array`). If the list has only two points, it will be interpreted as a rectangle [bottom left, top right].

`connects(column1, column2)`

Returns `True` if the geometry contains a connection connecting the two specified columns.

Parameters:

- **column1, column2**: `column`
  Two columns in the geometry.

`copy_layers_from(geo)`

Copies the layer structure from the geometry `geo` (deleting any existing layers first).

Parameters:

- **geo**: `mulgrid`
  The geometry to copy layers from.

`copy_wells_from(geo)`

Copies the wells from the geometry `geo` (deleting any existing wells first).

Parameters:

- **geo**: `mulgrid`
  The geometry to copy wells from.

`delete_column(colname)`

Deletes the column with the specified name from the grid.

Parameters:

- **colname**: string
  Name of the column to be deleted.

`delete_connection(colnames)`

Deletes the connection between the specified columns from the grid.

Parameters:

- **colnames**: tuple of string
  Tuple of two column names.

`delete_layer(layername)`

Deletes the layer with the specified name from the grid.

Parameters:

- **layername**: string
  Name of the layer to be deleted.

`delete_orphans()`

Deletes any orphaned nodes (those not belonging to any column) from the grid.

`delete_orphan_wells()`

Deletes any orphaned wells (those with wellheads outside the grid).

`delete_well(wellname)`

Deletes the well with the specified name from the grid.

Parameters:

- **layername**: string
  Name of the layer to be deleted.

`empty()`

Empties the grid of all its nodes, columns, layers, wells and connections. Other properties are unaffected.

`export_surfer(filename='', aspect=8.0, left=0.0)`

Exports the grid to files on disk useful for visualization in Surfer. Six files are written out:

- an Atlas BNA file (`filename.bna`) representing the grid columns
- a CSV file (`filename_column_names.csv`) containing the column names
- a Golden Software blanking file (`filename_layers.bln`) file representing the grid layers
- a CSV file (`filename_layer_bottom_elevations.csv`) containing the bottom elevations of the layers
- a CSV file (`filename_layer_centres.csv`) containing the elevations of the centres of the layers
- a CSV file (`filename_layer_names.csv`) containing the names of the layers

Parameters:

- **filename**: string
  Base name for the exported files. If it is not specified, the `filename` property of the `mulgrid` object itself is used (unless this is also blank, in which case a default name is used), with its extension removed.
- **aspect**: float
  Aspect ratio for the layer plot, so that the width is the total height of the grid divided by `aspect` (default 8.0).
- **left**: float
  Coordinate value of the left hand side of the layer plot (default zero).

`fit_surface(data, alpha=0.0, beta=0.0, columns=[], min_columns=[], grid_boundary=False, layer_snap=0.0)`

Fits column surface elevations from data, using bilinear least-squares finite element fitting with Sobolev smoothing. Smoothing is useful when data density is low in some areas of the grid, in which case least-squares fitting without smoothing can fail (e.g. if there are any columns which do not contain any data points).

Parameters:

- **data**: `np.array`
  Two-dimensional array of data to fit. Each row of the array should contain the x,y,z values for each data point. Such an array can be conveniently read from a text file using the `np.loadtxt()` method.

- **alpha**: float
  Smoothing parameter for first derivatives- increasing its value results in solutions with lower gradients (but may result in extrema being smoothed out).

- **beta**: float
  Smoothing parameter for second derivatives- increasing its value results in solutions with lower curvature.

- **columns**: list of string or `column`
  Columns, or names of columns to be fitted. If empty (the default), then all columns will be fitted.

- **min_columns**: list of string or `column`
  Columns, or names of columns for which elevations will be determined from the minimum of the fitted nodal elevations (elevations at all other columns are determined from the average of the fitted nodal elevations).

- **grid_boundary**: Boolean
  If `True`, test each data point first to see if it lies inside the boundary polygon of the grid. This can speed up the fitting process if there are many data points outside the grid, and the grid has a simple boundary (e.g. a rectangle). In general if there are many data points outside the grid, it is best to clip the data set before fitting, particularly if it is to be used more than once.

- **layer_snap**: float
  Smallest desired surface block thickness. Set to a positive value to prevent columns being assigned surface elevations that are very close to the bottom of a layer (resulting in very thin surface blocks). Default value is zero (i.e. no layer snapping).

`from_gmsh(`*`filename`*`, `*`layers`*`, `*`convention`*`=0, `*`atmosphere_type`*`=2, `*`top_elevation`*`=0)`

Imports a 2-D `gmsh` mesh into a geometry object. `gmsh` is grid generation program (see `http://geuz.org/gmsh/`). The horizontal structure of the geometry object is created from the `gmsh` mesh, while the layer structure is specified via the `layers` parameter, a list of layer thicknesses. The elevation of the top surface can also be specified, as well as the naming convention and atmosphere type.

Parameters:

- **filename**: string
  Name of the `gmsh` mesh file.

- **layers**: list
  List of floats containing the desired layer thicknesses.

- **convention**: integer
  Naming convention for grid columns and layers.

- **atmosphere_type**: integer
  Type of atmosphere.

- **top_elevation**: float
  Elevation of the top surface of the model (default is zero).

`layer_containing_elevation(`*`elevation`*`)`

Returns the grid layer containing the specified vertical elevation.

Parameters:

- **elevation**: float
  Vertical elevation

`layer_mapping(`*`geo`*`)`

Returns a dictionary mapping each layer name in the `mulgrid` object `geo` to the name of the nearest layer in the object's own geometry. (Note: this mapping takes no account of the grid surface, which may alter which layer is nearest in a given column.)

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` object to create a layer mapping from.

`layer_name(`*`block_name`*`)`

Gives the name of the layer corresponding to the specified block name, according to the naming convention of the grid.

Parameters:

- **block_name**: string
  Block name.

`layer_plot(`*`layer`*`, `*`variable`*`=None, `*`variable_name`*`=None, `*`unit`*`=None, `*`column_names`*`=None, `*`node_names`*`=None, `*`column_centres`*`=None, `*`nodes`*`=None, `*`colourmap`*`=None, `*`linewidth`*`=0.2, `*`linecolour`*`='black', `*`aspect`*`='equal', `*`plt`*`=None, `*`subplot`*`=111, `*`title`*`=None, `*`xlabel`*`='x (m)', `*`ylabel`*`='y (m)', `*`contours`*`=False, `*`contour_label_format`*`='%3.0f', `*`contour_grid_divisions`*`=(100,100), `*`connections`*`=None, `*`colourbar_limits`*`=None, `*`plot_limits`*`=None)`

Plots a variable over a layer of the grid, using the `matplotlib` plotting library. The required layer can be specified by name or as an elevation (in which case the routine will find the corresponding layer). Specifying the layer as `None` gives a plot over the ground surface of the geometry (i.e. the surface layer for each column). The variable can be a list or `np.array` containing a value for every block (or column) in the grid. If no variable is specified, only the grid in the layer is plotted, without shading. If the variable contains a value for each column in the grid, these values are extended down each column to fill the entire grid.

The name and units of the variable can optionally be specified, and the names of the columns and nodes can also optionally be displayed on the plot, as well as the column centres (represented by crosses). The colour map and limits of the variable shading, the line width of the grid columns and the aspect ratio of the plot can also be set, as can the title and x- and y-axis labels, and the plot limits.

When a variable is plotted over the grid, contours at specified levels can also be drawn, and optionally labelled with their values.

Parameters:

- **layer**: string, integer, float or `None`
  Name (string) of layer to plot, or elevation (float or integer). Specifying `None` gives a surface plot.
- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks or columns in the grid (or `None` just to plot the grid).
- **variable_name**: string
  Name of the variable (as it will appear on the scale of the plot).
- **unit**: string
  Units of the variable (as it will appear on the scale of the plot).
- **column_names**: Boolean or list
  Set to `True` if column names are to be indicated on the plot, or to a list of names of columns to be named.
- **node_names**: Boolean or list
  Set to `True` if node names are to be indicated on the plot, or to a list of names of nodes to be named.
- **column_centres**: Boolean or list
  Set to `True` if column centres are to be indicated on the plot (as crosses), or to a list of names of columns to be indicated.
- **nodes**: Boolean or list
  Set to `True` if nodes are to be indicated on the plot (as crosses), or to a list of names of nodes to be indicated.
- **colourmap**: string
  Name of `matplotlib` colour map to use for shading the variable.
- **linewidth**: float
  Line width to use for drawing the grid.
- **linecolour**: string
  Line colour to use for drawing the grid.
- **aspect**: string
  Aspect ratio to use for drawing the grid (default is 'equal' (i.e. 1:1).
- **plt**: `matplotlib.pyplot` instance
  An instance of the `matplotlib.pyplot` library, imported in the calling script using e.g. `import matplotlib.pyplot as plt`.
- **subplot**: integer
  Subplot number for multi-plots, e.g. set 223 to draw the third plot in a 2-by-2 multiplot (default is 111).

- **title**: string
  Plot title. If set to `None` (the default value), a title will be constructed from the other plot parameters. Set to '' for no title.

- **xlabel**: string
  x axis label (default is 'x (m)').

- **ylabel**: string
  y axis label (default is 'y (m)').

- **contours**: Boolean, list or `np.array`
  Set to `True` or to a list or array of contour values to draw contours on the plot (default `False`).

- **contour_label_format**: string
  Format string for contour labels (default '%3.0f').

- **contour_grid_divisions**: tuple (of integer)
  Number of divisions in the x- and y-directions in the regular grid superimposed on the model grid, and used to produce the contours (default (100,100)).

- **connections**: float (or `None`)
  Set non-zero to plot connections in the grid, shaded by absolute value of the connection angle cosine. The value specifies the lower cut-off value, above which connections will be plotted. Connections are shaded in greyscale from white (0.0) to black (1.0). This can be used to check orthogonality of grid connections, as less orthogonal connections (with larger angle cosine) will show up darker on the plot. If set to `None`, no connections will be plotted.

- **colourbar_limits**: tuple, list, `np.array` (or `None`)
  Specify a two-element tuple, list or `np.array` to set the limits of the colour scale. Default (`None`) will auto-scale.

- **plot_limits**: tuple or list (or `None`)
  Specify a two-element tuple (or list) of plot axis ranges, each itself being a tuple (or list) of minimum and maximum values, i.e. ((xmin,xmax),(ymin,ymax)). Default is `False` which will auto-scale.

Example:

```
geo.layer_plot(-500.,t,'Temperature','$^o$C', contours=np.arange(100,200,25))
```

plots the variable `t` at elevation -500 m over the grid, with the values as Temperature (°C), and with contours drawn from 100^{o}C to 200^{o}C with a contour interval of 25^{o}C.

`line_values(`*`start`*`, `*`end`*`, `*`variable`*`, `*`divisions`*`=100, `*`coordinate`*`=False, `*`qtree`*`=None)`

Returns values of a specified variable along an arbitrary line through the grid. The start and end points of the line (`start` and `end`) are 3-element lists, tuples or `np.arrays` specifying points in 3D. The variable can be a list or `np.array` containing a value for every block in the grid. The number of divisions along the line (default 100) can be optionally specified.

The routine returns a tuple of two arrays (l,v), the first (l) containing the distance from the start (or the appropriate coordinate (0,1, or 2) if `coordinate` is specified) for each point

along the line, and the second (v) containing the value of the variable at that point. The value of the variable at any point is the (block average) value at the block containing the point.

Parameters:

- **start**, **end**: list, tuple or `np.array` (of length 3)
  Start and end points of the line in 3D.
- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks in the grid.
- **divisions**: integer
  Number of segments the line is divided up into (default 100).
- **coordinate**: integer or Boolean
  If `False`, return distance along the line in first array, otherwise return specified coordinate (0,1 or 2) values.
- **qtree**: `quadtree`
  Quadtree object for fast searching of grid columns (can be constructed using the `column_quadtree()` method).

`line_plot(`*`start`*`=None, `*`end`*`=None, `*`variable`*`, `*`variable_name`*`=None, `*`unit`*`=None, `*`divisions`*`=100, `*`plt`*`=None, `*`subplot`*`=111, `*`title`*`='', `*`xlabel`*`='distance (m)')`

Plots a variable along a line through the grid, using the `matplotlib` plotting library. The line is specified by its start and end points in 3D. The variable can be a list or `np.array` containing a value for every block (or column) in the grid. If the variable contains a value for each column in the grid, these values are extended down each column to fill the entire grid. The name and units of the variable can optionally be specified, as well as the number of divisions the line is divided into (default 100), the plot title and the axis labels.

Parameters:

- **start**, **end**: list, tuple or `np.array`
  Start and end point of the line, each of length 3 (`None` to plot across the bounds of the grid).
- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks (or columns) in the grid.
- **variable_name**: string
  Name of the variable (as it will appear on the scale of the plot).
- **unit**: string
  Units of the variable (as it will appear on the scale of the plot).
- **divisions**: integer
  Number of divisions to divide the line into (default 100).
- **plt**: `matplotlib.pyplot` instance
  An instance of the `matplotlib.pyplot` library, imported in the calling script using e.g. `import matplotlib.pyplot as plt`.

- **subplot**: integer
  Subplot number for multi-plots, e.g. set 223 to draw the third plot in a 2-by-2 multiplot (default is 111).

- **title**: string
  Plot title. If set to `None` (the default value), a title will be constructed from the other plot parameters. Set to '' for no title.

- **xlabel**: string
  x axis label (default is 'distance (m)').

Example:

```
geo.line_plot([0.,0.,500.],[1000.,0.,500.],t,'Temperature','$^o$C')
```

plots the variable `t` along a line from (0,0,500) to (1000,0,500) through the grid, with the values as Temperature (°C).

`nodes_in_columns(`*`columns`*`)`

Returns a list of all nodes in a specified list of columns.

Parameters:

- **columns**: list (of `column`)
  List of columns in which to find nodes.

`nodes_in_polygon(`*`polygon`*`)`

Returns a list of all nodes inside the specified polygon or rectangle.

Parameters:

- **polygon**: list (of `np.array`)
  List of points defining the polygon (each point is a two-element `np.array`). If the list has only two points, it will be interpreted as a rectangle [bottom left, top right].

`node_nearest_to(`*`point, kdtree`*`=None)`

Returns the node nearest to a specified point. An optional kd-tree structure can be specified to speed searching- useful if searching for many points.

Parameters:

- **point**: `np.array`, list or tuple
  Array or list of length 2, specifying the required point in 2-D.

- **kdtree**: `cKDTree`
  kd-tree structure for searching for nodes. Such a tree can be constructed using the `node_kdtree` property of a `mulgrid` object. You will need the `scipy` library installed before you can use this property.

`optimize(`*`nodenames`*`=None, `*`connection_angle_weight`*`=1.0, `*`column_aspect_weight`*`=0.0, `*`column_skewness_weight`*`=0.0, `*`pest`*`=False)`

Adjusts positions of the specified nodes to optimize grid quality. If no nodes are specified, all node positions are optimized. Grid quality can be defined as a combination of connection angle cosine, column aspect ratio and column skewness. Increasing the weight for any of these increases its importance in the evaluation of grid quality.

Note that an error will result if the connection angle weight and either of the other two weights is set to zero- in this case there are not enough constraints to fit the parameters.

If the `pest` parameter is set to `True`, the PEST parameter estimation software is used to carry out the optimzation (this obviously requires that PEST is installed on your machine). Otherwise, the `leastsq` routine in the `scipy` Python library is used. Particularly for large problems, PEST may give quicker results. PEST is free software and may be downloaded from `http://www.pesthomepage.org/`. If PEST is used, a variety of intermediate files (named `pestmesh.*`) will be written to the working directory, including the PEST run record file (`pestmesh.rec`) which contains a detailed record of the optimization process.

Parameters:

- **nodenames**: list of string
  List of names of nodes to optimize. If not specified, all nodes in the grid are optimized.
- **connection_angle_weight**: float
  Weighting to be given to connection angle cosines. A higher value will place greater priority on making connections perpendicular to the column sides.
- **column_aspect_weight**: float
  Weighting to be given to column aspect ratios. A higher value will place greater priority on making column side ratios closer to 1.0.
- **column_skewness_weight**: float
  Weighting to be given to column skewness. A higher value will place greater priority on making column angle ratios closer to 1.0.
- **pest**: Boolean
  Set `True` to use the PEST parameter estimation software to perform the optimization.

`polyline_values(`*`polyline`*`, `*`variable`*`, `*`divisions`*`=100, `*`coordinate`*`=False, `*`qtree`*`=None)`

Returns values of a specified variable along an arbitrary polyline through the grid, defined as a list of 3-element lists or `np.arrays` specifying points in 3D. The variable can be a list or `np.array` containing a value for every block in the grid. The number of divisions along the line (default 100) can be optionally specified.

The routine returns a tuple of two arrays (`l`,`v`), the first (`l`) containing the distance from the start (or the appropriate coordinate (0, 1, or 2) if `coordinate` is specified) for each point along the polyline, and the second (`v`) containing the value of the variable at that point. The value of the variable at any point is the (block average) value at the block containing the point.

Parameters:

- **polyline**: list of 3-element lists or `np.arrays`
  Polyline points in 3D.

- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks in the grid.
- **divisions**: integer
  Number of segments the line is divided up into (default 100).
- **coordinate**: integer or Boolean
  If `False`, return distance along the line in first array, otherwise return specified coordinate (0, 1 or 2) values.
- **qtree**: `quadtree`
  Quadtree object for fast searching of grid columns (can be constructed using the `column_quadtree()` method).

`read(`*`filename`*`)`

Reads a `mulgrid` object from a MULgraph geometry file on disk.

Parameters:

- **filename**: string
  Name of the MULgraph geometry file to be read.

Example:

```
geo=mulgrid().read(filename)
```

creates a `mulgrid` object and reads its contents from file `filename`. This can be done more simply just by passing the filename into the `mulgrid` creation command:

```
geo=mulgrid(filename)
```

`rectangular(`*`xblocks`*`, `*`yblocks`*`, `*`zblocks`*`, `*`convention`*`=0, `*`atmos_type`*`=2, `*`origin`*`=[0,0,0], `*`justify`*`='r', `*`case`*`='l')`

Gives a `mulgrid` geometry object a rectangular grid structure. The grid sizes in the x, y and z directions can be non-uniform, and the grid column and layer naming convention, atmosphere type and origin can be specified. The optional `justify` and `case` parameters control the formatting of the character part of the block names.

Parameters:

- **xblocks**, **yblocks**, **zblocks**: list, tuple or `np.array`
  Lists (or arrays) of block sizes in the x, y and z directions.
- **convention**: integer
  Naming convention for grid columns and layers.
- **atmos_type**: integer
  Type of atmosphere.
- **origin**: list (or `np.array`)
  Origin of the grid (of length 3).
- **justify**: string
  Specify 'r' for the character part of the block names (first three characters) to be right-justified, 'l' for left-justified.

- **case**: string
  Specify 'l' for the character part of the block names (first three characters) to be lower case, 'u' for upper case.

Example:

```
geo=mulgrid().rectangular([1000]*10,[500]*20,[100]*5+[200]*10,
  origin=[0,0,2500])
```

creates a `mulgrid` object called `geo`, and fills it with a rectangular grid of 10 blocks of size 1000 m in the x-direction, 20 blocks of size 500 m in the x-direction, 5 layers at the top of thickness 100 m and 10 layers underneath of thickness 200 m, and with origin (0,0,2500) m. The grid will have the default naming convention (0) and atmosphere type (2).

`reduce(`*`columns`*`)`

Reduces a grid so that it contains only the specified list of columns (or columns with specified names).

Parameters:

- **columns**: list
  List of required columns or column names.

`refine(`*`columns`*`=[], `*`bisect`*`=False, `*`bisect_edge_columns`*`=[])`

Refines the specified columns in the grid. Appropriate transition columns are created around the refined region. If no columns are specified, all columns are refined. All columns in the region to be refined (and in the transition region) must be either triangular or quadrilateral. Each column in split into four, unless the `bisect` parameter is `True`, in which case each column in split into two. If `bisect` is 'x' or 'y', columns are split in the closest direction to the axis specified; or if `bisect` is `True`, between its longest sides.

The `bisect_edge_columns` parameter can be used to give more desirable column shapes in the transition region, if the original columns occupying the transition region have large aspect ratios. By default, these will become even worse when they are triangulated to form the transition columns, if they are connected to the refinement region by their shorter sides. Including them in `bisect_edge_columns` means they will be bisected (parallel to the edge of the refinement region) before the refinement is carried out, which should improve the aspect ratios of the transition columns.

Parameters:

- **columns**: list
  List of columns or column names to be refined.

- **bisect**: Boolean or string
  Set to `True` if columns are to be split into two, between their longest sides, instead of four (the default). Set to 'x' or 'y' to split columns along the specified axis.

- **bisect_edge_columns**: list
  List of columns or column names in the transition region (just outside the refinement area) to be bisected prior to the refinement, to improve the aspect ratios of the transition columns.

`refine_layers(`*`layers`*`=[], `*`factor`*`=2)`

Refines the specified layers in the grid. If no layers are specified, all layers are refined. Each layer is refined by the specified integer factor.

Parameters:

- **layers**: list
  List of layers or layer names to be refined.
- **factor**: integer
  Refinement factor: default is 2, which bisects each layer.

`rename_column(`*`oldcolname`*`, `*`newcolname`*`)`

Renames a grid column. Returns `True` if the column was found and renamed, or `False` if the specified column does not exist.

Parameters:

- **oldcolname**: string
  Name of the column to rename.
- **newcolname**: string
  New name of the column.

`rename_layer(`*`oldlayername`*`, `*`newlayername`*`)`

Renames a grid layer. Returns `True` if the layer was found and renamed, or `False` if the specified layer does not exist.

Parameters:

- **oldlayername**: string
  Name of the layer to rename.
- **newlayername**: string
  New name of the layer.

`rotate(`*`angle`*`, `*`centre`*`=None, `*`wells`*`=False)`

Rotates a grid by a specified angle (in degrees) clockwise in the horizontal plane. Any wells in the grid are also rotated. The centre of rotation can be optionally specified. If it is not specified, the centre of the grid is used as the centre of rotation. If the `wells` parameter is `True`, any wells in the grid are also rotated.

Parameters:

- **angle**: float
  Angle (in degrees) to rotate the grid, positive for clockwise, negative for anti-clockwise.
- **centre**: list, tuple or `np.array`
  Centre of rotation in the horizontal x,y plane (of length 2).
- **wells**: Boolean
  Set `True` to rotate wells.

Example:

```
geo.rotate(30)
```

rotates the grid `geo` clockwise by 30° about its centre in the horizontal plane.

slice_plot(*line*=None, *variable*=None, *variable_name*=None, *unit*=None, *block_names*=None, *colourmap*=None, *linewidth*=0.2, *linecolour*='black', *aspect*='auto', *plt*=None, *subplot*=111, *title*=None, *xlabel*='', *ylabel*='elevation (m)', *contours*=False, *contour_label_format*='%3.0f', *contour_grid_divisions*=(100,100), *colourbar_limits*=None, *plot_limits*=None, *column_axis*=False, *layer_axis*=False)

Plots a variable over a vertical slice through the grid, using the `matplotlib` plotting library. The required slice is specified by a horizontal line through the grid, defined as either a two-element list of (x,y) points (`np.arrays`), or as a string 'x' or 'y' which defines the x- or y-axes respectively, or as a northing (in degrees) through the centre of the grid. If no line is specified, the line is taken to be across the bounds of the grid. For slice plots along the x- or y-axis, the horizontal coordinate represents the x- or y-coordinate; for other slice directions it represents distance along the slice line.

The variable can be a list or `np.array` containing a value for every block (or column) in the grid. If no variable is specified, only the grid is plotted, without shading. If the variable contains a value for each column in the grid, these values are extended down each column to fill the entire grid.

The name and units of the variable can optionally be specified, and the name of each block can also optionally be displayed on the plot. The colour map and limits of the variable shading, the line width of the grid columns and the aspect ratio of the plot can also be set, as can the plot title and x- and z-axis labels, and the plot limits.

When a variable is plotted over the grid, contours at specified levels can also be drawn, and optionally labelled with their values.

Parameters:

- **line**: list, string or float
  List of two horizontal (x,y) points (`np.arrays`) defining the endpoints of the line, or string 'x' or 'y' to specify the x- or y-axis, or northing (float) through grid centre.

- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks (or columns) in the grid (or `None` just to plot the grid).

- **variable_name**: string
  Name of the variable (as it will appear on the scale of the plot).

- **unit**: string
  Units of the variable (as it will appear on the scale of the plot).

- **block_names**: Boolean or list
  Set to `True` if block names are to be indicated on the plot, or to a list of names of blocks to be named.

- **colourmap**: string
  Name of `matplotlib` colour map to use for shading the variable.

- **linewidth**: float
  Line width to use for drawing the grid.

- **linecolour**: string
  Line colour to use for drawing the grid.

- **aspect**: string
  Aspect ratio to use for drawing the grid (default is 'auto').

- **plt**: `matplotlib.pyplot` instance
  An instance of the `matplotlib.pyplot` library, imported in the calling script using e.g. `import matplotlib.pyplot as plt`.

- **subplot**: integer
  Subplot number for multi-plots, e.g. set 223 to draw the third plot in a 2-by-2 multiplot (default is 111).

- **title**: string
  Plot title. If set to `None` (the default value), a title will be constructed from the other plot parameters. Set to '' for no title.

- **xlabel**: string
  x axis label. If set to `None` (the default value), a label will be constructed according to the slice orientation- either 'x (m)', 'y (m)' or 'distance (m)' as appropriate.

- **ylabel**: string
  y axis label (default is 'elevation (m)').

- **contours**: Boolean, list or `np.array`
  Set to `True` or to a list or array of contour values to draw contours on the plot (default `False`).

- **contour_label_format**: string
  Format string for contour labels (default '%3.0f').

- **contour_grid_divisions**: tuple (of integer)
  Number of divisions in the x- and z-directions in the regular grid superimposed on the slice, and used to produce the contours (default (100,100)).

- **colourbar_limits**: tuple, list, `np.array` (or `None`)
  Specify a two-element tuple, list or `np.array` to set the limits of the colour scale. Default (`None`) will auto-scale.

- **plot_limits**: tuple or list (or `None`)
  Specify a two-element tuple (or list) of plot axis ranges, each itself being a tuple (or list) of minimum and maximum values, i.e. ((xmin,xmax),(zmin,zmax)). Default is `False` which will auto-scale.

- **column_axis**: Boolean
  If `True`, show column names instead of coordinates on the horizontal axis.

- **layer_axis**: Boolean
  If `True`, show layer names instead of coordinates on the vertical axis.

Example:

```
geo.slice_plot(45.,t,'Temperature','$^o$C',contours=[100,200])
```

plots the variable `t` through a SW–NE vertical slice (heading 45°) through the grid, with the values as Temperature ($^\circ C$) and contours drawn at 100^{o}C and 200^{o}C.

`snap_columns_to_layers(`*`min_thickness`*`=1.0, `*`columns`*`=[])`

Snaps column surfaces to the bottom of their layers, if the surface block thickness is smaller than a given value. This can be carried out over an optional subset of columns in the grid, otherwise over all columns.

Parameters:

- **min_thickness**: float
  Minimum surface block thickness. Blocks with thickness less than this value will be eliminated by 'snapping' the column surface elevation to the bottom of the surface layer. Values of `min_thickness` less than or equal to zero will have no effect.

- **columns**: list
  List of columns to process. If empty (the default), process all columns.

`split_column(`*`colname`*`, `*`nodename`*`)`

Splits a quadrilateral column with specified name into two triangular columns. The direction of the split is determined by specifying the name of one of the splitting nodes. The method returns `True` if the split was carried out successfully.

Parameters:

- **colname**: string
  Name of the quadrilateral column to be split. If the column is not quadrilateral, the method returns `False` and nothing is done to the column.

- **nodename**: string
  Name of one of the splitting nodes. The column is split across this node and the one on the opposite side of the column. If the specified node is not in the column, the method returns `False` and nothing is done to the column.

`translate(`*`shift`*`, `*`wells`*`=False)`

Translates a grid by a specified shift in the x, y and z directions. If the `wells` parameter is `True`, any wells in the grid are also translated.

Parameters:

- **shift**: list, tuple or `np.array`
  Distance to shift the grid in the x, y and z directions (of length 3).

- **wells**: Boolean
  Set `True` to translate wells.

Example:

```
geo.translate([10.e3,0.0,-1000.0])
```

translates the grid `geo` by 10 km in the x direction and down 1 km in the z direction.

`well_values(`*`well_name`*`, `*`variable`*`, `*`divisions`*`=1, `*`elevation`*`=False, `*`deviations`*`=False, `*`qtree`*`=None, `*`extend`*`=False)`

Returns values of a specified variable down a specified well. The variable can be a list or `np.array` containing a value for every block in the grid. The number of divisions between layer centres or along each well deviation (default 1) can be optionally specified (this can be increased to capture detail along a deviation that passes through several blocks). If `deviations` is `True`, values will be returned at the nodes of the well track, instead of at grid layer centres. If `extend` is `True`, the well trace is artificially extended to the bottom of the model.

The routine returns a tuple of two arrays (`d`,`v`), the first (`d`) containing the measured depth down the well (or elevation if the `elevation` parameter is set to `True`), and the second (`v`) containing the value of the variable at each point. The value of the variable at any point is the (block average) value at the block containing the point.

Parameters:

- **well_name**: string
  Name of the well.
- **variable**: list (or `np.array`)
  Variable to be plotted, of length equal to the number of blocks in the grid.
- **divisions**: integer
  Number of divisions each well deviation is divided up into (default 1).
- **elevation**: Boolean
  Set to `True` if elevation rather than measured depth is to be returned.
- **deviations**: Boolean
  Set to `True` to return values at deviation nodes, rather than intersections of layer centres with the well track.
- **qtree**: `quadtree`
  Quadtree object for fast searching of grid columns (can be constructed using the `column_quadtree()` method).
- **extend**: Boolean
  Set `True` to artificially extend the well trace to the bottom of the model.

`write(`*`filename`*`='')`

Writes a `mulgrid` object to a MULgraph geometry file on disk.

Parameters:

- **filename**: string
  Name of the MULgraph geometry file to be written. If no file name is specified, the object's own `filename` property is used.

`write_bna(`*`filename`*`='')`

Writes a geometry object to an Atlas BNA file on disk, for visualisation with Surfer or GIS tools.

Parameters:

- **filename**: string
  Name of the BNA file to be written. If no file name is specified, the object's own `filename` property is used, with the extension changed to *.bna. If the object's `filename` property is not set, the default name 'geometry.bna' is used.

`write_vtk(`*`filename`*`='', `*`arrays`*`=None, `*`wells`*`=False)`

Writes a `mulgrid` object to a VTK file on disk, for visualisation with VTK, Paraview, Mayavi etc. The grid is written as an 'unstructured grid' VTK object with optional data arrays defined on cells. A separate VTK file for the wells in the grid can optionally be written.

Parameters:

- **filename**: string
  Name of the VTK file to be written. If no file name is specified, the object's own `filename` property is used, with the extension changed to *.vtu. If the object's `filename` property is not set, the default name 'geometry.vtu' is used.
- **arrays**: dictionary or `None`
  Data arrays to be included in the VTK file. If set to `None`, default arrays (layer index, column index, column area, column elevation, block number and volume) are included.
- **wells**: Boolean
  If set to `True`, a separate VTK file is written representing the wells in the grid.

2.3 Other objects (`node`, `column`, `layer`, `connection` and `well`)

A `mulgrid` object contains lists of other types of objects: `node`, `column`, `layer`, `connection` and `well` objects. These classes are described below.

2.3.1 `node` objects

A `node` object represents a node (i.e. vertex) in a `mulgrid` object. A `node` object has three properties: `name`, which is a string property containing the name of the node, `pos` which is an `np.array` with three elements, containing the node's position in 3D, and `column` which is a set of the columns the node belongs to. A `node` object does not have any methods.

A `node` object `n` can be created for example using the command `n = node(name,pos)` where `name` is the node name and pos is an `np.array` (or list, or tuple) representing the node's position.

2.3.2 `column` objects

A `column` object represents a column in a `mulgrid` object. The properties of a `column` object are listed in Table 2.5.

The main properties defining a column are its `name` and `node` properties. The `name` is specified according to the naming convention of the `mulgrid` object that the column belongs to. The `node` property is a list of `node` objects (not node names) that belong to the column. A `column`'s `neighbour` property is a set of other `columns` connected to that column via a `connection` (see section 2.3.4), and its `connection` property is a set of connections the column is part of. The `neighbourlist` property is a list of neighbouring columns, with each

item corresponding to a column edge (`None` if the edge is on a grid boundary). A `column`'s `centroid` property returns the average of the positions of its vertices- which is what the `centre` property is set to, unless otherwise specified.

A `column` object has two properties measuring 'grid quality'. The `angle_ratio` property returns the ratio of largest to smallest interior angles in the column. The `side_ratio` property returns the ratio of largest to smallest side lengths (a generalisation of 'aspect ratio' to columns with any number of sides). Values as close as possible to 1.0 for both these measures are desirable (their values are both exactly 1.0 for any regular polygon, e.g. an equilateral triangle or square). Columns with large angle ratios will be highly skewed, while those with large side ratios will be typically highly elongated in one direction.

A `column` object `col` can be created for example using the command `col = column(name,nodes,centre,surfa` where `name` is the column name and `nodes` is a list of `node` objects defining the column. The `centre` and `surface` parameters are optional.

`column` objects have three methods, `contains_point`, `in_polygon` and `is_against`, as described below.

`contains_point(`*pos*`)`

Returns `True` if a 2D point lies inside the column, and `False` otherwise.

Parameters:

- **pos**: `np.array`
  Horizontal position of the point.

`in_polygon(`*polygon*`)`

Returns `true` if the column centre is inside the specified polygon or rectangle.

Parameters:

- **polygon**: list (of `np.array`)
  List of points defining the polygon (each point is a two-element `np.array`). If the list has only two points, it will be interpreted as a rectangle [bottom left, top right].

`is_against(`*othercolumn*`)`

Returns `true` if the column is 'against' `othercolumn`- that is, if it shares more than one node with it.

Parameters:

- **othercolumn**: `column`)
  Any other column in the geometry.

2.3.3 `layer` objects

A `layer` object represents a layer in a `mulgrid` object. The properties of a `layer` object are given in Table 2.6.

A `layer` object `lay` can be created for example using the command `lay = layer(name,bottom,centre,top)` where `name` is the layer name and `bottom`, `centre` and `top` specify the vertical position of the layer.

The methods of a `layer` object are as follows:

Property	Type	Description
angle_ratio	float	ratio of largest to smallest interior angles
area	float	horizontal area of the column
centre	np.array	horizontal centre of the column
centroid	np.array	average position of the column's vertices
connection	set	connections the column is in
name	string	name of the column
neighbour	set	set of neighbouring columns
neighbourlist	list	ordered list of neighbouring columns
node	list	list of nodes (vertices) belonging to the column
num_neighbours	integer	number of neighbouring columns
num_nodes	integer	number of nodes belonging to the column
num_layers	integer	number of layers in the column below the ground surface
side_ratio	float	ratio of largest to smallest side length
surface	float	surface elevation of the column (None if not specified)

Table 2.5: Properties of a column object

Property	Type	Description
bottom	float	elevation of the bottom of the layer
centre	float	elevation of the centre of the layer
thickness	float	layer thickness (top - bottom)
top	float	elevation of the top of the layer
name	string	name of the layer

Table 2.6: Properties of a layer object

contains_elevation(*z*)

Returns True if a point at a given elevation lies inside the layer, and False otherwise.

Parameters:

- **z**: float
  Elevation of the point.

translate(*shift*)

Translates a layer up or down by a specified distance.

Parameters:

- **shift**: float
  Distance to shift the layer (positive for up, negative for down).

2.3.4 connection objects

A connection object represents a connection between columns in a mulgrid object. It has three properties: column, which contains a two-element list of the column objects making up the connection, node, which contains a two-element list of the nodes on the face joining

Property	Type	Description
bottom	np.array	well bottom position
deviated	Boolean	whether well is deviated
head	np.array	well head position
name	string	well name
num_deviations	integer	number of deviations
num_pos	integer	number of well track nodes
pos	list	positions (3-D arrays) of well track nodes
pos_depth	np.array	downhole depths along well track

Table 2.7: Properties of a well object

Method	Type	Description
depth_elevation	float	elevation for a given downhole depth
depth_pos	np.array	position on well track for a given downhole depth
elevation_depth	float	downhole depth for a given elevation
elevation_pos	np.array	position on well track for a given elevation
pos_coordinate	np.array	array of coordinates for a given index

Table 2.8: Methods of a well object

the two columns in the connection, and angle_cosine, which gives the cosine of the angle between a line joining the nodes in the connection and a line joining the centres of the two columns. This is used as a measure of grid quality, these two lines should ideally be as close to perpendicular as possible, making the cosine of the angle zero. A connection has no methods.

A connection object con can be created for example using the command con = connection(cols) where cols is a two-element list of the column objects in the connection.

2.3.5 well objects

A well object represents a well in a mulgrid object. The properties of a well object are given in Table 2.7.

The well track can be deviated, and is defined as a list pos of (at least two) 3D positions (np.arrays). The num_deviations property returns the number of deviations in the track (one less than the num_pos property, which is the number of nodes in the pos list). The deviated property returns True if there is more than one deviation. The pos_depth property returns an array of the downhole depths at each node along the well track.

A well object w can be created simply with the command w = well(name,pos), where name is the well name and pos is a list of 3-element np.arrays (or lists, or tuples) representing the well trace (starting from the wellhead).

The methods of a well are listed in Table 2.8 and described below.

depth_elevation(*depth*)

Returns the elevation corresponding to the specified downhole depth (or None if depth is above the wellhead or below the bottom).

Parameters:

- **depth**: float
  Downhole depth.

`depth_pos(`*`depth`*`)`

Returns the 3D position of the point in the well with specified downhole `depth` (or `None` if `depth` is above the wellhead or below the bottom). The position is interpolated between the deviation locations.

Parameters:

- **depth**: float
  Downhole depth of the required point.

`elevation_depth(`*`elevation`*`)`

Returns the downhole depth corresponding to the specified `elevation` (or `None` if `elevation` is above the wellhead or below the bottom).

Parameters:

- **elevation**: float
  Elevation.

`elevation_pos(`*`elevation`*`, `*`extend`*`=False)`

Returns the 3D position of the point in the well with specified `elevation` (or `None` if `elevation` is above the wellhead or below the bottom). The position is interpolated between the deviation locations. If `extend` is `True`, return extrapolated positions for elevations below the bottom of the well.

Parameters:

- **elevation**: float
  Elevation of the required point.
- **extend**: Boolean
  If `True`, extrapolated positions will be returned for elevations below the bottom of the well (otherwise `None` will be returned).

`pos_coordinate(`*`index`*`)`

Returns an `np.array` of the well track node coordinates for the given index (0, 1 or 2). For example, `pos_coordinate(2)` returns an array containing the elevations of all well track nodes.

Parameters:

- **index**: integer
  Index required (0, 1 or 2).

2.4 Other functions: block name conversions

The `mulgrids` library contains two other functions connected with working with geometry files and TOUGH2 grids:

`fix_blockname(`*name*`)`

TOUGH2 always assumes that the last two characters of a block name represent a two-digit number. However, if that number is less than 10, the fourth character is not padded with zeros, so for example 'AA101' becomes 'AA1 1' when processed by TOUGH2.

The `fix_blockname` function corrects this by padding the fourth character of a block name with a zero if necessary. This is only done if the third character is also a digit, e.g. when naming convention 2 is used (two characters for layer followed by three digits for column).

Parameters:

- **name**: string
  Block name.

`unfix_blockname(`*name*`)`

This function reverses the effect of `fix_blockname()`.

Parameters:

- **name**: string
  Block name.

Chapter 3

TOUGH2 grids

3.1 Introduction

The `t2grids` library in PyTOUGH contains classes and routines for manipulating TOUGH2 grids. It can be imported using the command:

```
from t2grids import *
```

3.2 `t2grid` objects

The `t2grids` library defines a `t2grid` class, used for representing TOUGH2 grids. This gives access via Python to the grid's rock types, blocks, connections and other parameters.

Normally a TOUGH2 grid is not created directly, but is either read from a TOUGH2 data file, or constructed from a `mulgrid` geometry object (see chapter 2) using the `fromgeo` method (see section 3.2.2).

Printing a `t2grid` object (e.g. `print grid`) displays a summary of information about the grid: how many rock types, blocks and connections it contains.

3.2.1 Properties

The main properties of a `t2grid` object are listed in Table 3.1. Essentially a `t2grid` object contains collections of blocks, rock types and connections, each accessible either by name or by index. For example, block 'AB 20' in a `t2grid` called `grid` is given by `grid.block['AB 20']`.

Connections are slightly different from blocks or rock types, in that they are not named individually. However, they can be accessed by the names of the blocks connected by the connection. For example, the connection between blocks 'aa 10' and 'ab 10' in a `t2grid` called `grid` is given by `grid.connection['aa 10','ab 10']`.

The `rocktype_frequencies` property gives information about how frequently each rock type is used (i.e. how many blocks use that rock type). It returns a list of tuples, the first element of each tuple being the frequency of use, and the second element being a list of rock type names with that frequency. The list is given in order of increasing frequency.

The `rocktype_indices` property gives an `np.array` containing the index of the rocktype for each block in the grid. This can be used to give a plot of rock types, in conjunction with the `mulgrid` methods `layer_plot` or `slice_plot`.

Property	Type	Description
atmosphere_blocks	list	atmosphere blocks
blocklist	list	blocks (by index)
block	dictionary	blocks (by name)
block_centres_defined	Boolean	whether block centres have been calculated
connectionlist	list	connections (by index)
connection	dictionary	connections (by tuples of block names)
num_atmosphere_blocks	integer	number of atmosphere blocks
num_blocks	integer	number of blocks
num_connections	integer	number of connections
num_rocktypes	integer	number of rock types
num_underground_blocks	integer	number of non-atmosphere blocks
rocktypelist	list	rock types (by index)
rocktype	dictionary	rock types (by name)
rocktype_frequencies	list of tuples	frequencies of rock types
rocktype_indices	np.array	index of rock type for each block

Table 3.1: Properties of a t2grid object

3.2.2 Methods

The main methods of a t2grid object are listed in Table 3.2. Details of these methods are given below.

+

Adds two grids a and b together (i.e. amalgamates them) to form a new grid a+b. If any rock types, blocks or connections exist in both grids a and b, the value from b is used, so there are no duplicates.

Parameters:

- **a, b**: t2grid
  The two grids to be added together.

add_block(*block*)

Adds a block to the grid. If another block with the same name already exists, it is replaced.

Parameters:

- **block**: t2block
  Block to be added to the grid.

add_connection(*connection*)

Adds a connection to the grid. If another connection with the same column names already exists, it is replaced.

Parameters:

- **connection**: t2connection
  Connection to be added to the grid.

Method	Type	Description
+	t2grid	adds two grids together
add_block	–	adds a block to the grid
add_connection	–	adds a connection to the grid
add_rocktype	–	adds a rock type to the grid
block_index	integer	returns index of a block with a specified name
calculate_block_centres	–	calculates geometrical centre of all blocks in the grid
check	Boolean	checks grid for errors and optionally fixes them
clean_rocktypes	–	deletes any unused rock types from the grid
connection_index	integer	returns index of a connection with a specified pair of names
copy_connection_directions	–	copies connection permeability directions from another grid
delete_block	–	deletes a block from the grid
delete_connection	–	deletes a connection from the grid
delete_rocktype	–	deletes a rock type from the grid
embed	t2grid	embeds a subgrid inside one block of another
empty	–	empties contents of grid
fromgeo	t2grid	constructs a TOUGH2 grid from a mulgrid object
incons	t2incon	constructs initial conditions for the grid
radial	t2grid	constructs a radial TOUGH2 grid
rocktype_frequency	integer	frequency of use of a particular rock type
sort_rocktypes	–	sorts rock type list into alphabetical order by name
write_vtk	–	writes grid to VTK file

Table 3.2: Methods of a t2grid object

`add_rocktype(rock)`

Adds a rock type to the grid. If another rock type with the same name already exists, it is replaced.

Parameters:

- **rock**: `rocktype`
  Rock type to be added to the grid.

`block_index(blockname)`

Returns the block index (in the `blocklist` list) of a specified block name.

Parameters:

- **blockname**: string
  Name of the block.

`calculate_block_centres(geo)`

Calculates geometrical centres of all blocks in the grid, based on the specified geometry object `geo`.

Parameters:

- **geo**: `mulgrid`
  Geometry object associated with the grid.

`check(fix=False,silent=False)`

Checks a grid for errors and optionally fixes them. Errors checked for are: blocks not connected to any other blocks, and blocks with isolated rocktypes (not shared with any neighbouring blocks). Returns `True` if no errors were found, and `False` otherwise. If `fix` is `True`, any identified problems will be fixed. If `silent` is `True`, there is no printout (only really useful if `fix` is `True`).

Blocks not connected to any others are fixed by deleting them. Isolated-rocktype blocks are fixed by assigning them the most popular rocktype of their neighbours. Blocks with large volumes ($> 1E20$ m^3) are never considered isolated (because they often have a special rocktype, such as an atmosphere one, that their neighbours will never share).

Parameters:

- **fix**: Boolean
  Whether to fix any problems identified.

- **silent**: Boolean
  Whether to print out feedback or not.

`clean_rocktypes()`

Deletes any rock types from the grid which are not assigned to any block.

`connection_index(blocknames)`

Returns the connection index (in the `connectionlist` list) of the connection between a specified pair of block names.

Parameters:

- **blocknames**: tuple
  A pair of block names, each of type string.

`copy_connection_directions(geo,grid)`

Copies the connection permeability directions for horizontal connections from another grid. It is assumed that both grids have the same column structure, but may have different layer structures.

Parameters:

- **geo**: `mulgrid`
  Geometry object associated with the source grid.
- **grid**: `t2grid`
  The source grid from which the connection permeability directions are to be copied.

`delete_block(blockname)`

Deletes a block from the grid. This also deletes any connections involving the specified block.

Parameters:

- **blockname**: string
  Name of the block to be deleted from the grid.

`delete_connection(connectionname)`

Deletes a connection from the grid.

Parameters:

- **connectionname**: tuple (of string)
  Pair of block names identifying the connection to be deleted from the grid.

`delete_rocktype(rocktypename)`

Deletes a rock type from the grid.

Parameters:

- **rocktypename**: string
  Name of the rock type to be deleted from the grid.

`embed(subgrid, connection)`

Returns a grid with a subgrid embedded inside one of its blocks. The connection specifies how the two grids are to be connected: the blocks to be connected and the connection distances, area etc. between them.

Parameters:

- **subgrid**: `t2grid`
  Subgrid to be embedded.

- **connection**: `t2connection`
  Connection specifying how the subgrid is to be embedded, including the connection distances and area. The first block should be the host block, the second the connecting block in the subgrid.

`empty()`

Empties the grid of all its blocks, rock types and connections.

`fromgeo(`*`geo`*`)`

Returns a grid constructed from a `mulgrid` geometry object. (Any previous contents of the grid are first emptied.)

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` geometry object (see chapter 2).

`incons(`*`values`*`=(101.3e3,20.))`

Returns a `t2incon` set of initial conditions for the grid, using the supplied values. Initial conditions can be specified for only one block, in which case they will be applied to all blocks, or for each block, in an array.

Parameters:

- **values**: `tuple` or `np.array`
  Initial conditions values, either a `tuple` of values for one block, or an `np.array` with each row containing a set of values for one block.

`radial(`*`rblocks`*`, `*`zblocks`*`, `*`convention`*`=0, `*`atmos_type`*`=2, `*`origin`*`=[0,0],`
`justify`*`='r', `*`case`*`='l', `*`dimension`*`=2)`

Returns a radial TOUGH2 grid with the specified radial and vertical block sizes. Grid column and layer naming convention, atmosphere type and origin can be specified. The optional *justify* and *case* parameters control the formatting of the character part of the block names.

The *dimension* parameter sets the flow dimension for 'generalized radial flow', which can represent flow in fractured rocks and modifies the block volumes and areas (see Barker (1988)). The default *dimension* = 2 corresponds to standard radial flow.

Parameters:

- **rblocks**, **zblocks**: list (or `np.array`)
  Lists (or arrays) of block sizes in the r and z directions.

- **convention**: integer
  Naming convention for grid columns and layers- same as the naming convention for a `mulgrid` object.

- **atmos_type**: integer
  Type of atmosphere- also the same as the atmosphere type for a `mulgrid` object.

- **origin**: list (or `np.array`)
  Origin of the grid (of length 2 or 3). The first entry is the radial origin, i.e. the starting radius of the grid. The last entry is the vertical origin, i.e. the vertical position of the top of the grid. If of length 3, the middle entry is ignored.

- **justify**: string
  Specify 'r' for the character part of the block names (first three characters) to be right-justified, 'l' for left-justified.

- **case**: string
  Specify 'l' for the character part of the block names (first three characters) to be lower case, 'u' for upper case.

- **dimension**: float
  Dimension for 'generalized radial flow', which can take any (possibly non-integer) value between 1 and 3. Dimension 1 corresponds to flow in a linear 'pipe', dimension 2 corresponds to standard radial flow in a disc-shaped reservoir and dimension 3 corresponds to flow in a spherically symmetric reservoir.

`rocktype_frequency(`*`rockname`*`)`

Returns the frequency of use of the rock type with the specified name, i.e. how many blocks are assigned that rock type.

Parameters:

- **rockname**: string
  Name of the specified rock type.

`sort_rocktypes()`

Sorts the rocktype list into alphabetical order by name.

`write_vtk(`*`geo`*`, `*`filename`*`, `*`wells`*`=False)`

Writes a `t2grid` object to a VTK file on disk, for visualisation with VTK, Paraview, Mayavi etc. The grid is written as an 'unstructured grid' VTK object with data arrays defined on cells. The data arrays written, in addition to the defaults arrays for the associated `mulgrid` object, are: rock type index, porosity and permeability for each block. A separate VTK file for the wells in the grid can optionally be written.

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` geometry object associated with the grid. This is required as the `t2grid` object does not contain any spatial information, e.g. locations of block vertices.

- **filename**: string
  Name of the VTK file to be written. This is also required.

- **wells**: Boolean
  Set to `True` if the wells from the `mulgrid` object are to be written to a separate VTK file.

3.3 Other objects (`rocktype`, `t2block` and `t2connection`)

A `t2grid` object contains lists of other types of objects: `rocktype`, `t2block` and `t2connection`. These classes are described below.

3.3.1 `rocktype` objects

A `rocktype` object represents a TOUGH2 rock type. The properties of a `rocktype` object, and their default values, are given in Table 3.3.

The main familiar properties of a rock type are referred to in a natural way, e.g. the porosity of a rock type `r` is given by `r.porosity`. The permeability property is a 3-element `np.array`, giving the permeability in each of the three principal axes of the grid, so e.g. the vertical permeability of a rock type `r` would normally be given by `r.permeability[2]` (recall that array indices in Python are zero-based, so that the third element has index 2).

Some rock type properties are optional, and only need be specified when the property `nad` is greater than zero. An example is the relative permeability and capillarity functions that can be specified for a rock type when `nad` ≥ 2. The way these functions are specified is described in chapter 4.

Example:

```
r=rocktype(name='ignim',permeability=[10.e-15,10.e-15,2.e-15],
  specific_heat=850)
```

declares a rocktype object called `r` with name 'ignim', permeability of 10 mD in the first and second directions and 2 mD in the vertical direction, and specific heat 850 J.kg^{-1}.K^{-1}.

(Note that when declaring rock types, the permeability can for convenience be specified as a list, which will be converted internally to an `np.array`.)

3.3.2 `t2block` objects

A `t2block` object represents a block in a TOUGH2 grid. The properties of a `t2block` object are given in Table 3.4. These reflect the specifications of a TOUGH2 block as given in a TOUGH2 data file, with the exception of the `atmosphere`, `centre`, `connection_name`, `neighbour_name` and `num_connections` properties.

The `atmosphere` property determines whether the block is to be treated as an atmosphere block. The `centre` property can optionally be used to specify the coordinates of the centre of a block. Block centres are automatically calculated when a `t2grid` object is constructed from a `mulgrid` object using the `fromgeo` method (see section 3.2.2). The `connection_name` property is a set containing the names (as tuples of strings) of all connections involving the block.

A `t2block` object has no methods.

3.3.3 `t2connection` objects

A `t2connection` object represents a connections between two TOUGH2 blocks. The properties of a `t2connnection` object are given in Table 3.5. These correspond to the properties of a connection specified in a TOUGH2 data file. Note that the `block` property returns `t2block` objects, not just the names of the blocks in the connection. Hence, for example, the volume of the first block in a connection object `con` is given simply by `con.block[0].volume`.

A `t2connection` object has no methods.

Property	Type	Description	Default
capillarity	dictionary	capillarity function	–
compressibility	float	compressibility	0 $m^2.N^{-1}$
conductivity	float	heat conductivity	1.5 $W.m^{-1}.K^{-1}$
density	float	rock grain density	2600 $kg.m^{-3}$
dry_conductivity	float	dry heat conductivity	wet heat conductivity
expansivity	float	expansivity	0 K^{-1}
klinkenberg	float	Klinkenberg parameter	0 Pa^{-1}
nad	integer	number of extra data lines	0
name	string	rock type name	'dfalt'
permeability	np.array	permeability	np.array([10^{-15}]*3) m^2
porosity	float	porosity	0.1
relative_permeability	dictionary	relative permeability function	–
specific_heat	float	rock grain specific heat	900 $J.kg^{-1}.K^{-1}$
tortuosity	float	tortuosity factor	0
xkd3	float	used by EOS7R	0 $m^3.kg^{-1}$
xkd4	float	used by EOS7R	0 $m^3.kg^{-1}$

Table 3.3: Properties of a rocktype object

Property	Type	Description
ahtx	float	interface area for heat exchange (TOUGH2 only)
atmosphere	Boolean	whether block is an atmosphere block or not
centre	np.array	block centre (optional)
connection_name	set	names of connections involving the block
nadd	integer	increment between block numbers in sequence
name	string	block name
neighbour_name	set	names of neighbouring (connected) blocks
nseq	integer	number of additional blocks in sequence
num_connections	integer	number of connections containing the block
pmx	float	permeability modifier (TOUGH2 only)
rocktype	rocktype	rock type
volume	float	block volume

Table 3.4: Properties of a t2block object

Property	Type	Description
area	float	connection area
block	list	two-element list of blocks
dircos	float	gravity direction cosine
direction	integer	permeability direction (1, 2, or 3)
distance	list	two-element list of connection distances
nad1,nad2	integer	increments in sequence numbering
nseq	integer	number of additional connections in sequence
sigma	float	radiant emittance factor (TOUGH2 only)

Table 3.5: Properties of a t2connection object

3.4 Example

The following piece of Python script creates a rectangular 2-D slice TOUGH2 grid with two rock types, and assigns these rock types to blocks in the grid according to their position along the slice.

```
from t2grids import *

geo=mulgrid().rectangular([500]*20,[1000],[100]*20,atmos_type=0,convention=2)
geo.write('2Dgrd.dat')
grid=t2grid().fromgeo(geo)

grid.add_rocktype(rocktype('greyw',permeability=[1.e-15]*2+[0.1e-15]))
grid.add_rocktype(rocktype('fill ',permeability=[15.e-15]*2+[5.e-15]))

for blk in grid.blocklist[1:]:
    if 200<=blk.centre[0]<=400: blk.rocktype=grid.rocktype['fill ']
    else: blk.rocktype=grid.rocktype['greyw']
```

The first line just imports the required PyTOUGH library. (It is not necessary to import the mulgrids library explicitly, because it is used and therefore imported by the t2grids library.)

The second block of code creates a rectangular MULgraph geometry object with 20 columns (each 500 m wide) along the slice and 20 layers (each 100 m thick), writes this to a geometry file on disk, and creates a TOUGH2 grid from it.

Then the two rock types are created, 'greyw' and 'fill '. (Note that rock types are expected by TOUGH2 to have names 5 characters long, so it is necessary to add spaces to shorter names.)

The final part assigns the rock types to the blocks in the grid. The loop starts from 1 instead of 0, so that the atmosphere block is skipped. In this example, the blocks in the grid are assigned the 'fill' rock type if they are between 200 m and 400 m along the slice. Blocks outside this region are assigned the 'greyw' rock type.

Chapter 4

TOUGH2 data files

4.1 Introduction

The `t2data` library in PyTOUGH contains classes and routines for creating, editing and saving TOUGH2 or AUTOUGH2 data files. It can be imported using the command:

```
from t2data import *
```

4.2 `t2data` objects

The `t2data` library defines a `t2data` class, used for representing TOUGH2 data files.

Example:

```
dat=t2data()
```

creates an empty `t2data` object called `dat`.

```
dat=t2data(filename)
```

creates a `t2data` object called `dat` and reads its contents from file `filename`. (It is also possible to read the mesh part of the `t2data` object from separate files- see below.)

Because a `t2data` object contains a large number of different parameters, it is usually easier to load one from an existing TOUGH2 data file and edit it, rather than creating a new one from scratch.

4.2.1 Properties

The main properties of a `t2data` object are listed in Table 4.1. In general, each of these properties corresponds to an input block in a TOUGH2 data file. Most of these input blocks contain a number of different parameters, so that the `t2data` property corresponding to each input block is usually in the form of a dictionary, containing a number of keys representing sub-properties.

For example, the maximum number of time steps for the simulation is controlled by `max_timesteps` key in the `parameter` property, which for a `t2data` object called `dat` would be accessed by `dat.parameter['max_timesteps']`.

The details of all the `t2data` properties are given below.

Property	Type	Description	Input block
capillarity	dictionary	capillarity function	RELP
diffusion	list	diffusion coefficients	DIFFU
echo_extra_precision	Boolean	echoing extra precision sections to main data file (AUTOUGH2 only)	–
end_keyword	string	keyword to end file	ENDCY or ENDFI
extra_precision	list	data sections read from extra precision auxiliary file (AUTOUGH2 only)	–
filename	string	file name on disk	–
generator	dictionary	generators (by block name and generator name)	GENER
generatorlist	list	generators (by index)	GENER
grid	t2grid	model grid	ELEME, CONNE
history_block	list	history blocks (TOUGH2 only)	FOFT
history_connection	list	history connections (TOUGH2 only)	COFT
history_generator	list	history generators (TOUGH2 only)	GOFT
incon	dictionary	initial conditions	INCON
indom	dictionary	rocktype-specific initial conditions	INDOM
lineq	dictionary	linear equation solver options (AUTOUGH2 only)	LINEQ
meshfilename	string or tuple	file name(s) on disk containing mesh data	–
meshmaker	list	mesh generation options	MESHM
multi	dictionary	EOS configuration	MULTI
noversion	Boolean	suppressing printing of version summary	NOVER
num_generators	integer	number of generators	–
output_times	dictionary	times to write output	TIMES
parameter	dictionary	run-time parameters	PARAM
relative_permeability	dictionary	relative permeability function	RELP
selection	dictionary	selection parameters	SELEC
short_output	dictionary	short output (AUTOUGH2 only)	SHORT
simulator	string	simulator name (AUTOUGH2 only)	SIMUL
solver	dictionary	linear equation solver options (TOUGH2 only)	SOLVR
start	Boolean	run initialisation option	START
title	string	simulation title	TITLE

Table 4.1: Properties of a t2data object

Key	Type	Description	TOUGH2 parameter
`parameters`	array (7) of float	function parameters	CP
`type`	integer	type of capillarity function	ICP

Table 4.2: `capillarity` property keys

`capillarity` property

A dictionary property specifying the capillarity function used, corresponding to the second line of the **RPCAP** input block in the TOUGH2 data file. The individual keys of this property are given in Table 4.2.

`diffusion` property

A list property specifying diffusion coefficients for each mass component simulated, corresponding to the **DIFFU** input block in the TOUGH2 data file. The list has length `multi['num_components']` (i.e. NK in TOUGH2 terminology), and each element is a list of the diffusion coefficients for each component (with length `multi['num_phases']`, or NPH).

`echo_extra_precision` property

A Boolean property (AUTOUGH2 only) governing whether data written to an auxiliary extra-precision file is also echoed to the main data file. If `True`, all extra-precision data sections are echoed to the main file.

`extra_precision` property

A list property determining which data sections will be written to an auxiliary extra-precision file (AUTOUGH2 only). Recent versions of AUTOUGH2 support an additional data file containing some data written with extra precision. Possible extra-precision data sections are ROCKS, RPCAP and GENER. Typical usage of this extra-precision data is for automatic model calibration using PEST or similar software, where calculation of derivatives of model outputs with respect to model parameters requires higher precision than is possible with the standard TOUGH2 data file format.

The `extra_precision` parameter may be a list containing names of sections to be written in extra precision (e.g. ['RPCAP', 'GENER']), or set to `False` to disable extra precision (equivalent to []), or to `True` to specify that all possible sections should be written in extra precision.

The `read()` method of a `t2data` object determines whether extra precision data are available by searching for an additional file with the same base name as the data file itself, but with a '.pdat' or '.PDAT' extension (depending on the case of the main data file name). If no such file exists, then no extra precision data will be read.

`filename` property

A string property containing the name of the TOUGH2 data file on disk. (This does not correspond to any parameter in the TOUGH2 data file.)

`generator` **property**

A dictionary property containing the generators for the simulation, accessed by tuples of block name and generator name. Each generator is an object of type `t2generator`, which is described in section 4.3.

`generatorlist` **property**

A list property containing the generators for the simulation, accessed by index.

`grid` **property**

A `t2grid` object (see chapter 3) representing the simulation grid, corresponding to the **ELEME** and **CONNE** input blocks in a TOUGH2 data file.

`history_block` **property**

A list property containing blocks for which time history output is required, corresponding to the **FOFT** input block in a TOUGH2 data file. If the `t2data` object contains grid data, the items in this list are `t2block` objects; otherwise, they are block names (i.e. strings).

`history_connection` **property**

A list property containing connections for which time history output is required, corresponding to the **COFT** input block in a TOUGH2 data file. If the `t2data` object contains grid data, the items in this list are `t2connection` objects; otherwise, they are tuples of block names (i.e. tuples of strings).

`history_generator` **property**

A list property containing blocks in which generators are defined and for which time history output is required, corresponding to the **GOFT** input block in a TOUGH2 data file. If the `t2data` object contains grid data, the items in this list are `t2block` objects; otherwise, they are block names (i.e. strings).

`incon` **property**

A dictionary property representing the initial conditions for the simulation, accessed by block name, corresponding to the **INCON** input block in a TOUGH2 data file. The value of each element of the dictionary is a list consisting of the porosity of the block followed by the specified initial primary thermodynamic variables in the block.

For example, to specify porosity 0.1 and initial conditions (101.3E3, 20.0) in block `'AB105'` of a `t2data` object called `dat`, set `dat.incon['AB105'] = [0.1, 101.3e3, 20.0]`.

`indom` **property**

A dictionary property representing the initial conditions for the simulation, accessed by rocktype name, corresponding to the **INDOM** input block in a TOUGH2 data file. The value of each element of the dictionary is a list consisting of the specified initial primary thermodynamic variables for the rocktype.

Key	Type	Description	AUTOUGH2 parameter
`epsilon`	float	solver tolerance	EPN
`gauss`	integer	Gauss elimination parameter	IGAUSS
`max_iterations`	integer	max. number of iterations	MAXIT
`num_orthog`	integer	number of orthogonalisations	NORTH
`type`	integer	type of solver (1 or 2)	ISOLVR

Table 4.3: `lineq` property keys

`lineq` property

A dictionary property representing linear equation solver options, corresponding to the **LINEQ** input block in an AUTOUGH2 data file. The individual keys of this property are given in Table 4.3.

`meshfilename` property

A string property (or tuple of strings) containing the name(s) of files on disk containing the mesh data. (This does not correspond to any parameter in the TOUGH2 data file.) Its default value is an empty string which means mesh data will be read from the main data file.

If `meshfilename` is a single (non-empty) string, this is interpreted as the name of a formatted text file containing 'ELEME' and 'CONNE' sections specifying the mesh (e.g. the 'MESH' file created by TOUGH2 or TOUGH2_MP).

If `meshfilename` is a tuple of two strings, these are interpreted as the names of two binary files containing the mesh data, e.g. the 'MESHA' and 'MESHB' files created by TOUGH2_MP.

`meshmaker` property

A list property representing mesh generation options, corresponding to the **MESHM** input block in a TOUGH2 data file. For more detail on the use of **MESHM** data, consult the TOUGH2 users' guide (Pruess et al., 1999).

The **MESHM** data may contain multiple sections (e.g. creation of a rectilinear XYZ grid followed by MINC processing), so the `meshmaker` property is structured as a list of two-element tuples, each containing the type of section (`rz2d`, `xyz` or `minc`) followed by the section data itself.

The form of the section data varies depending on the section type. For the `rz2d` type it is also structured as a list, as these types may contain variable numbers of sub-sections. (For example, data for the `rz2d` type may contain multiple `logar` sub-sections for different logarithmic radial parts of the mesh.) Each sub-section is again a two-element tuple, consisting of the sub-section type (a string) followed by a dictionary containing the data for the sub-section.

Data for the `xyz` type are also structured as a list, with the first element containing the stand-alone `deg` parameter (a float), followed by the other sub-sections, corresponding to the **NX**, **NY** and **NZ** sub-sections in the TOUGH2 data file. The `minc` type does not have sub-sections so MINC data are not structured as a list but simply a dictionary.

Possible sub-section types for `rz2d` data are `radii`, `equid`, `logar` and `layer`, corresponding to their (uppercase) keyword counterparts in the TOUGH2 data file. Data keys for these

Key	Type	Description	TOUGH2 parameter
radii sub-section keys			
`radii`	list	specified mesh radii	RC
equid sub-section keys			
`dr`	float	radial increment	DR
`nequ`	integer	number of equidistant radii	NEQU
logar sub-section keys			
`dr`	float	reference radial increment	DR
`nlog`	integer	number of logarithmic radii	NLOG
`rlog`	float	largest radius	RLOG
layer sub-section keys			
`layer`	list	layer thicknesses	H

Table 4.4: `rz2d` data keys

Key	Type	Description	TOUGH2 parameter
`deg` parameter			
`deg`	float	angle between y-axis and horizontal	DEG
NX, NY and NZ keys			
`del`	float	constant grid increment	DEL
`deli`	list	variable grid increments	DEL
`no`	integer	number of grid increments	DR
`ntype`	string	axis direction ('NX', 'NY' or 'NZ')	NTYPE

Table 4.5: `xyz` data keys

types are given in Table 4.4. Data keys for the `xyz` and `minc` data are given in Tables 4.5 and 4.6.

Example: The easiest way to understand how the `meshmaker` property works is to read some example input data into a `t2data` object and examine the result. The **MESHM** data for the standard TOUGH2 test problem 'rhbc' ('Production from a geothermal reservoir with hypersaline brine') is represented as a `t2data meshmaker` property as follows:

```
[('rz2d',[
 ('radii', {'radii': [5.0]}),
 ('equid', {'dr': 2.0, 'nequ': 1}),
 ('logar', {'rlog': 100.0, 'nlog': 50}),
 ('logar', {'rlog': 1000.0, 'nlog': 20}),
 ('equid', {'dr': 0.0, 'nequ': 1}),
 ('layer', {'layer': [500.0]})
 ])
]
```

`multi` property

A dictionary property selecting the equation of state (EOS) module used and setting associated parameters, corresponding to the **MULTI** input block in a TOUGH2 or AUTOUGH2

Key	Type	Description	TOUGH2 parameter
dual	string	treatment of global matrix-matrix flow	DUAL
num_continua	integer	number of interacting continua	J
spacing	string	direction of volume fraction specification	PAR
type	string	proximity function type	TYPE
vol	list	volume fractions	VOL
where	string	direction of volume fraction specification	WHERE

Table 4.6: minc data keys

Key	Type	Description	TOUGH2 parameter
eos	string	EOS name (AUTOUGH2 only)	NAMEOS
num_components	integer	number of components	NK
num_equations	integer	number of equations	NEQ
num_inc	integer	number of mass components in INCON data (TOUGH2 only)	NKIN
num_phases	integer	number of phases	NPH
num_secondary_parameters	integer	number of secondary parameters	NB

Table 4.7: multi property keys

data file. The individual keys of this property are given in Table 4.7.

noversion property

A Boolean property specifying whether to suppress printing of version and date information, corresponding to the **NOVER** input block in a TOUGH2 data file.

num_generators property

A read-only integer property returning the number of generators.

output_times property

A dictionary property specifying the times at which model output is required, corresponding to the **TIMES** input block in a TOUGH2 data file. The individual keys of this property are given in Table 4.8.

parameter property

A dictionary property specifying run-time parameters, corresponding to the **PARAM** input block in a TOUGH2 data file. The individual keys of this property are given in Table 4.9.

The option parameter (MOP array in TOUGH2) is an array of 24 integers, and has a 1-based index so that its indices are the same as those in the TOUGH2 documentation.

Key	Type	Description	TOUGH2 parameter
`max_timestep`	float	maximum time step	DELAF
`num_times_specified`	integer	number of times specified	ITI
`num_times`	integer	total number of times	ITE
`time`	list of float	times at which output is required	TIS
`time_increment`	float	time increment after specified times	TINTER

Table 4.8: `output_times` property keys

(In fact it is really zero-based, like all other Python arrays, but has an extra unused zeroth element).

`relative_permeability` property

A dictionary property specifying the relative permeability function used, corresponding to the first line of the **RPCAP** input block in the TOUGH2 data file. The individual keys of this property are given in Table 4.10.

`selection` property

A dictionary property representing selection parameters for the simulation (only used by some EOS modules, e.g. EOS7, EOS7R, EWASG), corresponding to the **SELEC** block in the TOUGH2 data file.

The dictionary contains two keys: 'integer' and 'float', the first of which accesses a list of the integer selection parameters (the first line of the **SELEC** block), while the second accesses a list of the float selection parameters (the remaining lines of the **SELEC** block).

`short_output` property

A dictionary property representing blocks, connections and generators for which short output is required, corresponding to the **SHORT** input block in an AUTOUGH2 data file.

The dictionary contains four keys: 'frequency', 'block', 'connection' and 'generator'. The last three of these access lists of blocks, connections and generators respectively for short output. (Note that each of these lists contains `t2block`, `t2connection` or `t2generator` objects, rather than names.) The 'frequency' key accesses the time step frequency (an integer) for which short output is required.

`simulator` property

A string property specifying the type of simulator, corresponding to the **SIMUL** input block in an AUTOUGH2 data file.

`solver` property

A dictionary property representing linear equation solver options, corresponding to the **SOLVR** input block in a TOUGH2 data file. The individual keys of this property are given in Table 4.11.

Key	Type	Description	TOUGH2 parameter
absolute_error	float	absolute convergence tolerance	RE2
be	float	enhanced vapour diffusion	BE
const_timestep	float	time step length	DELTEN
default_incons	list of float	default initial conditions	DEP
derivative_increment	float	numerical derivate increment factor	DFAC
diff0	float	diffusive vapour flux (AUTOUGH2 only)	DIFF0
gravity	float	gravitational acceleration	GF
max_duration	integer	maximum simulation duration (machine seconds)	MSEC
max_iterations	integer	maximum number of iterations per time step	NOITE
max_timesteps	integer	maximum number of time steps	MCYC
max_timestep	float	maximum time step size	DELTMX
newton_weight	float	Newton-Raphson weighting factor	WNR
option	array(24) of integer	simulation options	MOP
pivot	float	pivoting parameter for linear solver	U
print_block	string	block name for short printout	ELST
print_interval	integer	time step interval for printing	MCYPR
print_level	integer	amount of printout	KDATA
relative_error	float	relative convergence tolerance	RE1
scale	float	grid scale factor	SCALE
texp	float	binary diffusion temperature parameter	TEXP
timestep_reduction	float	time step reduction factor	REDLT
timestep	list of float	specified time step sizes	DLT
tstart	float	start time (seconds)	TSTART
tstop	float	stop time	TIMAX
upstream_weight	float	upstream weighting factor	WUP

Table 4.9: `parameter` property keys

Key	Type	Description	TOUGH2 parameter
`parameters`	array (7) of float	function parameters	RP
`type`	integer	type of relative permeability function	IRP

Table 4.10: `relative_permeability` property keys

Key	Type	Description	TOUGH2 parameter
`closure`	float	convergence criterion	CLOSUR
`relative_max_iterations`	float	relative max. number of iterations	RITMAX
`type`	integer	solver type	MATSLV
`o_precond`	string	O-preconditioning type	OPROCS
`z_precond`	string	Z-preconditioning type	ZPROCS

Table 4.11: `solver` property keys

`start` property

A Boolean property specifying whether the flexible start option is used, corresponding to the **START** input block in a TOUGH2 data file.

`title` property

A string property containing the simulation title, corresponding to the **TITLE** input block in a TOUGH2 data file.

4.2.2 Methods

The main methods of a `t2data` object are listed in Table 4.12. Details of these methods are given below.

`add_generator(`*generator*`)`

Adds a generator to the data file object.

Parameters:

- **generator**: `t2generator`
  Generator to be added to the data file object.

`convert_to_TOUGH2(`*warn*`=True, `*MP*`=False)`

Converts an AUTOUGH2 data file for use with TOUGH2 (or compatible simulators such as TOUGH2_MP). Various parameter options are altered to try to make the TOUGH2 simulation give similar results to the original AUTOUGH2 simulation. This particularly affects AUTOUGH2 options related to backward compatibility with MULKOM. In particular, if these are used then the heat conductivities in the ROCKS block have to be altered to give the same results. Data blocks specific to AUTOUGH2 (e.g. SIMULATOR, LINEQ, and SHORT) are removed, and AUTOUGH2-specific generator types are converted to their TOUGH2 equivalents if possible, or otherwise deleted.

Method	Type	Description
add_generator	–	adds a generator
clear_generators	–	deletes all generators
convert_to_TOUGH2	–	converts from AUTOUGH2 input to TOUGH2
delete_generator	–	deletes a generator
delete_orphan_generators	–	deletes orphaned generators
generator_index	integer	returns index of generator with specified name and block name
read	t2data	reads data file from disk
run	–	runs a TOUGH2 simulation
specific_generation	np.array	generation per unit volume in each block
total_generation	np.array	total generation in each block
transfer_from	–	transfers data from another t2data object
write	–	writes to data file on disk

Table 4.12: Methods of a t2data object

Parameters:

- **warn**: Boolean
  If True, warnings will be printed regarding AUTOUGH2 options used in the original data file which are not supported in TOUGH2.
- **MP**: Boolean
  if True, converts to a TOUGH2_MP data file, which treats some of the parameters differently (e.g. MOP(20)).

clear_generators()

Deletes all generators from the data file object.

delete_generator(*blocksourcenames*)

Deletes the generator with the specified block and generator (source) name, if it exists.

Parameters:

- **blocksourcenames**: tuple
  Tuple of block name and generator name (both strings) of the generator to be deleted.

delete_orphan_generators()

Deletes all generators with block names that are not in the grid.

generator_index(*blocksourcenames*)

Returns the index (in the generatorlist list) of the generator with the specified block and generator name.

Parameters:

- **blocksourcenames**: tuple
  Tuple of block name and generator name (both strings) of the generator.

`read(`*`filename, meshfilename=''`*`)`

Reads a `t2data` object from a TOUGH2 data file on disk. The mesh data may optionally be read from auxiliary files, if it is not present in the main data file. (Note that if the main data file does contain mesh information (the 'ELEME' and 'CONNE' sections), any auxiliary mesh files will not be read.)

Parameters:

- **filename**: string
  Name of the TOUGH2 data file to be read.

- **meshfilename**: string or tuple
  Name of separate mesh file(s) to read, containing element and connection data. If empty, then mesh data will be read from the main data file. If a non-empty string is given, this is interpreted as the name of a formatted text file containing 'ELEME' and 'CONNE' data sections (as in the 'MESH' files created by TOUGH2 and TOUGH2_MP).

Note that it is possible to create a `t2data` object and read its contents in from disk files in one step, e.g.: `dat = t2data(filename,meshfilename)`.

`run(`*`save_filename=''`*`, `*`incon_filename=''`*`, `*`simulator=`*`'AUTOUGH2_2',`
`silent``=False)`

Runs an AUTOUGH2 or TOUGH2 (but not TOUGH2_MP) simulation using the data file corresponding to a `t2data` object. The contents of the `t2data` object must first have been written to disk using the `write` function. If the file names for the save file or initial conditions file are not specified, they are constructed by changing the file extension of the data file name. The name of the TOUGH2 executable can be specified.

For running TOUGH2 (rather than AUTOUGH2), the name of the TOUGH2 executable must be specified via the `simulator` parameter. However, the `save_filename` and `incon_filename` parameters do not need to be specified. Initial conditions will be read from the file INCON and final results written to SAVE. The listing file name will be the same as the data file name, but with the extension changed to *.listing.

Parameters:

- **save_filename**: string
  Name of the save file to be written to disk during the simulation. Default is *base.save* where the AUTOUGH2 data file name is *base.dat*.

- **incon_filename**: string
  Name of the initial conditions file for the simulation. Default is *base.incon* where the AUTOUGH2 data file name is *base.dat*.

- **simulator**: string
  Name of the AUTOUGH2 or TOUGH2 executable. Default is 'AUTOUGH2_2'.

- **silent**: Boolean
  Set to `True` to suppress output to the display while running (default is `False`).

`specific_generation(`*`type`*`='MASS', `*`name`*`='')`

Returns an `np.array` containing the total specific generation rate in each block (i.e. generation rate per unit volume) for the specified generator type and name.

Parameters:

- **type**: string
  Generation type ('HEAT', 'MASS' etc.)- default is 'MASS'.
- **name**: string
  Regular expression to match generator names (e.g. 'SP...' (or '^SP') will match all generators with names beginning with 'SP'.)

`transfer_from(`*`source`*`, `*`sourcegeo`*`, `*`geo`*`, `*`top_generator`*`=[], `*`bottom_generator`*`=[], `*`sourceinconfilename`*`='', `*`inconfilename`*`='', `*`rename_generators`*`=False, `*`preserve_generation_totals`*`=False)`

Transfers data from another `t2data` object, and its associated `mulgrid` object. Parameters, rock types and rock type assignments, and optionally initial conditions files are transferred. In general the data for a given block in the geometry is found by identifying the nearest block in the source geometry and transferring data from that block. There are, however, exceptions, such as for generators that need to remain on the surface or bottom of the model. The `top_generator` and `bottom_generator` lists specify the 'layer' part of the generator name for generators that should remain on the top or bottom of the model, respectively.

For generator types in which the `gx` and `rate` properties represent generation rates (as opposed to other types for which these properties are used to represent other things, e.g. productivity index for wells on deliverability), the values of `gx` and `rate` are scaled to account for the different volume of the block the generator has been mapped into. If `preserve_generation_totals` is `True`, and a generator with generation rate G is mapped into n blocks with volumes $V_1, V_2, \ldots, V_n$, then the generation rate for the new generator in block i will be $GV_i / \sum_{k=1}^{n} V_k$. This should preserve the total generation rate over the model. (For generator types matching the `bottom_generator` or `top_generator` specifications, the column area instead of the block volume is used to determine the appropriate scaling.) Note that of the columns a top or bottom generator is mapped into, only those with centres inside the source geometry are included in the scaling calculations. The generator types for which this scaling is carried out are: ' AIR', 'COM1', 'COM2', 'COM3', 'COM4', 'COM5', 'HEAT', 'MASS', 'NACL', 'TRAC' and ' VOL'.

If both `sourceinconfilename` and `inconfilename` are specified, a new initial conditions file with filename `inconfilename` is written to disk, with initial conditions transferred from the file `sourceinconfilename`.

Parameters:

- **source**: `t2data`
  The `t2data` object to transfer data from.
- **sourcegeo**: `mulgrid`
  The `mulgrid` object corresponding to `source`.
- **geo**: `mulgrid`
  The `mulgrid` object corresponding to the destination `t2data` object.

- **top_generator**: list
  A list of generator 'layer' identifier strings for generators that need to be kept at the top of the model (e.g. rain generators).

- **bottom_generator**: list
  A list of generator 'layer' identifier strings for generators that need to be kept at the bottom of the model (e.g. basement heat and mass inputs).

- **sourceinconfilename**: string
  Name of the (optional) initial conditions file to transfer initial conditions data from (corresponding to `source`).

- **inconfilename**: string
  Name of the (optional) initial conditions file to write, corresponding to the destination `t2data` object.

- **rename_generators**: Boolean
  If `False`, generators other than those at the top and bottom of the model retain their original names. Otherwise, they will be renamed according to their column names in the new grid.

- **preserve_generation_totals**: Boolean
  If `False` (the default), the transfer of generators will attempt to preserve the distribution of specific generation of the original model; otherwise, it will attempt to preserve the total generation over the model.

`total_generation(`*`type`*`='MASS', `*`name`*`='')`

Returns an `np.array` containing the total generation rate in each block for the specified generator type and name.

Parameters:

- **type**: string
  Generation type ('HEAT', 'MASS' etc.)- default is 'MASS'.

- **name**: string
  Regular expression to match generator names (e.g. 'SP...' (or '^SP') will match all generators with names beginning with 'SP'.)

`write(`*`filename`*`='', `*`meshfilename`*`='', `*`extra_precision`*`=None, `*`echo_extra_precision`*`=None)`

Writes a `t2data` object to a TOUGH2 data file on disk. If the `meshfilename` parameter is used, mesh information can be written to auxiliary mesh files.

Parameters:

- **filename**: string
  Name of the TOUGH2 data file to be written. If no file name is specified, the object's own `filename` property is used.

- **meshfilename**: string or tuple
  Name of auxiliary mesh file(s) to be written. If this is empty (the default), the object's own `meshfilename` property is used. Otherwise, if a single (non-empty) string is given, this in interpreted as the name of a file to write formatted mesh information to (as in

the 'MESH' files produced by TOUGH2 and TOUGH2_MP). If a tuple of two strings is given, this in interpreted as the names of two binary files (as in the 'MESHA' and 'MESHB' files produced by TOUGH2_MP).

- **extra_precision**: list or Boolean
  Controls whether to write extra precision data to auxiliary file (AUTOUGH2 only). If set to `True`, then all possible sections will be written to the extra precision file. Currently the possible extra-precision sections are the ROCKS, RPCAP and GENER sections. If set to `False` or [], then no extra-precision data will be written. If set to a list of section names (e.g. ['RPCAP', 'GENER']), then only those sections will be written in extra precision. If set to `None` (the default), then the value of the data object's `extra_precision` property is used. Otherwise, the value of this property is overwritten by the value specified here.

- **extra_precision**: Boolean or None
  Controls whether to echo all extra-precision data sections to the main data file (AUTOUGH2 only). If `None`, the value of the data object's `echo_extra_precision` property is used. Otherwise, the value of this property is overwritten by the value specified here.

4.3 `t2generator` objects

A `t2generator` object represents a generator in a TOUGH2 simulation (i.e. an item in the generation table). The properties of a `t2generator` object are given in Table 4.13. These correspond closely to the parameters specified in the TOUGH2 **GENER** input block. A `t2generator` object has no methods.

4.4 Example

The following piece of Python script opens a MULgraph geometry file and TOUGH2 data file, changes some TOUGH2 run-time parameters and assigns heat generators to the blocks in the bottom layer inside a defined area, with the specified total heat divided uniformly amongst the generators.

```
geo=mulgrid('gmodel.dat')
dat=t2data('model.dat')

dat.parameter['max_timesteps']=300
dat.parameter['print_interval']=dat.parameter['max_timesteps']/10
dat.parameter['option'][16]=5 # time step control

dat.clear_generators()
totalheat=10.e6
layer=geo.layerlist[-1]  # bottom layer
cols=[col for col in geo.columnlist if 10.e3<=col.centre[0]<=20.e3]
totalarea=sum([col.area for col in cols])
q=totalheat/totalarea

for col in cols:
    blockname=geo.block_name(layer.name,col.name)
```

Property	Type	Description	TOUGH2 parameter
block	string	name of block containing the generator	SL, NS
enthalpy	list of float	generation enthalpies (|ltab| >1, itab<>'')	F1
ex	float	enthalpy for injection	EX
gx	float	generation rate (or productivity index for deliverability)	GX
hg	float	layer thickness for deliverability	HG
fg	float	separator pressure/ injectivity etc.	FG
itab	string	blank unless table of specific enthalpies specified	ITAB
ltab	integer	number of generation times (or open layers for deliverability)	LTAB
nadd	integer	successive block increment	NADD
nads	integer	successive generator increment	NADS
name	string	generator name	EL, NE
nseq	integer	number of additional generators	NSEQ
rate	list of float	generation rates (|ltab| >1)	F2
time	list of float	generation times (|ltab| >1)	F1
type	string	generator type (default 'MASS')	TYPE

Table 4.13: Properties of a `t2generator` object

```
    gen=t2generator(name=' q'+col.name,block=blockname,type='HEAT',gx=q*col.area)
    dat.add_generator(gen)

dat.write()
```

Chapter 5

TOUGH2 initial conditions

5.1 Introduction

The `t2incons` library in PyTOUGH contains classes and routines for reading, editing and writing TOUGH2 initial conditions and files. It can be imported using the command:

```
from t2incons import *
```

The initial conditions files used by TOUGH2 and AUTOUGH2 have the same format.

5.2 `t2incon` objects

The `t2incons` library defines a `t2incon` class, used for representing TOUGH2 initial conditions.

Example:

```
inc=t2incon()
```

creates an empty `t2incon` object called `inc`.

```
inc=t2incon(filename)
```

creates a `t2incon` object called `inc` and reads its contents from file `filename`.

5.2.1 Properties

The main properties of a `t2incon` object are listed in Table 5.1. Once a set of initial conditions is loaded into a `t2incon` object, conditions for individual blocks can be accessed by block name or index. For example, for a `t2incon` object `inc`, the initial conditions in block *blockname* are given simply by `inc[`*`blockname`*`]`. This returns a `t2blockincon` object (see section 5.3). Similarly, `inc[`*`i`*`]` returns the initial conditions at the block with (zero-based) index *i*.

Each column in the initial conditions file can be accessed by adding an integer (zero-based) index after the `t2blockincon` object, so for example:

```
t=inc['aa 20'][1]
```

Property	Type	Description
blocklist	list	ordered list of block names in the initial conditions file
num_blocks	integer	number of blocks at which conditions are specified
num_variables	integer	number of thermodynamic variables specified at each block
porosity	np.array	array of porosity values specified at each block
timing	dictionary	additional timing information for restarting
variable	np.array	two-dimensional array of thermodynamic variable values at each block

Table 5.1: Properties of a t2incon object

assigns the variable t the value of the second primary thermodynamic variable (index 1) in block 'AA 20'. Initial conditions can be edited in a similar way, for example:

```
inc['aa 20'][0]=p
```

assigns the value of p to the first primary variable (usually pressure) in block 'AA 20'. For convenience, initial conditions for a given block can also be specified as a simple list or tuple of values, for example:

```
inc['ab 25']=(101.3e5,25.0)
```

sets the initial conditions at block 'ab 25' to the specified values. This will work even if no initial conditions have been previously specified for the given block.

An np.array of the values of the variables at all blocks can be found from the variable property. For example:

```
inc.variable[:,2]
```

returns an np.array of the third variable (index 2) in each block. The variable property can also be set to a given array. Note, however, that the whole array must be set, not just part of it. For example, adding an offset P0 to all pressures (variable 0) in the initial conditions could be done by:

```
v=inc.variable
v[:,0]+=P0
inc.variable=v
```

The porosity property may be set to assign values of porosity to all blocks. The assigned value may be an np.array with a value for each block, or a scalar float (in which case the same value is assigned to all blocks), or None which assigns the value in each block to None.

The timing property of a t2incon object contains the optional timing information at the end of the file. This is a dictionary property with keys 'kcyc', 'iter', 'nm', 'tstart' and 'sumtim', corresponding to the values stored on this line.

5.2.2 Methods

The main methods of a t2incon object are listed in Table 5.2. Details of these methods are given below.

Method	Type	Description
`add_incon`	–	adds a set of initial conditions for one block
`delete_incon`	–	deletes the initial conditions for one block
`empty`	–	deletes all initial conditions from the object
`insert_incon`	–	inserts initial conditions for one block at a specified index
`read`	–	reads initial conditions from file
`transfer_from`	–	transfers initial conditions from one grid to another
`write`	–	writes initial conditions to file

Table 5.2: Methods of a `t2incon` object

`add_incon(`*incon*`)`

Adds a set of initial conditions for a single block.

Parameters:

- **incon**: `t2blockincon`
  Initial conditions for the block.

`delete_incon(`*blockname*`)`

Deletes a set of initial conditions for a single block.

Parameters:

- **blockname**: string
  Name of the block at which initial conditions are to be deleted.

`empty()`

Deletes initial conditions for all blocks.

`insert_incon(`*index*`,`*incon*`)`

Inserts a set of initial conditions for a single block at the specified index.

Parameters:

- **index**: integer
  Index (zero-based) at which to insert the initial conditions.
- **incon**: `t2blockincon`
  Initial conditions for the block.

`read(`*filename*`)`

Reads initial conditions from file.

Parameters:

- **filename**: string
  Name of the initial conditions file to be read.

`transfer_from(`*`sourceinc`*`, `*`sourcegeo`*`, `*`geo`*`, `*`mapping`*`={}, `*`colmapping`*`={})`

Transfers initial conditions from another `t2incon` object `sourceinc`, using the two corresponding `mulgrid` geometry objects `sourcegeo` and `geo`, and optionally the block and column mappings between the two grids (which are created if not specified).

Parameters:

- **sourceinc**: `t2incon`
  Source initial conditions object.
- **sourcegeo**: `mulgrid`
  Geometry object corresponding to the source initial conditions.
- **geo**: `mulgrid`
  Geometry object for the grid to be transferred to.
- **mapping**: dictionary
  Dictionary mapping block names from `geo` to `sourcegeo`.
- **colmapping**: dictionary
  Dictionary mapping column names from `geo` to `sourcegeo`.

`write(`*`filename`*`, `*`reset`*`=True)`

Writes initial conditions to file.

Parameters:

- **filename**: string
  Name of the initial conditions file to be written.
- **reset**: Boolean
  Set to `False` if timing information is not to be reset- e.g. if restarting a transient simulation.

5.3 `t2blockincon` objects

A `t2blockincon` object represents the initial conditions for a particular block. A `t2blockincon` object has no methods, and three properties: `variable`, `block` and `porosity`. These are respectively list, string and float properties which hold the variable values, the block name and the (optional) porosity value for each block. If no value is given for porosity, its value is `None`.

The `variable` property of a `t2blockincon` can be more easily accessed simply by adding the required (zero-based) variable index after the object. For example, for a `t2blockincon` object `b`, the value of the second variable is given simply by `b[1]`.

5.4 Reading *.save files and converting to initial conditions

TOUGH2 writes a *.save file at the end of the simulation, which has a format almost the same as that of an initial conditions file and can be used to start a subsequent run. A *.save file generally has some extra timing information at the end which can be used to restart a

simulation at a particular time. However, in many cases, e.g when running natural state simulations, we want to restart at the original start time and this timing information must be discarded.

PyTOUGH will read a *.save file into a `t2incon` object. This can then be written to file, providing a simple way to convert *.save files into *.incon files. By default, the timing information is discarded when writing (it can be retained by setting the `reset` parameter of the `write` method to `False`). For example:

```
t2incon('model1.save').write('model2.incon')
```

will read the save file `'model1.save'`, convert it to initial conditions, and write it to the initial conditions file `'model2.incon'`.

5.5 Example

The following piece of Python script reads in a *.save file and prints out a table of block names and temperatures for the first 10 blocks. It then adds an extra variable to each initial condition and gives it a constant value (giving a new column in the initial conditions file), and finally writes out the edited initial conditions to a new file.

Adding a new variable to each initial condition can be useful when e.g. changing from one TOUGH2 equation of state (EOS) module to another, as different EOS modules may have different numbers of primary thermodynamic variables.

```
from t2incons import *
inc=t2incon('model1.save')
for blk in inc[0:10]:
    print 'Block %5s: temperature = %5.1f' % (blk.block,blk[1])
patm=101.3e3
for blk in inc: blk.variable.append(patm)
inc.write('model2.incon')
```

Chapter 6

TOUGH2 listing files

6.1 Introduction

The `t2listing` library in PyTOUGH contains classes and routines for reading TOUGH2 listing files. It can be imported using the command:

```
from t2listing import *
```

Listing files produced by AUTOUGH2, TOUGH2, TOUGH2_MP and TOUGH+ have different formats but are all supported.

6.2 `t2listing` objects

The `t2listing` library defines a `t2listing` class, used for representing TOUGH2 listing files.

Example:

```
lst=t2listing()
```

creates an empty `t2listing` object called `lst`.

```
lst=t2listing(filename)
```

creates a `t2listing` object called `lst` and reads its contents from file `filename`.

6.2.1 Properties

The main properties of a `t2listing` object are listed in Table 6.1.

Element, connection and generation tables

There are three main 'table' properties, corresponding to the **element**, **connection** and **generation** tables in the listing file. These are all of type `listingtable` (see section 6.3) and provide access to the simulation results. Not all of these tables will necessarily be present- this depends on the settings in the data file which produced the results. For TOUGH2 results, a fourth **primary** table may also be present, containing primary variables

and their changes, if the KDATA parameter is set to 3. TOUGH+ results can also contain additional element tables containing other calculated quantities; these are named **element1**, **element2** etc. A list of names of all available tables is given by the `table_names` property.

For example, for a `t2listing` object `lst`, `lst.element['AR210']['Temperature']` gives the temperature at block 'AR210', at the current time. Blocks can also be identified by index rather than name, so that `lst.element[120]['Pressure']` gives the pressure at the block with (zero-based) index 120.

These tables can also be accessed to give all results for a given block, or for a given column in the table. For example, `lst.element['AR210']` returns a dictionary containing all results at block 'AR210', referred to by the name of each table column. `lst.element['Temperature']` returns an `np.array` containing the temperatures at all blocks in the model. (Hence, `lst.element['Pressure'][120]` gives the same result as `lst.element[120]['Pressure']`.)

The connection and generation tables work very similarly to the element table, except that connections are referred to by tuples of block names (rather than single block names), and generators are referred to by tuples of block names and generator names. So for example, the mass flow rate between blocks 'AB300' and 'AC300' might be given by `lst.connection['AB300','AC300']['Mass flow']`.

The names of the columns for each table are read directly from the listing file, and will depend on the TOUGH2 equation of state (EOS) being used.

Skipping tables

The default behaviour is for a `t2listing` object to read all tables present in the listing file. However, it is possible to skip the reading of specified tables if required. This can be useful for speeding up reading of large listing files where not all tables are required. For example, sometimes the connection data are not required, but for large models the connection table is often much bigger than the others, so skipping it can make reading significantly faster. Data in skipped tables are not available either via their corresponding properties or via the `history()` method.

To skip tables, specify their table names (`element`, `connection` etc.) in the optional `skip_tables` parameter when creating the `t2listing` object. (By default, this parameter is an empty list.) For example, to read a listing file with name 'output.listing' into the object `lst` and skip reading the connection and generation tables:

```
lst = t2listing('output.listing',skip_tables=['connection','generation'])
```

Full and short output

AUTOUGH2 allows the use of 'short' output, in which a specified selection of block, connection or generator properties are printed at time steps between normal full output. A `t2listing` object will read short output results, if they are present, when producing time histories using the `history()` method. However it is not possible to navigate to short output results or access them via the `t2listing` table properties above.

TOUGH2, TOUGH2_MP and TOUGH+ do not support short output.

Navigating in time using `time`, `index` and `step`

The `time` property returns the time (in seconds) corresponding to the current set of results. It is also possible to set the `time` property to navigate to a specific set of full results. For

example, `lst.time=1.e9` navigates to the first set of full results for which `lst.time` is at least 10^9s.

The `index` property gives the index of the current set of results, and can take any value between 0 and `num_fulltimes`-1. The value of `index` can also be set to change to a different set of results in the listing file (e.g. `lst.index=12`). It can be incremented and decremented like any other Python integer variable, e.g. `lst.index+=1` or `lst.index-=2` to go to the next set of results, or the second to last set respectively.

The `step` property gives the time step number for the current set of results. This is the number of time steps carried out in the simulation up to the current set of results (recall that results are not necessarily written to the listing file at every time step). Again, its value can be set to navigate through the results, e.g. `lst.step=100` navigates to the first set of full results at which the time step number is at least 100.

The `times` property returns an `np.array` of all times at which results (including short output) are given in the listing file. It has length equal to `num_times`. The `fulltimes` property returns an `np.array` of times at which full results are given (not including short output), and has length equal to `num_fulltimes`.

A `t2listing` object also has methods (as well as properties) for navigating through time (see section 6.2.2).

Listing diagnostics

`t2listing` objects have two properties that provide diagnostics on the results of the TOUGH2 run.

The `convergence` property is a dictionary of the maximum absolute differences in the element table between the second to last and last sets of results in the listing file. This can be used to check convergence of steady-state simulations. For example:

```
lst.convergence['Temperature']
```

gives the largest absolute temperature change between the second to last and last sets of results.

The `reductions` property is a list of tuples of time step indices at which the time step size was reduced during the simulation, and the block name at which the maximum residual occurred prior to each reduction. This gives an indication of problematic times and blocks which caused time step reductions.

6.2.2 Methods

The main methods of a `t2listing` object are listed in Table 6.2. Details of these methods are given below.

`add_side_recharge(`*`geo`*`, `*`dat`*`)`

Adds side recharge generators to a `t2data` object `dat` for a production run, calculated according to the final results in the listing. These generators represent side inflows due to pressure changes in the blocks on the model's horizontal boundaries. Recharge generators are given the names of their blocks- any existing generators with the same names will be overwritten.

Parameters:

Property	Type	Description
connection	listingtable	connection table for current set of results
convergence	dictionary	maximum differences in element table between second to last and last sets of results
element	listingtable	element table for current set of results
element1 etc.	listingtable	additional element table for current set of results (TOUGH+ only)
filename	string	name of listing file on disk
fullsteps	np.array	array of time step numbers (integer) for full results
fulltimes	np.array	array of times (float) for full results
generation	listingtable	generation table for current set of results
index	integer	index of current set of results
num_fulltimes	integer	number of sets of full results
num_times	integer	number of sets of all results (full and short)
primary	listingtable	primary variable table for current set of results (TOUGH2 only)
reductions	list	time step indices at which time step was reduced during the simulation
short_types	list of string	types of short output present
step	integer	time step number of current set of results
steps	np.array	array of time step numbers (integer) for all results (full and short)
table_names	list	names of available tables
time	float	time of current set of results
times	np.array	array of times (float) for all results (full and short)
title	string	simulation title

Table 6.1: Properties of a t2listing object

Method	Type	Description
add_side_recharge	–	adds side recharge generators to a t2data object
first	–	navigates to the first set of full results
get_difference	dictionary	maximum differences in element table between two sets of results
history	list or tuple	time history for a selection of locations and table columns
last	–	navigates to the last set of full results
next	Boolean	navigates to the next set of full results
prev	Boolean	navigates to the previous set of full results
write_vtk	–	writes results to VTK file

Table 6.2: Methods of a t2listing object

- **geo**: `mulgrid`
  Geometry object associated with the listing.

- **dat**: `t2data`
  TOUGH2 data object for the side recharge generators to be added to.

`first()`

Navigates to the first set of full results in the listing file.

`get_difference(`*`indexa`*`=None, `*`indexb`*`=None)`

Returns dictionary of maximum differences, and locations of difference, of all element table properties between two sets of results.

Parameters:

- **indexa**, **indexb**: integer or `None`
  Indices of results between which the maximum differences are to be calculated. If both indexa and indexb are provided, the result is the difference between these two result indices. If only one index is given, the result is the difference between the given index and the one before that. If neither are given, the result is the difference between the last and penultimate sets of results.

`history(`*`selection`*`, `*`short`*`=True)`

Returns a list of time histories (as `np.arrays`) for specified locations and table columns in the element, connection or generation tables. For each selection, a tuple of two `np.arrays` is returned, one each for times and values. Short output (AUTOUGH2 only) can be omitted from the history results by setting the `short` parameter to `False`.

Parameters:

- **selection**: list of tuples
  Selection of listing tables, locations (or indices) and table columns to produce histories for. Each tuple contains three elements: the listing **table type** ('e', 'c', 'p' or 'g' for element, connection, primary or generation table respectively), the **block/ connection/ generator name** and the **table column name**. (If only a single tuple is given instead of a list of tuples, just the single tuple of times and values for that selection is returned.) For history of additional element tables in TOUGH+ results, use 'e1', 'e2' etc. instead of 'e'. Note that, as for listing tables, connection and generator names (or 'keys') are specified as two-element tuples (see Table 6.3).

- **short**: Boolean
  Whether short output (AUTOUGH2 only) is to be included in the history results-default is `True`.

Example:

```
[(tt,temp),(tq,q),(tg,g)] = lst.history([('e','AR210','Temperature'),
('c',('AB300','AC300'),'Mass flow'),('g',('BR110','SO  1'),'Generation rate')])
```

returns a list of three tuples of `np.arrays`, `(tt,temp)`, `(tq,q)` and `(tg,g)`, giving the times and values of temperature at block 'AR210', mass flow at the connection between blocks 'AB300' and 'AC300', and generation rate in the generator 'SO 1' in block 'BR110' respectively.

`last()`

Navigates to the last set of full results in the listing file.

`next()`

Navigates to the next set of full results in the listing file. Returns `False` if already at the last set of results (and `True` otherwise).

`prev()`

Navigates to the previous set of full results in the listing file. Returns `False` if already at the first set of results (and `True` otherwise).

`write_vtk(`*`geo`*`, `*`filename`*`, `*`grid`*`=None, `*`indices`*`=None, `*`flows`*`=False, `*`wells`*`=False, `*`start_time`*`=0, `*`time_unit`*`='s')`

Writes a `t2listing` object to a set of VTK files on disk, for visualisation with VTK, Paraview, Mayavi etc. The results in the listing object are written as an 'unstructured grid' VTK object with data arrays defined on cells. The data arrays written correspond to the variables given in the columns of the element table of the `t2listing` object. (For TOUGH+ results, variables from the additional element tables are also included.) In addition, data arrays from an associated `mulgrid` and (optionally) `t2grid` objects can be included.

If `flows` is `True` (and a `grid` is specified and the listing contains connection data), approximate block-average flux vectors at the centre of each block are also written, for all variables in the connection table with names ending in 'flow'.

One *.vtu file is produced for each time step in the `t2listing` object at which full results are present, and a *.pvd file is also written. This is usually the file that should actually be opened in Paraview or other software as it contains time information associated with each *.vtu file.

Optionally, only a subset of the time indices present in the `t2listing` can be written, according to the `indices` parameter. A start time and time unit for the output can optionally be specified.

Parameters:

- **geo**: `mulgrid`
  The `mulgrid` geometry object associated with the results. For flexibility, this geometry need not be fully compatible with the results- for example, it may contain only a subset of the blocks for which results are present, or the blocks may be in a different order. However, if it is not fully compatible, the writing process will be slower, and flux vectors will not be written (even if `flows` is set to `True`).

- **filename**: string
  Name of the *.pvd file to be written. Names of the individual *.vtu files for each time step are similar but with a time index appended and the file extension changed.

- **grid**: `t2grid`
  Name of optional `t2grid` object associated with the results.

- **indices**: list or tuple
  Optional specification of time indices to include in the output. If set to `None` (the default), all time indices will be included.

- **flows**: Boolean
  Set to `True` if approximate block-centred flux vectors are to be calculated and written, for visualising flows. Default is `False`. **Note**: flow vectors can only be calculated if a **grid** is specified.

- **wells**: Boolean
  Set to `True` if a separate VTK file for the wells in the `mulgrid` object is to be written. Default is `False`.

- **start_time**: float
  Optional start time of the simulation, i.e. time associated with the first set of results. Default is zero.

- **time_unit**: string
  Optional time unit for the output. TOUGH2 results are given at times in seconds, but this option allows them to be converted to other units. Options are: 's', 'h', 'd' and 'y', for seconds, hours, days and years respectively. Default is 's'.

6.3 `listingtable` objects

A `listingtable` object represents a table of results in a TOUGH2 listing file (whether it is an element, connection or generation table). The column headings of the table are taken directly from the corresponding table in the listing file. The rows of the table may be accessed either by (zero-based) index, or by the 'key' for the table row, which depends on the table type (see Table 6.3).

Table type	Key
`element`	block name
`connection`	(block name 1, block name 2)
`generation`	(block name, generator name)

Table 6.3: Keys for different listing table types

Hence, the value in the element table for a given block and column can be accessed by `lst.element[`*`blockname`*`][`*`columnname`*`]`, or by `lst.element[`*`blockindex`*`][`*`columnname`*`]` (for a `t2listing` object `lst`). Note that for connection and generation tables, the keys are tuples of two strings. For connection tables, the order of these two strings (the block names) is not important; if the listing file contains results for (block1, block2), then results for (block2, block1) can be accessed via the corresponding `listingtable` object (though the results will have the opposite sign to those in the file, as they will represent flows in the opposite direction).

The values for an entire row or column of the table can also be accessed, for example `lst.element[`*`blockname`*`]` gives the row in the table for a specified block, with the values arranged in a dictionary which can be accessed using the column names of the table (e.g. `lst.element['AR231']['Temperature']`). This dictionary for each row also contains an additional `'key'` item which returns the key for that row. Conversely, `lst.element[`*`columnname`*`]` gives the column in the table for a specified column name, with the values returned in an `np.array` (one value for each block in the grid, for an element table).

Property	Type	Description
column_name	list	column headings
num_columns	integer	number of columns
num_keys	integer	number of keys per row
num_rows	integer	number of rows
row_name	list	keys for each row

Table 6.4: Properties of a `listingtable` object

The properties of a `listingtable` object are given in Table 6.4. The entire list of key values for a `listingtable` may be accessed via the `row_name` property, which contains the key value for each row. The column headings of the table can similarly be accessed via the `column_name` list property. The `num_rows` and `num_columns` properties of a `listingtable` object return the numbers of rows and columns respectively. The `num_keys` property just returns the number of keys used to identify each row- generally 1 for an element table and 2 for connection and generation tables.

`listingtable` objects have one method as described below.

`rows_matching(`*`pattern`*`, `*`index`*`=0, `*`match_any`*`=False)`

Returns a list of rows in the table with keys matching the specified regular expression string, `pattern`.

For tables with multiple keys, `pattern` can be a list or tuple of regular expressions. If a single string pattern is given for a multiple-key table, the pattern is matched on the indexth key (and any value of the other key- unless the `match_any` option is used; see below).

If `match_any` is set to `True`, rows are returned with keys matching any of the specified patterns (instead of all of them). If this option is used in conjunction with a single string pattern, the specified pattern is applied to all keys.

Parameters:

- **pattern**: string, list or tuple
  Regular expression string specifying the pattern to match. For multiple-key tables, this can be a list or tuple of regular expression strings.
- **index**: integer
  Index of the key to which the pattern is to be applied, for multiple-key tables and when `pattern` is specified as a single string.
- **match_any**: Boolean
  If `False`, return only rows with keys matching *all* of their corresponding patterns. If `True`, return rows with keys matching *any* of the specified patterns- and if a single string pattern is given, apply this to all keys.

6.4 `t2historyfile` objects

In addition to the main listing file, TOUGH2 can optionally produce extra files containing time history data from selected blocks, connections or generators, named `FOFT`, `COFT` and `GOFT` files respectively. TOUGH+ can optionally name these files `Elem_Time_Series`, `Conx_Time_Series` and `SS_Time_Series` instead. (AUTOUGH2 does not produce separate

history files, but can instead produce 'short output' at selected blocks, connections or generators within the listing file itself.)

The `t2listing` module contains a `t2historyfile` class for reading and manipulating these history files. History files produced by TOUGH2, TOUGH2_MP and TOUGH+ are supported, although they all have different formats. The same class is used for FOFT, COFT and GOFT files. A history file of any of these types can be opened using a command such as:

```
hist = t2historyfile(filename)
```

where *`filename`* is the name of the file. It may contain wildcards (*) so that several files matching a pattern are read in to the same object. This is useful for reading output from TOUGH2_MP, which creates separate history files for each processor used in the calculation (e.g. `FOFT_P.000`, `FOFT_P.001`, etc.). It is assumed that all files opened are however of the same type (FOFT, COFT or GOFT).

Once a history file has been read in, history results for a particular key (i.e. block, connection or generator) can be extracted. For TOUGH2_MP, the keys are the block names for FOFT files, tuples of block names for COFT files, and tuples of block names and source names for GOFT files. For example:

```
foft = t2historyfile('FOFT_P.*')
blockname = 'fmq20'
results = foft[blockname]
```

This will return a dictionary containing an `np.array` for each column in the file, indexed by the column name. For example the temperature history at this block would be given by:

```
temp = foft[blockname]['TEMPERATURE']
```

Results at a particular time can also be found:

```
time = 3.156e7
result = foft[blockname,time]
```

Again, this will return a dictionary with one item for each column, but in this case each item is just a single floating point number instead of an array.

For **TOUGH2** rather than TOUGH2_MP, the keys are integer indices of blocks, connections or generators, rather than names or tuples of names. Similarly, the column names are just integers. This is because the key names and column names are not given in TOUGH2 history files. Aside from these differences, they can be used in the same way as TOUGH2_MP history files, for example:

```
foft = t2historyfile('FOFT')
blkindex = 123
temp = foft[blkindex][1]
```

For **TOUGH+** connection and generator history files (`COFT` and `GOFT`, or `Conx_Time_Series` and `SS_Time_Series`), multiple connections and generators can be specified as usual in the TOUGH2 input data file, but individual results for them are not written to the history file. Instead, the results for them are summed. As a result, there are no 'keys' as such for accessing individual results, and the `t2historyfile` works a little differently. An array containing the data in each column can be accessed by specifying the column name, for example:

Property	Type	Description
column_name	list	column headings
key_name	list	names of keys
num_times	integer	number of times
num_columns	integer	number of data columns
num_rows	integer	total number of data (for all keys)
simulator	string	detected simulator ('TOUGH2' or 'TOUGH2_MP')
times	np.array	times at which results are given
type	string	history type ('FOFT', 'COFT' or 'GOFT')

Table 6.5: Properties of a `t2historyfile` object

```
ct = t2historyfile('Conx_Time_Series')
qh = ct['HeatFlow']
```

The properties of a `t2historyfile` object are given in Table 6.5.

6.5 Examples

6.5.1 Slice plot of drawdown

This script shows a vertical slice plot along the model's x-axis of the difference in pressure (i.e. drawdown) between the start and end of a simulation.

```
from mulgrids import *
from t2listing import *
from copy import copy

geo=mulgrid('gmodel.dat')
results=t2listing('model.listing')

results.first()
p0=copy(results.element['Pressure'])
results.last()
p1=results.element['Pressure']

geo.slice_plot('x',(p1-p0)/1.e5,'Pressure\ difference','bar')
```

(Note: the `copy` command is needed, otherwise the arrays `p0` and `p1` would both contain the final values of pressure after the `results.last()` command.)

6.5.2 Pressure-temperature diagram

This script plots model results from a specified block on a pressure-temperature diagram.

```
from t2listing import *
import matplotlib.pyplot as plt

lst=t2listing('model.listing')
```

```
blk=‘ n 60’
[(tp,p),(tt,t)]=lst.history([(‘e’,blk,‘Pressure’),(‘e’,blk,‘Temperature’)])

plt.plot(t,p/1.e5,‘o-’)
plt.xlabel(‘T ($^o$C)’)
plt.ylabel(‘P (bar)’)
plt.show()
```

6.5.3 Comparing results of two models

This script reads grids and results for two different models, a coarse model and a fine one, and produces a comparison plot of the time history of temperature for both models at a given point.

```
from mulgrids import *
from t2listing import *
import matplotlib.pyplot as plt

geoc,geof=mulgrid(‘gcoarse.dat’),mulgrid(‘gfine.dat’)
coarse,fine=t2listing(‘coarse.listing’),t2listing(‘fine.listing’)

p=[47.e3,0.0,-7000.0]
blkc=geoc.block_name_containing_point(p)
blkf=geof.block_name_containing_point(p)

tc,tempc=coarse.history((‘e’,blkc,‘Temperature’))
tf,temp=fine.history((‘e’,blkf,‘Temperature’))

plt.plot(tc,tempc,‘o-’,label=‘coarse model’)
plt.plot(tf,tempf,‘s-’,label=‘fine model’)
plt.xlabel(‘time (s)’)
plt.ylabel(‘Temperature ($^o$C)’)
plt.legend()

plt.show()
```

Chapter 7

TOUGH2 thermodynamics

7.1 Introduction

The `t2thermo` library in PyTOUGH contains a Python implementation of the thermodynamic routines used in TOUGH2. These can be used to calculate the thermodynamic properties of water and steam under a range of conditions. They are based on a subset of the IFC-67 thermodynamic formulation.

The `t2thermo` library can be imported using the command:

```
from t2thermo import *
```

The functions available through the `t2thermo` library are listed in Table 7.1 and described below.

7.2 Thermodynamic functions

The thermodynamic routines used in TOUGH2 provide functions for liquid water and dry steam. These functions calculate secondary parameters from the primary thermodynamic variables. Their names follow the subroutine names used in the TOUGH2 code.

Function	Type	Description
`cowat`	tuple	density and internal energy of liquid water
`sat`	float	saturation pressure as a function of temperature
`separated_steam_fraction`	float	separated steam fraction for given enthalpy and separator pressure
`supst`	tuple	density and internal energy of dry steam
`tsat`	float	saturation temperature as a function of pressure
`visw`	float	dynamic viscosity of water
`viss`	float	dynamic viscosity of steam

Table 7.1: `t2thermo` functions

7.2.1 Liquid water: `cowat(t,p)`

The `cowat` function returns a two-element tuple (`d`,`u`) of density (kg/m^3) and internal energy (J/kg) of liquid water as a function of temperature `t` (°C) and pressure `p` (Pa).

Parameters:

- **t**: float
  Temperature (°C)
- **p**: float
  Pressure (Pa)

7.2.2 Dry steam: `supst(t,p)`

The `supst` function returns a two-element tuple (`d`,`u`) of density (kg/m^3) and internal energy (J/kg) of dry steam as a function of temperature `t` (°C) and pressure `p` (Pa).

Parameters:

- **t**: float
  Temperature (°C)
- **p**: float
  Pressure (Pa)

7.3 Viscosity

7.3.1 Liquid water: `visw(t,p,ps)`

The `visw` function returns the dynamic viscosity (Pa.s) of liquid water as a function of temperature `t` (°C), pressure (Pa) and saturation pressure (Pa).

Parameters:

- **t**: float
  Temperature (°C)
- **p**: float
  Pressure (Pa)
- **ps**: float
  Saturation pressure (Pa), calculated for example using the `sat` function.

7.3.2 Dry steam: `viss(t,d)`

The `viss` function returns the dynamic viscosity (Pa.s) of dry steam as a function of temperature `t` (°C) and density `d` (kg/m^3).

Parameters:

- **t**: float
  Temperature (°C)
- **d**: float
  Density (kg/m^3)

7.4 Saturation line: `sat(`*t*`)` and `tsat(`*p*`)`

7.4.1 `sat(`*t*`)`

The `sat` function returns the saturation pressure (Pa) at a given temperature `t` (°C), for temperatures below the critical temperature.

Parameters:

- `t`: float
  Temperature (°C)

7.4.2 `tsat(`*p*`)`

The `tsat` function returns the saturation temperature (°C) at a given pressure `p` (Pa), for pressures below the critical pressure.

Note that the IFC-67 formulation did not include an explicit formula for calculating saturation temperature as a function of pressure, so here (as in TOUGH2) this is calculated using an iterative root-finding process on the `sat` function. The root-finding function is from the `scipy` library, so this library must be installed before the `tsat` function will work.

Parameters:

- **p**: float
  Pressure (Pa)

7.5 Other functions

7.5.1 Separated steam fraction

`separated_steam_fraction(`*h, separator_pressure, separator_pressure2* `= None)`

Returns the separated steam fraction for a given enthalpy `h` and separator pressure. A second separator pressure may be specified in the case of two-stage flash.

Parameters:

- **h**: float
  Enthalpy (J/kg)
- **separator_pressure**: float
  Steam separator pressure (Pa)
- **separator_pressure2**: float (or `None`)
  Second separator pressure (Pa) for two-stage flash- set to `None` (the default) for single-stage.

7.6 Example

The following script reads in a geometry file and writes an initial conditions file with approximate hydrostatic conditions corresponding to a specified vertical temperature gradient. In this case, the model has a simple flat surface, so that each column has the same number of layers. The `cowat` function is used to calculate the fluid density at each layer, and hence the approximate vertical pressure distribution.

```
from mulgrids import *
from t2thermo import *

geo=mulgrid('gmodel.dat')

patm,tatm=101.325e3,15.0
pblk,tblk=np.zeros(geo.num_blocks)+patm,np.zeros(geo.num_blocks)+tatm
g=9.8
p,t=patm,tatm
thick=0.0
tgradient=30 # deg C/km
for lay in geo.layerlist[1:]:
    d=cowat(t,p)[0]
    thisthick=lay.top-lay.bottom
    h=0.5*(thick+thisthick)
    p+=d*g*h
    t+=tgradient/1.e3*h
    thick=thisthick
    for col in geo.columnlist:
        blkname=geo.block_name(lay.name,col.name)
        iblk=geo.block_name_index[blkname]
        pblk[iblk],tblk[iblk]=p,t
geo.write_incons('model.incon',[pblk,tblk])
```

Chapter 8

IAPWS-97 thermodynamics

8.1 Introduction

The `IAPWS97` library in PyTOUGH contains a Python implementation of the main functions of the International Association for the Properties of Water and Steam 1997 (IAPWS-97) thermodynamic formulation (Wagner et al., 2000). These can be used to calculate the thermodynamic properties of water, steam and supercritical water. The IAPWS-97 supersedes the IFC-67 formulation used in TOUGH2 (see section 7), being generally faster and more accurate, as well as having a simpler representation of the thermodynamic region around the critical point.

The operating range of the IAPWS-97 formulation is shown in the pressure-temperature plot below. It covers temperatures up to 800 °C and pressures up to 100 MPa, and is divided into four thermodynamic regions:

1. liquid water
2. dry steam
3. supercritical fluid
4. two-phase

The two-phase region (4) follows the saturation line on the pressure-temperature plot (the boundary between liquid water and dry steam), up to the critical point C ($T = 373.946$ °C, $P = 22.064$ MPa), where the distinction between liquid water and steam disappears. Region 3 covers supercritcal fluid (above the critical point) and also near-critical fluid, just below the critical point. The boundary between regions 1 and 3 (liquid water and supercritical) is aribitrarily set at $T = 350$ °C. The boundary between regions 2 and 3 (dry steam and supercritical) is described by the `b23p` and `b23t` functions, given in section 8.4.2.

The `IAPWS97` library can be imported using the command:

```
from IAPWS97 import *
```

The functions available through the `IAPWS97` library are listed in Table 8.1 and described below.

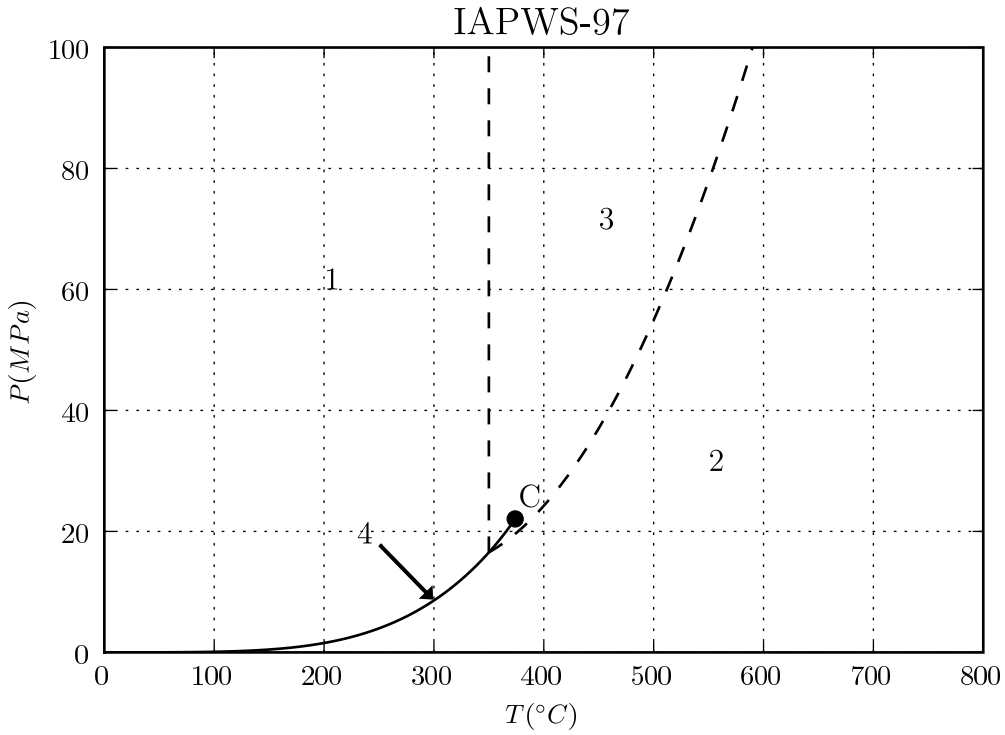


Figure 8.1: Operating range of the IAPWS-97 thermodynamic formulation

Function	Type	Description
b23p	float	pressure on boundary between steam and supercritical regions, as a function of temperature
b23t	float	temperature on boundary between steam and supercritical regions, as a function of pressure
cowat	tuple	density and internal energy of liquid water
density_temperature_plot	–	draws region boundaries on a density-temperature plot
pressure_temperature_plot	–	draws region boundaries on a pressure-temperature plot
sat	float	saturation pressure as a function of temperature
super	tuple	pressure and internal energy of supercritical fluid
supst	tuple	density and internal energy of dry steam
test	–	prints results of verification tests from Wagner et al. (2000)
tsat	float	saturation temperature as a function of pressure
visc	float	dynamic viscosity of water, steam or supercritical fluid

Table 8.1: IAPWS97 functions

8.2 Thermodynamic functions

The IAPWS-97 formulation provides thermodynamic functions for liquid water, dry steam and supercritical fluid. These functions calculate secondary parameters from the primary thermodynamic variables.

8.2.1 Liquid water: `cowat(t,p)`

The `cowat` function returns a two-element tuple (`d`,`u`) of density (kg/m^3) and internal energy (J/kg) of liquid water as a function of temperature `t` (°C) and pressure `p` (Pa).

Parameters:

- **t**: float
  Temperature (°C)
- **p**: float
  Pressure (Pa)

8.2.2 Dry steam: `supst(t,p)`

The `supst` function returns a two-element tuple (`d`,`u`) of density (kg/m^3) and internal energy (J/kg) of dry steam as a function of temperature `t` (°C) and pressure `p` (Pa).

Parameters:

- **t**: float
  Temperature (°C)
- **p**: float
  Pressure (Pa)

8.2.3 Supercritical fluid: `super(d,t)`

The `super` function returns a two-element tuple (`p`,`u`) of pressure (Pa) and internal energy (J/kg) of supercritical fluid as a function of density `d` (kg/m^3) and temperature `t` (°C).

Parameters:

- **d**: float
  Density (kg/m^3)
- **t**: float
  Temperature (°C)

8.3 Viscosity: `visc(d,t)`

The `visc` function returns the dynamic viscosity (Pa.s) of liquid water, dry steam or supercritical fluid as a function of density `d` (kg/m^3) and temperature `t` (°C). This function is based on the supplementary "IAPWS Formulation 2008 for the Viscosity of Ordinary Water Substance", without the critical enhancement of viscosity near the critical point.

Parameters:

- **d**: float
  Density (kg/m^3)

- **t**: float
  Temperature (°C)

8.4 Region boundaries

These functions describe the boundaries between the four thermodynamic regions of the IAPWS-97 formulation (see Figure 8.1). There is no equation for the boundary between regions 1 and 3 as this is simply the line $T = 350°C$.

8.4.1 Saturation line: `sat(t)` and `tsat(p)`

`sat(t)`

The `sat` function returns the saturation pressure (Pa) at a given temperature `t` (°C), for temperatures below the critical temperature.

Parameters:

- **t**: float
  Temperature (°C)

`tsat(p)`

The `tsat` function returns the saturation temperature (°C) at a given pressure `p` (Pa), for pressures below the critical pressure.

Parameters:

- **p**: float
  Pressure (Pa)

8.4.2 Steam/supercritical boundary

`b23p(t)`

The `b23p` function returns the pressure (Pa) on the boundary of the dry steam and supercritical regions (regions 2 and 3) at a given temperature `t` (°C).

Parameters:

- **t**: float
  Temperature (°C)

`b23t(p)`

The `b23t` function returns the temperature (°C) on the boundary of the dry steam and supercritical regions (regions 2 and 3) at a given pressure `p` (Pa).

Parameters:

- **p**: float
  Pressure (Pa)

8.5 Plotting functions

The `IAPWS97` library contains two functions used for including the IAPWS-97 thermodynamic region boundaries on plots.

8.5.1 `pressure_temperature_plot(`*plt*`)`

Draws the IAPWS-97 thermodynamic region boundaries on a pressure-temperature diagram.

Parameters:

- **plt**: `matplotlib.pyplot` instance
  An instance of the `matplotlib.pyplot` library, imported in the calling script using e.g. `import matplotlib.pyplot as plt`.

8.5.2 `density_temperature_plot(`*plt*`)`

Draws the IAPWS-97 thermodynamic region boundaries on a density-temperature diagram. (This function requires the Scientific Python (`scipy`) library to be installed.)

Parameters:

- **plt**: `matplotlib.pyplot` instance
  An instance of the `matplotlib.pyplot` library, imported in the calling script using e.g. `import matplotlib.pyplot as plt`.

8.6 Testing: `test()`

The `test` function prints the results of the verification tests in Wagner et al. (2000) for the `cowat`, `supst` and `super` functions.

References

Barker, J. (1988). A generalized radial flow model for hydraulic tests in fractured rock, *Water Resources Research* Vol. 24(10), 1796–1804.

O'Sullivan, M. and Bullivant, D. (1995). A graphical interface for the TOUGH2 family of flow simulators, *Proceedings of the TOUGH workshop 1995*, Lawrence Berkeley National Laboratory, University of California, Berkeley.

Pruess, K., Oldenburg, K. and Moridis, G. (1999). *TOUGH2 user's guide, version 2.0*, Lawrence Berkeley National Laboratory, University of California, Berkeley.

Wagner, W., Cooper, J., Dittman, A., Kijima, J., Kretzschmar, H.-J., Kruse, A., Mareš, R., Oguchi, K., Sato, H., Stöcker, I., Šifner, O., Takaishi, Y., Tanishita, I., Trübenbach, J. and Willkommen, T. (2000). The IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam, *Transactions of the ASME* Vol. 122, 150–182.

Zhang, K., Wu, Y.-S. and Pruess, K. (2008). *User's guide for TOUGH2-MP- a massively parallel version of the TOUGH2 code*, Lawrence Berkeley National Laboratory, University of California, Berkeley.