Using scriptit.py to define mb_cns simulations:  
a reference for the flow-description Python modules.

Mechanical Engineering Report 2005/10  
P. A. Jacobs  
Centre for Hypersonics  
The University of Queensland.

July 2005, Revised September 2005

Abstract

A flow simulation is set up by first describing the flow domain as one or more blocks of finite-volume cells and specifying both initial and boundary conditions. The simulation then proceeds in a number of stages: (1) start with the input script and prepare the block grids and initial (\(t = 0\)) flow solution; (2) run the simulation to produce snapshots of the flow solution at one or more subsequent times; and (3) postprocess the accumulated flow solution data to extract particular data that may be of interest.

The input script is written in the Python programming language and, when executed by the scriptit program, it creates various geometric and flow-condition objects as needed. This report describes the set of classes and functions that are available for creating the flow specification. Of course, the rest of the Python language is also available and may be used to assist (or automate) the set-up calculations.
# Contents

## Contents

1 Simulation Overview 4

2 Module scriptit 6
   2.1 Functions 6
   2.2 Variables 6
   2.3 Class Block2D 7
      2.3.1 Methods 8
      2.3.2 Instance Variables 9
   2.4 Class Face2D 9
      2.4.1 Methods 10
      2.4.2 Class Variables 10
   2.5 Class GlobalData2D 11
      2.5.1 Properties 11
      2.5.2 Instance Variables 11
   2.6 Class MetapostEnvironment 13
      2.6.1 Methods 14
   2.7 Class MultiBlock2D 14
      2.7.1 Methods 16
      2.7.2 Instance Variables 16

3 Module geom 18
   3.1 Functions 18
   3.2 Class Node 18
      3.2.1 Methods 18
      3.2.2 Properties 19
   3.3 Class Vector 19
      3.3.1 Methods 19
      3.3.2 Properties 20

4 Module gpath 22
   4.1 Functions 22
   4.2 Class Arc 22
      4.2.1 Methods 23
   4.3 Class Arc3 23
      4.3.1 Methods 24
   4.4 Class Bezier 24
      4.4.1 Methods 24
      4.4.2 Instance Variables 25
   4.5 Class ClosedSurfacePatch 25
      4.5.1 Methods 26
   4.6 Class Edge3D 27
      4.6.1 Methods 27
   4.7 Class Line 28
      4.7.1 Methods 28
   4.8 Class Polyline 29
      4.8.1 Methods 29
      4.8.2 Properties 30
   4.9 Class Spline 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.1</td>
<td>Methods</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Properties</td>
</tr>
<tr>
<td>5</td>
<td>Module flow_condition</td>
</tr>
<tr>
<td>5.1</td>
<td>Class FlowCondition</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Methods</td>
</tr>
<tr>
<td>6</td>
<td>Module cns_bc_defs</td>
</tr>
<tr>
<td>6.1</td>
<td>Variables</td>
</tr>
<tr>
<td>7</td>
<td>Module flux_dict</td>
</tr>
<tr>
<td>7.1</td>
<td>Variables</td>
</tr>
<tr>
<td>8</td>
<td>Module gas_dict</td>
</tr>
<tr>
<td>8.1</td>
<td>Variables</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
</tbody>
</table>

38
# 1 Simulation Overview

Setting up a simulation is mostly an exercise in writing a textual description of your flow and its bounding geometry. This description is presented to the `scriptit.py` program as a Python script, and is assumed to have a “.py” extension. Note that you will have full access to the Python programming language from within your script. This allows you to do some sophisticated calculations and automate repetitive parts of the specification (such as generating large numbers of blocks) but it also implies that you have to work to Python’s syntax\(^1\).

Once you have prepared your flow specification file, the simulation data is generated in a number of stages:

1a. Create the geometry definition with the command.

\[
\text{
$\text{scriptit.py -f job --do-mpost}$
}\]

**Input:** `job.py`  
**Program:** `scriptit.py`  
**Output:** `job.p`, `job.bez`, `job.mpost`

The file `job.p` contains the parameters that describe the logical connections of blocks, the flow conditions, and a number of parameters that will control the execution of the simulation. The file is plain text and may be edited manually if you wish. Its layout is documented in separate hypertext documentation\(^2\). The `job.bez` file contains the description of the block boundaries as sequences of lines, arcs and Bezier curves. Although it is also in plain text, this file is not so easy to understand. The `job.mpost` file contains a Metapost description of the geometry specification.

1b. Check the geometry definition (visually) by using Metapost to make a viewable postscript file containing labelled nodes, block boundary curves and blocks. Metapost is distributed as part of the \TeX\ document preparation system. It is most likely already installed on your UNIX/Linux/Cygwin system and there is a stand-alone binary for Win32 systems.

\[
\text{
$\text{mpost job.mpost}$
}\]

The `job.1` file contains a postscript rendering of your flow geometry that can be used for your documentation. It may also help in debugging your input script.

2. Generate an initial (i.e. \(t = 0\)) flow solution in file `job.s0` and a corresponding grid file `job.g`.

\[
\text{
$\text{mbprep.exe -f job}$
}\]

---

\(^1\)For example, if a list is specified as an argument to a particular function then you must supply a valid Python list object, possibly using the syntax `[item_0, item_1, item_3]` for a literal list.

3. Run the simulation code to produce flow data at subsequent times.
   
   ```bash
   $ mb_cns.exe -f job
   
   ```

   The whole-of-domain data are saved in `job.s` while history data at selected cells are saved in file `job.h`.

4. Extract subsets of the flow solution data for postprocessing. The specific command for this stage depends very much on what you want to do. The flow solution data is cell-averaged data associated with cell centres (i.e. the coordinates of the cell centres are kept with the flow data). You may extract the flow data for all cells at a particular time using `mb_post.exe` and reformat it for a particular plotting program or you may extract data along single grid lines (using `mb_prof.exe`) in a form ready for display with GNU-Plot or for further calculation. See the shell scripts in the examples for ideas on what can be done. The output of this stage is always a text file and, sometimes, it is convenient to look at the head of the file for hints as to what data is present.
2 Module scriptit

It is intended for the user to define the flow simulation in terms of the data objects defined in this program. As part of its initialization, scriptit.py will execute a user-specified job file that contains, in Python, the user’s script that defines both geometry and flow details.

The flow simulation definition is organised via the classes: GlobalData2D, FlowCondition, Face2D and Block2D. These classes provide places to store the configuration information and their function (or method) names appear as commands in the job description file. See the __init__() method for each class to determine what parameters can be specified when creating an object of that class.

The user will define the particular geometry in terms of the data objects defined in the geom and gpath modules. This geometry definition is created in a bottom-up approach by successively defining Nodes, simple path elements (such as Line, Arc and Bezier elements) and, possibly, compound path elements (such as Splines and Polylines). Finally, blocks of finite-volume cells covering the flow domain are defined via sets of four bounding faces. These faces also carry boundary-condition information.

Note: Physical quantities should be specified in MKS units.

2.1 Functions

**connect_blocks(A, faceA, B, faceB)**

Make the face-to-face connection between neighbouring blocks.

**Parameters**

- **A**: first block  
  *(type=Block2D object)*
- **faceA**: indicates which face of block A is to be connected. The constants NORTH, EAST, SOUTH, and WEST may be convenient to use.  
  *(type=int)*
- **B**: second block  
  *(type=Block2D object)*
- **faceB**: indicates which face of block B is to be connected. The constants NORTH, EAST, SOUTH, and WEST may be convenient to use.  
  *(type=int)*

**rad_to_degrees(rad)**

Convert radians to degrees.

This is a convenience function for the writing of the MetaPost file but may also be used in the user script.

**Parameters**

- **rad**: angle in radians.  
  *(type=float)*

**Return Value**

- angle in degrees

2.2 Variables
### Module scriptit

#### Class Block2D

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| AO   | Constant indicating use of the area-orthogonality grid generator.  
Value: 1 (type=int) |
| EAST | Constant indicating an east face.  
Value: 1 (type=int) |
| gdata | Contains the GlobalData2D information describing the simulation. Note that there is one such variable set up by the main program and the user’s script should directly set the attributes of this variable to adjust settings for the simulation.  
Value: <scriptit.GlobalData2D object at 0xb7ed07ac> (type=GlobalData2D) |
| mpost | A global variable holding the state of the MetapostEnvironment. Note that there is one such variable set up by the main program and the user’s script should directly set the attributes of this variable to adjust settings (such as scale and ranges of the axes) for the Metapost output.  
Value: <scriptit.MetapostEnvironment object at 0xb7ca09ac> (type=MetapostEnvironment) |
| NORTH | Constant indicating a north face.  
Value: 0 (type=int) |
| RULED | Constant indicating ruled-surface style of grid generation.  
Value: 2 (type=int) |
| SOUTH | Constant indicating a south face.  
Value: 2 (type=int) |
| TFI | Constant indicating transfinite interpolation style of grid generation. This type of interpolation surface is also known as a Coons patch.  
Value: 0 (type=int) |
| WEST | Constant indicating a west face.  
Value: 3 (type=int) |

#### 2.3 Class Block2D

```python
_builtins_object

Block2D
```

Python class to organise the setting of block parameters.

A block is defined by its four bounding Face2D objects with assumed positive directions as shown:

```
NORTH
1 +------->-------+
| |            |
s WEST | | EAST
| |            |
0 +------->-------+
| |            |
SOUTH
|
0------->-------
```

NORTH and SOUTH boundaries progress WEST to EAST while EAST and WEST boundaries progress SOUTH to NORTH. To reuse a Face2D object when building multiple blocks, you will need to pay
attention to the orientation of the blocks and the defined positive direction of the Face2D object.

2.3.1 Methods

```python
__init__(self, face_list=[], direction_list=[1, 1, 1, 1], fill_conditions=None, grid_type=0, hcell_list=[], turbulent_flag=0, label='')
```

Create a block from four faces.

**Parameters**

- **face_list**: List of the bounding faces which define the block. The order within the list is NORTH, EAST, SOUTH and WEST.
  
  *(type=list of Face2D objects)*

- **direction_list**: Each value of 1 indicates that we wish to use the corresponding Face2D in the direction that it presently has. A value of -1 indicates that we wish to reverse it’s direction for use in this block. Note that copies of the Face2D objects are made so that the original objects are not altered.
  
  *(type=list of int values)*

- **fill_conditions**: Either a single FlowCondition or a list of four FlowConditions. If a list is supplied, the order of items in the list is NE, SE, SW and NW.
  
  *(type=FlowCondition or a list of FlowCondition objects)*

- **grid_type**: Select the type of grid generator from RULED, TFI or AO.
  
  *(type=int or string)*

- **hcell_list**: List of (ix,iy) tuples specifying the cells (for this block) whose flow data is to be recorded in the history file. For an MPI simulation, there is one history file for each block but, for a shared-memory simulation, the history cells for all blocks are written to a single history file.
  
  *(type=list of tuples, each with two int values)*

- **turbulent_flag**: Set to 1 to activate the Baldwin-Lomax turbulence model. Note that the current implementation assumes that the north boundary is the only solid wall.
  
  *(type=int)*

- **label**: Optional label that will appear in the generated parameter file.
  
  *(type=string)*

Overrides: _builtin_.object.__init__

**Note**: The blocks are given their identity (counting from zero) according to the order in which they are created by the user’s script.
set_BC(self, face_name, type_of_BC, inflow_condition=None, sponge_flag=None, Twall=None, Pout=None)

Sets a boundary condition on a particular face of the block.
Sometimes it is good to be able to adjust properties after block creation; this function provides that capability.

**Parameters**

- **face_name**: Identifier to select the appropriate Face2D within the block.  
  *(type=string or int)*

- **type_of_BC**: Name or index value of the requested boundary condition. See module `cns_bc_defs` for the available options.  
  *(type=int or string)*

- **inflow_condition**: If the type of boundary requires the user to specify the inflow condition, this is the parameter to do so.  
  *(type=FlowCondition)*

- **sponge_flag**: Set to 1 to activate Andrew Denman’s damping layer near the boundary.  
  *(type=int)*

- **Twall**: If appropriate, specify the boundary-wall temperature in degrees Kelvin.  
  *(type=float)*

- **Pout**: If appropriate, specify the value of static pressure (in Pascals) just outside the boundary.  
  *(type=float)*

### 2.3.2 Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| blkId  | Index of the block. Blocks are numbered, from zero, in the order of creation. This numbering is used internally in the preprocessing, simulation and postprocessing stages. For the MPI simulations, it also the same as the rank of the process.  
  **Value**: `<member 'blkId' of 'Block2D' objects>` *(type=int)* |
| nnx    | Number of cells in the ix-direction. This value is normally obtained from the assigned north and south Face2D boundaries. Note that this ix index direction does not have to be aligned with the geometric x-direction.  
  **Value**: `<member 'nnx' of 'Block2D' objects>` *(type=int)* |
| nny    | Number of cells in the iy-direction. This value is normally obtained from the assigned east and west Face2D boundaries.  
  **Value**: `<member 'nny' of 'Block2D' objects>` *(type=int)* |

### 2.4 Class Face2D

```python
_builtin_object
```

Contains the information for one face of a block.
2.4.1 Methods

```python
__init__(self, path=None, direction=1, nn=10, cluster_tuple=(0, 0, 0.0), type_of_BC=3,
Twall=300.0, Pout=100000.0, inflow_condition=None, sponge_flag=0, label='')
```

Initialises a face consisting of a path, discretisation data and boundary-condition data.

**Parameters**

- **path:** may be a single path element or a list of path elements. The possible path elements include Line, Arc, Bezier, Polyline and Spline objects.
- **direction:** sense in which the path elements are assembled
- **nn:** number of cells to be distributed along the path
- **cluster_tuple:** clustering information consisting of (to-end-0, to-end-1, beta). to-end-0 is an integer (logical value) indicating whether the cells are clustered toward the t=0.0 end of the path. A value of 1 indicates that the cells are indeed to be clustered. to-end-1 is the same indicator for the t=1.0 end of the path. The beta parameter indicates the strength of the clustering. A value greater than 1.0 is used to obtain clustering, with a value approaching 1.0 indicating strong clustering. A value of zero for beta results in a uniformly distributed set of cells. See roberts.py and roberts.c (distribute_points_1) for further explanation of the parameters.
- **type_of_BC:** specifies the boundary condition See module `cns_bc_defs.py` for a dictionary of possible values.
- **Twall:** fixed wall temperature (in degrees K) that will be used if the boundary conditions needs such a value.
- **Pout:** fixed outside pressure (in Pascals) that will be used if the boundary conditions needs such a value.
- **inflow_condition:** the flow condition that will be applied if the specified boundary condition needs it
- **sponge_flag:** A value of 1 will activate Andrew Denman’s damping terms near the boundary. (type=int)
- **label:** optional label for the Face2D object (type=string)

Overrides: `__builtin__object.__init__`

```python
copy(self, direction=1)
```

Makes a complete copy of the face as a new object, optionally reversing the direction.

2.4.2 Class Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nmin</td>
<td>Minimum number of cells along any face and, correspondingly, across any block.</td>
</tr>
<tr>
<td></td>
<td><strong>Value:</strong> 2 (type=int)</td>
</tr>
</tbody>
</table>
2.5 Class GlobalData2D

Python class to organise the global data.

The user’s script should not create one of these but should specify the simulation parameters by altering the attributes of the global object `gdata`.

2.5.1 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gas.name</code></td>
<td>The (string) name of the gas.</td>
</tr>
</tbody>
</table>

2.5.2 Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>axisymmetric_flag</code></td>
<td>A value of 0 sets two-dimensional, planar flow. A value of 1 sets axisymmetric flow with the x-axis being the axis of symmetry. <strong>Value:</strong> <code>&lt;member 'axisymmetric_flag' of 'GlobalData2D' objects&gt;</code> <em>(type=int)</em></td>
</tr>
<tr>
<td><code>case_id</code></td>
<td>An identifier for special cases in which pieces of specialised code have been embedded into the main simulation program. If you don’t have such code to activate, the default value of 0 is fine. <strong>Value:</strong> <code>&lt;member 'case_id' of 'GlobalData2D' objects&gt;</code> <em>(type=int)</em></td>
</tr>
<tr>
<td><code>cfl</code></td>
<td>The ratio of the actual time step to the allowed time step as determined by the flow condition and grid. Typically the default value of 0.5 is good but you may want to try smaller values if you are having the solution go unstable, especially for viscous simulations. <strong>Value:</strong> <code>&lt;member 'cfl' of 'GlobalData2D' objects&gt;</code> <em>(type=float)</em></td>
</tr>
<tr>
<td><code>displacement_thickness</code></td>
<td>See Andrew Denman. <strong>Value:</strong> <code>&lt;member 'displacement_thickness' of 'GlobalData2D' objects&gt;</code> <em>(type=float)</em></td>
</tr>
<tr>
<td><code>dt</code></td>
<td>Size of the initial time step. After a few steps, the solver will have enough information to select a suitable time step, based on the cfl number. <strong>Value:</strong> <code>&lt;member 'dt' of 'GlobalData2D' objects&gt;</code> <em>(type=float)</em></td>
</tr>
<tr>
<td><code>dt_av</code></td>
<td>See Andrew Denman. <strong>Value:</strong> <code>&lt;member 'dt_av' of 'GlobalData2D' objects&gt;</code> <em>(type=float)</em></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dt_history</td>
<td>Period (in seconds) between writing the data for the selected cells to the history file.</td>
</tr>
<tr>
<td>dt_plot</td>
<td>Period between writing all of the flow field data to the solution file. Multiple instances can be written to the one file but be careful not to write too many and fill up your disk.</td>
</tr>
<tr>
<td>flux_calc</td>
<td>Specifies the form of flux calculation at cell interfaces. See module flux_dict.py for options.</td>
</tr>
<tr>
<td>gas_name</td>
<td>Select the thermochemistry model by setting this parameter. See module gas_dict for available options. If you don’t select something, the value will default to 'perf_air_14' the first time that you try to set a FlowCondition.</td>
</tr>
<tr>
<td>max_step</td>
<td>Time stepping will be terminated if the simulation reached this number of steps.</td>
</tr>
<tr>
<td>max_time</td>
<td>The (simulation) time (in seconds) at which time stepping should be terminated.</td>
</tr>
<tr>
<td>perturb_flag</td>
<td>See Andrew Denman.</td>
</tr>
<tr>
<td>perturb_frac</td>
<td>See Andrew Denman.</td>
</tr>
<tr>
<td>reacting_flag</td>
<td>A value of 1 will make Rowan Gollan’s finite-rate chemistry active if the appropriate gas_name (e.g. 'perf_gas_mix') has been specified.</td>
</tr>
<tr>
<td>t_order</td>
<td>Specifies the form of time stepping scheme. Select 1 for Euler stepping. Select 2 for predictor-corrector stepping.</td>
</tr>
<tr>
<td>tav_0</td>
<td>See Andrew Denman.</td>
</tr>
<tr>
<td>tav_f</td>
<td>See Andrew Denman.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_average_flag</td>
<td>See Andrew Denman.</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'time_average_flag' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=int)</em></td>
</tr>
<tr>
<td>title</td>
<td>A piece of text that will be propagated through the solution files and subsequently generated plots.</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'title' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=string)</em></td>
</tr>
<tr>
<td>turbulent_flag</td>
<td>Andrew Denman.</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'turbulent_flag' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=int)</em></td>
</tr>
<tr>
<td>viscous_delay</td>
<td>Sometimes, the viscous terms make it difficult to start a calculation without encountering numerical instability. Set this parameter to the delay (in seconds) from simulation start to the time at which the viscous terms will be allowed to become active (if viscous_flag was set to 1).</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'viscous_delay' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=float)</em></td>
</tr>
<tr>
<td>viscous_flag</td>
<td>Set to 1 to activate viscous transport terms. Set to 0 (the default) for inviscid flow simulation.</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'viscous_flag' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=int)</em></td>
</tr>
<tr>
<td>x_order</td>
<td>Specifies the form of reconstruction from cell-average data to cell interface data. Select 1 for low-order (i.e. no) reconstruction. Select 2 for a higher-order (limited quadratic) reconstruction.</td>
</tr>
<tr>
<td><strong>Value:</strong> &lt;member 'x_order' of 'GlobalData2D' objects&gt;</td>
<td><em>(type=int)</em></td>
</tr>
</tbody>
</table>

2.6 Class MetapostEnvironment

```
__builtin_object MetapostEnvironment
```

A place to put the metapost settings.

A metapost file will contain a rendering of the geometry objects defining the simulation domain. Since the coordinates for mb_cns are in metres, you will probably have to apply suitable scale factors to get a drawing that can be printed to an A4 page conveniently. Also, because the origin of a postscript figure is in the bottom-left corner of the page, you may have to reset the origin to see all of the geometry elements if some of them have negative coordinates.
2.6.1 Methods

### origin(self, x=0.0, y=0.0)

Set the origin on the page for the rendered picture. For example, it is sometimes good to select an origin of (0.05, 0.05) to get the origin 5 centimetres up and right from the bottom-left corner of the page.

### scales(self, xscale=None, yscale=None)

Set the scale factors for the drawing. Model coordinates are multiplied by these scales to get page coordinates.

### xaxis(self, xmin=None, xmax=None, xtic=None, xaxis_offset=None)

Set the x-axis scale parameters.

**Parameters**

- **xmin**: Minimum value for x-axis scale. *(type=float)*
- **xmax**: Maximum value for x-axis scale. *(type=float)*
- **xtic**: Interval between tic marks and labels. *(type=float)*
- **xaxis offset**: The vertical offset (from ymin) for drawing the length of the scale. Negative values will lower the x-axis scale. *(type=float)*

### yaxis(self, ymin=None, ymax=None, ytic=None, yaxis_offset=None)

Set the y-axis scale parameters.

**Parameters**

- **ymin**: Minimum value for y-axis scale. *(type=float)*
- **ymax**: Maximum value for y-axis scale. *(type=float)*
- **ytic**: Interval between tic marks and labels. *(type=float)*
- **yaxis offset**: The horizontal offset (from xmin) for drawing the length of the scale. Negative values will move the y-axis scale to the left. *(type=float)*

2.7 Class MultiBlock2D

```
__builtins__.object
   ^
MultiBlock2D
```

Allows us to specify a block of sub-blocks.

A number of internally-connected Block2D objects will be created when one MultiBlock2D object is created. Internal boundaries are defined by first using TFI interpolation to locate a set of intermediate points (that will become the corners of the individual blocks) and then fitting interpolating splines through this
array of points. Individual block boundaries are then subpaths of the original outer boundaries or of the newly created splines in the interior of the block cluster.

Note that the collection of Block2D objects will be stored in a list of lists with each inner-list storing a j-column of blocks:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
</tbody>
</table>
```

The user script may access an individual block within the MultiBlock2D object as `object.blks[i][j]`. This will be useful for connecting blocks within the MultiBlock cluster to other blocks as defined in the user's script.

Some properties, such as fill_conditions and grid_type, will be propagated to all sub-blocks. Individual sub-blocks can be later customised.
2.7.1 Methods

```python
__init__(self, face_list=[], direction_list=[1, 1, 1, 1], nb_w2e=1, nb_s2n=1, nn_w2e=None, nn_s2n=None, cluster_w2e=None, cluster_s2n=None, fill_conditions=None, grid_type=0, turbulent_flag=0, label='blk')
```

Create a cluster of blocks within a set of 4 boundaries.

**Parameters**

- **face_list**: List of the bounding faces which define the block. The order within the list is NORTH, EAST, SOUTH and WEST.
  *(type=list of Face2D objects)*

- **direction_list**: Each value of 1 indicates that we wish to use the corresponding Face2D in the direction that it presently has. A value of -1 indicates that we wish to reverse it's direction for use in this block. Note that copies of the Face2D objects are made so that the originals are not altered.
  *(type=list of int values)*

- **nb_w2e**: Number of sub-blocks from west to east.
  *(type=int)*

- **nb_s2n**: Number of sub-blocks from south to north.
  *(type=int)*

- **nn_w2e**: List of discretisation values for north and south boundaries of the sub-blocks. If a list is not supplied, the original number of cells for the outer boundary is divided over the individual sub-block boundaries.
  *(type=list of int values)*

- **nn_s2n**: List of discretisation values for west and east boundaries of the sub-blocks. If a list is not supplied, the original number of cells for the outer boundary is divided over the individual sub-block boundaries.
  *(type=list of int values)*

- **cluster_w2e**: If a list of cluster tuples is supplied, individual clustering will be applied to the corresponding south and north boundaries of each sub-block. If not supplied, a default of no clustering will be applied.
  *(type=list of cluster tuples)*

- **cluster_s2n**: If a list of cluster tuples is supplied, individual clustering will be applied to the corresponding west and east boundaries of each sub-block. If not supplied, a default of no clustering will be applied.
  *(type=list of cluster tuples)*

- **fill_conditions**: A single FlowCondition that is to be used for all sub-blocks
  *(type=a single FlowCondition object)*

- **grid_type**: Select the type of grid generator from Ruled, TFI or AO.
  *(type=int or string)*

- **turbulent_flag**: This flag will be propagated to all sub-blocks.
  *(type=int)*

- **label**: A label that will be augmented with the sub-block index and then used to label the individual Block2D objects.
  *(type=string)*

Overrides: __builtin__.object.__init__

2.7.2 Instance Variables
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| blks | This holds the collection of sub-blocks, each being a Block2D object.  
**Value:** `<member 'blks' of 'MultiBlock2D' objects>`  
*(type=a list of lists of Block2D objects)* |
Module geom

Provides basic 3D/2D Vector and Node classes for constructing geometric descriptions.
For 2D modelling, the z-coordinate can be omitted so that it takes its default value of 0.0.

3.1 Functions

cross($a$, $b$)
Vector cross product.
Return Value
vector product.

distance_between_nodes($a$, $b$)
Return the distance between Nodes $a$ and $b$.

dot($a$, $b$)
Vector dot product.
Return Value
scalar product.

3.2 Class Node

__builtin__ object

geom.Vector

Node

Defines a nodal-point in 3D space to be subsequently used in the definition of line segments.

3.2.1 Methods

__init__(self, $x=0.0$, $y=0.0$, $z=0.0$, label='')
Create a Vector from its Cartesian components.
Parameters
  x: x-component
      (type=float)
y: y-component
      (type=float)
z: z-component
      (type=float)
label: optional text label that will appear in the MetaPost file.
      (type=string)
Overrides: geom.Vector.__init__ (inherited documentation)
copy(self)

Create and return a new instance which is a copy of the original.

translate(self, dx, dy=0.0, dz=0.0)

Translate node position by displacement Vector dx or by displacement (dx, dy, dz) in Cartesian coordinates.

Parameters
- dx: displacement
  (type=either Vector or float)
- dy: (optional) Cartesian displacement in the y-direction
  (type=float)
- dz: (optional) Cartesian displacement in the z-direction
  (type=float)

Inherited from Vector: __abs__, __add__, __del__, __div__, __mul__, __neg__, __pos__, __rdiv__, __rmul__,
__sub__, getLabel, getX, getY, getZ, setLabel, setX, setY, setZ, sum, unit

3.2.2 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited from Vector: x (p. 19), y (p. 19), z (p. 19)</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Class Vector

`_builtin_.object` Vector

Known Subclasses: Node

Defines a vector 3D space.

The vector is created in the C module data space and new-style object properties to access the C-module values.
### 3.3.1 Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(self, x=0.0, y=0.0, z=0.0, label='')</code></td>
<td>Create a Vector from its Cartesian components.</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td><code>x</code></td>
<td>x-component</td>
</tr>
<tr>
<td><code>y</code></td>
<td>y-component</td>
</tr>
<tr>
<td><code>z</code></td>
<td>z-component</td>
</tr>
<tr>
<td><code>label</code></td>
<td>optional text label that will appear in the MetaPost file.</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td></td>
<td>(type=float)</td>
</tr>
<tr>
<td></td>
<td>(type=string)</td>
</tr>
<tr>
<td>Overrides</td>
<td><code>__builtin__.object.__init__</code></td>
</tr>
<tr>
<td><code>__abs__(self)</code></td>
<td>Absolute value is the magnitude of the Vector.</td>
</tr>
<tr>
<td><code>__add__(self, other)</code></td>
<td>Element by element addition.</td>
</tr>
<tr>
<td><code>__div__(self, other)</code></td>
<td>Element-by-element multiplication.</td>
</tr>
<tr>
<td><code>__mul__(self, other)</code></td>
<td>Negation of all elements for -x.</td>
</tr>
<tr>
<td><code>__neg__(self)</code></td>
<td>+x</td>
</tr>
<tr>
<td><code>__pos__(self)</code></td>
<td></td>
</tr>
<tr>
<td><code>__rdiv__(self, other)</code></td>
<td></td>
</tr>
<tr>
<td><code>__rmul__(self, other)</code></td>
<td></td>
</tr>
<tr>
<td><code>__sub__(self, other)</code></td>
<td>Element-by-element subtraction.</td>
</tr>
<tr>
<td><code>sum(self)</code></td>
<td>Add elements together.</td>
</tr>
</tbody>
</table>

unit(self)

Return Value
unit vector with same direction as this vector.

3.3.2 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>text label for the vector</td>
</tr>
<tr>
<td>x</td>
<td>x-coordinate of the vector</td>
</tr>
<tr>
<td>y</td>
<td>y-coordinate of the vector</td>
</tr>
<tr>
<td>z</td>
<td>z-coordinate of the vector</td>
</tr>
</tbody>
</table>
4 Module gpath

The user-specified geometry data is organised via the following classes: Node, Line, Arc, Bezier, Spline, and Polyline. This module builds on the Vector and Node classes provided by the module geom and provides curvilinear path-building classes.

The path elements Line, Arc, Bezier are a mix of Python top-level classes and lower-level C functions on arrays of points. Although it would have been much neater and more maintainable to use a pure Python implementation, we wanted to use the same basic code for both the C and the Python programs.

The compound Polyline and Spline objects are also a mix of Python classes and C functions on GPathPolyline structures.

4.1 Functions

**interpolate_TFI2D_mbcns**(edge_list, r, s)
Locate a point p(r,s) using transfinite interpolation.

**Parameters**
- **edge_list**: List of 4 paths in mb_cns order [N,E,S,W].
  (type=list of Polyline-derived objects.)
- **r**: interpolation parameter in the ix-index direction
  (type=float, 0.0 <= r <= 1.0)
- **s**: interpolation parameter in the iy-index direction
  (type=float, 0.0 <= s <= 1.0)

**interpolate_TFI3D**(edge_list, r, s, t)
Locate a point p(r,s,t) using transfinite interpolation.

**Parameters**
- **edge_list**: list of 12 paths in Elmer order.
  (type=list of Polyline-derived objects.)
- **r**: interpolation parameter in the i-index direction
  (type=float, 0.0 <= r <= 1.0)
- **s**: interpolation parameter in the j-index direction
  (type=float, 0.0 <= s <= 1.0)
- **t**: interpolation parameter in the k-index direction
  (type=float, 0.0 <= t <= 1.0)

4.2 Class Arc

```python
__builtin_object
  └── Arc
```

**Known Subclasses**: Arc3

Defines a circular-arc from Node a to Node b with centre at c.
4.2.1 Methods

__init__(self, a, b, c)

Parameters
  a: Starting point for arc.
     (type=Node object)
  b: Finish point for arc.
     (type=Node object)
  c: Centre of curvature.
     (type=Node object)

Overrides: builtin.object.__init__

Note: The radii c->a and c->b must match closely.

eval(self, t)

Locate a point on the arc.

Parameters
  t: interpolation parameter.
     (type=float, 0<=t<=1.0)

Return Value
  a Vector for the point location.

length(self)

Return Value
  the length of the arc.

translate(self, dx, dy=0.0, dz=0.0)

Displace the Arc in Cartesian space.

Parameters
  dx: displacement Vector representing (dx, dy, dz) or x-component of displacement.
     (type=Vector or float)
  dy: y-component of displacement (if dx was a scalar)
     (type=float)
  dz: z-component of displacement (if dx was a scalar)
     (type=float)

4.3 Class Arc3

__builtin__object

  gpath.Arc

Arc3

Defines a circular-arc from Node a through Node b ending at Node c.
4.3.1 Methods

```python
__init__(self, a, b, c)
```

**Parameters**

- `a`: Starting point for arc.
  
  *(type=Node object)*

- `b`: Intermediate point on arc.
  
  *(type=Node object)*

- `c`: Finish point for arc.
  
  *(type=Node object)*

**Overrides:** `gpath.Arc.__init__`

**Note:** The points must not be colinear.

Inherited from Arc: `eval`, `length`, `translate`

### 4.4 Class Bezier

```python
__builtin__ object

Bezier
```

Defines a Bezier polynomial curve.

**Note:** The curve goes through the end-points but that the intermediate points generally do not lie on the curve.

4.4.1 Methods

```python
__init__(self, B)
```

Defines a Bezier polynomial of order $N=\text{len}(B)-1$.

**Parameters**

- `B`: The control nodes of the curve, $B[0] \rightarrow B[-1]$.
  
  *(type=list of Node objects)*

**Overrides:** `__builtin__ object.__init__`

```python
eval(self, t)
```

Locate a point on the Bezier curve.

**Parameters**

- `t`: interpolation parameter.
  
  *(type=float, $0 \leq t \leq 1.0$)*

**Return Value**

- a `Vector` for the point location.
Module gpath

Class ClosedSurfacePatch

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length(self)</td>
<td>Return Value the length of the Bezier curve. Note: This is obtained by sampling the curve.</td>
</tr>
<tr>
<td>translate(self, dx, dy=0.0, dz=0.0)</td>
<td>Displace the Bezier curve. Parameters dx: displacement Vector representing (dx, dy, dz) or x-component of displacement. (type=Vector or float) dy: y-component of displacement (if dx was a scalar) (type=float) dz: z-component of displacement (if dx was a scalar) (type=float)</td>
</tr>
</tbody>
</table>

4.4.2 Instance Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>The order of the curve is N = len(B)-1. (type=int)</td>
</tr>
</tbody>
</table>

4.5 Class ClosedSurfacePatch

A ClosedSurfacePatch is defined by 4 bounding Polyline paths. It is intended mainly for the generation of 3D blocks by sweeping out volumes.
4.5.1 Methods

__init__(self, cA, cB, cC, cD)
Create a ClosedSurfacePatch.

Parameters
- cA: 'South' curve
  *(type=Polyline-derived object)*
- cB: 'North' curve
  *(type=Polyline-derived object)*
- cC: 'West' curve
  *(type=Polyline-derived object)*
- cD: 'East' curve
  *(type=Polyline-derived object)*

Overrides: __builtin_object__._init__

Notes:
- The logical layout for the bounding curves is:
  
  . cB
  . +---------+
  . |         |
  . cC ^       ^ cD
  . |         |
  . +---------+
  . cA

  These curves must meet at the corners.
- In the 3D simulation code, this ClosedSurfacePatch can represent any of the 6 faces of a block. Curves cA and cB progress in the positive i-index direction for Top, Bottom, North and South faces. Curves cC and cD progress in the positive j-index direction for Top and Bottom faces. Curves cA and cB progress in the positive j-index direction for West and East faces. Curves cC and cD progress in the positive k-index direction for North, South, West and East faces.

extrude(self, cE, direction)
Extrudes the ClosedSurfacePatch to form a closed volume.

Parameters
- cE: curve along which the extrusion is done.
  *(type=Polyline-derived object)*
- direction: provides a hint as to which way we want to extrude the surface.
  *(type=string being one of 'i', 'j', or 'k')*

Return Value
the list of 12 edges defining a 3D block.
Module gpath

Class Edge3D

interpolate_TFI(self, r, s)

Locate a point on the ClosedSurfacePatch.

Parameters

- r: interpolation parameter for along curves cA and cB
  
  \(0.0 < r < 1.0\)

- s: interpolation parameter for along curves cC and cD
  
  \(0.0 < s < 1.0\)

Return Value

a Vector value for the point.

4.6 Class Edge3D

An Edge3D is a specialized Polyline that contains some extra data for mesh generation in the 3D flow simulation code.

4.6.1 Methods

_init_(self, path=None, direction=1, t0=0.0, t1=1.0, cluster_tuple=(0, 0, 0.0), label='')

Initialises an edge consisting of a path, and cluster data that may be used for mesh generation

Parameters

- path: may be a single path element or a list of path elements. The possible path elements include Line, Arc, Bezier, Polyline and Spline objects.

- direction: sense in which the path elements are assembled

- cluster_tuple: clustering information consisting of (to-end-0, to-end-1, beta) See roberts.py and roberts.c (distribute_points_1) for an explanation of the parameters.

- label: optional label (string) for the object

Overrides: gpath.Polyline._init_

copy(self, direction=1)

Parameters

- direction: Set to -1 to reverse the sense of the path for this copy.

  \((type=int)\)

Return Value

a separate copy of the Edge3D object, possibly reversed.

Overrides: gpath.Polyline.copy

Inherited from Polyline: append, eval, get_t0, get_t1, length, nelements, set_t0, set_t1, translate

27
4.7 Class Line

```python
__builtin__object

Line
```

Defines a straight-line segment.

4.7.1 Methods

```python
__init__(self, a, b)
```

Define the directed line from Node a to Node b.

**Parameters**

- `a`: Starting point on line.
  
  \(\text{type=} \text{Node object}\)

- `b`: Finishing point on line.
  
  \(\text{type=} \text{Node object}\)

**Overrides:** `__builtin__object.__init__`

```python
eval(self, t)
```

Locate a point on the line.

**Parameters**

- `t`: Interpolation parameter.
  
  \(0\leq t\leq 1.0\)

**Return Value**

- A Vector for the point location.

```python
length(self)
```

**Return Value**

- The length of the line.

```python
translate(self, dx, dy=0.0, dz=0.0)
```

Displace the Line.

**Parameters**

- `dx`: Displacement Vector representing (dx, dy, dz) or x-component of displacement.
  
  \(\text{type=} \text{Vector or float}\)

- `dy`: y-component of displacement (if dx was a scalar)
  
  \(\text{type=} \text{float}\)

- `dz`: z-component of displacement (if dx was a scalar)
  
  \(\text{type=} \text{float}\)
4.8 Class Polyline

```
__builtin__object

Polyline
```

Known Subclasses: Edge3D, Spline

Polylines are composed of a number of gpath elements.
This is also the data-structure used in the C-functions that define the edges of the grid.

4.8.1 Methods

```
_init_(self, pathElements=[], direction=1, t0=0.0, t1=1.0)

Create, possibly initialising from a list of gpath elements.

Parameters

pathElements: either a list of gpath objects or a single such object
direction: sense in which to assemble the pathElements
t0: starting value for the evaluation parameter (defines a subpath)
t1: end value for the evaluation parameter (defines a subpath)

Overrides: __builtin__object_init
```

```
append(self, other, direction=1)

Parameters

other: The item to append.
   (type=a Polyline object or other gpath object)
direction: A value of -1 will reverse the sense of the appended object.
   (type=int)
```

```
copy(self, direction=1)

Parameters

direction: Set to -1 to reverse the sense of the copy.
   (type=int)

Return Value

a separate copy of the Polyline, possibly reversed.
```

```
eval(self, t)

Locate a point on the Polyline path.

Parameters

t: interpolation parameter.
   (type=float, 0<=t<=1.0)

Return Value

a Vector for the point location.
```
length(self)

Return Value
the Polyline length

Note: The length will be updated with the addition of each new element.

nelements(self)

Return Value
the number of elements in Polyline.

translate(self, dx, dy=0.0, dz=0.0)

Displace the Polyline.

Parameters
- dx: displacement Vector representing (dx, dy, dz) or x-component of displacement.
  (type=Vector or float)
- dy: y-component of displacement (if dx was a scalar)
  (type=float)
- dz: z-component of displacement (if dx was a scalar)
  (type=float)

4.8.2 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>Lower bound for subrange.</td>
</tr>
<tr>
<td>t1</td>
<td>Upper bound for subrange.</td>
</tr>
</tbody>
</table>

4.9 Class Spline

Defines a cubic-spline path.
4.9.1 Methods

```python
_init_(self, B)
```

Creates the spline as a set of Bezier segments, then casts it as a Polyline.

**Parameters**

- `B`: interpolation points, `B[0] \rightarrow B[-1]`.
  
  *(type=list of Node objects.)*

**Overrides:** gpath.Polyline._init_

**Note:** The internal representation is a set of N cubic Bezier segments that have the `B[j]` nodes as end points.

**Inherited from Polyline:** append, copy, eval, get_t0, get_t1, length, nelements, set_t0, set_t1, translate, write_to_file

4.9.2 Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited from Polyline: t0 (p. 29), t1 (p. 29)</td>
<td></td>
</tr>
</tbody>
</table>
5 Module flow\_condition

A Python module to specify the flow conditions that may be applied as initial and boundary conditions in the 2D and 3D flow codes.

It is intended that this module be imported by scriptit.py and elmer\_prep.py.

5.1 Class FlowCondition

```python
import object

FlowCondition
```

Python class to organise the setting of each flow condition.

5.1.1 Methods

```python
__init__(self, p=100000.0, u=0.0, v=0.0, w=0.0, T=300.0, mf=[1.0], Tv=None, Te=None, label='')
```

Create a FlowCondition.

**Parameters**

- `p`: static pressure, Pa  
  
  *(type=float)*  

- `u`: x-component of velocity, m/s  
  
  *(type=float)*  

- `v`: y-component of velocity, m/s  
  
  *(type=float)*  

- `w`: z-component of velocity, m/s  
  
  *(type=float)*  

- `T`: static temperature, degrees K  
  
  *(type=float)*  

- `mf`: mass fractions of the component species  
  
  *(type=list of float values)*  

- `Tv`: (optional) vibrational temperatures, degrees K  
  
  *(type=list of float values)*  

- `Te`: (optional) separate electron temperature, degrees K  
  
  *(type=float)*  

**Overrides**: `_builtin__object.__init__`

**Notes:**

- If a gas model has not already been selected, the `perf\_air\_14` model will be implicitly chosen.
- The lengths of the lists of mass fractions and vibrational temperatures (if relevant) must match the number of species in the previously selected gas model.

```python
_deepcopy_(self, visit)
```

Provides a deep copy mechanism for the flow state
6 Module cns_bc_defs

Dictionary to look up boundary-condition index from name or number.

Boundary conditions are implemented within the simulation by setting flow data in ghost cells to suitable values. This is done once per time step, before evaluating the fluxes.

6.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADIABATIC</td>
<td>A solid, no-slip wall without heat transfer. (i.e. the near-wall temperature is reflected in the ghost cells)</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>ADJACENT</td>
<td>This boundary joins that of another block. Normally, this boundary condition would be set implicitly when making block connections.</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>COMMON</td>
<td>Synonym for ADJACENT.</td>
<td>0</td>
<td>int</td>
</tr>
<tr>
<td>EXTRAPOLATE_OUT</td>
<td>Extrapolate all flow properties from just inside the boundary into the ghost-cells outside the boundary. This works fine for a strong supersonic outflow.</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>FIXED_P_OUT</td>
<td>Something like EXTRAPOLATE_OUT but with the pressure set to some user-specified value. It is probably best to set this pressure at the same value as the initial fill pressure so that this boundary condition will be passive until a wave arrives at the boundary.</td>
<td>11</td>
<td>int</td>
</tr>
<tr>
<td>FIXED_T</td>
<td>A solid, no-slip wall with a user-specified temperature.</td>
<td>5</td>
<td>int</td>
</tr>
<tr>
<td>RRM</td>
<td>Andrew Denman’s recycled and renormalised boundary condition.</td>
<td>12</td>
<td>int</td>
</tr>
<tr>
<td>SLIP</td>
<td>Synonym for SLIP_WALL</td>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>SLIP_WALL</td>
<td>A solid but inviscid wall. Effectively, this boundary condition copies the properties just inside the boundary into the ghost cells.</td>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>STATIC_PROF</td>
<td>A steady inflow boundary with a variable set of flow conditions across the boundary.</td>
<td>10</td>
<td>int</td>
</tr>
<tr>
<td>SUB_OUT</td>
<td>Synonym for SUBSONIC_OUT</td>
<td>7</td>
<td>int</td>
</tr>
<tr>
<td>SUBSONIC_IN</td>
<td>An inflow boundary for which the total pressure and temperature have been specified and the velocity from just inside the boundary is copied into the ghost cells.</td>
<td>6</td>
<td>int</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSONIC_OUT</td>
<td>An outflow boundary which will try to prevent wave reflection at the boundary in the presence of subsonic flow. (Doesn't work so well at present.)</td>
<td>7 (int)</td>
</tr>
<tr>
<td>SUP_IN</td>
<td>Fully-prescribed inflow (e.g. supersonic inflow).</td>
<td>1 (int)</td>
</tr>
<tr>
<td>SUP_OUT</td>
<td>Synonym for EXTRAPOLATE_OUT.</td>
<td>2 (int)</td>
</tr>
<tr>
<td>TRANSIENT_UNI</td>
<td>An transient inflow boundary which has a uniform flow condition applied across the full boundary.</td>
<td>8 (int)</td>
</tr>
</tbody>
</table>
7 Module flux_dict

Dictionary to look up flux-calculator index from name or number.

7.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAPTIVE</td>
<td>A switched AUSMDV/EFM scheme that uses EFM near shocks and AUSMDV elsewhere. This is a good all-rounder for shock-tunnel work.</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>AUSM</td>
<td>M. S. Liou’s AUSM approximate flux calculator. Fast but tends to be a bit noisy.</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>AUSMDV</td>
<td>A version of Wada and Liou’s AUSMDV scheme.</td>
<td>3</td>
<td>int</td>
</tr>
<tr>
<td>EFM</td>
<td>Mike Macrossan’s version of Dale Pullin’s equilibrium flux calculator as coded by Paul Petrie-Repar. When you need a dissipative scheme, this is a good one.</td>
<td>2</td>
<td>int</td>
</tr>
<tr>
<td>RIEMANN</td>
<td>An exact Riemann-solver-based flux calculator. It is slow and only works for ideal gas models at the moment.</td>
<td>0</td>
<td>int</td>
</tr>
</tbody>
</table>
8 Module gas_dict

Dictionary to look up gas-type index from name or number.
Either the string name or the integer index can be used to set the gas model.

8.1 Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARGON_LJ</td>
<td>Ideal argon with Lennard-Jones viscosity.</td>
<td>27</td>
<td>int</td>
</tr>
<tr>
<td>ARGON_POWERV</td>
<td>Argon with power-law viscosity for MNM.</td>
<td>23</td>
<td>int</td>
</tr>
<tr>
<td>EQ_AIR_TSP</td>
<td>Equilibrium Air, 1-specie, Tannehill equation of state.</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>EQ_CO2</td>
<td>Carbon-dioxide in chemical equilibrium.</td>
<td>9</td>
<td>int</td>
</tr>
<tr>
<td>EQ_N2</td>
<td>Nitrogen in chemical equilibrium (see n2eq.c).</td>
<td>7</td>
<td>int</td>
</tr>
<tr>
<td>IONIZE_AR_N2</td>
<td>Mix of ionizing argon (species 1) and inert nitrogen in vibrational equilibrium (species 0).</td>
<td>16</td>
<td>int</td>
</tr>
<tr>
<td>LOWT_AIR_14</td>
<td>Low temperature air, GAMMA = 1.4, fudged Sutherland constants.</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>LUT</td>
<td>Single species gas with a look-up-table for thermodynamic and transport properties. The simulation programs expect to find a binary version of the table in a file called 'lut.dat' in your current directory.</td>
<td>99</td>
<td>int</td>
</tr>
<tr>
<td>LUT2</td>
<td>Two species gas with look-up-tables for thermodynamic and transport properties. The simulation programs expect to find binary versions of the tables in files called 'lut-0.dat' and 'lut-1.dat'. If any particular table is not present, the programs then try 'lut-0.dat' and finally 'lut.dat' before giving up.</td>
<td>98</td>
<td>int</td>
</tr>
<tr>
<td>LUT_MIX</td>
<td>One LUT gas with a mix of 4 ideal gases. The species are 0=LUT, 1=argon 2=helium, 3=nitrogen and 4=air.</td>
<td>96</td>
<td>int</td>
</tr>
<tr>
<td>LUTN</td>
<td>Multiple-species (actually 5 species) gas with look-up-tables for all species. The simulation programs expect to find binary versions of the tables in files called 'lut-0.dat' through 'lut-4.dat'. If any particular table is not present, the programs then try 'lut-0.dat' and finally 'lut.dat' before giving up.</td>
<td>97</td>
<td>int</td>
</tr>
<tr>
<td>MULTI_T_GAS</td>
<td>Rowan's gas mixture with multiple temperatures translational + vibrational + electrons. Look in your 'species.dat' file for details.</td>
<td>26</td>
<td>int</td>
</tr>
<tr>
<td>NIT0</td>
<td>Another nitrogen model for MNM.</td>
<td>18</td>
<td>int</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIT1</td>
<td>Another nitrogen model for MNM.</td>
<td>Value: 19 (type=int)</td>
<td></td>
</tr>
<tr>
<td>NIT2</td>
<td>Another nitrogen model for MNM.</td>
<td>Value: 20 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_AIR_13</td>
<td>Perfect gas, air, GAMMA = 1.3.</td>
<td>Value: 2 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_AIR_14</td>
<td>Perfect gas, air, GAMMA = 1.4.</td>
<td>Value: 0 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_AR_167</td>
<td>Perfect gas Argon with GAMMA = 1.667.</td>
<td>Value: 6 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_AR_AIR</td>
<td>Mix of argon (species 1) and air (species 0), both as perfect gases.</td>
<td>Value: 15 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_CO2</td>
<td>Carbon-dioxide – perfect gas.</td>
<td>Value: 10 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_GAS_MIX</td>
<td>Rowan's perfect gas mix of species defined in 'species.dat'</td>
<td>Value: 22 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_HE_167</td>
<td>Perfect gas, Helium, GAMMA = 1.667.</td>
<td>Value: 3 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_HE_AIR</td>
<td>Perfect gas mix of helium (species 1) and air (species 0).</td>
<td>Value: 12 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_HE_N2</td>
<td>Perfect gas mix of helium (species 1) and nitrogen (species 0).</td>
<td>Value: 11 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_N2</td>
<td>Nitrogen – perfect gas.</td>
<td>Value: 8 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_N2_LOWG</td>
<td>Perfect gas nitrogen with low GAMMA to simulate high T.</td>
<td>Value: 17 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_NE_H</td>
<td>Perfect gas mix of neon atoms (species 0) and hydrogen atoms (species 1).</td>
<td>Value: 25 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_NE_H2</td>
<td>Perfect gas mix of neon (species 0) atoms and hydrogen (species 1) molecules.</td>
<td>Value: 24 (type=int)</td>
<td></td>
</tr>
<tr>
<td>PERF_R22_AIR</td>
<td>Perfect gas mix of R22 (species 1) and air (species 0).</td>
<td>Value: 21 (type=int)</td>
<td></td>
</tr>
<tr>
<td>VIBEQ_N2</td>
<td>Nitrogen molecules with vibrational equilibrium.</td>
<td>Value: 13 (type=int)</td>
<td></td>
</tr>
<tr>
<td>VIBEQ_N2_HE</td>
<td>Mix of helium (species 1) with nitrogen in vib. equilibrium (species 0).</td>
<td>Value: 14 (type=int)</td>
<td></td>
</tr>
<tr>
<td>WEIRD_167</td>
<td>Weird air with GAMMA = 1.667 to match the DSMC satellite simulations.</td>
<td>Value: 5 (type=int)</td>
<td></td>
</tr>
</tbody>
</table>
Index

cns_bc_defs (module), 33–34

flow_condition (module), 32
  FlowCondition (class), 32
    __deepcopy__ (method), 32
    __init__ (method), 32
flux_dict (module), 35

gas_dict (module), 36–37
gemom (module), 18–21
  cross (function), 18
  distance_between_nodes (function), 18
  dot (function), 18
Node (class), 18–19
  __init__ (method), 18
  copy (method), 18
  translate (method), 19
Vector (class), 19–21
  __abs__ (method), 20
  __add__ (method), 20
  __div__ (method), 20
  __init__ (method), 20
  __mul__ (method), 20
  __neg__ (method), 20
  __pos__ (method), 20
  __rdiv__ (method), 20
  __rmul__ (method), 20
  __sub__ (method), 20
  sum (method), 20
  unit (method), 20
gpath (module), 22–31
Arc (class), 22–23
  __init__ (method), 23
  eval (method), 23
  length (method), 23
  translate (method), 23
Arc3 (class), 23–24
  __init__ (method), 24
Bezier (class), 24–25
  __init__ (method), 24
  eval (method), 24
  length (method), 24
  translate (method), 25
ClosedSurfacePatch (class), 25–27
  __init__ (method), 26
  extrude (method), 26
  interpolate_TFI (method), 26
Edge3D (class), 27
  __init__ (method), 27
  copy (method), 27
  interpolate_TFI2D_mbcns (function), 22
interpolate_TFI3D (function), 22
Line (class), 27–28
  __init__ (method), 28
  eval (method), 28
  length (method), 28
  translate (method), 28
Polyline (class), 28–30
  __init__ (method), 29
  append (method), 29
  copy (method), 29
  eval (method), 29
  length (method), 29
  nelements (method), 30
  translate (method), 30
Spline (class), 30–31
  __init__ (method), 31
scriptit (module), 6–17
  Block2D (class), 7–9
    __init__ (method), 8
  set_BC (method), 8
  connect_blocks (function), 6
Face2D (class), 9–11
  __init__ (method), 10
  copy (method), 10
GlobalData2D (class), 11–13
MetapostEnvironment (class), 13–14
  origin (method), 14
  scales (method), 14
  xaxis (method), 14
  yaxis (method), 14
MultiBlock2D (class), 14–17
  __init__ (method), 16
  rad_to_degrees (function), 6