View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Queensland eSpace

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 (*i.e.* $t = 0$) 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 **2**

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 . . . 20
3.3.2 Properties . . . 21

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

4.9.1 Methods 31
4.9.2 Properties 31

5 Module flow_condition **32**
5.1 Class FlowCondition 32
5.1.1 Methods 32

6 Module cns_bc_defs **33**
6.1 Variables 33

7 Module flux_dict **35**
7.1 Variables 35

8 Module gas_dict **36**
8.1 Variables 36

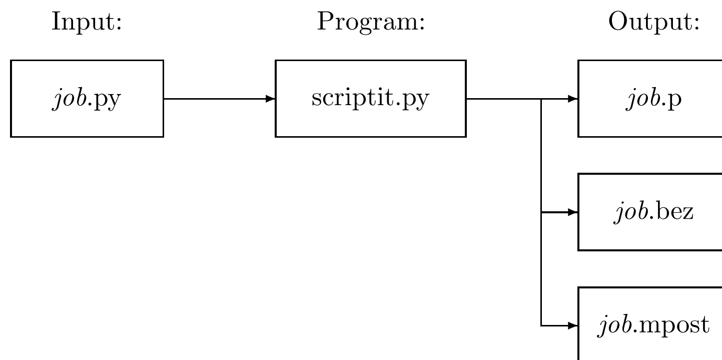
Index **38**

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].

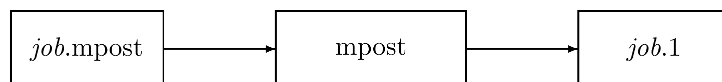
Once you have prepared your flow specification file, the simulation data is generated in a number of stages:

1a. Create the geometry definition with the command.
`$ scriptit.py -f job --do-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. It's 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.

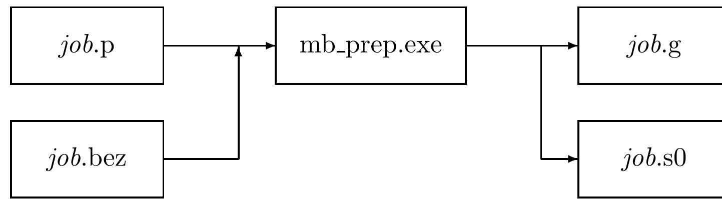
1b. Check the geometry definition (visually) by using Metapost to make a viewable postscript file containing labelled nodes, block boundary curves and blocks. Metapost is distributed as part of the TEX document preparation system. It is most likely already installed on your UNIX/Linux/Cygwin system and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



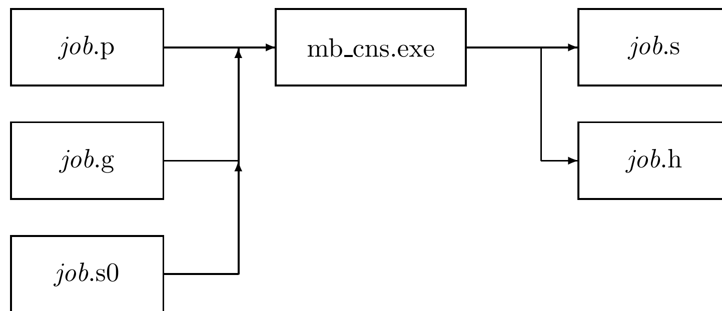
The *job*`.1` file contains a postscript rendering of your flow geometry that can be used for your documentation. It may also help in debugging your input script.

2. Generate an initial (*i.e.* $t = 0$) flow solution in file *job*.s0 and a corresponding grid file *job*.g.
`$ mb_prep.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.
[2]See http://www.mech.uq.edu.au/cfcfd/code/mb_cns/doc/.



3. Run the simulation code to produce flow data at subsequent times.
   `$ 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 `Node`s, simple path elements (such as `Line`, `Arc` and `Bezier` elements) and, possibly, compound path elements (such as `Spline`s and `Polyline`s). 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)

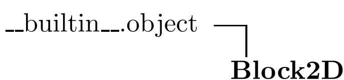
Return Value

angle in degrees

2.2 Variables

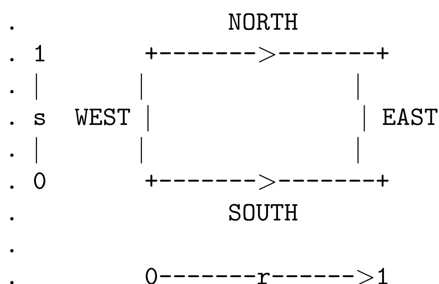
Name	Description
AO	Constant indicating use of the area-orthogonality grid generator. **Value:** `1` *(type=`int`)*
EAST	Constant indicating an east face. **Value:** `1` *(type=`int`)*
gdata	Contains the `GlobalData2D` information describing the simulation. Note that there is one such variable set up 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 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 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

__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

Name	Description
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

```
__builtin__.object ─┐
                    Face2D
```

Contains the information for one face of a block.

2.4.1 Methods

__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__

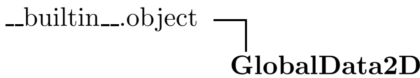
copy(*self*, *direction*=1)

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

2.4.2 Class Variables

Name	Description
nmin	Minimum number of cells along any face and, correspondingly, across any block. **Value:** 2 *(type=int)*

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

Name	Description
gas_name	The (string) name of the gas.

2.5.2 Instance Variables

Name	Description
axisymmetric_flag	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. **Value:** `<member 'axisymmetric_flag' of 'GlobalData2D' objects>` *(type=int)*
case_id	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. **Value:** `<member 'case_id' of 'GlobalData2D' objects>` *(type=int)*
cfl	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. **Value:** `<member 'cfl' of 'GlobalData2D' objects>` *(type=float)*
displacement_thickness	See Andrew Denman. **Value:** `<member 'displacement_thickness' of 'GlobalData2D' objects>` *(type=float)*
dt	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. **Value:** `<member 'dt' of 'GlobalData2D' objects>` *(type=float)*
dt_av	See Andrew Denman. **Value:** `<member 'dt_av' of 'GlobalData2D' objects>` *(type=float)*

continued on next page

Name	Description
dt_history	Period (in seconds) between writing the data for the selected cells to the history file. **Value:** `<member 'dt_history' of 'GlobalData2D' objects>` *(type=float)*
dt_plot	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. **Value:** `<member 'dt_plot' of 'GlobalData2D' objects>` *(type=float)*
flux_calc	Specifies the form of flux calculation at cell interfaces. See module `flux_dict.py` for options. **Value:** `<member 'flux_calc' of 'GlobalData2D' objects>` *(type=int or string)*
gas_name	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`.
max_step	Time stepping will be terminated if the simulation reached this number of steps. **Value:** `<member 'max_step' of 'GlobalData2D' objects>` *(type=int)*
max_time	The (simulation) time (in seconds) at which time stepping should be terminated. **Value:** `<member 'max_time' of 'GlobalData2D' objects>` *(type=float)*
perturb_flag	See Andrew Denman. **Value:** `<member 'perturb_flag' of 'GlobalData2D' objects>` *(type=int)*
perturb_frac	See Andrew Denman. **Value:** `<member 'perturb_frac' of 'GlobalData2D' objects>` *(type=int)*
reacting_flag	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. **Value:** `<member 'reacting_flag' of 'GlobalData2D' objects>` *(type=int)*
t_order	Specifies the form of time stepping scheme. Select 1 for Euler stepping. Select 2 for predictor-corrector stepping. **Value:** `<member 't_order' of 'GlobalData2D' objects>` *(type=int)*
tav_0	See Andrew Denman. **Value:** `<member 'tav_0' of 'GlobalData2D' objects>` *(type=float)*
tav_f	See Andrew Denman. **Value:** `<member 'tav_f' of 'GlobalData2D' objects>` *(type=float)*

continued on next page

Name	Description
time_average_flag	See Andrew Denman. **Value:** `<member ’time_average_flag’ of ’GlobalData2D’-objects>` *(type=int)*
title	A piece of text that will be propagated through the solution files and subsequently generated plots. **Value:** `<member ’title’ of ’GlobalData2D’ objects>` *(type=string)*
turbulent_flag	Andrew Denman. **Value:** `<member ’turbulent_flag’ of ’GlobalData2D’ objects>` *(type=int)*
viscous_delay	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). **Value:** `<member ’viscous_delay’ of ’GlobalData2D’ objects>` *(type=float)*
viscous_flag	Set to 1 to activate viscous transport terms. Set to 0 (the default) for inviscid flow simulation. **Value:** `<member ’viscous_flag’ of ’GlobalData2D’ objects>` *(type=int)*
x_order	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 higer-order (limited quadratic) reconstruction. **Value:** `<member ’x_order’ of ’GlobalData2D’ objects>` *(type=int)*

2.6 Class 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 postsript 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

__builtin__.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:

```
.                       North
.   1       +-------------+-------------+
.   |       |   [0][1]    |   [1][1]    |
.   s  West +-------------+-------------+ East
.   |       |   [0][0]    |   [1][0]    |
.   0       +-------------+-------------+
.                       South
.           0           --r-->          1
```

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

__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

Name	Description
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)*

3 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*)

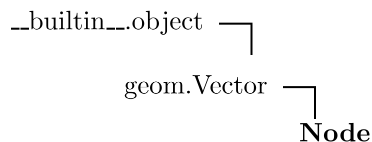
Return the distance between Nodes a and b.

dot(*a*, *b*)

Vector dot product.

Return Value
scalar product.

3.2 Class Node



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__ extit(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

Name	Description
Inherited from Vector: x *(p. 19)*, y *(p. 19)*, z *(p. 19)*	

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

__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: __builtin__.object.__init__

__abs__(*self*)

Absolute value is the magnitude of the Vector.

__add__(*self*, *other*)

Element by element addition.

__div__(*self*, *other*)

__mul__(*self*, *other*)

Element-by-element multiplication.

__neg__(*self*)

Negation of all elements for -x.

__pos__(*self*)

+x

__rdiv__(*self*, *other*)

__rmul__(*self*, *other*)

__sub__(*self*, *other*)

Element-by-element subtraction.

sum(*self*)

Add elements together.

unit(*self*)

Return Value

unit vector with same direction as this vector.

3.3.2 Properties

Name	Description
label	text label for the vector
x	x-coordinate of the vector
y	y-coordinate of the vector
z	z-coordinate of the vector

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_TFI_2D_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_TFI_3D(*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

```
__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.

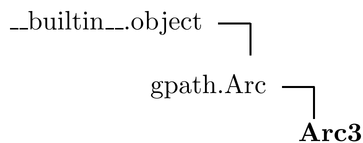
Parameters

`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



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

4.3.1 Methods

__init__(*self*, *a*, *b*, *c*)

Parameters

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

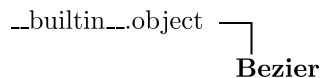
`b`: Intermediate point on arc.
(type=`Node` object)

`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



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

__init__(*self*, *B*)

Defines a Bezier polynomial of order N=len(B)-1.

Parameters

`B`: The control nodes of the curve, B[0] –> B[-1].
(type=list of `Node` objects)

Overrides: __builtin__.object.__init__

eval(*self*, *t*)

Locate a point on the Bezier curve.

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 Bezier curve.

Note: This is obtained approximately by sampling the curve.

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

Displacee 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)

4.4.2 Instance Variables

Name	Description
N	The order of the curve is N = len(B)-1. *(type=int)*

4.5 Class ClosedSurfacePatch

__builtin__.object
└─ **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)

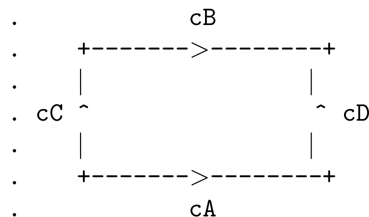
`cC`: 'West' curve
(type=`Polyline`-derived object)

`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.

interpolate_TFI(*self*, *r*, *s*)

Locate a point on the ClosedSurfacePatch.

Parameters

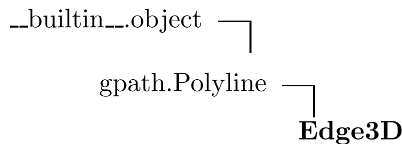
`r`: interpolation parameter for along curves cA and cB
(type=float, 0.0<=r<=1.0)

`s`: interpolation parameter for along curves cC and cD
(type=float, 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

4.7 Class Line

__builtin__.object ─┐
Line

Defines a straight-line segment.

4.7.1 Methods

__init__(*self*, *a*, *b*)

Define the directed line from `Node` a to `Node` b.

Parameters

`a:` Starting point on line.
(type=`Node` object)

`b:` Finishing point on line.
(type=`Node` object)

Overrides: __builtin__.object.__init__

eval(*self*, *t*)

Locate a point on the line.

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 line.

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.
(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 Class 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)

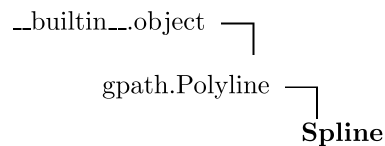
`dy`: y-component of displacement (if dx was a scalar)
(type=float)

`dz`: z-component of displacement (if dx was a scalar)
(type=float)

4.8.2 Properties

Name	Description
t0	Lower bound for subrange.
t1	Upper bound for subrange.

4.9 Class Spline



Defines a cubic-spline path.

4.9.1 Methods

__init__(*self*, *B*)

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

Parameters

`B`: interpolation points, B[0] –> 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

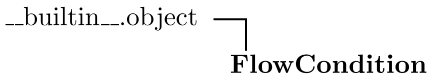
Name	Description
Inherited from Polyline: t0 *(p. 29)*, t1 *(p. 29)*	

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 class to organise the setting of each flow condition.

5.1.1 Methods

__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 with 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.

__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

Name	Description
ADIABATIC	A solid, no-slip wall without heat transfer. (i.e. the near-wall temperature is reflected in the ghost cells) **Value:** `4` *(type=int)*
ADJACENT	This boundary joins that of another block. Normally, this boundary condition would be set implicitly when making block connections. **Value:** `0` *(type=int)*
COMMON	Synonym for `ADJACENT`. **Value:** `0` *(type=int)*
EXTRAPOLATE_OUT	Extrapolate all flow properties from just inside the boundary into the ghost-cells outside the boundary. This works fine for a strong supersonic outflow. **Value:** `2` *(type=int)*
FIXED_P_OUT	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. **Value:** `11` *(type=int)*
FIXED_T	A solid, no-slip wall with a user-specified temperature. **Value:** `5` *(type=int)*
RRM	Andrew Denman's recycled and renormalised boundary condition. **Value:** `12` *(type=int)*
SLIP	Synonym for `SLIP_WALL` **Value:** `3` *(type=int)*
SLIP_WALL	A solid but inviscid wall. Effectively, this boundary condition copies reflects the properties just inside the boundary into the ghost cells. **Value:** `3` *(type=int)*
STATIC_PROF	A steady inflow boundary with a variable set of flow conditions across the boundary. **Value:** `10` *(type=int)*
SUB_OUT	Synonym for `SUBSONIC_OUT` **Value:** `7` *(type=int)*
SUBSONIC_IN	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. **Value:** `6` *(type=int)*

continued on next page

Name	Description
SUBSONIC_OUT	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.) **Value:** `7` *(type=int)*
SUP_IN	Fully-prescribed inflow (e.g. supersonic inflow). **Value:** `1` *(type=int)*
SUP_OUT	Synonym for `EXTRAPOLATE_OUT`. **Value:** `2` *(type=int)*
TRANSIENT_UNI	An transient inflow boundary which has a uniform flow condition applied across the full boundary. **Value:** `8` *(type=int)*

7 Module flux_dict

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

7.1 Variables

Name	Description
ADAPTIVE	A switched AUSMDV/EFM scheme that uses EFM near shocks and AUSMDV elsewhere. This is a good all-rounder for shock-tunnel work. **Value:** `4` *(type=int)*
AUSM	M. S. Liou's AUSM approximate flux calculator. Fast but tends to be a bit noisy. **Value:** `1` *(type=int)*
AUSMDV	A version of Wada and Liou's AUSMDV scheme. **Value:** `3` *(type=int)*
EFM	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. **Value:** `2` *(type=int)*
RIEMANN	An exact Riemann-solver-based flux calculator. It is slow and only works for ideal gas models at the moment. **Value:** `0` *(type=int)*

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

Name	Description
ARGON_LJ	Ideal argon with Lennard-Jones viscosity. **Value:** `27` *(type=int)*
ARGON_POWERV	Argon with power-law viscosity for MNM. **Value:** `23` *(type=int)*
EQ_AIR_1SP	Equilibrium Air, 1-specie, Tannehill equation of state. **Value:** `4` *(type=int)*
EQ_CO2	Carbon-dioxide in chemical equilibrium. **Value:** `9` *(type=int)*
EQ_N2	Nitrogen in chemical equilibrium (see n2eq.c). **Value:** `7` *(type=int)*
IONIZE_AR_N2	Mix of ionizing argon (species 1) and inert nitrogen in vibrational equilibrium (species 0). **Value:** `16` *(type=int)*
LOWT_AIR_14	Low temperature air, GAMMA = 1.4, fudged Sutherland constants. **Value:** `1` *(type=int)*
LUT	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. **Value:** `99` *(type=int)*
LUT2	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. **Value:** `98` *(type=int)*
LUT_MIX	One LUT gas with a mix of 4 ideal gases. The species are 0=LUT, 1=argon 2=helium, 3=nitrogen and 4=air. **Value:** `96` *(type=int)*
LUTN	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. **Value:** `97` *(type=int)*
MULTI_T_GAS	Rowan's gas mixture with multiple temperatures translational + vibrational + electrons. Look in your 'species.dat' file for details. **Value:** `26` *(type=int)*
NIT0	Another nitrogen model for MNM. **Value:** `18` *(type=int)*

continued on next page

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

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
- geom *(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_TFI_2D_mbcns *(function)*, 22
  - interpolate_TFI_3D *(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