The Eilmer3 Code: User Guide and Example Book.

Mechanical Engineering Report 2008/07
P. A. Jacobs*and R. J. Gollan†
with contributions‡from a cast of many, including:
Peter Blyton, Arianna Bosco, Djamel Boutamine, Laurie Brown, David Buttsworth,
Wilson Chan, Sam Chiu, Chris Craddock, Brian Cook, Jason Czapla,
Carlos de Miranda-Ventura, Andrew Denman, David Gildfind, Richard Goozeé, Stefan Hess,
Carolyn Jacobs, Ian Johnston, Ojas Joshi, Rainer Kirchhartz, Matt McGilvray,
David Mee, Luke Montgomery, Jan-Pieter Nap, Brendan O'Flaherty, Paul Petrie-Repar,
Dan Potter, Deepak Ramanath, Michael Scott, Umar Sheikh, Ben Stewart,
Joseph Tang, Katsu Tanimizu, Paul van der Laan, Jaidev Vesudevan, Mike Wendt,
Vince Wheatley, Adrian Window, Hannes Wojciak, Fabian Zander

December 1, 2011

Preface

Eilmer3 is an integrated collection of programs for the simulation of transient, compressible flow in two and three spatial dimensions. It provides a preparation program that can be used to set up a database of simulation parameters, a block-structured grid defining the flow domain and an initial flow field. These items are then used as a starting point for the main simulation program which computes a series of snapshots of the evolving flow. Eilmer3 is part of the larger collection of compressible flow simulation codes found at `http://www.mech.uq.edu.au/cfcfd/`.

This user guide contains a collection of example simulations: scripts, results and commentary. It may be convenient for new users of the code to identify an example close to the situation that they wish to model and then adapt the scripts for that example.

*Queensland Geothermal Energy Centre of Excellence, The University of Queensland, Brisbane, Australia.
†Centre for Hypersonics, The University of Queensland, Brisbane, Australia.
‡These contributions have come in the form of examples, debugging, proof-reading and constructive comments on the codes and this document, and code for special cases.

Contents

I User guide 9

1 Introduction 9

2 Building and installing the programs 10

3 Running simulations 10
3.1 Data preparation (with e3prep.py) . . . 11
3.2 Running the simulation (with e3shared.exe) . . . 13
3.3 Running the simulation in parallel (e3mpi.exe) . . . 14
3.4 Restarting a simulation . . . 14
3.5 Postprocessing (with e3post.py) . . . 15
3.6 Supervisory GUI . . . 18

4 Input Script Overview 19

5 Thermochemical model and flow conditions 21
5.1 10 second version: just tell me how to select perfect air . . . 21
5.2 2 minute version: tell me about other simple models . . . 21
5.3 10 minute version: the detail of gas model configuration . . . 22
5.4 Selecting a simple model and adjusting it . . . 23
5.5 Specifying chemically reacting flow . . . 24
5.6 Defining flow conditions . . . 24
5.7 Using flow conditions from other simulations . . . 25
5.8 Using mole fractions and species dictionaries . . . 26

6 Boundary representation of the gas domain 27
6.1 Geometric elements . . . 27
6.1.1 Paths . . . 28
6.1.2 Surfaces . . . 29
6.1.3 Volumes . . . 30
6.2 Two-dimensional grids . . . 31
6.3 Three-dimensional grids . . . 36

7 Setting conditions for individual boundaries 41

8 Special zones and history points 42

9 Simulation control parameters 43

10 Parameters for a 2D sketch of the flow domain 47

II Examples for 2D flow 49

11 Mach 1.5 flow over a 20° cone – Simple boundaries **50**
11.1 Input script (.py) . . . 53
11.2 Shell scripts . . . 54
11.3 More specialized postprocessing . . . 54
11.4 Notes . . . 57

12 Viscous Flow Along a Cylinder **59**
12.1 Input script (.py) . . . 61
12.2 Shell scripts . . . 62
12.3 Notes . . . 62

13 Mach 3 flow over a sharp-nosed two-dimensional body **63**
13.1 Input script (.py) . . . 66
13.2 Shell scripts . . . 67
13.3 Notes . . . 67

14 Sharp-nosed 2D body – PyFun version **69**
14.1 Input script (.py) . . . 69
14.2 Notes on using Python for the input script . . . 70

15 Hypersonic flow of ideal air over a blunt wedge **71**
15.1 Input script (.py) . . . 74
15.2 Shell scripts . . . 76
15.3 Notes . . . 77

16 Pressure on a flat-faced cylinder **79**
16.1 Input script (.py) . . . 81
16.2 Shell scripts . . . 82
16.3 Awk scripts . . . 83
16.4 Notes . . . 83

17 Flow through a conical nozzle **85**
17.1 Input script (.py) . . . 88
17.2 Shell scripts . . . 89
17.3 Notes . . . 91

18 Flow of equilibrium air over a sphere **93**
18.1 Input script (.py) . . . 95
18.2 Shell scripts . . . 97
18.3 Notes . . . 98

19 Heat transfer to a sphere in equilibrium air **99**
19.1 Template input script (.py) . . . 104
19.2 Coordinating script (.py) . . . 105
19.3 Shell script for postprocessing . . . 109
19.4 Notes . . . 110

20 Dissociating N_2 flow over a 2D cylinder **111**
20.1 Input script (.py) . . . 113
20.2 Reaction scheme file (.lua) . . . 114
20.3 Shell scripts . . . 114
20.4 Notes . . . 115

21 Flow of detonable mixture over a sphere **117**
21.1 Input script (.py) . . . 119
21.2 Reaction scheme file (.lua) . . . 123
21.3 Notes . . . 126

22 MNM implosion problem **127**
22.1 Input script (.py) . . . 129
22.2 Shell scripts . . . 130
22.3 Notes . . . 130

23 Periodic Shear Layer **131**
23.1 Input script (.py) . . . 133
23.2 Shell scripts . . . 134
23.3 Notes . . . 134

24 Mach 1.5 flow over a 20° cone – UDF boundaries **135**
24.1 Input script (.py) . . . 138
24.2 Boundary-condition files (.lua) . . . 139
24.3 Shell scripts . . . 142
24.4 Notes . . . 142

25 A section of an ideal compressible-flow vortex **143**
25.1 Input script (.py) . . . 145
25.2 Boundary condition file (.lua) . . . 145
25.3 Shell scripts . . . 147
25.4 Notes . . . 147

26 Method of manufactured solutions – Euler flow **151**
26.1 Input script (.py) . . . 152
26.2 Boundary condition file (.lua) . . . 152
26.3 Source term file (.lua) . . . 154
26.4 Shell scripts . . . 156
26.5 Python reference-function files . . . 157
26.6 Notes . . . 158

27 Oblique detonation wave **159**
27.1 Input script (.py) . . . 160
27.2 gas-model file (binary-gas.lua) . . . 161
27.3 Source term file (.lua) . . . 163
27.4 Shell scripts . . . 163
27.5 Python reference function files . . . 164
27.6 Notes . . . 168

28 Subsonic compressor blade – sc10 **169**
28.1 Input script (.py) . . . 169
28.2 Boundary-condition files (.lua) . . . 177
28.3 Shell scripts . . . 179
28.4 Notes . . . 179

29 Subsonic compressor blade – PyFun version **181**
29.1 Input scripts (.py) . . . 181
29.2 Notes . . . 188

III Examples for 3D flow **189**

30 Mach 1.5 flow over a 10° ramp **190**
30.1 Input script (.py) . . . 192
30.2 Shell script . . . 193
30.3 Postprocessing program . . . 193
30.4 Notes . . . 194

31 Sod shock tube problem in 3D **195**
31.1 Input script (.py) . . . 196
31.2 Shell script . . . 197
31.3 Notes . . . 197

32 Injection of hydrogen into a nitrogen stream **199**
32.1 Input script (.py) . . . 201
32.2 Shell script . . . 202
32.3 Notes . . . 203

33 Flow of nitrogen over a cylinder of finite length **205**
33.1 Input script (.py) . . . 208
33.2 Reaction scheme file (.lua) . . . 210
33.3 Shell script . . . 211
33.4 Postprocessing program . . . 211
33.5 Notes . . . 212

34 Spherically-blunted cone **213**
34.1 Input script (.py) . . . 214

35 Katsu's scramjet combustor and nozzle **217**
35.1 Input script (.py) . . . 219

36 Titan aeroshell using imported grids **221**
36.1 Input script (.py) . . . 222

IV References and Appendices **225**

A README.txt: Instructions for installation **227**

B Surviving the Linux Command Line **231**

C Just enough Python to be dangerous **233**

D Make your own debugging cube **237**

E Notes on running MPI jobs on cluster computers **239**
E.1 The Blackhole (SUN Rack) cluster 239
E.2 The barrine.hpcu.uq.edu.au SGI cluster 240

F cfpylib modules **243**
F.1 Billig shock shape correlation 243

G Gas models: specification by configuration file **245**
G.1 User-defined gas model 245
G.1.1 An example minimal user-defined gas model 250
G.2 Equilibrium gas based on a look-up table 251
G.2.1 Selecting a look-up table for the gas model 252
G.2.2 Building your own look-up table 253

H Chemical reactions: specification by configuration file **255**
H.1 Overview of input file format 256
H.2 Details of the `reaction` table 257
H.3 Extra control of the chemistry scheme 259

I User-defined functions for run-time customization **263**
I.1 Boundary conditions 263
I.2 Source terms 265

J Hints for Solution Visualisation with ParaView **267**
J.1 Plotting Streamlines and Streamtubes 267
J.2 Moving Blocks 268

Part I
User guide

1 Introduction

Eilmer3 is an integrated collection of programs for the simulation of transient, compressible flow in two and three dimensions. It provides a preparation program (`e3prep.py`) that can be used to set up a database of simulation parameters, a block-structured, body-fitted grid defining the flow domain and an initial flow field. These items are then used as a starting point for the main simulation program (`e3shared.exe`) which computes a series of snapshots of the evolving flow. Finally, a rudimentary but versatile postprocessing program (`e3post.py`) makes the flow data available for further analysis.

Eilmer3 is a derivative of the code mbcns2 which, in turn was an experiment in writing the mb_cns code in C++. Once it was determined that there were clear benefits in using C++, our three-dimensional flow code Elmer was then reworked in C++ as Elmer2. At the same time, we experimented with using the Python language for the user's input script and embedding the Lua language in order to make some of the boundary conditions programmable. Of course, these codes being experiments in C++, we soon decided that it could all be done much more cleanly and be made much more versatile if we just reworked some of the basic modules. Thus, the thermochemistry was reworked and the separate two and three dimensional codes merged into Eilmer3. The name change is to avoid a naming clash with the Elmer finite-element code from Finland.[1]

The following sections provide example input scripts and shell scripts for a number of simulations. These are intended to be starting points for your own simulations and should be studied together with the other manuals that can be found in the documentation section of the Compressible Flow CFD Group web site: `http://www.mech.uq.edu.au/cfcfd/`. Study these scripts carefully; some of the interesting bits of the documentation are embedded within them.

For a description of the methods coded into `Eilmer3`, see the companion report [1] which covers the gas-dynamic formulation and the basic thermochemistry components.

[1] http://www.csc.fi/elmer

2 Building and installing the programs

The core solver and its modules are mainly written in C/C++ for speed and the benefits of compile time checking. The pre- and post-processing programs are mainly Python so that we get flexibility and convenient customization. There is also a little Tcl/Tk and Lua.

Our main development environment is Linux but the programs can be deployed on Linux, flavours of Unix such as MacOS-X and MS-Windows (using Cygwin). The main requirement is that the C/C++ compilers, the Tcl and Python interpreters be available, along with their supporting libraries and various extensions. The source code of the Lua interpreter is included in with Eilmer3. The `README.txt` file (Appendix A) or the corresponding `README.html` file in the root directory of the source code provides more detail including the actual commands needed to build and install the programs.

If you are not accustomed to working with Unix/Linux, have a look at Appendix B for a brief introduction to working on the command line.

3 Running simulations

Setting up a simulation is mostly an exercise in writing a text-based description of your flow and its bounding geometry. This *input script* is presented to the preparation program as a Python source file, often with the extension ".py". Once you have prepared your flow specification as an input script using your favourite text editor, the simulation data is generated by the `Eilmer3` programs in a number of stages:

1 Create the geometry definition, a grid and the initial flow state. For simple to moderately complex geometries, the built-in geometry tools (described later in this manual) are adequate. For complex geometries, you may find it convenient to import block-structured grids, possibly from a specialized gridding tool such as `Gridgen` or `ICEMCFD`.

2 Run the simulation code to produce flow data at subsequent times.

3 Reformat the flow solution data to produce files suitable for a data viewing program such as Paraview or GNU-Plot.

3.1 Data preparation (with e3prep.py)

Create the geometry definition and a grid with the command

`$ e3prep.py --job=`*job* `--do-svg`

The italics word *job* in the command should be replaced by whatever job name that you have chosen. That name is then used as a base to derive specific names for each of the files associated with the simulation. At a minimum, you have an input script called *job*`.py` with the `.py` extension, indicating that the script is written in Python. The files from the preparation stage are:

- *job*`.config`: A database of configuration parameters in INI format. Parameters are specified, one per line, as *parameter-name* = *value*. A hierarchical structure is given to the set of parameters via named subsections in the file. Although you would probably never assemble one of these parameter files from scratch manually, it is sometimes convenient to alter a value or two and rerun a simulation without invoking `e3prep.py`.

- *job*`.control`: A small database of parameters to control the time-stepping, the final time, and the intervals between writing of solutions and history data. The content of this file is also in INI format and it is parsed at the start of every time step. This way, a user can alter the simulation behaviour (by editing this file) without having to restart the simulation. To stop a simulation cleanly, set the `halt_now` entry to 1. Other control parameters are marked with ‡ in Section 9.

- *job*`.times`: A mapping of time stamps to actual times at which the simulation data was written. After the preparation stage, there should be only the zero-time entry.

- *job*`.svg` or *job*`.wrl`: Sometimes it is convenient to see a graphical representation of

the flow domain and boundary conditions. These options produce a SVG or VRML rendering of the block boundaries and the boundary-condition labels. The `--do-svg` will invoke the rendering of two-dimensional blocks to a scalable-vector-graphics file while `--do-vrml` will render three-dimensional blocks to a virtual-reality-modeling-language file. For two-dimensional simulations, the SVG file can be edited in a program such as `Inkscape` (`http://www.inkscape.org`) and the result used as part of your documentation for a particular simulation.

- *job*`.grid.b0000.t0000`, *job*`.grid.b0001.t0000` : The grid of finite-volume cells, one file for each block that defines part of the flow domain. The grids are written as plain text files in a relatively simple format. The spatial coordinates for points within each file are associated with cell vertices of the structured grid.[2]

- one flow-data file for each block: *job*`.flow.b0000.t0000`, *job*`.flow.b0001.t0000`, ... containing the initial flow state within each of the finite-volume cells. Look at the first couple of lines of a flow file to see what data elements are written for each cell. Variable names appear on the second line and units are `SI`.

Note that the grid and flow data files are written to subdirectories of the same names. The grid is written once (at time zero, subdirectory `grid/t0000/`) and the flow files are written to a new subdirectory (`flow/tnnnn/`) at each output time. This is to keep the main job directory clean and to allow easy copying or moving of individual solution times. Also, these files are stored in "gzip" format with a ".gz" extension by default.

[2]Note that, in recent versions of the programs, the grid and flow files are written to subdirectories within the job directory.

3.2 Running the simulation (with e3shared.exe)

Run the simulation code to produce flow data at subsequent times.[3]

```
$ e3shared.exe --job=job --run
```

The output files are:

- *job*`.flow.b`*nnnn*`.t`*mmmm*: The flow data for all cells at the times requested. As the simulation proceeds, whole-field solutions are written to new files with *nnnn* representing the block number and *mmmm* representing a time stamp. Look up the *job*`.times` file to see what time values belong to each time stamp (or tindx). Just as for the grid files, each flow solution file is written as a plain text file with a simple layout, not too different from the Tecplot point-format for a structured-block grid. In these files, the spatial coordinates of points within the file are associated with the cell centres.

- *job*`.hist.b`*nnnn*: Data at particular "history locations" and at times requested. This data is typically used to simulate the signals recorded by pressure and heat-transfer sensors mounted on model surfaces. When restarting a simulation, the program will append to existing history files rather than clobbering them. Note that, if you are running a simulation from the start multiple times, you will need to manually remove the history files before each run. The command ``rm -r ./hist/''

[3] If the simulation finishes too quickly (possibly without taking any steps at all), it may be that the initial time step size is too large and the calculation is unstable. One symptom of this is that the final value for `dt` is reported as being the excessively large value of `1e+6` seconds. Choose a suitably small value and try again.

should do the job.

- *job*`.times`: A mapping of time stamps to actual times at which the simulation data was written. The main simulation appends lines to this file. This file may assist when automating some of the postprocessing operations.
- *job*`.finish`: An INI-format file giving some information about the time-stepping parameters at the end of the simulation. These may be useful for starting a follow-on simulation.

For viscous simulations, surface heat flux and cell Reynolds number files are also written to the subdirectory `heat`. See the `--heat-flux-list` option in Section 3.5 for a hint at how to extract the data and then have a look in the data files to see what specific data has been captured.

3.3 Running the simulation in parallel (e3mpi.exe)

One can build and run the distributed-memory version of the program, `e3mpi.exe`, on computers with the MPI (Message Passing Interface) library[4] and runtime environment. The notes in Appendix A show how to build the Eilmer3 executable for OpenMPI. There are also some notes in Appendix E on batch commands and job output files for the Blackhole and Barrine clusters at UQ.

3.4 Restarting a simulation

By default, the simulation program picks up the flow solution for `tindx` equal to 0 but it can be told to pick up any other `tindx` snapshot. To pick up a solution and continue, it is probably best to do a little house-keeping with the command

```
e3post.py --job=name --prepare-restart
```

This renames the 9999 flow files and tidies up the *job*`.times` file to reflect the changes. Then you should edit the *job*`.control` file and change the parameters `dt` `max_time` and `max_steps` to suitable values. Do not run `e3prep.py` again, else it will write all over your newly edited *job*`.control` file. At this point, you should be ready to run the main simulation program again. Remember to supply the relevant `tindx` value on the command line for your restart. For example:

```
e3shared.exe --job=name --tindx=5 --run
```

Also, with restarts, be careful that you have consistent modelling requirements and settings. Restarting a laminar simulation as a turbulent simulation with the $k - \omega$ model would lead to inconsistent data. It may be better to start a new job and use `ExistingSolution` objects[5] (see Section 5.7) to pick up the old data.

[4] See, for example, http://www.open-mpi.org/.

[5] We need to make `ExistingSolution` smart enough to fill in missing values.

3.5 Postprocessing (with e3post.py)

Postprocessing of the simulation data is the most unstructured of the simulation activities. We provide a postprocessing program, `e3post.py` that has the basic capabilities of picking up the simulation data and writing flow field files in formats suitable for `Paraview`, `Visit`, `Tecplot`, the venerable `Plot3D` or `gnuplot`[6].

To reformat the flow solution data into one unstructured grid containing all of the flow data for the domain and write this data in a format suitable for `Paraview` or `Visit`, use the command:

`$ e3post.py --job=`*`job`*` --vtk-xml --tindx=all`

The postprocessing program (`e3_post.py`) started as a fairly simple script that picked up solution data and reformatted it for plotting, however, it has continued to sprout features and has become a bit complex to describe. To see its command-line options, just run it without any options at all. It should then print a *usage* message which provides some hints. As at October 2009, the start of this message is:

```
Begin e3post.py...

Usage: e3post.py [--help] [--job=<jobFileName>] [--tindx=<index|all>]
                 [--zip-files|--no-zip-files]
                 [--vtk-xml] [--tecplot] [--plot3d] [--prepare-restart]
                 [--put-into-folders]
                 [--ref-function=<python-script>]
                 [--report-norms]
                 [--per-block-norm-list="jb,var-name,norm-name;..."
                 [--global-norm-list="var-name,norm-name;..."
                 [--compare-job=<jobFileName> [--compare-tindx=<index>]]
```

[6]See the web sites `http://www.paraview.org` `https://wci.llnl.gov/codes/visit/` `http://www.tecplot.com` `http://people.nas.nasa.gov/řogers/plot3d/intro.html` and `http://www.gnuplot.info`

```
[--output-file=<profile-data-file>]
[--slice-list="blk-range,i-range,j-range,k-range;..."]
[--slice-at-point="blk-range,index-pair,x,y,z;..."]
[--slice-along-line="x0,y0,z0,x1,y1,z1,N"]
[--surface-list="blk,surface-name;..."]
[--add-pitot-p] [--add-total-p] [--add-mach] [--add-total-enthalpy]
[--probe="x,y,z;..."]
[--heat-flux-list="blk-range,surf-range,i-range,j-range,k-range;..."]
```

The options can be combined in fairly complex ways; some experimentation on the part of the user may be required to get the desired effect.

- `--help` just prints the usage message. No other options are relevant.
- `--job=<jobFileName>` specifies the root name of the solution files
- `--tindx=<index|all` You may pick up one solution time via its index or you may specify all solution times via the keyword "`all`".
- `--zip-files|--no-zip-files` The default behaviour is to use gzipped files for the grid and flow data files, however, earlier version of the code used plain text files that were not zipped.
- `--vtk-xml` The XML format for the Visualization Tool Kit (VTK) is readable by both `Paraview` and `Visit`.
- `--tecplot` This produces an ASCII file that can be read by `Tecplot`.
- `--plot3d` This is also an ASCII format file that many visualization and flow simulation packages read and write. Two grid files are generated. The first, with `.grd` extension, is the true grid as used by the simulation with mesh location at the nodes. The second, with extension `.g`, has cell-centred values and accompanies the cell-centred values in the `.f` file.
- `--prepare-restart` does some house-keeping in the data files so that a simulation may be restarted cleanly. This is mainly dealing with the `9999` file and adjusting the `.times` file.
- `--put-into-folders` puts an old solution (which has its files all sitting in the current directory) into the current directory structure where the grid, flow and plot files have their own subdirectories.
- `--ref-function=<python-script>` compares the flow solution with a supplied Python function. The difference is output.
- `--report-norms` returns a dictionary of norms for all of the flow variables. The available norms are `L1`, `L2`, and `Linf` (maximum magnitude).

- `--per-block-norm-list="jb,var-name,norm-name;..."` returns the specified norms for particular variables and blocks. Sometimes just a little bit of information is required.

- `--global-norm-list="var-name,norm-name;..."` returns the specified norms, computed over the whole flow domain.

- `--compare-job=<jobFileName> [--compare-tindx=<index>]` compares one flow data set with another. The difference is output. This option combined with the computation of norms is a convenient way to check convergence in of a simulation.

- `--output-file=<profile-data-file>` specifies the name of a file in which to dump the requested data. This naming option is relevant to the various slice options and also to the the surface-list option where where it is used as the root name of the generated VTK files. This will allow you to make a number of sliced data sets for plotting.

- `--slice-list="blk-range,i-range,j-range,k-range;..."` allows one to extract subsets of the data. A Python-like slicing notation is used in the specification string which should be enclosed in quotes, as shown. Several slices (separated by semicolons) may be specified in the one string. Each slice specification consists of 4 indices or index ranges separated by commas. An index is a single integer value and may be negative to indicate counting from the end. A value of `-1` indicates the maximum value. An index range may be a colon-separated pair of integers, a colon and one limit or just a colon by itself (to indicate the full range). Note that the range limits are inclusive. So, for example, to extract the EAST strip of cells from block 0 in a 2D simulation, you would use the string `"0,-1,:,0"`.

- `--slice-at-point="blk-range,index-pair,x,y,z;..."` allows one to extract a slice/plane of data through a particular point. The index-pair is one of ij, jk or ki. The program sets these indices to zero and searches along the remaining index to find the cell nearest the specified (x,y,z) point. Once found, the slice over the index pair is selected for output (by adding it to the slice-list. Be aware that, for each block selected, slice-at-point will always select a slice to output, even if it is not very close. Again, use quotes to hold the string together as it passed through the shell interpreter.

- `--slice-along-line="x0,y0,z0,x1,y1,z1,N"` generates a list of `N` sampled points between the specified end points. The sampled data is taken from the nearest cell-centre for eash sample point. No higher-order interpolation is done.

- `--surface-list="blk,surface-name;..."` extracts a set of surfaces from the full flow field and writes them as VTK files. Sometimes we want convenient access to the bounding surfaces of the blocks. Use `NORTH`, `EAST`, `SOUTH`, `WEST`, `TOP` and `BOTTOM` as the surface names.

- `--add-pitot-p`, `--add-total-p`, `--add-mach` and `--add-total-enthalpy` add the named variable to the plotting data set, either for the full field (VTK, Tecplot and Plot3D format) or for sliced data. These flow variables are not in the Eilmer3 native flow solution file and must be reconstructed by `e3post.py`.

- `--probe="x,y,z;..."` reports the sampled data for the specified points. The selected data is written in gnuplot format.

- `--heat-flux-list="blk-range,surf-range,i-range,j-range,k-range;..."` [7] extracts surface heat flux and cell Reynolds number data. The syntax is the same as the `--slice-list` option except that the second argument is the boundary index (`NORTH`, `SOUTH`, `WEST`, `EAST`, `TOP` or `BOTTOM`). For 2D simulations, the block and boundary indices are sufficient to define the edge, so you can then leave the `i-range`, `j-range` and `k-range` arguments blank. For 3D simulations you would need to specify either `i`, `j` or `k` to get a single line of cells. For any range, it is sufficient to give just a colon to get the full range. For the surface range, the order of the boundary names comes into play with `NORTH`=0 and `BOTTOM`=5.

Note that you must use double-quotes on some specification strings to prevent the command shell from pulling the string apart (or otherwise changing it) before giving it to `e3post.py`.

Ad hoc postprocessing is possible by picking up the cell-centre flow data with your own custom postprocessing program written in Python. Two Python modules (`e3_flow.py` and `e3_grid.py`) are available for picking up individual blocks of data and storing selected flow properties in numpy arrays. Note that three-dimensional arrays are always used, even for two-dimensional simulations where the k-index has the single value 0. The examples that make up the bulk of this manual show some of the things that are possible. Two specific applications of writing a custom postprocessing script are the estimation of surface force on the 10° ramp case (Section 30) and finding the location of the bow shock for the finite cylinder simulation.

3.6 Supervisory GUI

To ease new-comers into the use of the codes, the `e3console.tcl` program provides a graphical view of the simulation process. It provides straight-forward automation of the

[7] Dan Potter's heat flux code writes the heat fluxes for a collection of surfaces. This was part of his PhD work.

Figure 1: Screen shot of the `e3console.tcl` GUI running on PJ's workstation.

simple case of running a simulation from scratch and then reformatting the entire flow-field data for plotting. Figure 1 shows the state of the GUI just after running the cone20 simulation. The Python input file is shown in the top text frame of the main window, with the log of the standard output from the simulation shown in the lower text frame. The tab for the postprocessor is visible in the "Options" window. It indicates that `e3post.py` will reformat all the flow data into the XML file format for the VTK plotting library (as used by Paraview). Also, note the text in the console window which shows the underlying commands that have been used.

4 Input Script Overview

Currently, `e3prep.py` is implemented as a Python program that has a library of classes specialized for constructing geometric regions and specifying flow conditions. Because your specification script, *`job`*`.py`, becomes a part of that program when it runs, it is worth the effort to learn just enough Python to be dangerous. The web site `http://www.python.org` is a good starting point for learning about the Python programming language, however, Appendix C may have enough information to get you started.

After doing some initialization, `e3prep.py` executes your script file and assembles the geometry and flow specification data into a form that can be given to the main

simulation code `e3shared.exe`[8]. The advantage of this approach is that you have the full capability of the Python interpreter available to you from within your script. You can perform calculations so that you can parameterize your geometry, for example, or you can use Python control structures to make repetitive definitions much more concise. Additionally, you may use Python comments and print statements to add documentation to the script file. An input script usually does the following:

1. selects gas model
2. optionally, creates geometric elements to assist in defining the boundary representation of the gas domain
3. creates blocks within the gas domain and specifies their discretization and, optionally, specifies boundary conditions along some block surfaces (in 3D) or edges (in 2D)
4. specifies remaining boundary conditions, if any
5. sets some simulation control parameters

Most examples in this manual do just these things, however, it is possible to do much more. The example that computes the heat transfer to a sphere (Section 19) uses a top-level Python script to coordinate a number of simulations with increasingly-refined grids as a crude multigrid simulation.

To aid with debugging, it is easy to process part of your input script and then temporarily put the interpreter into an interactive mode where you may type python commands and expressions at the prompt (`>>>`). To do so, add the following lines at the appropriate point in your input script.

```
from code import interact
interact('Start interactive mode (Ctrl-D to return)', local=locals())
```

Now you can interact with the Python environment and the objects that your input script has defined so far. For example, to find out a bit about defining `Block3D` objects, type:

```
>>> help(Block3D)
```

To get out of the interactive mode and continue processing the input script, type `Control-D` at the prompt.

[8]The "shared" tag indicated that we are using the shared-memory version of the code. There is also a distributed-memory version, `e3mpi.exe`, based on message passing (MPI) that can be used for running the main simulation.

5 Thermochemical model and flow conditions

The thermochemical models are provided by the `libgas` module. This is primarily a C++ module but it has a SWIG-generated Python interface so that its objects and methods can be accessed from the user's input script.

5.1 10 second version: just tell me how to select perfect air

Place the following text (which is a function call) in your script *before* specifying any `FlowCondition` objects:

```
select_gas_model(model='ideal gas', species=['air'])
```

If this is the only gas model that interests you for the present, then proceed to page 24 which discusses the specification of a `FlowCondition`.

5.2 2 minute version: tell me about other simple models

To select a gas model, the user calls the function `select_gas_model`. This function accepts three keyword arguments: `model`, `species`, and `fname`. In the vast majority of cases, only the first two keyword arguments will be used. This function must be called *before* specifying any `FlowCondition` objects so that the complete thermodynamic state can be computed.

A second example: to select an ideal mixture of nitrogen and oxygen call:

```
select_gas_model(model='ideal gas', species=['N2', 'O2'])
```

Note that the only difference between selecting a mixture and a single component gas is the addition of extra species in the species list and the extra computation that the main simulation program needs to do.

In general, the `model` keyword accepts a string describing the gas model behaviour. The available gas models are:

- `'ideal gas'`: a gas with ideal behaviour: modelled as having perfectly elastic collisions and constant specific heats
- `'thermally perfect gas'`: a gas with thermally perfect behaviour: modelled as having perfectly elastic collisions but with specific heats that are functions of temperature

The `species` keyword accepts a list of strings; each string denotes a species in the mixture. The order of this list is important: the order of species in this list corresponds

to the order in which the species mass fractions are specified in other parts of the input. To get a list of available species, look at the selection of species which are placed in the `$HOME/e3bin/species` area during the install, that is, at a command prompt type:

```
> ls $HOME/e3bin/species
```

The names of these files (excluding the `.lua` extension) correspond to the names of available species. The `defaults.lua` file is not a species name. Rather, this file provides a set of default values when no other data is available.

5.3 10 minute version: the detail of gas model configuration

In the last two examples, the `select_gas_model` function was called using the two keyword arguments `model` and `species`. Behind the scenes, this function calls an auxiliary set of tools to build a stand-alone text file which is a configuration file for the gas model. This configuration file is a Lua-style file: it is read directly by the C++ code (with embedded Lua interpreter) in order to configure the gas model. By default, the created configuration file is called `gas-model.lua`. This file will sit in your working directory after a successful call to `select_gas_model` using only the `model` and `species` keyword arguments. The configuration file contains all the necessary details to completely specify the gas. Thus, this file serves as a record of the gas model input parameters used in your simulation.

You are encouraged to open the file `gas-model.lua` and take a look. It contains not only the input parameters for the gas model but also references for the data where possible. Some amount of effort has been made to design a configuration file that properly documents the input data. The use of Lua as the configuration language has aided this effort.

Alternatively, the `select_gas_model` function may also be called with `fname` as a keyword argument. This argument, `fname`, accepts a string which names a Lua-style configuration file for the gas model. Thus, if you have a gas model configuration file from a previous simulation, you could set the gas model with the call:

```
select_gas_model(fname='gas-model.lua')
```

This assumes your configuration file is called `gas-model.lua` and resides in the same directory as your main simulation script.

Finally, for certain advanced gas models (like multi-temperature gas mixtures), the only means to configure these models is via the preparation of a Lua-style configuration file by hand. After building a file by hand (that is, in a text editor), one would use the

fname keyword argument in the call to select_gas_model to set the gas model. The list of gas models which are set by directly creating a configuration file are:

- user-defined gas (by specification of callable Lua functions)
- an equilibrium gas, based on a look-up table
- a reacting mixture of thermally perfect gases

Further discussion of gas models which are set by direct creation and manipulation of a configuration file is given in Appendix G.

5.4 Selecting a simple model and adjusting it

The simple ideal gas model of air as discussed above has $\gamma = 1.4$. You can get an air model with $\gamma = 1.3$ by selecting the species as 'air13' or you can adjust the value of γ directly for the ideal gas model. This can be done from within the Python input script by calling the function change_ideal_gas_attribute(), and telling it which species, which attribute and what new value to use. The function actually does a string substitution within the gas-model.lua file that was generated behind the scenes when the select_gas_model() function was called.

For an example of use, see the MNM Implosion problem in Section 22. There, the value of ratio of specific heats is changed with the lines

```
gas_gamma = 5.0/3.0
select_gas_model(model='ideal gas', species=['air'])
change_ideal_gas_attribute('air', 'gamma', gas_gamma)
```

You might also like to change the gas constant but, since that is not an actual parameter in the gas-model.lua file, it needs to be set indirectly, via the molecular mass (in units of kg/mol).

```
Rgas = 300.0
MM = R_u / Rgas
change_ideal_gas_attribute('air', 'M', Rgas)
```

Note that 'M' is the label for molecular mass in the gas-model.lua file and R_u is the universal gas constant made available by the thermochemistry module to the Python input script.

5.5 Specifying chemically reacting flow

For chemically reacting flow simulations, the following function call is required:

```
set_reaction_scheme(config_file, reacting_flag=1)
```

where `config_file` is a string naming the configuration file for the chemical reaction scheme. This configuration file specifies all of the chemical reactions between the various species and is built by hand by the user. By default, the reactions are turned on, however, the user may elect to turn off chemical reaction updates by setting `reacting_flag=0`.

An example of a reacting flow simulation is given in Section 20. The details of building a chemistry input file are provided in Appendix H.

5.6 Defining flow conditions

Because `Eilmer3` is a flow *simulation* code, initial gas flow conditions need to be specified throughout the domain. Also, depending on your model, free-stream inflow boundary conditions may need to be specified on appropriate boundary surfaces. To define such a flow condition in your input script for one or both of these purposes, create a `FlowCondition` object[9] as:

```
my_flow = FlowCondition(p=1.0e5, u=0.0, v=0.0, w=0.0, T=[300.0,],
                        massf=None, label="", tke=0.0, omega=1.0,
                        S=0, add_to_list=1)
```

- `p`: pressure in Pa, default value 100 kPa.
- `u`: x-coordinate velocity in m/s, default value 0.0.
- `v`: y-coordinate velocity in m/s, default value 0.0.
- `w`: z-coordinate velocity in m/s, default value 0.0.
- `T`: list of temperatures in degrees K, default value [300.0,]. For gas models with multimodal energies, these are the corresponding temperatures. For a gas model with only one internal energy mode, you may specify a scalar value for temperature.
- `massf`: mass fractions of the component species. These may be provided in a number of ways:
  (a) full list of floats. The length of the list of mass fractions must match the number of species in the previously selected gas model.

[9] The `FlowCondition` class is defined in source file `e3_flow.py`

(b) single float or integer that gets used as the first element, the rest being set 0.0

(c) dictionary of species names with mass fraction values, the remainder being set 0.0. See the example in Section 5.8.

(d) None provided, results in the first element being 1.0 and the rest 0.0

Note that the mass fractions supplied must sum to 1.0 (within a tolerance of 1.0×10^{-6}.

- `label`: (optional) text label for the FlowCondition object.
- `tke`: turbulent kinetic energy per unit mass in m^2/s^2 or J/kg, default value 0.0.
- `omega`: turbulence vorticity in 1/s, default value 1.0.
- `mu_t`: turbulence viscosity in Pa.s, default value 0.0.
- `k_t`: turbulence thermal conductivity, default value 0.0. This might be conveniently computed as $C_p\mu_t/Pr_t$.
- `S`: integer shock indicator value, default value 0. A value of 1 indicates the presence of a shock through the cell.
- `add_to_list`: flag to indicate that this FlowCondition object should be added to the flowList. Sometimes we don't want to accumulate objects in this list, for example, when using many FlowCondition objects in a user-defined flow evaluation function. default value 1.

Simulations involving nonequilibrium chemistry require and extra input file describing the participating gas species and their reactions. Preparation of this file is described in Appendix H.

5.7 Using flow conditions from other simulations

There are occasions where you might like to use flow data from an old simulation as initial conditions for some or all of your blocks in your new simulation. A typical use case is to restart a simulation with a finer, or otherwise changed, mesh. For this, you may pick up the old simulation data using:

```
old_flow = ExistingSolution(rootName, solutionWorkDir, nblock, tindx,
                            dimensions=2, assume_same_grid=0, zipFiles=1,
                            add_velocity=Vector(0.0,0.0,0.0))
```

where the arguments and their possible values are:

- `rootName`: job name that will be used to build file names
- `solutionWorkDir`: the directory where we'll find our existing solution files.
- `nblock`: number of blocks in the existing solution data set
- `tindx`: the time index to select 0..9999. Do not specify with leading zeros because the Python interpreter will assume that you want to count the time index in octal.
- `dimensions`: number of spatial dimensions for the existing solution
- `assume_same_grid`: decide how to locate corresponding cells

  `0` : searches for corresponding cells. As Rainer found, this can be agonisingly slow for large grids.

  `1` : omits the search for the corresponding cell. Definitely the option for the impatient.

- `zipFiles`: to use gzipped files (1), or not (0)
- `add_velocity`: value to be added to each cell's velocity, for changing frame of reference.

The process of writing the data into each cell of the new grid uses a fairly naive search for the nearest cell in the existing solution. Although it is robust, the search is extremely slow and the preparation of sew grids has been known to take hours of CPU time. If the new simulation is a continuation of the old simulation, it may be appropriate to set `gdata.t0` to a nonzero value. See Section 9.

5.8 Using mole fractions and species dictionaries

When simulating flows with mixes of gas species, it may be more convenient to specify the gas mix via mole fractions rather than mass fractions and via a dictionary rather than a list. With large numbers of species in the gas model, specification of the mix via dictionary is far easier to read and check than when using a list of numerical values.

There are a number of functions attached to the `Gas_model` object that make the conversion to a list of mass fractions easy. Here is an extract from Umar's standing-shock script showing the creation of a fairly complex gas mix using a dictionary of mole fractions.

```
select_gas_model(model='thermally perfect gas',
                 species=['O', 'N', 'N2', 'O2', 'NO', 'N_plus', 'O_plus', 'N2_plus',
                          'O2_plus', 'NO_plus', 'e_minus', 'Ar', 'Ar_plus'])
set_reaction_scheme("gupta_etal_air_reactions.lua", reacting_flag=1)
gmodel = get_gas_model_ptr()

# Pre-shock gas: mass fractions for an ideal air mixture.
mi = {'N2':0.769, 'O2':0.231}
# Post-shock: mole fractions from a CEA calculation.
X = {'O':1.6936e-1, 'N':5.9784e-1, 'N2':6.9757e-5, 'O2':4.7543e-8, 'NO':2.5654e-3,
     'N_plus':9.6331e-2, 'O_plus':1.7562e-2, 'N2_plus':7.7688e-6, 'O2_plus':5.0837e-8,
     'NO_plus':1.4459e-5, 'e_minus':1.1436e-1, 'Ar':4.0026e-3, 'Ar_plus':4.4835e-4}

initial = FlowCondition(p=2700.0, u=0.0, v=0.0, T=300.0, massf=mi)
inflow  = FlowCondition(p=4464.0, u=10284.0, v=0.0, T=10140.42,
                        massf=gmodel.to_massf(X))
```

6 Boundary representation of the gas domain

Most of the effort required to set up a simulation goes into defining the "body-fitted" grid of finite-volume cells that completely fills the flow domain. The top-level geometry description given to the grid generator is in terms of "patches" for 2D flow and "parametric volumes" for 3D flow. These are regions of space that may be traversed by a set of parametric coordinates $0 \leq r < 1$, $0 \leq s < 1$ (in 2D) and with the thrid parameter $0 \leq t < 1$ in 3D. These patches or volumes can be imported as VTK structured grids or they can be constructed as a "boundary representation" from lower-dimensional geometric entities such as paths and points.

6.1 Geometric elements

The most fundamental class of geometric object is the `Vector` (or `Vector3` as it is defined in the C++ module `libgeom2`). A `Vector` represents a point in 3D space and has the usual behaviour of a geometric vector (as opposed to the `vector` collection class in C++). See, for example, the postprocessing program in the `simple_ramp` simulation (Section 30.3). If you want a point to be rendered with a label, you can define it as a `Node`. Examples of use include: $a =$ `Vector`(x, y, z) and $b =$ `Node`$(x, y, z,$ `label='B'`$)$.

If you look into the file `cfcfd2/lib/geometry2/source/geom.hh`, you will see that the `Vector3` objects support the usual vector operations of addition, subtraction and the like. Also, you can `clone` and transform a point. For example, to create a point and its mirror image in the (x,z)-plane, you could use

```
a = Vector(0.5, 0.6)
b = a.clone().mirror_image(Vector(0.0,0.0), Vector(0.0,1.0)
```

6.1.1 Paths

The next level of dimensionality is the `Path` class. A path object is a parametric curve in space, along which points can be specified via the single parameter $0 \leq t < 1$. Types of paths that are available include:

- `Line`(a, b): a straight line between points a and b.
- `Arc`(a, b, c): a circular arc from a to b around centre, c. Be careful that you don't try to make an `Arc` with included angle of 180° or greater. For such a situation, create two circular arcs and join as a `Polyline` path.
- `Arc3`(a, b, c): a circular arc from a through b to c. All three points lie on the arc.
- `Helix`(a_0, a_1, x_{local}, r_0, r_1, $d\theta$): a helical path about a specified axis, start and end radii and angle through which the path extends.
- `Helix`(p_0, p_1, a_0, a_1): a helical path through specified points and about a specified axis. Internally, it is stored as the helical path described above.
- `Bezier`($[b_0, b_1, ..., b_n]$): a Bezier curve from b_0 to b_n.
- `Nurbs`($CP[.]$, $w[.]$,$degree$, $U[.]$): nonuniform rational B-spline with control points vector $CP[.]$, weights vector $w[.]$, and knot vector $U[.]$.
- `Polyline`($[p_0, p_1, ..., p_n]$): a composite path made up of the segments p_0, through p_n. The individual segments are reparameterised, based on arc length, so that the composite curve parameter is $0 \leq t < 1$.
- `Polyline2`(*args): a composite path constructed from path elements and/or `Vector` points. If there are gaps between the elements and points, they will be filled with `Line` segments.
- `Spline`($[b_0, b_1, ..., b_n]$): a cubic spline from $b0$ through $b1$, to bn. A `Spline` is actually a specialized `Polyline`.
- `Spline2`(filename): a spline constructed from a file containing $x(, y(, z))$ coordinates of the interpolation points, one point per line. If the y or z values are missing, they are assumed to be zero.

- `PathOnSurface`(S, f_r, f_s): a path on the ParametricSurface $S(r, s)$, defined by the univariate functions $r = f_r(t)$ and $s = f_s(t)$.

- `PolarPath`(P, H): A path in 3D space made from another path, P, such that the neutral plane at height H is wrapped around a cylinder aligned with the x-axis.

- `PyFunctionPath`(f): a path defined by the user-supplied Python function, $f(t)$. The user function returns a tuple of three values representing the point in space for parameter value t.

Most `Path` objects (except `PyFunctionPath`) support the transformation methods `translate(displacement)`, `reverse()`, `mirror_image(point, normal)` and `rotate_about_zaxis(radians)`. Look in the source code files `gpath.hh` and `gpath.cxx` for details. These may be found in the directory `cfcfd2/lib/geometry2/source/`.

6.1.2 Surfaces

The `ParametricSurface` class represents two-dimensional objects which can be constructed from `Path` objects. Examples are:

- `CoonsPatch`(p_S, p_N, p_W, p_E): a transfinite interpolated surface between the four paths. It is expected that the paths join at the corners of the patch, such that $p_S(0) = p_W(0) = p00$, $p_S(1) = p_E(0) = p10$, $p_N(0) = p_W(1) = p01$ and $p_N(1) = p_E(1) = p11$. See the left part of Figure 2 for the layout of this surface. Note that, although we are using subscripts aligned with the BOTTOM and TOP surfaces in this description, the same order is used for the other surfaces when the local surface parametric directions are aligned with the relevant index directions. See the debugging cube in Appendix D. Be aware that the order of the supplied paths for each surface is (SOUTH, NORTH, WEST, EAST), which is different to the order accepted by the `make_patch()` function that is used to make two-dimensional grids in the following section. Finally, it is important to be careful with the orientation of the `Path` elements that form the patch boundaries. The NORTH and SOUTH boundaries progress WEST to EAST as shown in Figure 2 (in the following section). The WEST and EAST boundaries progress SOUTH to NORTH. If the `e3prep.py` program complains that the corners of your patch are "open", that may be a symptom of having one, or more, of your bounding paths having incorrect orientation.

- `AOPatch`(p_S, p_N, p_W, p_E): an interpolated surface that tries to keep the grid orthogonal near the edges and also tries to keep equal areas across the surface.

- `MeshPatch`: a surface defined over a structured mesh of quadrilateral facets. This might be useful for generating new grids from files imported from an external grid generator.

- `TrianglePatch`: a surface defined over an unstructured mesh of triangular facets. When the surface is really too complex to describe as a simpler form, this type of surface can conform (approximately) to just about anything.

- `BezierPatch`: a surface defined over a tensor product of Bezier curves.

- `RevolvedSurface`(p): a surface defined by rotating Path p about the x-axis. When calling the `eval(r,s)` method for this surface, the first parameter, `r`, is along the path and the second parameter, `s`, is the angle in the (y, z)-plane.

- `MappedSurface`(S_{query}, S_{true}): points on the query surface are projected onto the true surface. The final surface is a subset of the true surface. Usually the query surface is something simple like a `CoonsPatch` that is close to the shape of the desired grid and the true surface could be constructed as a `RevolvedSurface` which is a bit difficult to grid regularly.

- `PolarSurface`(S, H): A surface in 3D space made from another surface, S, such that the neutral plane at height H is wrapped around a cylinder aligned with the x-axis.

- `SurfaceThruVolume`(V,f_r,f_s,f_t): a surface through the ParametricVolume $V(r, s, t)$, defined by the univariate functions $r = f_r(t)$, $s = f_s(t)$ and $t = f_t(t)$.

- `NurbsSurface`: a surface defined as the tensor product of non-uniform rational B-splines.

- `PyFunctionSurface`(f): a surface defined by the user-supplied Python function, $f(r, s)$. The user function returns a tuple of three values representing the point in 3D space for parameter values r and s. If you are trying to build a 2D simulation, just return the z-coordinate as zero.

Except for `PyFunctionSurface`, most of the surface objects can be cloned and transformed with `translate`, `mirror_image` and `rotate_about_zaxis` methods. Again, see the source code for details.

6.1.3 Volumes

Finally, in its most general form, a `ParametricVolume`($S_N, S_E, S_S, S_W, S_T, S_B$) can be constructed from a set of six parametric surfaces to form a body-fitted hexahedral volume. More restricted forms of a volume can be constructed as

- `WireFrameVolume`($p_{01}, p_{12}, p_{32}, p_{03}, p_{45}, p_{56}, p_{76}, p_{47}, p_{04}, p_{15}, p_{26}, p_{37}$): is defined by its 12 edges (paths). Note the implied directions in the subscripts. The subscripts correspond to the labelled points in Figure 3.

- `WireFrameVolume`($surf, p$): consists of a surface $surf$ extruded along path p. The extrusion is actually done be forming a set of 6 surfaces by copying the original surface and then constructing four `CoonsPatch` surfaces between them.

- `SimpleBoxVolume`($p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7$): consists of a straight-edged hexahedral box defined by its 8 corner points (as shown in Figure 3).

- `MeshVolume`: consists of a `ParametricVolume` interpolated in an existing mesh. This mesh may be specified as an array of points or it may be read in from a VTK file.

There is an alternative approach to defining the ParametricVolume via a user-supplied Python function as

- `PyFunctionVolume`(f): a volume defined by the user-supplied Python function, $f(r, s, t)$. The user function returns a tuple of three values representing the point in 3D space for parameter values r, s and t.

Again, transform methods such as `translate` and `rotate_about_zaxis` may help in reducing the amount of user input script required to build complex regions out of multiple `ParametricVolume` objects.

6.2 Two-dimensional grids

The grid defining the discretized gas domain is block structured. In 2D, each block is a patch bounded by 4 edges (NORTH, EAST, SOUTH and WEST) such that we are looking at a plan-view of the flow domain as shown in Fig. 2.

To define a block in your input script for a 2D simulation, create a `Block2D` object as:

```
my_2d_block = Block2D(psurf=None, grid=None,
                      import_grid_file_name=None, nni=2, nnj=2,
                      cf_list=[None,]*4, bc_list=[SlipWallBC(),]*4,
                      fill_condition=None, hcell_list=[],
                      xforce_list=[0,]*4, label="", active=1)
```

where the assignment to the name `my_2d_block` allows easy referencing of the block at later times, say, for adding boundary conditions. The names of the actual arguments given above match the actual arguments in the `e3prep.py` program and these represent[10]:

- `psurf`: a region of 2D space bounded by 4 edges. This region is often constructed from 4 geometric paths via a call to `make_patch(north, east, south, west,`

[10]The definitive source is, of course, the `Block2D` class definition in `e3_block.py`.

Figure 2: A two-dimensional patch containing the structured mesh for a Block2D object (left) and a collection of sub-blocks defined via a SuperBlock2D or MultiBlock2D constructor (right). The orientations of the bounding paths are important: WEST and EAST paths progress from SOUTH to NORTH; SOUTH and NORTH paths progress from WEST to EAST.

`grid_type)` where the default value for `grid_type` is "TFI" *i.e.* transfinite interpolation or Coons' patch. Another possible form of grid is "AO", the area-orthogonality grid. This is the usual way of specifying the flow domain, which will be discretized using `nni, nnj, and cf_list`. Note that all geometric elements should have zero values for their z-components when doing a 2D flow simulation. Since most constructors will have a default value of zero for the z-component, this detail can usually be ignored.

- `grid`: a `StructuredGrid` object may be supplied (defaults to None).
- `import_grid_file_name` defaults to None. If a name is supplied, this file is read to obtain the grid directly. The assumed file format in the legacy (ASCII) VTK format for a structured grid.
- `nni` is the number of finite-volume cells in the i-index direction. See the left part of Figure 2 for the orientation of the index. Note that, when placing one block against another, the blocks must conform in
  - the number of cells along corresponding edges
  - the clustering of those cells along the edges

- the path defining the corresponding edges.

The minimum number of cells is 2, because of the way that the cell-interface values are reconstructed from cell-centred data.

- `nnj` is the number of finite-volume cells in the j-index direction.
- `cf_list` is an optional list of 4 `UnivariateFunction` objects that specify a (possibly) nonuniform distribution of cells along each particular edge. For each object, there is an `eval(t)` method which returns a transformed (new) value of t. The options available are:
  - `LinearFunction(m, c)` where $t_{new} = m \times t_{old} + c$.
  - `LinearFunction2(y0, y1)` where $t_{new} = y0 \times (1 - t_{old}) + y1 \times t_{old}$.
  - `RobertsClusterFunction(end0, end1, beta)` where the `end0, end1` integer flags indicate which end (possibly both) we wish to cluster toward. The value of `beta` > 1.0 specifies the strength of the clustering, with the clustering being stronger for smaller values of `beta`. For example, a value of 1.3 would be relatively weak clustering while a value of 1.01 is quite strong clustering.
  - `ValliammaiFunction(dL0, dL1, L, n)` See Adriaan's source code for definitions.

  See the files `lib/nm/source/fobject.cxx` and `lib/nm/source/fobject.hh` for details. The order of appearance of boundaries in the list is NORTH, EAST, SOUTH and WEST. Note that a full list of 4 items is required. If you don't want to specify one (or more) of the items in the list, specify `None` as that item.

- `bc_list` is an optional list of BoundaryCondition objects.[11] Available boundary conditions are:
  - `AdjacentBC()` for cases where one block interfaces with another.
  - `SupInBC(inflow_condition, label='')` where we want to specify the inflow condition that gets copies into the ghost cells each time step. The optional `label` has an empty default value but may be used to group boundary surfaces symbolically in the postprocessing stage. Paul Petrie-Repar has made use of these labels in his `CGNS` postprocessing program.
  - `ExtrapolateOutBC(sponge_flag=0, label='')` where we want a (mostly supersonic) outflow condition. Flow data is effectively copied from just inside the boundary to the ghost cells just outside the boundary, every time step. In subsonic flow, this can lead to unphysical bahaviour.

[11] Note that, when creating these objects in the Python input script, the Python language requires the parentheses even for the cases where no arguments, such as `Twall`, are required.

- `SlipWallBC(label='')` where we want a solid wall with no viscous effects. This is the default boundary condition where no other condition is specified.
- `AdiabaticBC(label='')` where we want viscous effects to impose no-slip at the wall but where there is no heat transfer.
- `FixedTBC(Twall, label='')` where we want viscous effects to impose a no-slip velocity condition and a fixed wall temperature.
- `SubsonicInBC(stagnation_condition, label='')` where the flow is assumed subsonic and we specify the stagnation pressure and temperature, but take the velocity from just inside the boundary.
- `TransientUniBC(filename, label='')` where we want to specify the time-history of the inflow condition.
- `StaticProfileBC(filename, label='')` where we want to apply a steady-state inflow which may vary in space.
- `FixedPOutBC(Pout, label='')` is like `ExtrapolateOutBC()` but with a specified back pressure. This can be analogous to a vacuum pump that removes gas at the boundary to maintain a fixed pressure in the ghost cells.
- `UserDefinedBC(filename, is_wall=0, use_udf_flux=0, label='')`: allows the user to define the ghost-cell flow properties and/or interface fluxes at run time. This is done via a set of functions defined by the user, and written in the Lua programming language. These functions are provided in the file given by `filename`. The flag `is_wall` indicates whether the boundary is to be considered a wall for the application of turbulence-model fudges and the like (default 0). The flag `use_udf_flux` indicates whether the user is supplying the fluxes at the boundary interfaces (default 0). If not, the internal flux calculator is used together with the supplied ghost-cell data. This boundary condition is the Jack of all trades and master of none. It can be used to emulate any of the other boundary conditions and then build variations, however, it is going to cost quite a lot in computational time. See Appendix I for the details of setting up this boundary condition.
- `AdjacentPlusUDFBC(other_block, other_face, orientation, filename, is_wall=0, use_udf_flux=0, label='')`: is a combination of the `AdjacentBC` and `UserDefinedBC`. At each time step, the flow data is first exchanged, as per the usual `AdjacentBC`. Then the user-defined functions are applied. This is one way of getting fancy boundary conditions, such as slowly-opening diaphragms, into the simulation.

These boundary conditions may also be set, one at a time, as described in the Section 7.

- `fill_condition` is the `FlowCondition` object with which to define the initial flow state within the volume. See Section 5 for defining a suitable flow condition. You may alternatively provide a Python function that supplies the flow properties as a function of position or you may use an `ExistingSolution()` object.

- `hcell_list` is a list of (i, j)-tuples specifying which cells should be monitored at simulation time. Data from the specified cells will be written to a "history" file for the block and may be used at the postprocessing stage to provide flow data as if there was a sensor located in the cell.

- `xforce_list` is an optional list of zeros/ones that indicate if we want the force to be calculated for each of the four edges and written to the `e3shared.log` log file. See the notes in the 20° cone test case (Section 11) for an example of how to extract this data from the log file.

- `label` is an optional text label for the block. This label will be embedded in the block definition and some of the postprocessing programs may use it. For example, the `e3cgns.py` postporcessing program uses labels to group block boundaries symbolically.

Note that, when lists of items are provided for the four boundaries, the order of the boundaries is NORTH, EAST, SOUTH and WEST.

When defining large domains and running simulations of a parallel computer, it may be convenient to define many Block2D objects with one call. The first of two constructors for this situation is

```
my_block_list = SuperBlock2D(psurf=None, nni=2, nnj=2, nbi=1, nbj=1,
                             cf_list=[None,]*4, bc_list=[SlipWallBC(),]*4,
                             fill_condition=None, hcell_list=[], label="sblk")
```

which generates a single grid over `psurf` and then subdivided that grid into `nbi` $\times$ `nbj` Block2D sub-blocks. References to all of these sub-blocks are returned as a list of lists, such that a particular sub-block may be obtained as `my_block_list.blks[i][j]`. The second constructor is

```
my_block_list = MultiBlock2D(psurf=None, nni=None, nnj=None,
                             bc_list=[SlipWallBC(),]*4,
                             nb_w2e=1, nb_s2n=1, nn_w2e=None, nn_s2n=None,
                             cluster_w2e=None, cluster_s2n=None,
                             fill_condition=None, label="blk")
```

which first subdivides the parametric patch into sub-patches and then generates an individual grid over each sub-patch. Here, a set of `nb_w2e` $\times$ `nb_s2n` sub-blocks are generated

and, if lists of integers are provided for `nn_w2e` and `nn_s2n`, these will be used as the numbers of cells along the edges of the sub-blocks. If these lists are not supplied, `nni` × `nnj` cells will be divided across the sub-blocks. In both of these constructors, the interior boundaries for the sub-blocks are connected (as `AdjacentBC` boundary conditions).

When assembling large numbers of blocks for complex geometries, there is a function

```
identify_block_connections(block_list=None, exclude_list=[],
                           tolerance=1.0e-6)
```

that performs a brute-force search for all adjacent blocks and sets `AdjacentBC` boundary conditions for pairs of edges that have coinciding corners (to within a given tolerance). If you don't want the search to be over all blocks generated so far, supply a list to the `block_list` argument. Alternatively, supply a list for blocks that should be excluded.

In some situations, you may want to manually connect particular blocks. You can use the function

```
connect_blocks_2D(A, faceA, B, faceB, with_udf=0, filename=None,
                  is_wall=0, use_udf_flux=0)
```

where `A` and `B` are references to the individual `Block2D` objects and `faceA` and `faceB` are their adjoining edges (NORTH, EAST, SOUTH or WEST) Most of the time you can just ignore the default arguments associated with user-defined functions (*i.e.* `with_udf, filename, is_wall, use_udf_flux`). These are used to implement slowly-opening diaphragms and the like.

6.3 Three-dimensional grids

In 3D, life is just that bit more complicated with each block defined by 6 surfaces (NORTH, EAST, SOUTH, WEST, TOP and BOTTOM) fitted to the actual surfaces of the domain. Figure 3 shows the "index-space" view with cell indices i,j and k taking values $0 \leq i < nni$, $0 \leq j < nnj$ and $0 \leq k < nnk$ respectively.[12] The corner vertices of the block are numbered 1 through 7 as shown. These points are used in the search to determine block connectivity if the flow domain is defined as consisting of more than one block. Subdividing a complex flow domain into simpler subdomains is often done because the mapping from parametric space to physical space is limited to a simple transfinite interpolation.

To assist in understanding the orientation of the corners, surfaces and indices, you can build a model block from the development plan in Appendix D. This should bring back fond memories of kindergarten and primary school, at least it did for us.

[12] The i, j and k indices are related to the r, s and t parameters used within the 3D geometric functions. In some places, the corner points are identified by their (r, s, t) coordinates. For example, in the simple-ramp postprocessing script (section 30.3), point 0 would be identified as $p000$, point 1 as $p100$, etc.

Figure 3: Two views of the hexahedral block containing the structured mesh. These figures are ambiguous but each is supposed to show a hollow box with the *far* surfaces in each view being labelled. The *near* surfaces are transparent and unlabelled. To get your hands on an unambiguous representation, build the debugging cube drawn in Appendix D .

To define a block in your input script, create a `Block3D` object as:

```
my_3d_block = Block3D(parametric_volume=None, grid=None,
                      import_grid_file_name=None,
                      nni=None, nnj=None, nnk=None,
                      cf_list=[None,]*12, bc_list=[SlipWallBC(),]*6,
                      fill_condition=None,
                      hcell_list=None, xforce_list=[0,]*6,
                      label="", active=1, omegaz=0.0)
```

where the assignment to the name `my_3d_block` allows easy referencing of the block at later times, say, for adding boundary conditions. The names of the actual arguments given above match the actual arguments in the `e3prep.py` program and these represent[13]:

- `parametric_volume`: a region of 3D space bounded by 6 surfaces. This is the usual way of specifying the flow domain, which will be discretized using `nni, nnj, nnk and cf_list`. See the following section for a guide to constructing `parametric_volume` objects.
- `grid`: a `StructuredGrid` object may be supplied (defaults to None).
- `import_grid_file_name` defaults to None. If a name is supplied, this file is read to obtain the grid directly. The assumed file format in the legacy (ASCII) VTK

[13] Again, the definitive source is, of course, the `Block3D` class definition in `e3_block.py`.

Table 1: Directions for the edges of a `Block3D` object.

edge	from point	to point	comment
0	p_0	p_1	i-direction, bottom surface
1	p_1	p_2	j-direction, bottom surface
2	p_3	p_2	i-direction, bottom surface
3	p_0	p_3	j-direction, bottom surface
4	p_4	p_5	i-direction, top surface
5	p_5	p_6	j-direction, top surface
6	p_7	p_6	i-direction, top surface
7	p_4	p_7	j-direction, top surface
8	p_0	p_4	k-direction
9	p_1	p_5	k-direction
10	p_2	p_6	k-direction
11	p_3	p_7	k-direction

format for a structured grid. There is also an external tool (`p2e.py`) that can be used to convert Plot3D format files to Eilmer's native format.

- `nni` is the number of finite-volume cells in the i-index direction as shown in Figure 3. This is only used when diecretizing a `parametric_volume`. When importing or supplying a grid, this data (`nni, nnj and nnk`) is ignored. Note that, when placing one block against another, the blocks must conform in
  - the number of cells along corresponding edges
  - the clustering of those cells along the edges
  - the path defining the corresponding edges.
- `nnj` is the number of finite-volume cells in the j-index direction.
- `nnk` is the number of finite-volume cells in the k-index direction.
- `cf_list` is a list of `Function` objects that specify a (possibly) nonuniform distribution of cells along a particular edge of the `parametric_volume`. The order of the edges is shown in Table 1. See page 33 for a more complete description of the cluster functions.
- `bc_list` is an optional list of BoundaryCondition objects for the six bounding surfaces (`NORTH, EAST, SOUTH, WEST, TOP, BOTTOM`). Available boundary conditions are the same as for Block2D objects
- `fill_condition` is the `FlowCondition` object with which to define the initial flow state within the volume. See Section 5 for defining a suitable flow condition. This

may also be a callable function that supplies the flow properties as a function of position.

- `hcell_list` is a list of (i, j, k)-tuples specifying which cells should be monitored at simulation time. Data from the specified cells will be written to a "history" file for the block and may be used at the postprocessing stage to provide flow data as if there was a sensor located in the cell.

- `xforce_list` is an optional list of zeros/ones that indicate if we want the force to be calculated for each of the six surfaces and written to the `e3shared.log` log file. The order of the boundaries is the same as for `bc_list`.

- `label` is an optional text label for the block. This label will be embedded in the block definition and some of the postprocessing programs may use it.

- `omegaz` is the rotational speed of the volume about the z-axis. This parameter is non-zero only for rotating components of the turbomachine grids.

To manually connect particular Block3D objects, you can use the function

```
connect_blocks_3D(A, B, vtx_pairs, with_udf=0, filename=None,
                  is_wall=0, use_udf_flux=0)
```

where `A` and `B` are references to the individual `Block3D` objects and `vtx_pairs` is a list of 4 pairs (tuples) of vertex indices. For example, the list `[(3,2),(7,6),(6,7),(2,3)]` specifies a `NORTH`-to-`NORTH` connection with orientation `0`. The definitions of all allowable connections is listed near the top of the file `e3_block.py`. You will see that there are *many* more combinations in 3D compared with 2D.

As for the 2D grids, there are two composite-block generation functions. The first takes a volume, grids it and then subdivides the newly generated grid:

```
my_3d_block = SuperBlock3D(parametric_volume=None, cf_list=[None,]*12,
                           fill_condition=None,
                           nni=2, nnj=2, nnk=2,
                           nbi=1, nbj=1, nbk=1,
                           bc_list=[SlipWallBC(),]*6, label="sblk",
                           hcell_list=None, omegaz=0.0)
```

where `nbi`, `nbi` and `nbk` are the number of basic blocks in each of the index directions. The values for `nni`, `nnj` and `nnk` specify the number of cells for the grid generated over the whole volume. The second composite block takes a volume, subdivides that volume and then generates a separate grid within each subvolume:

```
my_3d_block = MultiBlock3D(parametric_volume=None,
                           fill_condition=None,
                           nni=None, nnj=None, nnk=None,
                           nbi=1, nbj=1, nbk=1,
                           clusteri=None, clusterj=None, clusterk=None,
                           bc_list=[SlipWallBC(),]*6, label="blk",
                           hcell_list=None, omegaz=0.0)
```

Here, `nni`, `nnj` and `nnk` may be integer values or lists of integer values. If they are simple integers, they represent the number of cells over the whole volume. If they are lists of integers, they specify the number of cells each of the subblocks. The `clusteri`, `clusterj` and `clusterk` may be lists of cluster functions at get applied to the subblocks in the respective index directions.

Note the the composite-block objects contain a member `blks` that refers to the list of basic blocks that form the composite block. Any further setting of boundary conditions, and the like, needs to be done to the individual blocks within this list. See the input script for the finite-cylinder case (on page 208) for an example of this.

When assembling large numbers of blocks for complex geometries, the function

```
identify_block_connections(block_list=None, exclude_list=[],
                           tolerance=1.0e-6)
```

also works for 3D blocks. As for 2D blocks, it performs a brute-force search for all adjacent blocks and sets `AdjacentBC` boundary conditions for pairs of faces that have coinciding corners (to within a given tolerance). The rotational orientation of the joined faces is also determined automatically. If you don't want the search to be over all blocks generated so far, supply a list to the `block_list` argument. Alternatively, supply a list for blocks that should be excluded.

Be aware that the `identify_block_connections()` function is unaware of the form of the actual paths or surfaces connecting the corner points. It may be that the corners coincide but the paths and surfaces do not conform. If you want more control over the process of joining blocks, you can manually connect blocks using the `connect_blocks_3D()` function which makes the logical connection without looking at the geometric locations of the corners. This situation might arise, for example, when you want to apply periodic boundary conditions in the cross-stream direction of a flow domain. Then, the boundaries that you want to connect have corners and faces that really don't coincide.

7 Setting conditions for individual boundaries

If you have not already set all appropriate boundary conditions through the `bc_list` argument of the block constructor, you may apply boundary conditions to specific faces of a `Block2D` or `Block3D` object by calling its method

```
set_BC(face_name, type_of_BC, inflow_condition=None, sponge_flag=None,
       Twall=None, Pout=None, filename=None, is_wall=0, use_udf_flux=0,
       label='')
```

and specifying the face and type of boundary condition. When this function is called, it creates a suitable boundary condition object (as discussed in the previous section) and binds it to the appropriate block boundary. There is no difference in the end result compared with the approach of specifying the boundary conditions when the block is created.

- `face_name`: one of `NORTH`, `EAST`, `SOUTH`, `WEST`, `TOP`, `BOTTOM`
- `type_of_BC`: one of
  - `ADJACENT`: there is another block abutting this face. This boundary condition is usually set by the block-conection functions.
  - `SUP_IN`: supersonic inflow using the `inflow_condition` properties.
  - `EXTRAPOLATE_OUT`: (assumed) supersonic-outflow where the ghost-cell flow properties are simply copies of the adjacent interior cell properties.
  - `SLIP_WALL`: an inviscid solid wall where the normal velocity in the ghost cells is a reflection of the velocity in the interoir cell.
  - `ADIABATIC`: a no-slip wall where the wall temperature is the same as the cell-centre temperature.
  - `FIXED_T`: a no-slip wall where the wall temperature is specified by `Twall` in degrees K.
  - `SUBSONIC_IN`: subsonic inflow where the stagnation pressure and temperature is specified and the velocity is taken from the interior cell.
  - `TRANSIENT_UNI`: a transient flow condition applied uniformly across the face of the block.
  - `STATIC_PROF`: a time-invariant flow condition that has spatial variation across the face of the block.
  - `FIXED_P_OUT`: something like the `EXTRAPOLATE_OUT` condition with the pressure in the ghost cells set to `Pout`.

- RRM: rescaled and recycled data for Andrew Denman's LES simulations.
- USER_DEFINED: the user-supplied Lua functions are used to determine ghost-cell flow properties and or interface fluxes. These functions are provided in the file given by filename. The flag is_wall indicates whether the boundary is to be considered a wall for the application of turbulence-model fudges and the like (default 0). The flag use_udf_flux indicates whether the user is supplying the fluxes at the boundary interfaces (default 0). If not, the internal flux calculator is used together with the supplied ghost-cell data.
- ADJACENT_PLUS_UDF:

- inflow_condition: the flow condition used for SUP_IN, default value None.
- sponge_flag: Andrew Denman's flag, default value None.
- Twall: static temperature of the wall in degrees K, default value None.
- Pout: static pressure in Pa applied to the ghost cells, default value None.
- label: symbolic label for the boundary, default value is an empty string.

You need only specify the properties that are relevant to the specific boundary condition.

8 Special zones and history points

Zones of heating or cooling may be defined within the flow domain as rectangular (2D) or regular hexahedral (3D) patches which are specified by two diagonally-opposite corners (point0 and point1). For example, we could specify

```
HeatZone(qdot, point0, point1, label="")
```

where qdot is the heat addition per unit volume in W/m^3. The corners of each hexahedral zone are given by the Vector values point0 and point1. If the centre of a cell lies within the heat zone, qdot is added to the source term in the energy equation every time step during the simulation. When using a HeatZone it is necessary to give at least heat_time_stop a positive non-zero value and heat_time_start and heat_factor_increment can also be modified as appropriate. A HeatZone might be used to model the deposition of energy into a small volume from a high-power laser, for example.

Similarly, zones of reaction are defined with

```
ReactionZone(point0, point1, label="")
```

where the finite-rate reactions will be allowed to proceed. Outside of these zones, the finite-rate chemical update will be suppressed and the species concentrations will be effectively frozen. If no such zones are specified, reactions are permitted for the entire flow field.

Also, when running turbulent flow simulations, the turbulence model can also be restricted to being applied to specific zones using
`TurbulenceZone(point0, point1, label="")`
The turbulence model (say, the $k - \omega$ model) is active throughout the flow but its effect on the flow field is masked outside of the `TurbulenceZone`s. This is achieved by the code setting the turbulence vicsosity and conductivity to zero for finite-volume cells that fall outside of all regions defined as a `TurbulenceZone`. If there a no such defined regions, all of the flowfield may have nonzero turbulence viscosity.

As well as being identified by their cell indices when defining a block, history points can be located by their Cartesian coordinates using:
`HistoryLocation(x, y, z=0.0, i_offset=0, j_offset=0, k_offset=0, label="")`
where the offset indices allow you to select a cell a known number of cells away from another.

9 Simulation control parameters

A number of other parameters can be set in order to control the behaviour of the simulation. These parameters are mainly collected into the `gdata` object[14] which is accessible to the user's input script. Grouped by theme, the possible attributes include[15]:

Geometry

- `dimensions`: number of geometric dimensions (2 or 3). If unspecified, the default is 2.
- `axisymmetric_flag`: 1=2D-axisymmetric geometry with x-axis being the axis of symmetry, 0=2D-planar geometry, default value 0.

Time stepping

- `sequence_blocks`: 0=normal time iteration on all blocks, 1=integrate one block at a time, default value 0.
- `dt`‡: the initial time step (in seconds) that will be used for the first few steps of the simulation process. Be careful to set a value small enough for the time-stepping to be stable. Since the time stepping is synchronous across all parts of the flow domain, this time step size should be smaller than half of the smallest time for a signal (pressure wave) to cross any cell in the flow domain.

[14] The `gdata` object is an instance of the `GlobalData` class defined in `e3prep.py`. Most of the attributes are discussed here, however, see the source code for that class for a full list of attributes.

[15] Attributes that are stored in the control file are denoted by a ‡ symbol. The rest go into the config file.

- `dt_chem`: suggested time-step for finite-rate chemistry update; default value of -1.0 indicates that we want the code to work it out.

- `dt_therm`: default value -1.0.

- `t_order`‡: 1=Euler time stepping, 2=predictor-corrector time-stepping, default value 1. If you want time-accurate solutions, use predictor-corrector stepping, otherwise, Euler stepping is half the computational expense (but you may get less accuracy and the code will not be as robust).

- `fixed_time_step`‡: 1=do not change time step from that specified, 0=allow time step size to be determined from cell conditions and cfl number, default value 0.

- `cfl`‡: ratio of the smallest signal time to the actual time step, default value 0.5.

- `stringent_cfl`‡: 1=use the smallest cross-cell distance in the CFL check, 0=use different cell widths in each index direction, default is 0.

- `dt_reduction_factor`‡: if the CFL condition is violated, scale the time-step size down by this factor, default value 0.2.

- `cfl_count`: number of time steps between checks of the CFL condition, default value 10. This check is expensive so we don't want to do it too frequently but, then, we have to be careful that the time step does not become unstable.

- `max_time`‡: the simulation will be terminated on reaching this value of time, default value 1.0×10^{-3}.

- `t0`: starting time for simulation, may be useful to change when restarting from another job, default value 0.0.

- `max_step`‡: the simulation will be terminated on reaching this number of time steps, default value 10.

- `dt_plot`‡: the whole flow solution will be written to disk when this amount of simulation time has elapsed, default value 1.0×10^{-3}s.

- `dt_history`‡: the history-point data will be written to disk when this amount of time has elapsed, default value 1.0×10^{-3}s.

Spatial reconstruction/interpolation

- `x_order`‡: 1=no reconstruction of intra-cell flow properties before applying the flux calculator, 2=high-order reconstruction applied, default value 2

- `apply_limiter_flag`: 1=apply reconstrction limiter, default value 1.
- `interpolation_type`: string to choose the set of interpolation variables to use in the interpolation, options are "rhoe", "rhop", "rhoT", "pT", default value "rhoe".

Flux calculator

- `flux_calc`: selects the flavour of the flux calculator, default value `ADAPTIVE`. The ADAPTIVE scheme is a good all-round scheme that uses AUSMDV away from shocks and EFM near shocks.
- `compression_tolerance`: value of relative velocity change (normalised by local sound-speed) across a cell-interface that triggers the shock-point detector. A negative value indicates a compression. When the ADAPTIVE flux calculator is used and the shock detector is triggered, the EFM flux calculation will be used in place of the default AUSMDV calculation. A value of -0.05 seems OK for the sod and cone20 inviscid flow simulations, however, a higher value is needed for cases with viscous boundary layers, where it is important to not have too much dissipation in the boundary layer region. The default value is -0.30.
- `shear_tolerance`: value of the relative tangential-velocity change (normalised by local sound speed) across a cell-interface that suppresses the use of EFM even if the shock detector indicates that EFM should be used for the ADAPTIVE flux calculator. The default value is experimentally set at 0.20 to get smooth shocks in the stagnation region of bluff bodies. A smaller value (say, 0.05) may be needed to get strongly expanding flows to behave when regions of shear are also present.

Viscous effects

- `viscous_flag`: 1=viscous terms are active, 0=inviscid simulation, default value 0.
- `viscous_delay`: the time (in seconds) to wait before applying the viscous terms. This might come in handy when trying to start blunt-body simulations.
- `viscous_factor_increment`: per-time-step increment of the viscous effects, once t > viscous_delay, default value 0.01.
- `diffusion_flag`: 1=compute multicomponent diffusion of species, default value 0.
- `diffusion_model`: string, default value "None".
- `turbulence_flag`: 1=activate turbulence model, 0=laminar viscosity only, default value 0.

- `turbulence_model`: string specifying which model to use, "k_omega", "baldwin_lomax", default "k_omega".
- `turbulence_prandtl_number`: default value 0.89
- `turbulence_schmidt_number`: default value 0.75
- `max_mu_t_factor`: turbulent viscosity is limited to laminar viscosity multiplied by this factor, default value 300.0.
- `transient_mu_t_factor`: default value 1.0.

Chemistry

- `reacting_flag`: flag to indicate that the finite-rate chemical reactions are active. It has a default value of 0, however, it gets set to 1 if the call to `set_reaction_scheme()` is made. This is the usual way of setting it.
- `reaction_update`: File name for reaction scheme configuration. (More conveniently set by calling `set_reaction_scheme()`.)
- `reaction_time_start`: time after which finite-rate reactions are allowed to start, default value 0.0.

Miscellaneous

- `title`: a title string that may appear in a number of places. For example, in plots made during the postprocessing stage.
- `max_invalid_cells`: the maximum number of bad cells that will be tolerated on decoding conserved quantities. It this number is exceeded, the simulation will stop. default value 10.
- `udf_source_vector_flag`: 1=apply user-defined source terms as supplied in a Lua file, default value 0.
- `udf_file`: name of the Lua file for the user-defined source terms, default value "".
- `print_count`‡: number of time steps between printing status information to the console, default value 20.
- `heat_time_start`: default value 0.0, in seconds. For a description of HeatZones, see Section 8.
- `heat_time_stop`: a non-zero value indicates that we wish to add heat through the HeatZones, default value 0.0, in seconds.

- `heat_factor_increment`: the fraction of full heat load that will be added with each step after t=heat_time_start, default value 0.01.

10 Parameters for a 2D sketch of the flow domain

The `sketch` object holds parameters that set the view and scale of the SVG (scalable vector graphic) rendering of the two-dimensional flow domain. The method

```
sketch.window(xmin=0.0, ymin=0.0, xmax=1.0, ymax=1.0,
              page_xmin=0.05, page_ymin=0.05, page_xmax=0.17, page_ymax=0.17)
```

sets the mapping from the lower-left to upper-right points in simulation geometry to the corresponding points on a page. Axes may also be drawn with:

```
sketch.xaxis(x0, x1, xtic, y_offset)
sketch.yaxis(y0, y1, ytic, x_offset)
```

where small negative values may be given for the offset values in order to move the axes clear of the main sketch elements.

Of course, this sketch environment is only available for 2D simulations. For 3D rendering, there is an option for most geometric elements to be rendered as Virtual Reality Markup Language (VRML) strings. These VRML strings may be manually written to a file as part of the user's input script.

Part II
Examples for 2D flow

These examples are graded from simple geometry specification and gas model specification to more complex. Initially, simple box regions and single-specied ideal gas models are used, followed by examples with curved boundaries, equilibrium gas models and, also, multi-species thermally-perfect gases with finite-rate chemical kinetics. Later examples also use more of Python's capabilities with the input script for the heat-transfer to a sphere, for example, being written as a template script and a top-level coordinating script that runs the simulation a number of times with better grid resolution.

11 Mach 1.5 flow over a 20° cone – Simple boundaries

This is a small (in both memory and run time) example that is useful for checking that the simulation and plotting programs have been built or installed correctly. Assuming that you have the program executable files built and accessible on your system's search `PATH`, try the following commands:

```
$ cd ~/cfcfd2/examples/eilmer3/2D/cone20-simple/
$ ./cone20_run.sh
```

and, within a minute or so, you should end up with a number of files with various solution data plotted. The grid and initial solution are created and the time-evolution of the flow field is computed for 5 ms (with 1105 time steps being required). The commands invoke the shell scripts displayed in subsection 11.2.

Figure 4: Schematic diagram of the geometry for a cone with 20 degree half-angle. This PDF figure was generated from the SVG file with some edits to move the boundary labels to nicer positions.

The free-stream conditions ($p_\infty = 95.84\,\text{kPa}$, $T_\infty = 1103\,\text{K}$ and $u_\infty = 1000\,\text{m/s}$) are related to the shock-over-ramp test problem in the original ICASE Report [2] and are set to give a Mach number of 1.5. From Chart 5 in Ref. [3], the expected steady-state shock

wave angle is 49° and, from Chart 6, the pressure coefficent is

$$\frac{p_{cone-surface} - p_\infty}{q_\infty} \approx 0.387$$

and the dynamic pressure for the specified free stream is $q_\infty = \frac{1}{2}\rho_\infty u_\infty^2 \approx 151.38\,\text{kPa}$. Figure 7 shows the pressure coefficient estimated as

$$C_p = \frac{f_x - p_\infty A}{q_\infty A}$$

from the simulated axial force, f_x, written into the simulation log file and frontal area of the cone, A.

Figure 5: Pressure data for flow over a cone with 20 degree half-angle. The shock profile is not yet straight and the pressure field near the cone surface is not conically symmetric, although it would become more so if we continued the simulation.

Figure 6: Shock-sensor data for flow over a cone with 20 degree half-angle. For the `adaptive` flux calculator, this sensor indicates the regions of the flow where the more dissipative scheme should be used.

Figure 7: Evolution of the axial (drag) force for flow over a cone with 20 degree half-angle for two mesh resolutions.

11.1 Input script (.py)

```
## \file cone20.py
## \brief Simple job-specification file for e3prep.py
## \author PJ, 08-Feb-2005
##
## 15-Sep-2008 -- simplified version for Eilmer3
##
## We have set this file up very much like the cone20.sit file
## so that users may more-easily see the correspondence between
## the Tcl and Python elements.

job_title = "Mach 1.5 flow over a 20 degree cone."
print job_title

# We can set individual attributes of the global data object.
gdata.dimensions = 2
gdata.title = job_title
gdata.axisymmetric_flag = 1
gdata.stringent_cfl = 1  # to match the old mb_cns behaviour

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])

# Define flow conditions
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)
inflow  = FlowCondition(p=95.84e3, u=1000.0, v=0.0, T=1103.0)

# Set up two quadrilaterals in the (x,y)-plane be first defining
# the corner nodes, then the lines between those corners and then
# the boundary elements for the blocks.
# The labelling is not significant; it is just to make the SVG picture
# look the same as that produced by the Tcl scriptit program.
a = Node(0.0, 0.0, label="A")
b = Node(0.2, 0.0, label="B")
c = Node(1.0, 0.29118, label="C")
d = Node(1.0, 1.0, label="D")
e = Node(0.2, 1.0, label="E")
f = Node(0.0, 1.0, label="F")

ab = Line(a, b); bc = Line(b, c) # lower boundary including cone surface
fe = Line(f, e); ed = Line(e, d) # upper boundary
af = Line(a, f); be = Line(b, e); cd = Line(c, d) # vertical lines

# Define the blocks, boundary conditions and set the discretisation.
nx0 = 10; nx1 = 30; ny = 40
blk_0 = Block2D(make_patch(fe, be, ab, af), nni=nx0, nnj=ny,
                fill_condition=initial, label="BLOCK-0")
blk_1 = Block2D(make_patch(ed, cd, bc, be, "A0"), nni=nx1, nnj=ny,
                fill_condition=initial, label="BLOCK-1",
                hcell_list=[(9,0)], xforce_list=[0,0,1,0])
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(inflow, label="inflow-boundary") # one way to set a BC
blk_1.set_BC(EAST, EXTRAPOLATE_OUT, label="outflow-boundary")   # another way

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.compression_tolerance = -0.05 # the old default value
gdata.max_time = 5.0e-3  # seconds
gdata.max_step = 3000
gdata.dt = 1.0e-6
gdata.dt_plot = 1.5e-3
gdata.dt_history = 10.0e-5

sketch.xaxis(0.0, 1.0, 0.2, -0.05)
sketch.yaxis(0.0, 1.0, 0.2, -0.04)
sketch.window(0.0, 0.0, 1.0, 1.0, 0.05, 0.05, 0.17, 0.17)
```

11.2 Shell scripts

```
#! /bin/sh
# cone20_run.sh
# exercise the Navier-Stokes solver for the cone20 test case.
# It is assumed that the path is set correctly.

# Prepare the simulation input files (parameter, grid and initial flow data).
# The SVG file provides us with a graphical check on the geometry
e3prep.py --job=cone20 --do-svg
if [ "$?" -ne "0" ] ; then
    echo "e3prep.py ended abnormally."
    exit
fi

# Integrate the solution in time,
# recording the axial force on the cone surface.
time e3shared.exe -f cone20 --run --verbose
if [ "$?" -ne "0" ] ; then
    echo "e3shared.exe ended abnormally."
    exit
fi

# Extract the solution data and reformat.
# If no time is specified, the final solution found is output.
e3post.py --job=cone20 --vtk-xml

# Extract the average coefficient of pressure from the axial force
# records that were written to the simulation log file.
awk -f cp.awk e3shared.log > cone20_cp.dat

# Plot the average coefficient of pressure on the cone surface.
# We assume that the high-resolution data file is also available.
gnuplot <<EOF
set term postscript eps enhanced 20
set output "cone20_cp.ps"
set style line 1 linetype 1 linewidth 3.0
set title "20 degree cone in Mach 1.5 flow"
set xlabel "time, ms"
set ylabel "average C_p"
set xtic 1.0
set ytic 0.1
set yrange [0:0.5]
set key bottom right
set arrow from 5.2,0.387 to 5.8,0.387 nohead linestyle 1
set label "Value from\nNACA 1135\nChart 6" at 5.0,0.3 right
set arrow from 5.0,0.3 to 5.5,0.387 head
plot "cone20_cp.dat" using 1:2 title "10x40+30x40", \
     "cone20_cp_hi-res.dat" using 1:2 title "20x80+60x80" with lines
EOF

echo "At this point, we should have a solution that can be viewed with Paraview."
```

11.3 More specialized postprocessing

Beyond the usual slice-and-dice type of postprocessing, it may be useful to do specialized calculations on the flow data. Here, for example, the shock is expected to be straight and at an angle of 49.7^o with respect to the free-stream direction (Chart 5 in Ref. [3]). The `estimate_shock_angle.py` script uses the Python code libraries that the `e3post.py` is built upon to pick up the data, locate the shock position along each strip of cells in the x-direction, and then fit a straight line to the collected points. Note that the points from

the top right of the flow solution are omitted from the straight-line fit because the top boundary has interfered with the flow. The shock points and the fitted line are shown in Fig.8

Figure 8: Shock shape for Mach 1.5 flow over the 20-degree cone.

The script below uses the data reading and storage capability provided by the class `StructuredGridFlow`, which is imported from `e3_flow.py`. Given a file containing the flow data for a block of cells, this class has a `read` method that picks up the data. The flow and position data is stored in a dictionary, with one multidimensional numpy array for each variable. Access to the pressure in cell `i,j,k` of block `ib,jb` is achieved by putting these indices together as `blockData[ib][jb].data['p'][i,j,k]`. The core of the data handling is in the function `locate_shock_front()` in the middle of the script.

```
#!/usr/bin/env python
# estimate_shock_angle.py
# PJ, 11-Jan-2011

import sys, os, gzip
sys.path.append(os.path.expandvars("$HOME/e3bin"))
import math
import numpy
from getopt import getopt
from e3_flow import StructuredGridFlow

#-----------------------------------------------------------------
def locate_shock_along_strip(x, y, p):
    """
    Shock location is identified as a pressure rise along a strip of points.

    Input:
    x: sequence of float values, x-coordinate of each point
    y: sequence of float values, y-coordinate
    p: sequence of float values, static pressure in Pa
    Returns:
    x and y coordinates of a point on the shock.
```

```
    This function taken from the example 2D/sphere-heat-transfer.
    """
    n = len(x)
    p_max = max(p)
    p_trigger = p[0] + 0.3 * (p_max - p[0])
    x_old = x[0]; y_old = y[0]; p_old = p[0]
    for i in range(1,n):
        x_new = x[i]; y_new = y[i]; p_new = p[i]
        if p_new > p_trigger: break
        x_old = x_new; y_old = y_new; p_old = p_new
    frac = (p_trigger - p_old) / (p_new - p_old)
    x_loc = x_old * (1.0 - frac) + x_new * frac
    y_loc = y_old * (1.0 - frac) + y_new * frac
    return x_loc, y_loc

def locate_shock_front(jobName, nbi, nbj):
    """
    Reads flow blocks and returns the coordinates of the shock front.

    Input:
    jobName: string name used to construct file names
    nbi: number of blocks in the i-index direction
    nbj: number of blocks in the j-index direction

    It is assumed that the shock front will be located by scanning
    along the i-index direction, with j being constant for each search.

    This function taken from the example 2D/sphere-heat-transfer
    and is a bit more general than needed for the cone20 case.
    """
    blockData = []
    for ib in range(nbi):
        blockData.append([])
        for jb in range(nbj):
            blkindx = ib*nbj + jb
            fileName = 'flow/t9999/%s.flow.b%04d.t9999.gz' % (jobName, blkindx)
            fp = gzip.open(fileName, "r")
            blockData[ib].append(StructuredGridFlow())
            blockData[ib][-1].read(fp)
            fp.close()
    x_shock = []; y_shock = []
    for jb in range(nbj):
        nj = blockData[0][jb].nj
        for j in range(nj):
            x = []; y = []; p = [];
            for ib in range(nbi):
                ni = blockData[ib][jb].ni
                k = 0 # 2D only
                for i in range(ni):
                    x.append(blockData[ib][jb].data['pos.x'][i,j,k])
                    y.append(blockData[ib][jb].data['pos.y'][i,j,k])
                    p.append(blockData[ib][jb].data['p'][i,j,k])
            xshock, yshock = locate_shock_along_strip(x, y, p)
            x_shock.append(xshock)
            y_shock.append(yshock)
    return x_shock, y_shock

#-----------------------------------------------------------------
print "Begin estimate_shock_angle.py"

xs_all, ys_all = locate_shock_front("cone20", nbi=2, nbj=1)
# print "xs_all=", xs_all, "ys=", ys_all
# The shock interacts with the NORTH boundary and so bends after x=0.9m.
xs = [x for x in xs_all if x < 0.9]
ys = ys_all[0:len(xs)] # trim y-coordinate list to match
# print "xs=", xs, "ys=", ys
print "len(xs)=", len(xs), "len(ys)=", len(ys)

# Fit a straight-line to the computed shock points.
m, b = numpy.polyfit(xs, ys, 1)
print "m=", m, "b=", b
y2 = [m*x+b for x in xs]
shock_angle = math.atan(m)
```

```
print "shock_angle_rad=", shock_angle
print "shock_angle_deg=", shock_angle*180/math.pi

# Generate some points on the cone surface.
tan20 = math.tan(20.0*math.pi/180.0)
ycone = [tan20*(x-0.2) for x in xs]

# Average deviation of CFD shock points from fitted line.
d = 0
for i in range(len(xs)):
    d += abs(y2[i] - ys[i])
d /= len(xs)
print "average_deviation_metres=", d

# Optionally do the plot.
if len(sys.argv) > 1 and sys.argv[-1] == "--do-plot":
    import pylab
    pylab.plot(xs, ys, 'o', label='Eilmer3')
    pylab.hold(True)
    pylab.plot(xs, y2, label='fitted line')
    pylab.plot(xs, ycone, label='cone surface')
    pylab.title('Shock position')
    pylab.xlabel('x, m')
    pylab.ylabel('y, m')
    pylab.legend(loc='upper left')
    pylab.show()

print "Done."
```

11.4 Notes

- Remember that long-format command-line options start with two dashes. For example `--job=cone20`. These double dashes are a little hard to distinguish in the shell scripts.

- Run time is approximately 79 seconds for 1126 steps on a computer with an Intel Dual Pentium E2160, 1.6 GHz processor. As of September 2008, we have much optimisation to do. Of course, the shared-memory code does not make use of the second processor, however, there is an MPI version of the code that can.

- This cone20.py file really has full access to the Python interpreter on your system. Later examples will show how to use Python to write data files from within the input script. Be careful.

- Python is a dynamic language. It is easy to bind names to new objects within your script. Be careful that you do not rebind essential names that will be later used by the `e3prep.py` program. Where this might happen in a non-obvious way is in the importing of foreign modules (to do something interesting in your script) with the command "from *module-name* import *".

- Awk script for extracting x-force data from the simulation log file. New users might like to use an equivalent program written in Python.

```
# cp.awk
# Extract the simulation times and axial force values from the log file.
# The relevant lines in mb_cns.log start with the string "XFORCE"
# and are of the form:
#    XFORCE: t n jb ibndy fx_p fx_v [jb ibndy fx_p fx_v [jb ...]]
# Present the axial force as an average coefficient of pressure to
# compare with that obtained from NACA 1135.

BEGIN {
    p_inf = 95.84e3;  # Pa
    T_inf = 1103.0;   # K
    u_inf = 1000.0;   # m/s
    R = 287;          # J/kg.K
    r_base = 0.29118; # m
    rho_inf = p_inf / (R * T_inf);  # kg/m**3
    q_inf = 0.5 * rho_inf * u_inf * u_inf;  # Pa
    A = 3.14159 * r_base * r_base;  # m**2
    print "# time (ms)  Cp";
    print "# rho_inf= ", rho_inf, " q_inf= ", q_inf, " A= ", A
}

/XFORCE/ {
    # Select just the simulation time and the force on the cone surface.
    t = $3;  # in seconds
    f = $9;  # pressure force in Newtons
    # The coefficient of pressure is based on the difference
    # between the cone surface pressure and the free-stream pressure.
    Cp = (f / A - p_inf) / q_inf;
    print t*1000.0, Cp;
}
```

- The script `cone20_run_mpi.sh` is available for running the simulation with the parallel version of the code on a machine with LAM-MPI installed. This script is essentially the same as shown for the shared-memory simulation with the MPI simulation being started with the commands:

  ```
  mpirun -np 2 e3mpi.exe -f cone20 --run
  ```

  The only other modification required is to look for the surface-force data in the log file `e3mpi.0001.log` rather than `e3shared.log`.

- It turned out to be handy (while trying to debug one of those nasty memory segmentation violations) to be able to run the debug version of the MPI code within the valgrind instrumentation and profiling environment with the commands:

  ```
  $ cd ~/work/eilmer3/2D/cone20-simple/
  $ script xxxx
  $ mpirun -np 2 valgrind --tool=memcheck e3mpi.exe -f cone20 --run
  $ exit
  ```

  The `script` and `exit` start and stop recording of all of the console output to the file `xxxx`. If you have a few errors, valgrind output can be verbose.

12 Viscous Flow Along a Cylinder

This case (`2D/axi-cylinder/` computes the flow for a supersonic laminar boundary layer growing along a hollow cylinder. It was used in the original report[2] to verify the implementation of the viscous and axisymmetric terms in the code.

Figure 9: Flow domain for viscous flow along a cylinder.

The flow geometry consists of a hollow cylinder, 1.0m long with radius 0.005m, aligned with the x-axis. The flow domain shown in Figure 9 is defined by a quadrilateral with corners (1.0, 0.005), (1.0, 0.7), (0.0, 0.06), (0.0, 0.005). This region is shaped to capture the weak leading-edge-interaction shock while concentrating cells near the cylinder surface for the early part of the boundary layer development. The grid consists of 50×50 cells which are clustered toward the leading edge of the cylinder and (even more strongly in the y-direction) toward the cylinder surface.

The free stream is a uniform supersonic flow of air, modelled as a perfect gas with conditions

$$\rho = 0.00404 \text{ kg/m}^3, \quad u_x = 597.3 \text{ m/s}, \quad u_y = 0, \quad e = C_v T = 1.592 \times 10^5 \text{ J/kg},$$

$$T = 222 \text{ K}, \quad p = 257 \text{ Pa}, \quad M = 2.$$

This free stream condition is applied to the West and North boundaries, the East boundary is a supersonic outflow boundary and the South boundary (along the cylinder surface) is a no-slip boundary with temperature fixed at $T = 222$ K. The Reynolds number at the end of the plate is 1.65×10^5.

Initially, the flow throughout the block is set at the same conditions as the free stream and the governing equations are integrated in time. Figure 10 shows the pressure and temperature fields after a period of 8 ms. The weak leading-edge interaction shock is most clearly seen in the pressure field and the boundary layer on the cylinder surface is evident in the temperature field.

Figure 10: Pressure and temperature fields for viscous flow along a cylinder.

Figure 11 shows the x-velocity and temperature profiles through the boundary layer at x=0.916 m, 48 cells from the leading edge of the cylinder. The simulation data from `Eilmer3` are compared with data produced by David Pruett's spectral boundary layer code.

This case requires a fairly large computational effort of about 4 hours to reach a simulation time of 8 ms.

Figure 11: Velocity and temperature profiles at $x = 0.916\,\mathrm{m}$ for viscous flow along a cylinder.

12.1 Input script (.py)

```
## \file cyl50.py
## \author PJ
## \version 14-Aug-2006 updated from Tcl script
## \version 18-Jan-2010 updated for Eilmer3

gdata.title = "Mach 2 flow along the axis of a 5mm cylinder."
print gdata.title
gdata.axisymmetric_flag = 1

# Accept defaults giving perfect air (R=287 J/kg.K, gamma=1.4)
select_gas_model(model='ideal gas', species=['air'])

inflow  = FlowCondition(p=257.3, u=597.3, v=0.0, T=222.0)

# Set up a quadrilateral in the (x,y)-plane.
#  y     c
#  ^   / |
#  | /   |
#  d     |
#  a-----b
#  0------------> x
a = Node(0.0,0.005); b = Node(1.0,0.005); c = Node(1.0,0.7); d = Node(0.0,0.06)
south = Line(a,b); north = Line(d,c); west = Line(a,d); east = Line(b,c)

# The following lists are in order [N, E, S, W]
bndry_list = [SupInBC(inflow), ExtrapolateOutBC(), FixedTBC(222.0), SupInBC(inflow)]
cluster_list = [RobertsClusterFunction(1,0,1.1), RobertsClusterFunction(1,0,1.01),
                RobertsClusterFunction(1,0,1.1), RobertsClusterFunction(1,0,1.01)]

# Assemble the block from the geometry, discretization and boundary data.
blk = Block2D(psurf=make_patch(north, east, south, west, grid_type="AO"),
              nni=50, nnj=50,
              bc_list=bndry_list, cf_list=cluster_list,
              fill_condition=inflow)

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 8.0e-3  # seconds
gdata.max_step = 230000
gdata.dt = 3.0e-8
gdata.dt_plot = 4.0e-3

sketch.xaxis(0.0, 1.0, 0.2, -0.05)
sketch.yaxis(0.0, 1.0, 0.2, -0.04)
```

```
sketch.window(0.0, 0.0, 1.0, 1.0, 0.05, 0.05, 0.17, 0.17)
```

12.2 Shell scripts

```
#! /bin/sh
# cyl50_run.sh
e3prep.py --job=cyl50 --do-svg
time e3shared.exe --job=cyl50 --run
e3post.py --job=cyl50 --tindx=2 --vtk-xml

# Extract the profile near the downstream end of the cylinder.
e3post.py --job=cyl50 --tindx=2 --slice-list="0,47,:,0"

echo "At this point, we should have a new solution"
echo "Run cyl50_plot.sh next"
```

```
# cyl50_plot.sh
# Plot the profiles of temperature and velocity toward the end of the plate.

gnuplot <<EOF
set term postscript eps enhanced 20
set output "cyl50_profile_T.eps"
set style line 1 linetype 1 linewidth 3.0
set title "cyl50: Profile at x=0.917m"
set ylabel "y-R, mm"
set key top right
set xlabel "Temperature, K"
set yrange [0:25]
set xrange [200:270]
set style line 1 linetype 1 linewidth 4.0
plot "profile.data" using (\$20):(\$2-0.005)*1000 title "50x50 grid" with points 4, \
     "cyl50_dimensional.dat" using (\$2):(\$1-0.005)*1000.0 title "spectral" with lines ls 1
EOF

gnuplot <<EOF
set term postscript eps enhanced 20
set output "cyl50_profile_ux.eps"
set style line 1 linetype 1 linewidth 1.0
set title "cyl50: Profile at x=0.917m"
set ylabel "y-R, mm"
set xlabel "ux, m/s"
set key top left
set yrange [0:25]
set xrange [0:700]
set style line 1 linetype 1 linewidth 4.0
plot "profile.data" using (\$6):(\$2-0.005)*1000 title "50x50 grid" with points 4, \
     "cyl50_dimensional.dat" using (\$3):(\$1-0.005)*1000.0 title "spectral" with lines ls 1
EOF
```

12.3 Notes

- None

13 Mach 3 flow over a sharp-nosed two-dimensional body

The specifications for this example come from section 5.2 in JD Anderson's Hypersonics book [4]. It shows the use of a `spline` curve as well as being a source of test data for the Method-of-Characteristics for rotational flow. Data for the spline points was computed from

$$\frac{y}{y_e} = -0.008333 + 0.609425\left(\frac{x}{y_e}\right) - 0.092593\left(\frac{x}{y_e}\right)^2$$

where $y_e = 1.0$.

Figure 12: Schematic diagram of the geometry for the sharp body.

The surface pressure (shown in Fig. 14) has been extracted from the solution file by `e3post.py` by selecting the south-most line of cells of block 1. The pressure field (Fig. 15) shows the curved shock clearly.

Figure 13: Mesh, coloured by pressure, for the sharp body exercise.

Figure 14: Pressure data along the body surface.

Figure 15: The pressure field for flow over a sharp body. The data has been transformed from cells to points in Paraview. Note that the shock reflects from the upper boundary, which has a SLIP_WALL boundary condition by default.

13.1 Input script (.py)

```
# sharp.py
# PJ, 14-Dec-2006
#     16-Sep-2008 ported to Eilmer3
job_title = "Mach 3.0 flow over a curved 2D-planar body."
print job_title
gdata.title = job_title

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
# Define flow conditions
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)
inflow  = FlowCondition(p=95.84e3, u=2000.0, v=0.0, T=1103.0)

# Geometry
def shape(x):
    return -0.008333 + 0.609425*x - 0.092593*x*x

a = Node(-1.0, 0.0, label="A")
b = Node( 0.0, 0.0, label="B")
x_list = [0.5, 1.5, 2.5, 3.291]
b_list = [b,] # to accumulate points in the spline
for x in x_list:
    b_list.append(Node(x, shape(x)))
c = Node(10.0, b_list[-1].y, label="C") # extend at same y-value
d = Node(10.0, 7.0, label="D")
e = Node( 0.0, 7.0, label="E")
f = Node(-1.0, 7.0, label="F")

north0 = Line(f, e)
e0w1 = Line(b, e)
south0 = Line(a, b)
west0 = Line(a, f)
south1 = Polyline([Spline(b_list), Line(b_list[-1], c)])
north1 = Line(e, d)
east1 = Line(c, d)

# Define the blocks, boundary conditions and set the discretisation.
ny = 60
clustery = RobertsClusterFunction(1, 0, 1.3)
clusterx = RobertsClusterFunction(1, 0, 1.2)
blk_0 = Block2D(make_patch(north0, e0w1, south0, west0),
                nni=16, nnj=ny,
                cf_list=[None,clustery,None,clustery],
                fill_condition=initial)
blk_1 = Block2D(make_patch(north1, east1, south1, e0w1),
                nni=80, nnj=ny,
                cf_list=[clusterx,None,clusterx,clustery],
                fill_condition=initial)
identify_block_connections()
blk_0.bc_list[WEST]=SupInBC(inflow)
blk_1.bc_list[EAST]=ExtrapolateOutBC()


# Do a little more setting of global data.
gdata.flux_calc = ADAPTIVE
gdata.max_time = 15.0e-3  # seconds
gdata.max_step = 2500
gdata.dt = 1.0e-6

sketch.xaxis(0.0,10.0, 2.0, -0.6)
sketch.yaxis(0.0, 8.0, 2.0, -1.6)
sketch.window(0.0, 0.0, 10.0, 10.0, 0.05, 0.05, 0.17, 0.17)
```

13.2 Shell scripts

```
#! /bin/sh
# sharp_prep.sh
# A sharp axisymmetric body as described in Andersons Hypersonics text.

e3prep.py --job=sharp --do-svg

# Extract the initial solution data and reformat so that we can plot the grid.
e3post.py --job=sharp --tindx=0 --vtk-xml

echo At this point, we should be ready to start the simulation.
```

```
#! /bin/sh
# sharp_run.sh
# Exercise the Navier-Stokes solver for a sharp 2D body.

# Integrate the solution in time.
time e3shared.exe --job=sharp --run

echo At this point, we should have a final solution in sharp.b0000.t0015
```

```
#! /bin/sh
# sharp_post.sh
# Sharp 2D body, extract data and plot it.

# Extract the solution data over whole flow domain and reformat.
e3post.py --job=sharp --tindx=15 --vtk-xml

# Extract surface pressure and plot.
e3post.py --job=sharp --output-file=sharp_surface.dat --tindx=15 \
    --slice-list="1,:,0,0"

gnuplot <<EOF
set term postscript eps 20
set output "sharp_surface_p.eps"
set title "Sharp 2D Body in Mach 3 Freestream"
set xlabel "x, m"
set ylabel "Pressure, kPa"
set xrange [0.0:10.0]
set yrange [0.0:800]
plot "sharp_surface.dat" using 1:(\$9/1000) with lines
EOF

echo At this point, we should have a plotted data.
```

13.3 Notes

- For mbcns2, this simulation reached a final time of 15 ms in 1801 steps and, on a Pentium-M 1.73 Ghz system, taking 2 min, 48 s of CPU time.
- For Eilmer3, this simulation required 5 min, 22 sec on a single core of a Pentium 1.6 GHz processor. It reached the same time of 15 ms in 1838 steps. As of September 2008, we clearly have some optimisation to do.

14 Sharp-nosed 2D body – PyFun version

This is the same flow specifications as for the previous example but we directly use the functional form of the sharp body as supplied by Ref. [4].

$$\frac{y}{y_e} = -0.008333 + 0.609425\left(\frac{x}{y_e}\right) - 0.092593\left(\frac{x}{y_e}\right)^2$$

where $y_e = 1.0$. In the input script, the path is defined as a `PyFunctionPath` object that receives a function `xypath`. The function `xypath` accepts a parameter value $0.0 \leq t \leq 1.0$ and returns a corresponding point along the path as the Python tuple $(x(t), y(t), z(t))$. Note that it is *not* a `Vector` object as most of the other geometry objects expect.

14.1 Input script (.py)

```
# sharp-pyfun/sharp.py
# PJ, 14-Dec-2006
#     16-Sep-2008 ported to Elmer3
#     29-Apr-2009 PyPath used instead of spline.
#
gdata.title = "Mach 3.0 flow over a curved 2D-planar body."
print gdata.title

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
# Define flow conditions.
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)
inflow  = FlowCondition(p=95.84e3, u=2000.0, v=0.0, T=1103.0)
# One can get access to the details of the FlowCondition.
print "inflow M=", inflow.flow.u / inflow.flow.gas.a

# Geometry of flow domain.
def y(x):
    "(x,y)-space path for x>=0"
    if x <= 3.291:
        return -0.008333 + 0.609425*x - 0.092593*x*x
    else:
        return 1.0

def xypath(t):
    "Parametric path with 0<=t<=1."
    global y
    x = 10.0 * t
    return (x, y(x), 0.0)

a = Node(-1.0, 0.0, label="A")
b = Node( 0.0, 0.0, label="B")
c = Node(10.0, 1.0, label="C")
d = Node(10.0, 7.0, label="D")
e = Node( 0.0, 7.0, label="E")
f = Node(-1.0, 7.0, label="F")

north0 = Line(f, e)
e0w1   = Line(b, e)
south0 = Line(a, b)
west0  = Line(a, f)
south1 = PyFunctionPath(xypath)
north1 = Line(e, d)
east1  = Line(c, d)

# Define the blocks, grid resolution and boundary conditions.
```

```
ny = 60
clustery = RobertsClusterFunction(1, 0, 1.3)
clusterx = RobertsClusterFunction(1, 0, 1.2)
blk_0 = Block2D(make_patch(north0, e0w1, south0, west0),
                nni=16, nnj=ny,
                cf_list=[None,clustery,None,clustery],
                fill_condition=initial)
blk_1 = Block2D(make_patch(north1, east1, south1, e0w1),
                nni=80, nnj=ny,
                cf_list=[clusterx,None,clusterx,clustery],
                fill_condition=initial)
identify_block_connections()
blk_0.bc_list[WEST]=SupInBC(inflow)
blk_1.bc_list[EAST]=ExtrapolateOutBC()

# Do a little more setting of global data.
gdata.flux_calc = ADAPTIVE
gdata.max_time = 15.0e-3  # seconds
gdata.max_step = 2500
gdata.dt = 1.0e-6

sketch.xaxis(0.0,10.0, 2.0, -0.6)
sketch.yaxis(0.0, 8.0, 2.0, -1.6)
sketch.window(0.0, 0.0, 10.0, 10.0, 0.05, 0.05, 0.17, 0.17)
```

14.2 Notes on using Python for the input script

- The script runs in the context set up by the `e3prep.py` program. This means that data elements such as `gdata` are available for manipulation by the user's script.
- Comments can be used in the script as a form of documentation on the simulation.
- We can get intermediate results printed as the script is processed. This is useful for debugging and for documentation of the situation.
- It is often convenient to set up small functions that get passed as arguments to other functions. For example, the function `y` (brought over from the previous simulation) is passed into `xypath` which is, in turn, passed in to PyFunctionPath to construct a Path element.

15 Hypersonic flow of ideal air over a blunt wedge

This example is a partial solution to the CFD exercise for the MECH4470 class in 2004. Because the original specification was given in nondimensional form, an arbitrary 10 mm nose radius has been selected for the inviscid simulation. This is also a reasonable size for a possible wind tunnel experiment. The free-stream condition was specified as having a Mach number of 5 and the gas was specified as ideal air. Choosing particular values of $p_\infty = 100\,\text{kPa}$, $T_\infty = 100\,\text{K}$, lead to a free-stream velocity of $u_\infty = 1002\,\text{m/s}$ and a dynamic pressure of $q_\infty = 1.75\,\text{MPa}$.

Figure 16: Schematic diagram of the geometry for the blunted 10 degree wedge.

The simulation is started with low-pressure conditions throughout the flow domain and free-stream conditions applied to the inflow boundary (the west boundary of blk-0 and the north boundary of blk-1). The flow data is allowed to evolve until $t_{final} = 399\,\mu\text{s}$, which corresponds to a particle of the free-stream travelling 40 nose radii. The axial force (shown in Fig.18) is seen to settle to a value of 28590 N in that time. This corresponds to a drag coefficient of 0.674.

The surface pressure (shown normalised in Fig. 19) has been extracted from the solution file by `e3post.py` by selecting the east-most line of cells of the first block and the south-most line of cells of the second block. The selected data is filtered by an Awk script to produce the normalised data (and the Newtonian reference data) as plotted.

Figure 17: Mesh for the blunt wedge exercise.

Figure 18: History of the axial forces for the blunt-wedge exercise.

Figure 19: Surface pressure coefficient data for the blunt-wedge exercise.

Figure 20: Mach number data for the blunt-wedge exercise.

15.1 Input script (.py)

```
# bw.py
# MECH4470/CFD Exercise: Hypersonic flow over a blunt wedge.
# PJ, 07-Dec-2006
#     31-Jan-2010 ported to Eilmer3

from math import sqrt, sin, cos, tan, pi

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])

# Free stream
g_gas = 1.4                     # Ideal Air
R_gas = 287.0
M_inf = 5.0                     # Specified Mach number
p_inf = 100.0e3                 # Select a static pressure
T_inf = 100.0                   # and a temperature
a_inf = sqrt(T_inf * R_gas * g_gas) # determine sound speed
u_inf = M_inf * a_inf                    # and velocity
# Also, handy to know dynamic pressure for nondimensionalization
# of the pressures and drag forces.
q_inf = 0.5 * g_gas * p_inf * M_inf * M_inf
print "Free-stream velocity, u_inf=", u_inf
print "    static  pressure, p_inf=", p_inf
print "    dynamic pressure, q_inf=", q_inf
free_stream = FlowCondition(p=p_inf, u=u_inf, v=0.0, T=T_inf)
# For transient simulation, we start with a low pressure.
initial = FlowCondition(p=1000.0, u=0.0, v=0.0, T=100.0)

# Geometry
Rn = 10.0e-3                    # radius of cylindrical nose
xEnd = 8.0 * Rn                 # downstream extent of wedge
alpha = 10.0 / 180.0 * pi       # angle of wedge wrt free stream
delta = 10.0e-3                 # offset for inflow boundary

# First, specify surface of cylinder and wedge
a = Node(0.0, 0.0, label='a') # Centre of curvature for nose
b = Node(-Rn, 0.0, label='b')
c = Node(-Rn*sin(alpha), Rn*cos(alpha), label='c')
bc = Arc(b, c, a)
# Down-stream end of wedge
d = Node(xEnd, c.y+(xEnd-c.x)*tan(alpha), label='d')
print "height at end of plate yd=", d.y
cd =Line(c, d)

# Outer-edge of flow domain has to contain the shock layer
# Allow sufficient for shock stand-off at the stagnation line.
R2 = Rn + delta
e = Node(-R2, 0.0, label='e')
# The shock angle for a 10 degree ramp with sharp leading edge
# is 20 degrees (read from NACA 1135, chart 2),
# however, the blunt nose displaces the shock a long way out
# so we allow some more space.
# We need to set the boundary high enough to avoid the shock
R3 = Rn + 2.0 * delta
f = Node(-R3*sin(alpha), R3*cos(alpha), label='f')
# Now, put in intermediate control points so that we can use
# cubic Bezier curve for the inflow boundary around the nose
# and a straight line downstream of point f.
e1 = Node(e.x, delta, label='e1')
alpha2 = 40.0 / 180.0 * pi
f1 = Node(f.x-delta*cos(alpha2), f.y-delta*sin(alpha2), label='f1')
ef = Bezier([e, e1, f1, f])
g = Node(xEnd, f.y+(xEnd-f.x)*tan(alpha2), label='g')
fg = Line(f,g)

# Define straight-line segments between surface and outer boundary.
eb = Line(e, b); fc = Line(f, c); dg = Line(d, g)
```

```
# Define the blocks using the path segments.
# Note that the EAST face of region0 wraps around the nose and
# that the NORTH face of region0 is adjacent to the WEST face
# of region1.
region0 = make_patch(fc, bc, eb, ef)
cf = fc.copy(); cf.reverse() # common boundary but opposite sense
region1 = make_patch(fg, dg, cd, cf)
cluster0 = RobertsClusterFunction(0, 1, 1.2)
cluster1 = RobertsClusterFunction(1, 0, 1.2)
nni0 = 40
nnj0 = 40
nni1 = 100
blk_0 = Block2D(region0, nni=nni0, nnj=nnj0,
                cf_list=[cluster0,None,cluster0,None],
                fill_condition=initial,
                xforce_list=[0,1,0,0])
blk_1 = Block2D(region1, nni=nni1, nnj=nnj0,
                cf_list=[None,cluster1,None,cluster1],
                fill_condition=initial,
                xforce_list=[0,0,1,0])
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(free_stream)
blk_1.bc_list[NORTH] = SupInBC(free_stream)
blk_1.bc_list[EAST] = ExtrapolateOutBC()


# We can set individual attributes of the global data object.
job_title = "Blunt Wedge Rn=" + str(Rn)
job_title += (" q_inf=%12.3e" % q_inf) + (" d.y=%10.5f" % d.y)
print job_title
gdata.title = job_title
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 40.0 * Rn / u_inf
print "Final time=", gdata.max_time
gdata.max_step = 5000
gdata.dt = 1.0e-8
gdata.dt_plot = gdata.max_time
gdata.dt_history = gdata.max_time / 100.0
HistoryLocation(b.x-0.001, b.y) # just in front of the stagnation point

sketch.xaxis(-0.020, 0.080, 0.020, -0.004)
sketch.yaxis(0.0, 0.100, 0.020, -0.004)
sketch.window(-0.02, 0.0, 0.08, 0.10, 0.05, 0.05, 0.17, 0.17)
```

15.2 Shell scripts

```
# bw_prep.sh
#
e3prep.py --job=bw --do-svg
```

```
# bw_run.sh
#
time e3shared.exe --job=bw --run
mv e3shared.log bw.e3shared.log
echo "Done"
```

```
# bw_post.sh

e3post.py --job=bw --tindx=9999 --vtk-xml --add-mach

# Plot the surface pressure on the wedge
# We want the EAST edge of block 1 and the SOUTH edge of block 1
e3post.py --job=bw --tindx=9999 --output-file=bw_surface.data \
    --slice-list="0,-1,:,0;1,:,0,0"
awk -f surface_pressure.awk bw_surface.data > bw_surface_p_coeff.data

gnuplot <<EOF
set term postscript eps enhanced 20
set output "bw_surface_pressure.eps"
set title "Blunted wedge: surface pressure coefficient."
set xlabel "s/R_n"
set ylabel "Pressure Coefficient, (p - p_{/Symbol \245})/q_{/Symbol \245}"
set yrange [0.0:2.0]
plot "bw_surface_p_coeff.data" using 1:2 title "CFD at t=399us" with lines, \
     "bw_surface_p_coeff.data" using 1:3 title "Modified Newtonian" with lines
EOF

# Plot the axial force coefficient.
awk -f xforce.awk bw.e3shared.log > bw_xforce.data

gnuplot <<EOF
set term postscript eps 20
set output "bw_xforce.eps"
set title "Blunted wedge: x-force history"
set xlabel "t, microseconds"
set ylabel "x-force, N"
set yrange [0:35000]
set key top left
plot "bw_xforce.data" using 1:2 title "total" with lines, \
     "bw_xforce.data" using 1:3 title "cylinder" with lines, \
     "bw_xforce.data" using 1:4 title "wedge" with lines
EOF
```

15.3 Notes

- This simulation reaches a final time of 399 μs. For `mbcns2` on an Intel Pentium-M 1.73 Ghz system, this took 6 min, 39 s of CPU time for 3722 steps. However, for `Eilmer3` on an Intel E2140 1.6Ghz system it now takes 15 m, 23 s for 3759 steps.

- Selection of the `e3shared.log` file showing some x-force data as written during the simulation. Pressure and viscous forces are written separately. Note that the lines are written with several items separated by spaces and the format is mostly self-documenting. The only extra bit of information is that BNDY values are 0, 1, 2 and 3 for boundaries NORTH, EAST, SOUTH and WEST, respectively.

```
Step=    420 t= 2.747e-05 dt= 9.100e-08 WC=102.0 WCtFT=991.8 WCtMS=1112.3
CFL_min = 1.862345e-03, CFL_max = 4.958796e-01, dt_allow = 9.100331e-08
Smallest CFL_max so far = 3.381457e-02 at t = 1.000000e-07
 dt[0]=9.100331e-08 dt[1]=1.500771e-07
There are 2 active blocks.
RESIDUAL mass block 0 max: 4.899825e-02 at (-0.00280173,0.0146712,0)
RESIDUAL energy block 0 max: 5.025321e-02 at (-0.00280173,0.0146712,0)
RESIDUAL mass block 1 max: 1.656031e-01 at (0.0254165,0.0185377,0)
RESIDUAL energy block 1 max: 4.834703e-01 at (0.0262722,0.0181336,0)
RESIDUAL mass global max: 1.656031e-01 step 420 time 2.74667e-05
RESIDUAL energy global max: 4.834703e-01 step 420 time 2.74667e-05
XFORCE: TIME 2.801336e-05 BLOCK 0 BNDY 1 FX_P 2.415181e+04 FX_V 2.204480e+00
XFORCE: TIME 2.801336e-05 BLOCK 1 BNDY 2 FX_P 9.973561e+02 FX_V 9.133770e+00
```

- Awk filter for extracting the x-force data from the simulation log file. Note that there are two pattern-action rules, one for each block.

```
# xforce.awk
# Extract the simulation times and axial force values from the log file.
#

BEGIN {
    print "# time (microseconds)  x-force-total  only-cylinder only-wedge";
}

/XFORCE/ && $5 == 0 {
    # Select just the simulation time and the pressure forces for block 0.
    t = $3;  # in seconds
    fx_p_0 = $9;  # force on cylinder in Newtons
    # Don't do anything until we pick up the wedge data (block 1).
}

/XFORCE/ && $5 == 1 {
    # Select just the simulation time and the pressure forces for block 1.
    t = $3;  # in seconds
    fx_p_1 = $9; # wedge surface in Newtons
    print t*1.0e6, fx_p_0 + fx_p_1, fx_p_0, fx_p_1;
}
```

- Awk filter for normalising the surface pressure data.

```
# surface_pressure.awk
# Normalise the surface pressure with free-stream dynamic pressure and
# compute the distance around from the stagnation point.

BEGIN {
    q_inf = 1.750e6;   # free-stream dynamic pressure, Pa
    p_inf = 100.0e3;   # free-stream static pressure, Pa
    Rn = 10.0e-3;      # nose radius
    xold = -Rn;        # location of the stagnation point
    yold = 0.0;
```

```
    s = 0.0;           # distance around from stagnation point
    count = 0;
    pi = 3.1415927;
    wedge_angle = 10.0/180.0 * pi;
    print "# s/Rn  Cp(CFD)  Cp(Newton)  x(m)  y(m)";
}

$1 != "#" {
    count += 1;
    x = $1;            # cell-centre position
    y = $2;
    p = $9;            # cell-centre pressure
    if ( count == 1 ) p_pitot = p;  # Close enough to the stagnation point.
    dx = x - xold;
    dy = y - yold;
    s += sqrt(dx * dx + dy * dy);
    # Estimate Cp using Modified Newtonian Model.
    theta = 0.5 * pi - (s/Rn);  # local angle of surface
    if (theta < wedge_angle) theta = wedge_angle;
    Cp_MN = (p_pitot - p_inf) / q_inf * sin(theta) * sin(theta);
    print s/Rn, (p - p_inf)/q_inf, Cp_MN, x, y;
    xold = x;
    yold = y;
}
```

16 Pressure on a flat-faced cylinder

This example models the bar gauge type of pressure sensor as used in the expansion-tube facilities. It also shows the application of a multiple-block grid to describe the flow domain (Figure 21) around a flat-faced cylinder whose axis is aligned with the free-stream flow direction. The free-stream Mach number is 4.76 to match one of the higher Mach number conditions reported in Ref.[5].

Figure 21: Schematic diagram of the full flow domain around the flat-faced cylinder.

The simulation is started with low pressure stationary gas throughout the domain and the inflow conditions are applied to the west boundaries of blocks "Front" and "Outer". After allowing 50 μs for the flow to reach steady state, the pressure distribution throughout the domain is shown in Fig. 22. The stand-off distance was determined by searching for the pressure jump along the row of cells adjacent to the centreline. See the `locate_shock.awk` script below. If the trigger for the pressure jump is 200 kPa, the stand-off distance is 2.815 mm but, if we use a level of 1.5 MPa, the estimated stand-off distance is 2.756 mm. The difference is about 70% of one cell width.

Figure 23 shows the distribution of pressure across the face of the cylinder. The simulation data agrees closely with Kendall's measurements except in the region the sharp corner where there is inadequate resolution and an absence of viscous effects in the simulation.

Figure 22: Pressure and Mach number within the flow domain at 50 μs.

Figure 23: Normalised pressure across the face of the cylinder compared with experimental measurements [5].

16.1 Input script (.py)

```
# bar.py
# PJ
# 14-Dec-2006
# 03-Feb-2010 ported to Eilmer3 examples

gdata.title = "Bar gauge M=4.76 in air."
print gdata.title
# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
# Define flow conditions: low pressure ambient with M=4.76 inflow
initial = FlowCondition(p=30.0,    u=0.0,    v=0.0, T=300.0)
inflow  = FlowCondition(p=100.0e3, u=1653.0, v=0.0, T=300.0)

# Geometry
R = 5.0e-3    # radius of bar in metres
a = Node(-2*R, 0.0, label="A")
b = Node(-2*R, R,   label="B")
c = Node(-2*R, 6*R, label="C")
d = Node( 0.0, 0.0, label="D")
e = Node( 0.0, R,   label="E")
f = Node( 0.0, 6*R, label="F")
g = Node( 6*R, R,   label="G")
h = Node( 6*R, 6*R, label="H")

ad=Line(a,d); be=Line(b,e); cf=Line(c,f); eg=Line(e,g); fh=Line(f,h)
ab=Line(a,b); bc=Line(b,c); de=Line(d,e); ef=Line(e,f); gh=Line(g,h)

# Define the blocks, boundary conditions and set the discretisation.
nx0 = 120; nx2 = 120; ny0 = 40; ny1=80
cfy = RobertsClusterFunction(1, 0, 1.2)
cfx = RobertsClusterFunction(1, 0, 1.1)
blk_0 = Block2D(make_patch(be, de, ad, ab), nni=nx0, nnj=ny0,
                fill_condition=initial, label="Front",
                hcell_list=[(nx0,1),(nx0,5),(nx0,10)],
                xforce_list=[0,1,0,0])
blk_1 = Block2D(make_patch(cf, ef, be, bc), nni=nx0, nnj=ny1,
                cf_list=[None,cfy,None,cfy],
                fill_condition=initial, label="Outer")
blk_2 = Block2D(make_patch(fh, gh, eg, ef), nni=nx2, nnj=ny1,
                cf_list=[cfx,cfy,cfx,cfy],
                fill_condition=initial, label="After")
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(inflow)
blk_1.bc_list[WEST] = SupInBC(inflow)
blk_2.bc_list[EAST] = ExtrapolateOutBC()

# We can set individual attributes of the global data object.
gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 50.0e-6  # seconds
gdata.max_step = 15000
gdata.dt = 2.0e-8
gdata.dt_plot = 5.0e-6
gdata.dt_history = 0.5e-6

sketch.xaxis(-0.010, 0.030, 0.010, -0.002)
sketch.yaxis( 0.000, 0.030, 0.010, -0.002)
sketch.window(-0.010, 0.0, 0.030, 0.040, 0.05, 0.05, 0.17, 0.17)
```

16.2 Shell scripts

```
#!/bin/bash
e3prep.py --job=bar --do-svg
```

```
#!/bin/bash
# run_sumulation.sh
# catch both stdout and stderr
nohup time e3shared.exe --job=bar --run &> LOGFILE &
```

```
#!/bin/bash
# post_simulation.sh

# Extract the stagnation line data from the steady flow field.
e3post.py --job=bar --output-file=stag_line.data --tindx=9999 \
     --slice-list="0,:,1,0"
awk -f locate_shock.awk stag_line.data > result.txt

# Create a VTK plot file of the steady flow field.
e3post.py --job=bar --tindx=all --vtk-xml --add-mach --add-pitot-p

# Extract the flow data across the face of the bar gauge.
e3post.py --job=bar --output-file=raw_profile.data --tindx=9999 \
     --slice-list="0,-1,:,0"
awk -f normalize.awk raw_profile.data > norm_profile.data

gnuplot <<EOF
set output "bar_norm_p.eps"
set term postscript eps 20
set xrange [0:1.1]
set yrange [0:1.2]
set title "Normalized surface pressure over cylinder face, M=4.76."
set xlabel "r/r-max"
set ylabel "p/p-centre"
set key bottom left
plot "norm_profile.data" using 1:2 title "simulation" with lines, \
     "kendall_profile.data" using 1:2 title "experiment" with points 4
EOF
```

16.3 Awk scripts

```
# normalize.awk
# Normalize the surface pressure over the centreline static pressure.
BEGIN {
  p_centre = -1.0;
}

$1 != "#" {
  p = $9;
  r = $2;
  if (p_centre < 0.0) p_centre = p;
  print r/0.005, p/p_centre;
}
```

```
# locate_shock.awk

BEGIN {
    p_old = 0.0;
    x_old = -2.0;   # dummy position
    y_old = -2.0;
    p_trigger = 1.5e6; # something midway between free stream and stagnation
    shock_found = 0;
}

$1 != "#" { # for any non-comment line, do something
    p_new = $9;
    x_new = $1;
    y_new = $2;
    # print "p_new=", p_new, "x_new", x_new, "y_new", y_new
    if ( p_new > p_trigger && shock_found == 0 ) {
        shock_found = 1;
        frac = (p_new - p_trigger) / (p_new - p_old);
        x = x_old + frac * (x_new - x_old);
        y = y_old + frac * (y_new - y_old);
        print "shock-location= ", x, y
    }
    p_old = p_new;
    x_old = x_new;
    y_old = y_new;
}

END {
    if ( shock_found == 0 ) {
        print "shock not located";
    }
    print "done."
}
```

16.4 Notes

- The `mbcns2` version of this simulation reaches a final time of 50 μs in 2932 steps and, on a Pentium-M 1.73 Ghz system, this takes 19 min, 27 s of CPU time. This is equivalent to 17.8 μs per cell per predictor-corrector time step.

- The `Eilmer3` simulation takes 2929 steps and 19 min, 6 s on an Intel Core 2 Duo E8400 at 3GHz. We have some optimization to do...

17 Flow through a conical nozzle

Good quality experimental data for wall pressure distribution in a conical nozzle with a circular-arc throat profile and a 15° divergent section is available in Ref. [6]. In the original experiment the flow of air through the facility was allowed to reach steady state and static pressures were measured at a large number of points along the nozzle wall. In contrast, the present simulation is transient with just the transonic plus supersonic parts of the flow field reaching steady state.

Figure 24 shows the outline of the simulated flow domain which is set up to approximate the largest subsonic area ratio used in the experiment. Note that the geometric calculation of the tangent arcs is done within the input script. This makes use of Python, beyond just being an input format, and allows the specification to be fully parametric. This makes the initial setup of the script a bit more complex than absolutely necessary but does make the running of the simulation for other radii of curvature very simple.

The relatively long upstream part of the simulated tube provides the gas through an unsteady expansion from zero speed and pressure of 500 kPa (state 4) up to a small Mach number (state 3). These state labels refer to the those for the hypothetical shock tube problem in which state 1 is the initial low-pressure condition, state 4 is the initial high-pressure condition, state 2 is the post-shock condition of the low-pressure gas, and state 3 is the expanded high-pressure gas condition. Assuming that flow in the subsonic and transonic regions of the nozzle is steady, the expected Mach number is $M_3 = 0.13812$ for an area ratio of $A_3/A_* = 4.2381$.

Figure 24: Schematic diagram of the full flow domain for the duct and conical nozzle.

Along the C_+ characteristic connecting states 4 and 3, the Riemann invariant can be written as

$$J_+ = u + \frac{2a}{\gamma - 1} \quad ,$$

and so the relation between states 4 and 3 can be written as

$$\frac{a_4}{a_3} = 1 + \frac{\gamma - 1}{2} M_3 \quad .$$

Because the unsteady expansion is isentropic and $a = \sqrt{\gamma RT}$, the pressure ratio can be written as

$$\frac{p_4}{p_3} = \left[1 + \frac{\gamma - 1}{2} M_3\right]^{2\gamma/(\gamma-1)}$$

which gives a specific pressure ratio of $p_4/p_3 = 1.2102$. Since the experiment used a steady expansion from a large reservoir at stagnation conditions to the equivalent of our state 3, the corresponding stagnation pressure for state 3 is computed from

$$\frac{p_{03}}{p_3} = \left[1 + \frac{\gamma - 1}{2} M_3^2\right]^{\gamma/(\gamma-1)}$$

which gives $p_{03} = 418.7\,\text{kPa}$ in the current simulation.

Figure 25 shown part of the mesh around the nozzle overlaid with pressure contours. Figure 26 shows the pressure distribution throughout the flow domain at $t = 1.0\,\text{ms}$. A shock, starting at the transition from circular arc to straight wall in the early supersonic part of the nozzle, can be seen propagating toward the centreline as the flow proceeds to the exit plane.

Figure 25: Downstream-end of the mesh generated for the axisymmetric nozzle simulation.

The flow in the nozzle is reasonably steady, as indicated by the histories shown in Fig. 27 but the unsteady expansion can be seen reflecting from the inflow boundary at $x = -0.245\,\text{m}$ in Fig.26.

Figure 28 shows that the simulation matches the experimental data closely. The reflected expansion is shown clearly in the left figure and indicates that the subsonic boundary condition is not working as well as it should.

Figure 26: Pressure contours within the flow domain at 1.0 ms.

Figure 27: Development of the flow at a "history point" near the centre of the exit plane: (a) Mach number; (b) static pressure.

Figure 28: Normalised pressure distribution along the nozzle wall: (a) full length of flow domain; (b) just the supersonic part of the nozzle.

17.1 Input script (.py)

```
# back.py
# Conical nozzle from Back, Massier and Gier (1965)
gdata.title = "Flow through a conical nozzle."
print gdata.title

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
stagnation_gas = FlowCondition(p=500.0e3, T=300.0)
low_pressure_gas = FlowCondition(p=30.0, T=300.0)

# Define geometry.
# The original paper specifies sizes in inches, Eilmer3 works in metres.
inch = 0.0254 # metres
L_subsonic = 10.0 * inch
L_nozzle = 3.0 * inch
R_tube = 1.5955 * inch
R_throat = 0.775 * inch
R_curve = 1.55 * inch # radius of curvature of throat profile
theta = 15.0 * math.pi / 180.0 # radians

# Compute the centres of curvature for the contraction profile.
height = R_throat + R_curve
hypot = R_tube + R_curve
base = math.sqrt(hypot*hypot - height*height)
centre_A = Node(0.0, height, label="centre_A")
centre_B = Node(-base, 0.0, label="centre_B")
fraction = R_tube/hypot
intersect_point = centre_B + Vector(fraction*base, fraction*height)

# The following Nodes will be rendered in the SVG file.
z0 = Node(-L_subsonic, 0.0)  # assemble from coordinates
p0 = Node(-L_subsonic, R_tube)
z1 = Node(centre_B)  # initialize from a previously defined Node
p1 = Node(centre_B + Vector(0.0,R_tube))  # vector sum
p2 = Node(intersect_point)
z2 = Node(p2.x, 0.0)  # on the axis, below p2
z3 = Node(0.0, 0.0)
p3 = Node(0.0, R_throat)
# Compute the details of the conical nozzle
p4 = Node(R_curve*math.sin(theta), height - R_curve*math.cos(theta))
z4 = Node(p4.x, 0.0)
L_cone = L_nozzle - p4.x
```

```
p5 = Node(p4 + Vector(L_cone, L_cone*math.tan(theta)))
z5 = Node(p5.x, 0.0)

north0 = Polyline([Line(p0,p1),Arc(p1,p2,centre_B),Arc(p2,p3,centre_A)])
east0west1 = Line(z3, p3)
south0 = Line(z0, z3)
west0 = Line(z0, p0)
north1 = Polyline([Arc(p3,p4,centre_A), Line(p4,p5)])
east1 = Line(z5, p5)
south1 = Line(z3, z5)

# Define the blocks, boundary conditions and set the discretisation.
nx0 = 180; nx1 = 60; ny = 30
subsonic_region = Block2D(make_patch(north0, east0west1, south0, west0),
                          nni=nx0, nnj=ny,
                          fill_condition=stagnation_gas,
                          label="subsonic-region")
supersonic_region = Block2D(make_patch(north1, east1, south1, east0west1),
                            nni=nx1, nnj=ny,
                            fill_condition=low_pressure_gas,
                            label="supersonic-region")
identify_block_connections()
subsonic_region.bc_list[WEST] = SubsonicInBC(stagnation_gas)
supersonic_region.bc_list[EAST] = ExtrapolateOutBC()

# Flow-history to be recorded at the following points.
HistoryLocation(0.001, 0.002, label="nozzle-throat")
HistoryLocation(L_nozzle-0.001, 0.002, label="nozzle-exit")

# Do a little more setting of global data.
gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 1.0e-3  # seconds
gdata.max_step = 5000
gdata.dt = 1.0e-7
gdata.dt_plot = 0.2e-3
gdata.dt_history = 10.0e-6

sketch.xaxis(-0.250, 0.080, 0.050, -0.01)
sketch.yaxis( 0.0,   0.050, 0.020, -0.015)
sketch.window(-0.25, 0.0, 0.10, 0.05, 0.03, 0.03, 0.24, 0.06)
```

17.2 Shell scripts

```
#!/bin/sh
#  back_run.sh
#  Exercise the Navier-Stokes solver for the conical nozzle
#  as used by Back, Massier and Gier (1965) AIAA J. 3(9):1606-1614.
e3prep.py --job=back --do-svg
time e3shared.exe --job=back --run
e3post.py --job=back --tindx=9999 --vtk-xml
```

```
# back_profile.sh
# Extract the flow data along the nozzle wall,
# scale it so that it can be lotted with the experimental data
# and plot it using gnuplot.

e3post.py --job=back --output-file=raw_profile.data --tindx=9999 \
    --slice-list=":,:,-1,0"

awk -f normalize.awk raw_profile.data > norm_profile.data
```

```
gnuplot <<EOF
set term postscript eps 20
set output 'back_profile_whole.eps'

set title 'Pressure along the nozzle wall'
set xlabel 'distance from throat (inches)'
set ylabel 'p/pt'

set yrange [0:1.2]
plot 'norm_profile.data' using 1:2 title "simulation" with lines, \
     'back-exp.data' using 1:2 title "experiment" with points
EOF

gnuplot <<EOF
set term postscript eps 20
set output 'back_profile_supersonic.eps'

set title 'Pressure along the nozzle wall'
set xlabel 'distance from throat (inches)'
set ylabel 'p/pt'

set xrange [-1.0:3.0]
set yrange [0:0.6]
plot 'norm_profile.data' using 1:2 title "simulation" with lines, \
     'back-exp.data' using 1:2 title "experiment" with points
EOF
```

```
# back_history.sh
# Extract the flow history data at the nozzle exit plane.
# This is then plotted using gnuplot and an assessment
# can be made as to whether the flow has reached steady state.

awk -f extract-history.awk < hist/back.hist.b0001 > nozzle-exit.data

gnuplot <<EOF
set term postscript eps 20
set output 'back_history_M.eps'

set title 'Mach number history at the nozzle exit'
set xrange [0.0:1.0]
set xlabel 'time, ms'
set ylabel 'M'

plot 'nozzle-exit.data' using 1:2 with lines
EOF

gnuplot <<EOF
set term postscript eps 20
set output 'back_history_p.eps'

set title 'Static pressure history at the nozzle exit'
set key bottom right
set xrange [0.0:1.0]
set xlabel 'time, ms'
set ylabel 'p, kPa'

plot 'nozzle-exit.data' using 1:3 with lines
EOF
```

17.3 Notes

- The simulation reaches a final time of 1 ms in 1448 steps and, on an Intel E2140 1.60 Ghz system, this takes 5 min, 16 s of CPU time.

- When run using `mbcns2`, the simulation reaches a final time of 1 ms in 1431 steps and, on a Pentium-M 1.73 Ghz system, this takes 2 min, 50 s of CPU time. This is equivalent to 16.5 μs per cell per predictor-corrector time step.

- The pressure is normalised with respect to the stagnation pressure using the following AWK script.

```
# normalize.awk
# Normalize the surface pressure over the length of the nozzle.
BEGIN {
    p0 = 418.7e3
    print "# Normalized surface pressure for the Back nozzle (simulation)"
    print "# x(inches) p/pt"
}

$1 != "#" {  # For non-comment lines in the data file do...
   p = $9
   r = $2
   x = $1
   print x/0.0254, p/p0
}
```

- The history data for all of the history cells in a particular block are written to the one file. A particular cell can be extracted as shown by the following AWK script.

```
# extract-history.awk
BEGIN {
    print "# t(ms)  Mach  p(kPa)";
}

$2==59 && $3==1 {
    t = $1;
    u = $10;
    v = $11;
    a = $14;
    p = $13;
    print t*1000.0, sqrt(u*u+v*v)/a, p/1000.0;
}
```

18 Flow of equilibrium air over a sphere

This example is a good starting-point for the modelling of hypersonic flow over blunt bodies. It shows the use of arcs and the use of a look-up table as the equation of state for a gas in chemical equilibrium but it remains geometrically simple by using a single-block grid. Also, the .py file makes use of the Python language to parameterize the simulation's specification.

Figure 29: Schematic diagram of the geometry for a sphere wrapped by a single-block grid.

The free-stream condition ($p_\infty = 20\,\text{kPa}$, $T_\infty = 296\,\text{K}$, $u_\infty = 4.68\,\text{km/s}$) corresponds to Case 3 in Ref. [7] with $M_\infty = 13.6$. According to Sawada & Dendou [7], the air is close to being in chemical equilibrium and there is a very thin boundary layer. The results show that the inviscid simulation does indeed capture some of the high-temperature chemistry influence. Ideal stagnation temperature would be 11204 K whereas the simulated temperature along the stagnation line rises to only 6081 K. Secondly, the stand-off distance for an ideal gas is expected to be approximately 4.63 mm. In Fig. 31 the simulated shock stand-off distance is 2.66 mm near the stagnation point. This is within 3% of the experimental value obtained by D. Reda in Sandia's Ballistics Range (see [7]).

Figure 30: Mesh for flow over a sphere.

Figure 31: Temperature field and shock-detector (S) for equilibrium-air flow over a sphere.

18.1 Input script (.py)

```
# file: ss3.py
#
# Sphere in equilibrium air modelling Case 3 from
#     K. Sawada & E. Dendou (2001)
#     Validation of hypersonic chemical equilibrium flow calculations
#     using ballistic-range data.
#     Shock Waves (2001) Vol. 11, pp 43--51
#
# Experimental shock stand-off distance is 2.59mm
# Sawada & Dendou CFD value:               2.56mm
#
# This script derived from rbody, 22-Jan-2004.
# and the Python version: ss3.py, 04-Apr-2005, 10-Aug-2006, 27-Nov-2006
# PJ
#
# The grid is a bit wasteful because the shock lies close to
# the body for equilibrium air, however, this grid layout
# (as used in rbody) allows us to play with perfect-gas models
# without hitting the inflow boundary with the shock.
#
# Updated: 12-Nov-2008 by RJG for use in Elmer3


# The following JOB name is used to build file names at the end.
JOB = "ss3"

# Radius of body
R = 31.8e-3                     # m
T_body = 296.0                  # surface T, not relevant for inviscid flow
body_type = "sphere"            # choose between "cylinder" and "sphere"

# Free-stream flow definition
p_inf = 20.0e3                  # Pa
T_inf = 296.0                   # degrees K
u_inf = 4.68e3                  # flow speed, m/s

# For equilibrium chemistry, use the look-up-table (which has
# been previously created).
print "About to select gas model."
select_gas_model(fname='cea-lut-air.lua.gz')
print "Gas model selection: done."

# Define flow conditions
inflow = FlowCondition(p=p_inf, u=u_inf, v=0.0, T=T_inf)
initial = FlowCondition(p=0.3*p_inf, u=0.0, v=0.0, T=T_inf)

# Job-control information
do_viscous = 0                  # flag for viscous/inviscid calc
nn = 60                         # grid resolution, both ix and iy
t_final = 10.0 * R / u_inf      # allow time to settle at nose
t_plot = t_final / 1.0          # plot only once
TitleText = "Blunt Body " + JOB + ": R=" + str(R) + ", gas='equilibrium air'" + \
            ", p=" + str(p_inf) + ", v=" + str(u_inf) + ", T=" + str(T_inf) + \
            ", viscous=" + str(do_viscous)
gdata.title = TitleText
gdata.case_id = 0
if do_viscous:
    gdata.viscous_flag = 1
    gdata.viscous_delay = t_plot
if body_type == "sphere":
    gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = t_final
gdata.max_step = 400000
gdata.dt = 1.0e-8
gdata.cfl = 0.40
gdata.dt_plot = t_plot
gdata.dt_history = 1.0e-6
```

```
# Begin geometry details...
# Note that mbcns_prep.py has already imported the math module.
deg2rad = math.pi / 180.0
alpha1 = 135.0 * deg2rad
alpha2 = 50.8 * deg2rad
# The node coordinates are scaled with the body radius.
# The labels are not required but make the MetaPost plot
# look a little like the plot produced by scriptit.tcl.
a = Node(0.0, 0.0, label="a")
b = Node(-1.0 * R, 0.0, label="b")
c = Node(math.cos(alpha1) * R, math.sin(alpha1) * R, label="c")
d = Node(0.0, R, label="d")
e = Node(math.cos(alpha2) * R, math.sin(alpha2) * R, label="e")
f = Node(1.4 * R, 1.5 * R, label="f")
g = Node(1.5 * R, 2.5 * R, label="g")
h = Node(1.5 * R, 3.5 * R, label="h")
i = Node(-1.5 * R, 0.0, label="i")
j = Node(-1.5 * R, 1.5 * R, label="j")
k = Node(-1.0 * R, 2.8 * R, label="k")

east0  = Polyline([Arc(b, c, a), Arc(c, d, a), Arc(d, e, a)])
north0 = Bezier([e, f, g, h,]); north0.reverse()
south0 = Line(i, b)
west0  = Bezier([i, j, k, h,])

print "ss3: block to be defined."
cluster_functions = [RobertsClusterFunction(0, 1, 1.2),
                     RobertsClusterFunction(1, 0, 1.1),
                     RobertsClusterFunction(0, 1, 1.2),
                     RobertsClusterFunction(1, 0, 1.1)]
boundary_conditions = [ExtrapolateOutBC(), FixedTBC(T_body),
                       SlipWallBC(), SupInBC(inflow)]

blk_0 = Block2D(psurf=make_patch(north0, east0, south0, west0),
                fill_condition=initial,
                nni=nn, nnj=nn,
                cf_list=cluster_functions,
                bc_list=boundary_conditions,
                label="BLOCK-0", hcell_list=[(nn,1)])

# Some hints to scale and place the sketch.
# If you change the radius, you'll probably have to adjust the axes.
sketch.xaxis( -0.060, 0.050, 0.020, -0.010)
sketch.yaxis(  0.0,   0.110, 0.020, 0.0)
sketch.window(-1.5*R, 0.0, 1.5*R, 3.0*R, 0.05, 0.05, 0.15, 0.15)
```

18.2 Shell scripts

The `ss3_setup_lut.sh` script assumes a "standard" location for the `e3bin` directories in order to find the files for the look-up-table gas model. The first form of the look-up-table has been generated as a regular array of sample points over ranges of density and temperature. When reformatting the table to have a regular array of data points over density and internal-energy, there is an option `--extrapolate` to instruct the program to extrapolate when necessary. When this option is not given, the final table covers smaller ranges of density and internal-energy that fall completely within the original sampled data.

```
#! /bin/sh
# file: ss3_setup_lut.sh

build-cea-lut --case=air

echo "We should now have a Look-Up-Table for air"
```

```
# ss3_run_py.sh
# Shell script to set up and run Sawada & Dendou's sphere case 3.

# For a clean start
e3prep.py --job=ss3.py --do-svg

# The main event
time e3shared.exe --job=ss3 --run
```

```
# ss3_post.sh
# By default, e3post.py grabs the solution at final time.
e3post.py --job=ss3 --vtk-xml

e3post.py --job=ss3 --slice-list="0,:,0,:" --output-file=ss3_stag_line.data
awk -f locate_shock.awk ss3_stag_line.data > ss3.result
```

18.3 Notes

- Going back a couple of years, the mbcns2 simulation finished at a final time of 67.95 μs in 4548 steps and, on a Pentium-M 1.73 Ghz system, this took 5 min, 6 s of CPU time. Eilmer3 is a bit slower, requiring 8 min, 38 s of CPU time on a Pentium E2140 (1.6 GHz) for 4556 steps.

- Awk script for extracting the shock location from the stagnation-line flow data.

```
# locate_shock.awk

BEGIN {
    p_old = 0.0;
    x_old = -2.0;   # dummy position
    y_old = -2.0;
    p_trigger = 2.0e6; # something midway between free stream and stagnation
    shock_found = 0;
}

$1 != "#" { # for any non-comment line, do something
    p_new = $9;
    x_new = $1;
    y_new = $2;
    # print "p_new=", p_new, "x_new", x_new, "y_new", y_new
    if ( p_new > p_trigger && shock_found == 0 ) {
        shock_found = 1;
        frac = (p_new - p_trigger) / (p_new - p_old);
        x = x_old + frac * (x_new - x_old);
        y = y_old + frac * (y_new - y_old);
        print "shock-location= ", x, y
    }
    p_old = p_new;
    x_old = x_new;
    y_old = y_new;
}

END {
    if ( shock_found == 0 ) {
        print "shock not located";
    }
    print "done."
}
```

19 Heat transfer to a sphere in equilibrium air

This example continues the modelling of hypersonic flow over blunt bodies and looks at the heat transfer to a spherical probe [8] in high temperature equilibrium air. It takes use of the Python language further by automating the process of running a simulation, adjusting the grid and then running a subsequent simulation on the the adjusted grid. The specific input file for each stage of the overall simulation is constructed from a template in which a few parameters are left unspecified. Most of the effort has gone into the coordinating script which has functions for running stages of the simulation as subprocesses and also has functions which fit a Bezier curve to the shock located in the flow field.

Figure 32: Schematic diagram of the geometry for a sphere wrapped by a SuperBlock2D grid.

The original experiments used a probe with a spherical nose, located in a small shock tube. The free-stream flow was initiated with the arrival of a strong shock and the useful test period in the experiments was terminated with the arrival of driver gas. From Figure 12 in Rose and Detra's paper [8], we choose the point corresponding to $p_1 = 1\,\text{cm}$ Hg (1.33 kPa) and M_s=8 which has a stagnation-point heat transfer of $30 \pm 2.0\,\text{MW/m}^2$. To keep the grid resolution requirements small, we will start with an initial test gas pressure $p_1 = 6.7\,\text{Pa}$ much lower than that used in the original experiments. Assuming that the chemistry doesn't change too much with the change in pressure, we can scale the

stagnation-point heat transfer as $\dot{q}_{s-sim} = \left(\frac{p_{1-sim}}{p_{1-expt}}\right)^{0.5} \dot{q}_{s-expt}$ to get an expected value of $2.212 \pm 0.14\,\mathrm{MW/m^2}$ for our low-pressure simulation.

For a $M_s = 8$ incident shock in air at 296 K, the post-shock, free-stream conditions are $p_\infty = 535.6\,\mathrm{kPa}$, $T_\infty = 2573.5\,\mathrm{K}$, and $u_\infty = 2436.5\,\mathrm{m/s}$. This assumes fully-equilibrium chemistry for the gas. The snap-shots of results for the staged simulation are shown in Figures 33 through 37 which show the temperature field at the end of each stage and the mesh used for that stage.

Figure 33: Temperature field and mesh for stage 0. The control points for the Bezier curve have been set so as to accommodate a shock in ideal (nonreacting) air and the clustering is fairly strong so that the boundary layer on the sphere surface may be resolved. The wall-clock time required to run this simulation 10 body lengths (27μs) is 23 seconds on 4 processors of `geyser`. 10103 time steps were made and the size of the time step at the end of the simulation is 2.479 ns. At the end of the simulation, the estimated value of stagnation-point heat transfer is $\dot{q}_s = 2.156\,\mathrm{MW/m^2}$ and the cell Reynolds number at the stagnation point is $\mathrm{Re}_{wall} = \frac{\rho_{wall} a_{wall} \Delta x}{\mu_{wall}} = 3.85$. Here Δx is the width of the cell out from the wall.

Figure 38 shows the distribution of heat transfer around the nose compared with the experimental data reported Kemp, Rose and Detra [9]. In the simulation data, there are small distrubances at the corners of blocks (at approximately 20 degrees and then again approaching 90 degrees) but they are quite small.

Figure 34: Temperature field and mesh for stage 1. The Bezier points have been adapted to the shock from stage 0 but the number of cells in each direction remains at 20×20, as for stage 0. With the finer cells, the size of the time step decreased and this stage required 55 seconds of wall-clock time to extend the simulation a further 10 body lengths in 23125 time steps. $\dot{q}_s = 2.260\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 2.94$

Figure 35: Temperature field and mesh for stage 2. The Bezier points have not been adapted further for this stage but the number of cells has been increased to 30×30. The size of the time step decreased further and this stage required 130 seconds of wall-clock time to extend the simulation only 5 body lengths (13.5 μs) in 24293 time steps. $\dot{q}_s = 2.257\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 2.39$

Figure 36: Temperature field and mesh for stage 3. The Bezier points have been adapted to the shock from stage 2 and the cells have been increased to 40×40. The size of the time step is now 0.319 ns and this stage required 469 seconds of wall-clock time to extend the simulation only 5 body lengths (13.5 μs) in 42660 time steps. $\dot{q}_s = 2.260\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 1.99$

.

Figure 37: Temperature field and mesh for stage 4. The Bezier points have not been further adapted but the number of cells has been increased to 80×80 to test the sensitivity of the heat transfer estimate. The size of the time step is now 0.086 ns and this stage required 8950 seconds of wall-clock time to extend the simulation a further 5 body lengths (13.5 μs) in 157420 time steps. $\dot{q}_s = 2.217\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 1.25$

.

Figure 38: Temperature around the sphere for stage 4. The experimental data is from Kemp, Rose and Detra [9].

.

19.1 Template input script (.py)

```
# file: sphere.input.template
#
# Sphere in equilibrium air modelling the Kemp Rose and Detra experiment.
# This input file template is specialized for each stage of the simulation.
# The $$name items seen below will be substituted with specific values.
#
# PJ, 22-Feb-2010
jobName = '$jobName'; stage = $stage; np = $np

# For equilibrium chemistry, use the look-up-table.
select_gas_model(fname='cea-lut-air.lua.gz')
inflow = FlowCondition(p=$p_inf, u=$u_inf, T=$T_inf)
initial = FlowCondition(p=$p_init, T=$T_inf)

# Job-control information
t_final = $body_lengths * $R / $u_inf  # allow time to settle at nose
t_plot = t_final / 5.0           # plot several times
gdata.title = "Spherical Blunt Body: R=" + str($R) + \
            ", p=" + str($p_inf) + ", v=" + str($u_inf) + \
            ", T=" + str($T_inf) + ", viscous=" + str($viscous_flag)
gdata.viscous_flag = $viscous_flag
gdata.viscous_delay = $viscous_delay
gdata.viscous_factor_increment = 0.02
gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = t_final
gdata.max_step = 800000
gdata.dt = 1.0e-10
gdata.cfl = 0.30
gdata.dt_plot = t_plot
gdata.dt_history = 1.0e-6

# Begin geometry details for a single region around a spherical nose.
# The node coordinates are scaled with the body radius.
a = Node(0.0, 0.0, label="a")
b = Node(-1.0*$R, 0.0, label="b")
c = Node(0.0, $R, label="c")
# The inflow boundary is a Bezier curve.
d = []; x_d = $x_d; y_d = $y_d
for i in range(len(x_d)):
    d.append(Node(x_d[i], y_d[i], label="d[%d]"%i))
# order of boundaries: N, E, S, W
flow_domain0 = make_patch(Line(d[-1],c), Arc(b,c,a), Line(d[0],b), Bezier(d))
cluster_functions0 = [RobertsClusterFunction(0, 1, 1.02), RobertsClusterFunction(1, 0, 1.06),
                      RobertsClusterFunction(0, 1, 1.02), RobertsClusterFunction(1, 0, 1.05)]
boundary_conditions0 = [ExtrapolateOutBC(), FixedTBC($T_body),
                        SlipWallBC(), SupInBC(inflow)]
if stage == 0:
    # We start from scratch.
    fill_condition0 = initial
else:
    # We start with the previous solution.
    rootName = jobName + str(stage-1)
    fill_condition0 = ExistingSolution(rootName, '.', np, 9999)
blk = SuperBlock2D(psurf=flow_domain0, fill_condition=fill_condition0,
                   nni=$ni, nnj=$nj, nbi=$nbi, nbj=$nbj,
                   cf_list=cluster_functions0, bc_list=boundary_conditions0,
                   label="blk")

sketch.xaxis(-15.0e-3, 5.0e-3, 5.0e-3, -0.002)
sketch.yaxis(0.0, 20.0e-3, 5.0e-3, 0.0)
sketch.window(-1.5*$R, 0.0, 1.5*$R, 3.0*$R, 0.05, 0.05, 0.15, 0.15)
```

19.2 Coordinating script (.py)

```
#!/usr/bin/env python
# run_adaptive_simulation.py
#
# Top-level script to coordinate the running of the
# solution-adaptive simulation in stages.
# This approximates a form of solution adaptivity in that
# the grid is adjusted to the shock occasionally.
# The grid is also refined with the stages.
#
# PJ, 22-Feb-2010

import shlex, subprocess, string
from subprocess import PIPE
import sys, os, gzip
sys.path.append(os.path.expandvars("$HOME/e3bin"))
from e3_flow import StructuredGridFlow

#-----------------------------------------------------------------
def prepare_input_script(substituteDict, jobName, stage):
    """
    Prepare the actual input file from a template.
    """
    stageName = jobName + str(stage)
    templateFileName = jobName + ".input.template"
    scriptFileName = stageName + ".py"
    fp = open(templateFileName, 'r')
    text = fp.read()
    fp.close()
    template = string.Template(text)
    text = template.substitute(substituteDict)
    fp = open(scriptFileName, 'w')
    fp.write(text)
    fp.close()
    return

def run_command(cmdText):
    """
    Run the command as a subprocess.
    """
    print "About to run cmd:", cmdText
    args = shlex.split(cmdText)
    p = subprocess.Popen(args)
    stdoutData, stderrData = p.communicate()
    # wait until the subprocess is finished
    return

def run_stage(paramDict, jobName, stage):
    """
    Set up and run one stage of the simulation as a normal job.
    """
    prepare_input_script(paramDict, jobName, stage)
    stageName = jobName+str(stage)
    run_command("/home/peterj/e3bin/e3prep.py --job=%s --do-svg" % (stageName,))
    run_command("mpirun -np %d /home/peterj/e3bin/e3mpi.exe --job=%s --run"
                % (np, stageName,))
    return

#-----------------------------------------------------------------
def locate_shock_along_strip(x, y, p):
    """
    Shock location is identified as a pressure rise
    along a strip of points.
    """
    n = len(x)
    p_max = max(p)
    p_trigger = p[0] + 0.3 * (p_max - p[0])
    x_old = x[0]; y_old = y[0]; p_old = p[0]
    for i in range(1,n):
```

```
        x_new = x[i]; y_new = y[i]; p_new = p[i]
        if p_new > p_trigger: break
        x_old = x_new; y_old = y_new; p_old = p_new
    frac = (p_trigger - p_old) / (p_new - p_old)
    x_loc = x_old * (1.0 - frac) + x_new * frac
    y_loc = y_old * (1.0 - frac) + y_new * frac
    return x_loc, y_loc

def locate_shock_front(stageName, nbi, nbj):
    """
    Reads all flow blocks and returns the coordinates
    of the shock front in lists of coordinates.
    """
    blockData = []
    for ib in range(nbi):
        blockData.append([])
        for jb in range(nbj):
            blkindx = ib*nbj + jb
            fileName = 'flow/t9999/%s.flow.b%04d.t9999.gz' \
                % (stageName, blkindx)
            fp = gzip.open(fileName, "r")
            blockData[ib].append(StructuredGridFlow())
            blockData[ib][-1].read(fp)
            fp.close()
    x_shock = []; y_shock = []
    for jb in range(nbj):
        nj = blockData[0][jb].nj
        for j in range(nj):
            x = []; y = []; p = [];
            for ib in range(nbi):
                ni = blockData[ib][jb].ni
                k = 0 # 2D only
                for i in range(ni):
                    x.append(blockData[ib][jb].data['pos.x'][i,j,k])
                    y.append(blockData[ib][jb].data['pos.y'][i,j,k])
                    p.append(blockData[ib][jb].data['p'][i,j,k])
            xshock, yshock = locate_shock_along_strip(x, y, p)
            x_shock.append(xshock)
            y_shock.append(yshock)
    return x_shock, y_shock

#-----------------------------------------------------------------
def define_bezier_points(alpha, x_s, y_s):
    """
    It is assumed that the centre of the circular body is at (0,0)
    and that we have a third-order Bezier curve that goes through
    the start and finish of the shock.
    """
    import math
    x0 = x_s[0]; y0 = 0.0         # the first point coincides with the shock
    x3 = 0.0; y3 = y_s[-1]        # locate final point also on shock
    x1 = x0; y1 = 0.5 * y3
    L = 0.4 * y3
    x2 = x3 - L * math.cos(alpha)
    y2 = y3 - L * math.sin(alpha)
    return [x0, x1, x2, x3], [y0, y1, y2, y3]

def fit_bezier(x_s, y_s):
    """
    Fits a Bezier curve to the shock coordinates
    and returns lists of coordinates.
    """
    from cfpylib.nm.line_search import minimize
    import math
    #
    def objective(alpha, x_s=x_s, y_s=y_s):
        """
        Objective function for the optimizer.
        """
        from libprep3 import Bezier, Vector
        bx, by = define_bezier_points(alpha, x_s, y_s)
        bpath = Bezier([Vector(bx[0],by[0]),Vector(bx[1],by[1]),
                        Vector(bx[2],by[2]),Vector(bx[3],by[3])])
```

```
        nbez = 1000
        pbez = []
        for i in range(nbez):
            t = 1.0/nbez * i
            pbez.append(bpath.eval(t))
        n = len(x_s)
        sum_sq_err = 0.0
        for j in range(n):
            min_dist = (x_s[j]-pbez[0].x)**2 + (y_s[j]-pbez[0].y)**2
            for i in range(1,nbez):
                dist = (x_s[j]-pbez[i].x)**2 + (y_s[j]-pbez[i].y)**2
                if dist < min_dist: min_dist = dist
            sum_sq_err += min_dist
        # print "alpha=", alpha, "sum_sq_err=", sum_sq_err
        return sum_sq_err
    #
    alphaL, alphaR = minimize(objective, 0.0, math.pi/4)
    best_alpha = 0.5*(alphaL+alphaR)
    return define_bezier_points(best_alpha, x_s, y_s)

#-----------------------------------------------------------------
# main...

jobName = 'sphere'
R = 6.6e-3                    # nose radius of sphere
T_body = 296.0                # surface T

# Free-stream flow definition,
# We have initially static gas, processed by a Mach 8 shock.
# Inflow conditions are thus post-shock conditions.
p_init = 6.7                  # Pa
p_inf = 535.6                 # Pa
T_inf = 2573.5                # degrees K
u_inf = 2436.5                # flow speed, m/s

# Initial simulation using guessed inflow boundary position.
stage = 0
x_d = [-1.5*R, -1.5*R, -1.0*R, 0.0]
y_d = [0.0, 1.0*R, 2.0*R, 3.0*R]
factor = 2; ni_basic = 10; nj_basic = 10
nbi = 2; nbj = 2
np = nbi * nbj  # number of processes for MPI simulation
paramDict = {'jobName': jobName, 'stage':stage,
             'R':R, 'x_d':x_d, 'y_d':y_d,
             'T_body':T_body, 'p_init':p_init,
             'p_inf':p_inf, 'T_inf':T_inf, 'u_inf':u_inf,
             'ni':factor*ni_basic, 'nj':factor*nj_basic,
             'nbi':nbi, 'nbj':nbj, 'np':np,
             'viscous_flag':1, 'viscous_delay':10.0e-6,
             'body_lengths':10}  # 20 body_lengths normally
run_stage(paramDict, jobName, stage)

# Restart from stage 0 flow data,
# bringing grid in closer to the shock.
stage = 1
paramDict['stage'] = stage
x_shock, y_shock = locate_shock_front(jobName+str(stage-1), nbi, nbj)
x_d, y_d = fit_bezier(x_shock, y_shock)
# Scale out so that we are sure to capture the shock.
paramDict['x_d'] = [1.05*x for x in x_d]
paramDict['y_d'] = [1.1*y for y in y_d]
paramDict['viscous_delay'] = 0.0
run_stage(paramDict, jobName, stage)

# Restart from stage 1 flow data, refining grid
stage = 2
paramDict['stage'] = stage
factor = 3
paramDict['ni'] = factor*ni_basic
paramDict['nj'] = factor*nj_basic
paramDict['body_lengths'] = 5.0
run_stage(paramDict, jobName, stage)
```

```
# Restart from stage 2 flow data,
# adjusting the inflow boundary and refining grid
stage = 3
paramDict['stage'] = stage
x_shock, y_shock = locate_shock_front(jobName+str(stage-1), nbi, nbj)
x_d, y_d = fit_bezier(x_shock, y_shock)
# Scale out so that we are sure to capture the shock.
paramDict['x_d'] = [1.05*x for x in x_d]
paramDict['y_d'] = [1.1*y for y in y_d]
factor = 4
paramDict['ni'] = factor*ni_basic
paramDict['nj'] = factor*nj_basic
paramDict['body_lengths'] = 5.0
run_stage(paramDict, jobName, stage)

# Restart from stage 3 flow data, refining the grid only.
stage = 4
paramDict['stage'] = stage
factor = 8
paramDict['ni'] = factor*ni_basic
paramDict['nj'] = factor*nj_basic
paramDict['body_lengths'] = 5.0
run_stage(paramDict, jobName, stage)

print "Done at top-level."
```

19.3 Shell script for postprocessing

```
#!/bin/bash
# plot_heat_transfer.sh

# Get the heat-flux data around the surface of the sphere.
# These come from blk-1-0 (block 2) and blk-1-1 (block 3)

stages="0 1 2 3 4"
for STAGE in ${stages}
do
    echo "Stage $STAGE:"
    e3post.py --job=sphere${STAGE} --tindx=9999 --heat-flux-list="2:3,1,-1,:,0" \
        --output-file=sphere_heat_transfer_${STAGE}.dat
done

# Scale current physical simulation to compare with theory and experiment.
awk '$1 != "#" {print $1/0.0066*180.0/3.14159, $2/2.217e6}' \
    sphere_heat_transfer_4.dat > sphere_normalised_heat_transfer.dat

gnuplot <<EOF
set term postscript eps enhanced 20
set output "sphere_norm_heat_transfer.eps"
set style line 1 linetype 1 linewidth 3.0
set xlabel "angle from stagnation point, degrees"
set ylabel "q/q_s"
set logscale y
# set yrange [0.1:2.0]
set yrange [0.1:1.2]
set title "Normalised heat transfer to R6.6mm sphere with Ms=8"
# set key top right
set key bottom left
plot "sphere_normalised_heat_transfer.dat" using 1:2 \
          title "Eilmer3 simulation" with lines ls 1, \
     "kemp_theory.dat" using 1:2 title "Kemp-Rose-Detra theory" \
          with linespoints, \
     "kemp_experiment.dat" using 1:2 title "Kemp-Rose-Detra experiment" \
          with points
EOF
```

19.4 Notes

- The look-up table for the equilibrium air equation of state is set up as for the Sawada sphere example.

20 Dissociating N_2 flow over a 2D cylinder

High speed flow of nitrogen over a 2D cylinder is a signature experiment for shock tunnel and expansion tube facilities. This example shows the construction of a simple flow domain around a circular cylinder and the set up of a finite-rate reacting model for dissociating nitrogen. The data for comparison has come from our colleagues ar DLR-Göttingen.

Figure 39: Schematic diagram of the geometry for the bluff body.

Figure 40: Mesh, coloured by pressure, for the n90 bluff body exercise.

Figure 41: Temperature data along the stagnation streamline.

Figure 42: Temperature and mass fraction of nitrogen atoms for the n90 bluff body exercise.

20.1 Input script (.py)

```
# n90.py
# Updated version of n90.sit (an mb_cns example)
#
# R.J.Gollan
# Updated on 12-Mar-2008
# PJ - Elmer3 port, July 2008
# RJG - updated for new kinetics library, Nov 2008
# PJ - different ways to set mass fractions, July 2009

gdata.title = "Cylinder in Mach 10 nitrogen flow"

# Gas model selection
species_list = select_gas_model(model='thermally perfect gas',
                                species=['N2', 'N'])
print "species_list=", species_list
set_reaction_scheme("nitrogen-2sp-2r.lua",reacting_flag=1)

# Flow conditions
# mf = [1.0, 0.0]
# mf = 1.0
# mf = 1
mf = {'N2':1.0}
inflow = FlowCondition(p=500.0, u=5000.0, v=0.0, T=700.0, massf=mf)
initial = FlowCondition(p=5.0, u=0.0, v=0.0, T=300.0, massf=mf)
print "inflow=", inflow
print "initial=", initial

# Geometry
a = Node( 0.045,    0.0,       label="a")
b = Node( 0.0,      0.0,       label="b")
c = Node( 0.013181, 0.031820,  label="c")
d = Node( 0.045,    0.045,     label="d")
e = Node( 0.0675,   0.038972,  label="e")
f = Node(-0.020,    0.0,       label="f")
g = Node(-0.020,    0.050625,  label="g")
h = Node(-0.016875, 0.106875,  label="h")
i = Node( 0.045,    0.135,     label="i")
j = Node( 0.07875,  0.095625,  label="j")
k = Node( 0.084375, 0.0675,    label="k")

bc = Arc(b, c, a, "ab")
cd = Arc(c, d, a, "cd")
```

```
de = Arc(d, e, a, "de")
east = Polyline([bc, cd, de], "east")
west = Bezier([f, g, h, i], "west")
south = Line(f, b, "south")
north = Bezier([i, j, k, e], "north")

# Block setup
NNR = 60
NNT = 40

blk = Block2D(make_patch(north, east, south, west),
              nni=NNR, nnj=NNT,
              fill_condition=initial)
blk.set_BC(WEST, SUP_IN, inflow)
blk.set_BC(NORTH, EXTRAPOLATE_OUT)

# Simulation parameters
gdata.flux_calc = ADAPTIVE
gdata.max_time = 100.0e-6
gdata.max_step = 40000
gdata.dt = 1.0e-8
gdata.cfl = 0.5
gdata.dt_plot = 20.0e-6

sketch.xaxis(-0.02, 0.10, 0.02, -0.005)
sketch.yaxis(0.0, 0.14, 0.02, -0.005)
sketch.window(-0.02, 0.0, 0.10, 0.12, 0.05, 0.05, 0.17, 0.17)
```

20.2 Reaction scheme file (.lua)

```
-- nitrogen-2sp-2r.lua
--
-- This chemical kinetic system provides
-- a simple nitrogen dissociation mechanism.
--
-- Author: Rowan J. Gollan
-- Date: 13-Mar-2009 (Friday the 13th)
-- Place: NIA, Hampton, Virginia, USA
--
-- History:
--   24-Mar-2009 - reduced file to minimum input

reaction{
   'N2 + N2 <=> N + N + N2',
   fr={'Arrhenius', A=7.0e21,  n=-1.6, T_a=113200.0},
   br={'Arrhenius', A=1.09e16, n=-0.5, T_a=0.0}
}

reaction{
   'N2 + N <=> N + N + N',
   fr={'Arrhenius', A=3.0e22,  n=-1.6, T_a=113200.0},
   br={'Arrhenius', A=2.32e21, n=-1.5, T_a=0.0}
}
```

20.3 Shell scripts

```
#!/bin/bash
e3prep.py --job=n90 --do-svg
```

```
#!/bin/bash

time e3shared.exe --job=n90 --run
```

```
#!/bin/bash
# post_simulation.sh

# Extract the stagnation line data from the steady flow field.
e3post.py --job=n90 --output-file=n90_100_iy1.data --tindx=5 \
    --slice-list="0,:,1,0"
gnuplot plot_comparison.gnu

# Create a VTK plot file of the steady flow field.
e3post.py --job=n90 --tindx=5 --vtk-xml
```

```
set term postscript eps 20
set output "n90_compare_T_stag_line.eps"
set title "Stagnation line to 90mm Cylinder in M10 nitrogen flow"
set xlabel "x, m"
set ylabel "temperature, K"
set xrange [-0.020:0.0]
set xtics 0.005
set yrange [0:20000]
set ytics 5000.0
set key left top
plot "stag_sebo.dat" using 1:7 title "CEVCATS-N Reference" with lines, \
     "n90_100_iy1.data" using 1:22 title "Nonequilibrium"
```

20.4 Notes

- For Eilmer3, this simulation required 9 min, 21 sec on a single core of a Pentium 1.6 GHz processor to reach a final time of 100μs in 3406 steps.

21 Flow of detonable mixture over a sphere

Interesting things can happen when the chemical-reaction time scales are of the same order as the flow time scales. This example simulates the flow of a stoichiometric mixture of hydrogen and air over the spherical nose of a projectile as used by Lehr in some ballistic range experiments [10]. For a range of Mach numbers, the combustion of the hydrogen is unsteady so it provides an interesting test of the interaction of the gas dynamics and chemical kinetics modules of the code. Figure 43 shows the periodic structure caused by the unsteady combustion of the gas mixture over the projectile.

Figure 43: Shadowgraph of the unsteady flow of reacting hydrogen and air over a ballistic-range projectile at a Mach number of 4.79. This particular image has been scanned from Greg Wilson's PhD thesis [11].

The free-stream condition is $p_\infty = 320\,\mathrm{mm\,Hg}$, $T_\infty = 292\,\mathrm{K}$, $u_\infty = 1.931\,\mathrm{km/s}$, corresponding to a Mach number of 4.79 in a stoichiometric mixture of hydrogen and air (nitrogen + oxygen only). The small calculation to get actual mass fraction of each species is done as part of the user input script. For this flight condition, Lehr observed a frequency of oscillation of 0.72 MHz.

Because this example is just a demonstration of the code capability and not a validation and because the case takes a day or two to run on 4 processors of geyser, we settle for the use of a reduced chemical reaction scheme in which nitrogen is assumed to be a non-reacting diluent gas. This is done near the top of the the reaction scheme file 21.2 by selecting `INERT_N2` as the model.

If we start very reasonably, with a low-resolution grid and do the calculation fairly in short order, we get something like the left image in Figure 45. The solution is all very steady, well behaved, and quite wrong. The reaction front has merged into the shock and

Figure 44: Schematic diagram of the geometry for a sphere wrapped by a SuperBlock2D grid. Although 2×2 blocks are shown here, we are typically impatient and use 4×6 blocks as shown in the input scripts.

the whole shock layer has inflated to a stand-off distance significantly larger than that observed in the experiment. Just increasing the grid resolution (by a factor of 10 in each direction) provided a solution as shown in the right side of Figure 45. Here, the reaction front is clearly separated from the inicdent shock, and it is not smooth. Although it is not clear from this particular image, there is a periodic large scale disturbance to the reaction front, and a slightly smaller disturbance to the shock front.

Figure 45: Temperature field at t=190 μs for 20×30 cells (left) and 200×300 cells (right).

Figure 46 shows the density field for a few frames of the second-stage simulation (`lehr1.py`) over roughly one period of the large-scale oscillation. Although the periodic nature of the flow is captured, the detailed behaviour of the reaction front is quite sensitive to grid resolution and the details of the reaction mechanism. The frequency of the large scale oscillation in this simulation is a long way short of the 0.72 MHz observed by Lehr. It is left as an exercise for the reader to try the more complete reaction schemes to see if the frequency of the flow oscillation can be better approximated.

21.1 Input script (.py)

```
# file: lehr.py
#
# Spherical nose of Lehr's projectile in detonable gas.
#
# PJ, 27-Feb-2010
# Adapted bits from sphere-heat-transfer and Rowan's mbcsn2/lehr_sphere.

gdata.title = "Lehr experiment M=4.79"
R = 7.5e-3  # Nose radius, metres

p_inf = 320.0/760.0*101325.0  # Pascals
u_inf = 1931  # m/s
T_inf = 292 # degrees K
```

Figure 46: Density field at t=11, 14, 17, 20 μs in the `lehr1` second-stage simulation.

```
p_init = p_inf / 5

select_gas_model(model='thermally perfect gas',
                 species=['O2','N2','H2','O','H','H2O','OH','HO2'])
# species index              0    1    2   3   4     5    6     7
set_reaction_scheme("Evans_Schexnayder.lua",reacting_flag=1)

# Calculation: convert mole fractions to mass fractions.
MW_O2 = 3.19988000e-02  # kg/mole
MW_N2 = 2.80134800e-02
MW_H2 = 2.01588000e-03
# moles for a stoichiometric mix
m_O2 = 1.0; m_N2 = 3.76; m_H2 = 2.0
# mole fractions
mole_tot = m_O2 + m_N2 + m_H2
X_O2 = m_O2 / mole_tot
X_N2 = m_N2 / mole_tot
X_H2 = m_H2 / mole_tot
MW_mix = X_O2 * MW_O2 + X_N2 * MW_N2 + X_H2 * MW_H2
# mass fractions
mf = {'O2':X_O2*(MW_O2/MW_mix),
      'N2':X_N2*(MW_N2/MW_mix),
      'H2':X_H2*(MW_H2/MW_mix)}
print "mass fractions=", mf

inflow = FlowCondition(p=p_inf, u=u_inf, T=T_inf, massf=mf)
initial = FlowCondition(p=p_init, T=T_inf, massf=mf)

# Job-control information
t_final = 50 * R / u_inf  # allow time to establish
ni = 200; nj = 300
gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = t_final
gdata.max_step = 800000
gdata.dt = 1.0e-10
gdata.cfl = 0.40
gdata.dt_plot = 10.0e-6
gdata.dt_history = 0.1e-6  # want to capture MHz frequency

# Begin geometry details for a single region around a spherical nose.
# The node coordinates are scaled with the body radius.
a = Node(0.0, 0.0, label="a")
b = Node(-1.0*R, 0.0, label="b")
c = Node(0.0, R, label="c")
# The inflow boundary is a Bezier curve.
```

```
d = [Node(-1.5*R,0), Node(-1.5*R,R), Node(-R,2*R), Node(0,3*R)]
# order of boundaries: N, E, S, W
flow_domain0 = make_patch(Line(d[-1],c), Arc(b,c,a), Line(d[0],b), Bezier(d))
boundary_conditions0 = [ExtrapolateOutBC(), SlipWallBC(),
                        SlipWallBC(), SupInBC(inflow)]
blk = SuperBlock2D(psurf=flow_domain0, fill_condition=initial,
                   nni=ni, nnj=nj, nbi=4, nbj=6,
                   bc_list=boundary_conditions0,
                   label="blk")
HistoryLocation(-R,0.0)
HistoryLocation(-R,0.001)

sketch.xaxis(-15.0e-3, 5.0e-3, 5.0e-3, -0.002)
sketch.yaxis(0.0, 20.0e-3, 5.0e-3, 0.0)
sketch.window(-1.5*R, 0.0, 1.5*R, 3.0*R, 0.05, 0.05, 0.15, 0.15)
```

The second input script continues the simulation on a grid where the inflow boundary has been moved in toward the bow shock. This makes better use of the computational resources as more cells are now within the shock layer.

```
# file: lehr1.py
#
# Spherical nose of Lehr's projectile in detonable gas -- continued.
#
# PJ, 27-Feb-2010
# Adapted bits from sphere-heat-transfer and Rowan's mbcsn2/lehr_sphere.
# This is the continuation of the simulation on a better fitted grid.

gdata.title = "Lehr experiment M=4.79"
R = 7.5e-3  # Nose radius, metres

p_inf = 320.0/760.0*101325.0  # Pascals
u_inf = 1931  # m/s
T_inf = 292 # degrees K
p_init = p_inf / 5

select_gas_model(model='thermally perfect gas',
                 species=['O2','N2','H2','O','H','H2O','OH','HO2'])
# species index             0    1    2   3   4     5    6     7
set_reaction_scheme("Evans_Schexnayder.lua",reacting_flag=1)

# Calculation: convert mole fractions to mass fractions.
MW_O2 = 3.19988000e-02  # kg/mole
MW_N2 = 2.80134800e-02
MW_H2 = 2.01588000e-03
# moles for a stoichiometric mix
m_O2 = 1.0; m_N2 = 3.76; m_H2 = 2.0
# mole fractions
mole_tot = m_O2 + m_N2 + m_H2
X_O2 = m_O2 / mole_tot
X_N2 = m_N2 / mole_tot
X_H2 = m_H2 / mole_tot
MW_mix = X_O2 * MW_O2 + X_N2 * MW_N2 + X_H2 * MW_H2
# mass fractions
mf = {'O2':X_O2*(MW_O2/MW_mix),
      'N2':X_N2*(MW_N2/MW_mix),
      'H2':X_H2*(MW_H2/MW_mix)}
print "mass fractions=", mf

inflow = FlowCondition(p=p_inf, u=u_inf, T=T_inf, massf=mf)
initial = ExistingSolution('lehr', '.', 24, 9999)

# Job-control information
t_final = 50.0e-6
ni = 200; nj = 300
gdata.axisymmetric_flag = 1
gdata.flux_calc = ADAPTIVE
```

```
gdata.max_time = t_final
gdata.max_step = 800000
gdata.dt = 1.0e-9
gdata.cfl = 0.40
gdata.dt_plot = 1.0e-6
gdata.dt_history = 0.01e-6  # want to capture MHz frequency

# Begin geometry details for a single region around a spherical nose.
# The node coordinates are scaled with the body radius.
a = Node(0.0, 0.0, label="a")
b = Node(-1.0*R, 0.0, label="b")
c = Node(0.0, R, label="c")
# The inflow boundary is a Bezier curve.
d = [Node(-1.3*R,0), Node(-1.3*R,0.7*R), Node(-0.87*R,1.4*R), Node(0,2.1*R)]
# order of boundaries: N, E, S, W
flow_domain0 = make_patch(Line(d[-1],c), Arc(b,c,a), Line(d[0],b), Bezier(d))
boundary_conditions0 = [ExtrapolateOutBC(), SlipWallBC(),
                        SlipWallBC(), SupInBC(inflow)]
blk = SuperBlock2D(psurf=flow_domain0, fill_condition=initial,
                   nni=ni, nnj=nj, nbi=4, nbj=6,
                   bc_list=boundary_conditions0,
                   label="blk")
HistoryLocation(-R,0.0)
HistoryLocation(-R,0.001)

sketch.xaxis(-15.0e-3, 5.0e-3, 5.0e-3, -0.002)
sketch.yaxis(0.0, 20.0e-3, 5.0e-3, 0.0)
sketch.window(-1.5*R, 0.0, 1.5*R, 3.0*R, 0.05, 0.05, 0.15, 0.15)
```

21.2 Reaction scheme file (.lua)

```
-- Author: Rowan J. Gollan
-- Date: 02-Feb-2010
-- Place: Poquoson, Virginia, USA
--
-- Adapted from Python file: evans_scheznayder.py
--
-- This file provides four chemical kinetic descriptions
-- of hydrogen combustion.  You can select between the various
-- options below by setting the 'model' variable below to one of
-- the strings listed below.
--
-- REDUCED  : a 7-species, 8-reactions description of hydrogen
--            combustion in pure oxygen
-- PURE_O2  : a 7-species, 16-reactions description of hydrogen
--            combustion in pure oxygen
-- IN_AIR   : a 12-species, 25-reactions description of hydrogen
--            combustion in air (N2 and O2)
-- INERT_N2 : an 8-species, 16-reactions description of hydrogen
--            combustion in air with inert N2 (acting as diluent only).
--
-- The numbering of reactions in this file corresponds to
-- Table 1 in Evans and Schexnayder (1980).
--
-- Reference:
--  Evans, J.S. and Shexnayder Jr, C.J. (1980)
--  Influence of Chemical Kinetics and Unmixedness
--  on Burning in Supersonic Hydrogen Flames
--  AIAA Journal 18:2 pp 188--193
--
-- History:
--  07-Mar-2006 -- first prepared
--

options = {
   REDUCED=true,
   PURE_O2=true,
   IN_AIR=true,
   INERT_N2=true
}

-- User selects model here
model = 'INERT_N2'

-- Check that selection is valid
if options[model] == nil then
   print("User selected model: ", model)
   print("is not valid.")
   print("Valid models are:")
   for m,_ in pairs(options) do
      print(m)
   end
end

reaction{
   'HNO2 + M <=> NO + OH + M',
   fr={'Arrhenius', A=5.0e17, n=-1.0, T_a=25000.0},
   br={'Arrhenius', A=8.0e15, n=0.0, T_a=-1000.0},
   label='r1'
}

reaction{
   'NO2 + M <=> NO + O + M',
   fr={'Arrhenius', A=1.1e16, n=0.0, T_a=32712.0},
   br={'Arrhenius', A=1.1e15, n=0.0, T_a=-941.0},
   label='r2'
}

reaction{
```

```
   'H2 + M <=> H + H + M',
   fr={'Arrhenius', A=5.5e18, n=-1.0, T_a=51987.0},
   br={'Arrhenius', A=1.8e18, n=-1.0, T_a=0.0},
   label='r3'
}

reaction{
   'O2 + M <=> O + O + M',
   fr={'Arrhenius', A=7.2e18, n=-1.0, T_a=59340.0},
   br={'Arrhenius', A=4.0e17, n=-1.0, T_a=0.0},
   label='r4'
}

reaction{
   'H2O + M <=> OH + H + M',
   fr={'Arrhenius', A=5.2e21, n=-1.5, T_a=59386.0},
   br={'Arrhenius', A=4.4e20, n=-1.5, T_a=0.0},
   label='r5'
}

reaction{
   'OH + M <=> O + H + M',
   fr={'Arrhenius', A=8.5e18, n=-1.0, T_a=50830.0},
   br={'Arrhenius', A=7.1e18, n=-1.0, T_a=0.0},
   label='r6'
}

reaction{
   'HO2 + M <=> H + O2 + M',
   fr={'Arrhenius', A=1.7e16, n=0.0, T_a=23100.0},
   br={'Arrhenius', A=1.1e16, n=0.0, T_a=-440.0},
   label='r7'
}

reaction{
   'H2O + O <=> OH + OH',
   fr={'Arrhenius', A=5.8e13, n=0.0, T_a=9059.0},
   br={'Arrhenius', A=5.3e12, n=0.0, T_a=503.0},
   label='r8'
}

reaction{
   'H2O + H <=> OH + H2',
   fr={'Arrhenius', A=8.4e13, n=0.0, T_a=10116.0},
   br={'Arrhenius', A=2.0e13, n=0.0, T_a=2600.0},
   label='r9'
}

reaction{
   'O2 + H <=> OH + O',
   fr={'Arrhenius', A=2.2e14, n=0.0, T_a=8455.0},
   br={'Arrhenius', A=1.5e13, n=0.0, T_a=0.0},
   label='r10'
}

reaction{
   'H2 + O <=> OH + H',
   fr={'Arrhenius', A=7.5e13, n=0.0, T_a=5586.0},
   br={'Arrhenius', A=3.0e13, n=0.0, T_a=4429.0},
   label='r11'
}

reaction{
   'H2 + O2 <=> OH + OH',
   fr={'Arrhenius', A=1.7e13, n=0.0, T_a=24232.0},
   br={'Arrhenius', A=5.7e11, n=0.0, T_a=14922.0},
   label='r12'
}

reaction{
   'H2 + O2 <=> H + HO2',
   fr={'Arrhenius', A=1.9e13, n=0.0, T_a=24100.0},
   br={'Arrhenius', A=1.3e13, n=0.0, T_a=0.0},
```

```
   label='r13'
}

reaction{
   'OH + OH <=> H + HO2',
   fr={'Arrhenius', A=1.7e11, n=0.5, T_a=21137.0},
   br={'Arrhenius', A=6.0e13, n=0.0, T_a=0.0},
   label='r14'
}

reaction{
   'H2O + O <=> H + HO2',
   fr={'Arrhenius', A=5.8e11, n=0.5, T_a=28686.0},
   br={'Arrhenius', A=3.0e13, n=0.0, T_a=0.0},
   label='r15'
}

reaction{
   'OH + O2 <=> O + HO2',
   fr={'Arrhenius', A=3.7e11, n=0.64, T_a=27840.0},
   br={'Arrhenius', A=1.0e13, n=0.0, T_a=0.0},
   label='r16'
}

reaction{
   'H2O + O2 <=> OH + HO2',
   fr={'Arrhenius', A=2.0e11, n=0.5, T_a=36296.0},
   br={'Arrhenius', A=1.2e13, n=0.0, T_a=0.0},
   label='r17'
}

reaction{
   'H2O + OH <=> H2 + HO2',
   fr={'Arrhenius', A=1.2e12, n=0.21, T_a=39815.0},
   br={'Arrhenius', A=1.7e13, n=0.0, T_a=12582.0},
   label='r18'
}

reaction{
   'O + N2 <=> N + NO',
   fr={'Arrhenius', A=5.0e13, n=0.0, T_a=37940.0},
   br={'Arrhenius', A=1.1e13, n=0.0, T_a=0.0},
   label='r19'
}

reaction{
   'H + NO <=> N + OH',
   fr={'Arrhenius', A=1.7e14, n=0.0, T_a=24500.0},
   br={'Arrhenius', A=4.5e13, n=0.0, T_a=0.0},
   label='r20'
}

reaction{
   'O + NO <=> N + O2',
   fr={'Arrhenius', A=2.4e11, n=0.5, T_a=19200.0},
   br={'Arrhenius', A=1.0e12, n=0.5, T_a=3120.0},
   label='r21'
}

reaction{
   'NO + OH <=> H + NO2',
   fr={'Arrhenius', A=2.0e11, n=0.5, T_a=15500.0},
   br={'Arrhenius', A=3.5e14, n=0.0, T_a=740.0},
   label='r22'
}


reaction{
   'NO + O2 <=> O + NO2',
   fr={'Arrhenius', A=1.0e12, n=0.0, T_a=22800.0},
   br={'Arrhenius', A=1.0e13, n=0.0, T_a=302.0},
   label='r23'
}
```

```
reaction{
   'NO2 + H2 <=> H + HNO2',
   fr={'Arrhenius', A=2.4e13, n=0.0, T_a=14500.0},
   br={'Arrhenius', A=5.0e11, n=0.5, T_a=1500.0},
   label='r24'
}

reaction{
   'NO2 + OH <=> NO + HO2',
   fr={'Arrhenius', A=1.0e11, n=0.5, T_a=6000.0},
   br={'Arrhenius', A=3.0e12, n=0.5, T_a=1200.0},
   label='r25'
}

reactions_list = {}

if model == 'REDUCED' then
   reactions_list = {'r3', 'r4', 'r5', 'r6', 'r8', 'r9', 'r10', 'r11'}
end

if model == 'PURE_O2' or model == 'INERT_N2' then
   reactions_list = {'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10',
                     'r11', 'r12', 'r13', 'r14', 'r15', 'r16', 'r17', 'r18'}
end


if model ~= 'IN_AIR' then
   -- For all other models we select only a subset.
   select_reactions_by_label(reactions_list)
end
```

21.3 Notes

- None

22 MNM implosion problem

This example shows the use of the Python functions to set up a very simple flow geometry with a reasonably complex initial flow state and then to test the symmetry of the computed flow solution. This test was suggested by Dr. Michael Macrossan. The flow field should be axisymmetric but it is computed on a square grid, so any grid-aligned flux calculation problems should be highlighted. Run the case with the following commands:

```
$ cd ~/cfcfd2/examples/eilmer3/2D/implosion/
$ ./imp_run.sh
```

and, within a couple of minutes, you should end up with a number of files with various solution data plotted. Figure 47 shows the initial density field in a quiescent gas. This field is approximately axisymmetric and is initialized by computing an estimate for fraction of each cell inside the nominal radius of 1.0 and then weighting the density value by that fraction. The code for this calculation dominates the imput script in Section 22.1

Figure 47: Initial density field for the implosion problem.

Figure 48 shows the density field at the end of time-stepping, when $t = 0.296\ \frac{L}{a}$ where a is the initial sound speed of the gas and L is a nominal length scale. At this time the shock has propagated into the origin, reflected and passed back out through the contact surface. Symmetry of the solution is not perfect but it is pretty good. Figure 49 shows the density profiles for a number of radial slices.

Figure 48: Final density field for the implosion problem.

Figure 49: Density profiles at the final time for the implosion problem.

22.1 Input script (.py)

```
## \file imp.py
## \author PJ, 19-Mar-2009

job_title = "MNM Implosion Problem."
print job_title
gdata.dimensions = 2

# Use a fudged air model
gas_gamma = 5.0/3.0
select_gas_model(model='ideal gas', species=['air'])
change_ideal_gas_attribute('air', 'gamma', gas_gamma)

L = 1.0
radius = L/2
pL = 100.0e3  # low pressure is 1 atm
pH = 10.0*pL  # high pressure

def my_domain(r, s, t=0.0):
    """
    The overall domain is a square of side L.

    User-defined function for the parametric volume maps from
    parametric space to physical space.
    Note that a (Python) tuple of coordinates is returned.
    """
    global L
    return (L*r, L*s, 0.0)

N = 100   # MNM's guess
dL = L/N  # cell width

def my_gas(x, y, z):
    """
    There is a circular region of low-pressure gas embedded in
    a larger, square region of high-pressure gas.
    Only one quarter of the full problem is simulated.

    User-defined function for the initial gas state
    works in physical space.
    Note that this function returns a dictionary
    of flow properties.
    """
    global dL, radius, pL, pH
    r2 = radius*radius
    x0 = x - 0.5*dL; x1 = x0 + dL
    y0 = y - 0.5*dL; y1 = y0 + dL
    r00 = x0*x0 + y0*y0
    r10 = x1*x1 + y0*y0
    r11 = x1*x1 + y1*y1
    r01 = x0*x0 + y1*y1
    if r00 < r2 and r10 < r2 and r11 < r2 and r01 < r2:
        # Fill the lower-left corner with low-pressure gas.
        p = pL
    elif r00 >= r2 and r10 >= r2 and r11 >= r2 and r01 >= r2:
        # and the outer-part of the field with high-pressure gas.
        p = pH
    else:
        # The cell is cut by the circular boundary.
        # Subdivide the cell to work out how much is inside radius.
        fcount = 0
        ddL = dL/10
        for i in range(10):
            xx = x0 + (i+0.5)*ddL
            for j in range(10):
                yy = y0 + (j+0.5)*ddL
                if xx*xx + yy*yy < r2:
                    fcount += 1
        f = float(fcount)/100.0
```

```
        p = f*pL + (1.0-f)*pH
    # We use the FlowCondition object to conveniently set all of
    # the relevant properties.
    return FlowCondition(p=p, u=0.0, v=0.0, T=300.0, add_to_list=0).to_dict()

# Define a single block for the domain.
Block2D(PyFunctionSurface(my_domain), nni=N, nnj=N, fill_condition=my_gas)

# We can set individual attributes of the global data object.
# These are often used to control the simulation process.
gdata.title = job_title
gdata.flux_calc = AUSMDV
sound_speed = sqrt(gas_gamma*287.1*300.0)
print "sound_speed=", sound_speed
gdata.max_time = 0.296*L/sound_speed  # to match Fig.2 of Macrossan et al.
gdata.max_step = 600
gdata.dt = dL/5.0/sound_speed  # probably safe
gdata.dt_plot = gdata.max_time/4  # want some intermediate plots
print "low density is", pL/(287.1*300.0), "kg/m**3"
```

22.2 Shell scripts

```
#!/bin/sh
# imp_run.sh
e3prep.py --job=imp
e3shared.exe --job=imp --run
e3post.py --job=imp --vtk-xml --tindx=all --add-mach

e3post.py --job=imp --tindx=9999 --add-mach --output-file=xaxis_profile.data \
    --slice-along-line="0.0,0.0,0.0,1.0,0.0,0.0,99"
e3post.py --job=imp --tindx=9999 --add-mach --output-file=diagonal_profile.data \
    --slice-along-line="0.0,0.0,0.0,1.0,1.0,0.0,100"
e3post.py --job=imp --tindx=9999 --add-mach --output-file=degree30_profile.data \
    --slice-along-line="0.0,0.0,0.0,0.8660,0.50,0.0,100"

gnuplot <<EOF
set term postscript eps 20
set output "density-vs-radius.eps"
set title "MNM Implosion Problem"
set xlabel "r/L"
set ylabel "Normalised density"
set xrange [0.0:1.0]
set yrange [0.0:12.0]
plot "xaxis_profile.data" using (sqrt(\$1*\$1+\$2*\$2+\$3*\$3)/1.0):(\$5/1.161) \
     title "x-axis" with lines, \
     "diagonal_profile.data" using (sqrt(\$1*\$1+\$2*\$2+\$3*\$3)/1.0):(\$5/1.161) \
     title "45 degrees" with lines, \
     "degree30_profile.data" using (sqrt(\$1*\$1+\$2*\$2+\$3*\$3)/1.0):(\$5/1.161) \
     title "30 degrees" with lines
EOF
```

22.3 Notes

- This example shows how to change an attribute of the ideal gas model, specifically, the ratio of specific heats. Look for the call to the function `change_ideal_gas_attribute` in the input script.
- It also shows off the `--slice-along-line` option for the postprocessor `e3post.py`.

23 Periodic Shear Layer

This example shows the use of the Python functions to set up a simple shear flow with a linear variation of velocity through the finite-width shear layer. There is also a variation of species mass fractions of helium and air between the counterflowing streams. The basic flow is intended to be somewhat representative of the fuel-air mixing layers encountered in shock-tunnel tests of scramjets, however, the model flow here is made periodic in the x-direction by connecting the block faces as shown in Fig. 50.

Figure 50: Computational domain for the periodic shear layer.

Figure 51 shows the mass fraction of helium at several time instants through the evolution of the shear layer. The layer has started with almost parallel flow, with a relatively small velocity perturbation, as defined in the function `initial_gas()` in the input script (see Sec. 23.1).

Figure 51: Mass fraction for helium across the periodic shear layer at times 0, 5.1 ms, 10.2 ms and 15 ms.

Figure 52 shows the evolution of the vorticity field. Vorticity is not part of the flow data files but can be computed within `Paraview` by applying the following filters to the cell data:

- `Gradient of Unstructured DataSet` selecting `vel.x` as the scalar and `du` as the result array name.
- `Gradient of Unstructured DataSet` selecting `vel.y` as the scalar and `dv` as the result array name.
- `Calculator` with `Cell Data` as the attribute mode, `dv_X - du_Y` as the expression to compute, and `vorticity` as the result array name.

Note that there is a small defect in the vorticity values at the block boundaries. This is an artifact of the `Paraview` calculation and not of the original simulation flow field.

Figure 52: Vorticity field for the periodic shear layer at times 0, 5.1 ms, 10.2 ms and 15 ms.

23.1 Input script (.py)

```
# psl.py
gdata.title = "Periodic shear layer"

select_gas_model(model='ideal gas', species=['He', 'air'])
gdata.diffusion_model = "FicksFirstLaw"
gdata.diffusion_flag = 1

H = 0.010 # layer thickness in metres
L = 0.100 # wavelength in metres

def initial_gas(x, y, z):
    """
    Top and bottom layer of different gases with a basic velocity shear
    across the interface, plus a streamwise-periodic perturbation
    that decays away from the interface.
    """
    global H, L
    from math import sin, exp, pi
    p = 100.0e3
    T = 300.0
    U0 = 1000.0
    #
    # The top and bottom streams.
    if y < 0.0:
        massf = {'He':0.1, 'air':0.9}
    else:
        massf = {'air':1.0}
    #
    # The basic velocity shear.
    if y < -H:
        u = -U0
    elif y < H:
        u = y/H * U0
    else:
        u = U0
    #
    # Add perturbation
    V0 = 50.0
    v = V0 * sin(x/L*pi) * exp(-abs(y)/H)
    flow = FlowCondition(p=p, T=T, u=u, v=v, massf=massf, add_to_list=0)
    return flow.to_dict()

#
# Geometry
ymin = -15.0 * H
ymax = 15.0 * H
xmin = -L
xmax = L

a0 = Node(xmin, ymin); a1 = Node(xmin, ymax)
b0 = Node(xmax, ymin); b1 = Node(xmax, ymax)
domain = make_patch(Line(a1,b1), Line(b0,b1), Line(a0,b0), Line(a0,a1))

nnx = 150; nny = 300
nbi = 2; nbj = 2

superblk = SuperBlock2D(psurf=domain, nni=nnx, nnj=nny,
                        bc_list=[SlipWallBC(),]*4,
                        fill_condition=initial_gas,
                        nbi=nbi, nbj=nbj, label="blk")
# Make the domain periodic in the x-direction.
for j in range(nbj):
    connect_blocks_2D(superblk.blks[-1][j], EAST, superblk.blks[0][j], WEST)

gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 15.0e-3  # seconds
gdata.max_step = 150000
```

```
gdata.dt = 1.0e-9
gdata.dt_plot = gdata.max_time / 50.0

sketch.xaxis(-0.1, 0.1, 0.05, -0.03)
sketch.yaxis(-0.15, 0.15, 0.05, -0.03)
sketch.window(-0.15, -0.15, 0.15, 0.15, 0.05, 0.05, 0.17, 0.17)
```

23.2 Shell scripts

```
#!/bin/sh
# prep_simulation.sh

e3prep.py --job=psl --do-svg
```

```
#!/bin/bash
# run_sumulation.sh
#$ -S /bin/bash
#$ -N PeriodicShear
#$ -pe orte 4
#$ -cwd
#$ -V

job=psl
np=4

echo "Start time: "; date
mpirun -np $np e3mpi.exe --job=$job --run
# e3shared.exe --job=$job --run
echo "Finish time: "; date
```

23.3 Notes

- This simulation take 33707 steps and 28945 seconds on 4 cores of geyser (AMD processors).

24 Mach 1.5 flow over a 20° cone – UDF boundaries

This is a small (in both memory and run time) example that shows the implementation of user-defined boundary conditions. It is otherwise equivalent to the case in Section 11. Use the following commands:

```
$ cd ~/cfcfd2/examples/eilmer3/2D/cone20-udf
$ ./cone20_run.sh
```

and, within a minute or so, you should end up with a number of files with various solution data plotted. The grid and initial solution are created and the time-evolution of the flow field is computed for 5 ms (with 1105 time steps being required). The commands invoke the shell scripts displayed in subsection 24.3.

Figure 53: Schematic diagram of the geometry for a cone with 20 degree half-angle and user-defined boundaries.

The free-stream conditions ($p_\infty = 95.84\,\text{kPa}$, $T_\infty = 1103\,\text{K}$ and $u_\infty = 1000\,\text{m/s}$) are related to the shock-over-ramp test problem in the original ICASE Report [2] and are set to give a Mach number of 1.5. From Chart 5 in Ref. [3], the expected steady-state shock wave angle is 49° and, from Chart 6, the pressure coefficent is

$$\frac{p_{cone-surface} - p_\infty}{q_\infty} \approx 0.387$$

and the dynamic pressure for the specified free stream is $q_\infty = \frac{1}{2}\rho_\infty u_\infty^2 \approx 151.38\,\text{kPa}$. Figure 56 shows the pressure coefficient estimated as

$$C_p = \frac{f_x - p_\infty A}{q_\infty A}$$

from the simulated axial force, f_x, written into the simulation log file and frontal area of the cone, A.

Figure 54: Pressure data for flow over a cone with 20 degree half-angle. The shock profile is not yet straight and the pressure field near the cone surface is not conically symmetric, although it would become more so if we continued the simulation.

Figure 55: Shock-sensor data for flow over a cone with 20 degree half-angle. For the `adaptive` flux calculator, this sensor indicates the regions of the flow where the more dissipative scheme should be used.

Figure 56: Evolution of the axial (drag) force for flow over a cone with 20 degree half-angle for two mesh resolutions.

24.1 Input script (.py)

```
## \file cone20.py
## \brief Test job-specification file for e3prep.py
## \author PJ, 08-Feb-2005
##
## 15-Jan-2008 -- demonstrate user-defined boundary conditions

job_title = "Mach 1.5 flow over a 20 degree cone."
print job_title
gdata.title = job_title
gdata.axisymmetric_flag = 1
gdata.stringent_cfl = 1  # to match the old mb_cns behaviour

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])

# Define flow conditions
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)
inflow  = FlowCondition(p=95.84e3, u=1000.0, v=0.0, T=1103.0)

# Set up two quadrilaterals in the (x,y)-plane be first defining
# the corner nodes, then the lines between those corners and then
# the boundary elements for the blocks.
# The labelling is not significant; it is just to make the MetaPost
# picture look the same as that produced by the Tcl scriptit program.
a = Node(0.0, 0.0, label="A")
b = Node(0.2, 0.0, label="B")
c = Node(1.0, 0.29118, label="C")
d = Node(1.0, 1.0, label="D")
e = Node(0.2, 1.0, label="E")
f = Node(0.0, 1.0, label="F")

ab = Line(a, b); bc = Line(b, c) # lower boundary including cone surface
fe = Line(f, e); ed = Line(e, d) # upper boundary
af = Line(a, f); be = Line(b, e); cd = Line(c, d) # vertical lines

# Define the blocks, boundary conditions and set the discretisation.
nx0 = 10; nx1 = 30; ny = 40
# help()
# help(make_patch)
# help(Block2D)
blk_0 = Block2D(make_patch(fe, be, ab, af), nni=nx0, nnj=ny,
                fill_condition=initial, label="BLOCK-1")
blk_1 = Block2D(make_patch(ed, cd, bc, be, "AO"), nni=nx1, nnj=ny,
                fill_condition=initial, label="BLOCK-1",
                hcell_list=[(9,0)], xforce_list=[0,0,1,0])
identify_block_connections()
blk_0.set_BC(WEST, USER_DEFINED, filename="udf-supersonic-in.lua", use_udf_flux=1)
blk_0.set_BC(SOUTH, USER_DEFINED, filename="udf-slip-wall.lua", is_wall=1)
blk_1.set_BC(EAST, USER_DEFINED, filename="udf-extrapolate-out.lua")
blk_1.set_BC(SOUTH, USER_DEFINED, filename="udf-slip-wall.lua", is_wall=1)

# Do a little more setting of global data.
gdata.udf_file = "udf-process.lua"
gdata.udf_source_vector_flag = 0
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 5.0e-3  # seconds
gdata.max_step = 3000
gdata.dt = 1.0e-6
gdata.dt_plot = 1.5e-3
gdata.dt_history = 10.0e-5
HistoryLocation(1.0, 2.0, i_offset=-2, j_offset=1, label="here")

sketch.xaxis(0.0, 1.0, 0.2, -0.05)
sketch.yaxis(0.0, 1.0, 0.2, -0.04)
sketch.window(0.0, 0.0, 1.0, 1.0, 0.05, 0.05, 0.17, 0.17)
```

24.2 Boundary-condition files (.lua)

```
-- udf-supersonic-in.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedGhostCell BC.
--
-- This particular example is defining the constant supersonic inflow
-- for the cone20 test case.


function ghost_cell(args)
   -- Function that returns the flow states for a ghost cells.
   -- For use in the inviscid flux calculations.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   -- but we don't happen to us any of them.
   --
   -- Set constant conditions across the whole boundary.
   ghost = {}
   ghost.p = 95.84e3 -- pressure, Pa
   ghost.T = {}  -- temperatures, K (as a table)
   ghost.T[0] = 1103.0
   ghost.u = 1000.0  -- x-velocity, m/s
   ghost.v = 0.0     -- y-velocity, m/s
   ghost.w = 0.0
   ghost.massf = {} -- mass fractions to be provided as a table
   ghost.massf[0] = 1.0 -- mass fractions are indexed from 0 to nsp-1
   return ghost, ghost
end


function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   -- but we don't happen to us any of them.
   wall = {}
   wall.u = 1000.0
   wall.v = 0.0
   wall.w = 0.0
   wall.T_wall = 1103.0
   wall.massf = {}
   wall.massf[0] = 1.0
   return wall
end


function flux(args)
   -- Function that returns the fluxes of conserved quantities.
   -- For use in the inviscid flux calculations.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   --
   -- Set constant conditions across the whole boundary.
   R = 287          -- gas constant J/(kg.K)
   g = 1.4          -- ratio of specific heats
   Cv = R / (g - 1) -- specific-heat, constant volume
   p = 95.84e3      -- pressure, Pa
   T = 1103.0       -- temperature, K
   rho = p/(R*T)    -- density, kg/m**3
   u = 1000.0       -- x-velocity, m/s
   v = 0.0          -- y-velocity, m/s
   w = 0.0
   massf = {}       -- mass fractions to be provided as a table
   massf[0] = 1.0   -- mass fractions are indexed from 0 to nsp-1
   -- Assemble flux vector
   F = {}
   F.mass = rho * (u*args.csX + v*args.csY) -- kg/s/m**2
   F.momentum_x = p * args.csX + u * F.mass
```

```
   F.momentum_y = p * args.csY + v * F.mass
   F.momentum_z = 0.0
   F.total_energy = F.mass * (Cv*T + 0.5*(u*u+v*v) + p/rho)
   F.species = {}
   F.species[0] = F.mass * massf[0]
   F.renergies = {}
   F.renergies[0] = F.mass * (Cv*T)
   return F
end
```

```
-- udf-extrapolate-out.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedGhostCell BC.
--
-- This particular example is defining the supersonic outflow
-- for the cone20 test case.


function ghost_cell(args)
   -- Function that returns the flow state for a ghost cell
   -- for use in the inviscid flux calculations.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   --
   -- Sample the flow field at the current cell
   -- which is beside the boundary.
   cell = sample_flow(block_id, args.i, args.j, args.k)
   return cell, cell
end


function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   return sample_flow(block_id, args.i, args.j, args.k)
end
```

```
-- udf-slip-wall.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedBC boundary condition.
--
-- This particular example is defining the slip-wall condition
-- for the cone20 test case.

function reflect_normal_velocity(ux, vy, cosX, cosY)
   -- Copied from cns_bc.h.
   un = ux * cosX + vy * cosY;     -- Normal velocity
   vt = -ux * cosY + vy * cosX;    -- Tangential velocity
   un = -un;                       -- Reflect normal component
   ux = un * cosX - vt * cosY;     -- Back to Cartesian coords
   vy = un * cosY + vt * cosX;
   return ux, vy
end

function ghost_cell(args)
   -- Function that returns the flow state for a ghost cell
   -- for use in the inviscid flux calculations.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   i = args.i; j = args.j; k = args.k
   cell1 = sample_flow(block_id, i, j, k)
   cell1.u, cell1.v = reflect_normal_velocity(cell1.u, cell1.v, args.csX, args.csY)
   if args.which_boundary == NORTH then
      j = j - 1
   elseif args.which_boundary == EAST then
```

```
      i = i - 1
   elseif args.which_boundary == SOUTH then
      j = j + 1
   elseif args.which_boundary == WEST then
      i = i + 1
   end
   cell2 = sample_flow(block_id, i, j, k)
   cell2.u, cell2.v = reflect_normal_velocity(cell2.u, cell2.v, args.csX, args.csY)
   return cell1, cell2
end


function zero_normal_velocity(ux, vy, cosX, cosY)
   -- Just the interesting bits from reflect_normal_velocity().
   vt = -ux * cosY + vy * cosX;    -- Tangential velocity
   ux = -vt * cosY;                -- Back to Cartesian coords
   vy =  vt * cosX;                -- just tangential component
   return ux, vy
end

function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   cell = sample_flow(block_id, args.i, args.j, args.k)
   cell.u, cell.v = zero_normal_velocity(cell.u, cell.v, args.csX, args.csY)
   return cell
end
```

```
-- udf-process.lua
-- This file sets up functions that will be called
-- from the main time-stepping loop.

print("Hello from the set-up stage of udf-process.")
print("nblock=", nblock)

function at_timestep_start(args)
   -- Here, we could do things like run an external program
   -- and set up data files for use in the boundary conditions.
   --
   -- args contains {t, step}
   if (args.step % 100) == 0 then
      print("At start of timestep ", args.step, " t=", args.t)
   end
   return
end

function at_timestep_end(args)
   if (args.step % 100) == 0 then
      print("At end of timestep ", args.step, " t=", args.t)
   end
   return
end

function source_vector(args, cell_data)
   -- args contains {t...}
   src = {}
   src.mass = 0.0
   src.momemtum_x = 0.0
   src.momentum_y = 0.0
   src.momentum_z = 0.0
   src.energy = {1.0e+6,}  -- J/m**3
   src.species = {0.0,}
   return src
end
```

24.3 Shell scripts

```
#! /bin/sh
# cone20_run.sh
# exercise the Navier-Stokes solver for the Cone20 test case.
# It is assumed that the path is set correctly.

# Prepare the simulation input files (parameter, grid and initial flow data).
# The SVG file provides us with a graphical check on the geometry
e3prep.py --job=cone20 --do-svg

# Integrate the solution in time,
# recording the axial force on the cone surface.
time e3shared.exe -f cone20 --run --verbose

# Extract the solution data and reformat.
# If no time is specified, the final solution found is output.
e3post.py --job=cone20 --vtk-xml

# Extract the average coefficient of pressure from the axial force
# records that were written to the simulation log file.
awk -f cp.awk e3shared.log > cone20_cp.dat

# Plot the average coefficient of pressure on the cone surface.
# We assume that the high-resolution data file is also available.
gnuplot <<EOF
set term postscript eps enhanced 20
set output "cone20_cp.ps"
set style line 1 linetype 1 linewidth 3.0
set title "20 degree cone in Mach 1.5 flow"
set xlabel "time, ms"
set ylabel "average C_p"
set xtic 1.0
set ytic 0.1
set yrange [0:0.5]
set key bottom right
set arrow from 5.2,0.387 to 5.8,0.387 nohead linestyle 1
set label "Value from\nNACA 1135\nChart 6" at 5.0,0.3 right
set arrow from 5.0,0.3 to 5.5,0.387 head
plot "cone20_cp.dat" using 1:2 title "10x40+30x40", \
     "cone20_cp_hi-res.dat" using 1:2 title "20x80+60x80" with lines
EOF

echo "At this point, we should have a solution that can be viewed with Paraview."
```

24.4 Notes

- Run time is approximately 94 seconds for 1126 steps on a computer with an Intel Dual Pentium E2160, 1.6 GHz processor. As would be expected, the calling of the user-defined (Lua) functions carries some cost, but not much.

25 A section of an ideal compressible-flow vortex

This flow example was used by Ian Johnston in his thesis and it comes with an analytic solution [12]. With respect to `Eilmer3`, it illustrates the use of a specified flow profile as an input and it shows the use of profile extraction, again.

The flow domain (Fig. 57) includes only part of the first quadrant of an ideal vortex flow in inviscid air with $R = 287\text{J/kg·K}$, γ=1.4). The NORTH and SOUTH boundaries are specified as reflecting walls at radii r_o and r_i, representing the outer and inner radii of the vortex segment that is centred at node A. The WEST boundary has the specified inflow as a function of radius

$$
\begin{aligned}
\rho(r) &= \rho_i \left[1 + \frac{\gamma - 1}{2} M_i^2 \left\{ 1 - \left(\frac{r_i}{r} \right)^2 \right\} \right]^{\frac{1}{\gamma-1}} \quad , \\
p(r) &= p_i \left(\frac{\rho}{\rho_i} \right)^{\gamma} \quad , \\
u(r) &= u_i \frac{r_i}{r} \quad ,
\end{aligned}
$$

with $r_o = 1.384 r_i$ and the properties at the inner radius being $M_i = 2.25$, $\rho_i = 1.0\,\text{kg/m}^3$ and $p_i = 100\,\text{kPa}$.

Figure 57: Schematic diagram of the first quadrant domain for the compressible-flow vortex.

Figure 58 shows the radial distributions of flow properties and highlight some of the problems with the crude reflecting-wall boundary condition. Other than at the boundaries, there is close agreement between the analytic and numerical solutions. The errors

at the inner and outer radii stand out clearly because we know that the trends of the flow property variations should continue at these boundaries and not mirror what is just inside the flow domain.

Figure 58: Radial distributions of normalized pressure, temperature and velocity. Also, the bottom right image shows the flow speed over the simulated domain

25.1 Input script (.py)

```
# file: vtx.py
# PJ, 14-Dec-2006
#     01-Feb-2010 ported to Eilmer3
gdata.title = "Inviscid supersonic vortex -- flow in a bend."

# Geometry
R_inner = 1.0
R_outer = 1.384
a = Node(0.0, 0.0)
b = Node(0.0, R_inner)
c = Node(0.0, R_outer)
d = Node(R_inner, 0.0)
e = Node(R_outer, 0.0)
north0 = Arc(c, e, a)
east0 = Line(d, e)
south0 = Arc(b, d, a)
west0 = Line(b, c)

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
# The following flow condition is not really important because
# the actual data will be taken from the user-defined boundaries.
initial = FlowCondition(p=1000.0, u=0.0, v=0.0, T=348.43)

blk_0 = Block2D(psurf=make_patch(north0, east0, south0, west0),
                fill_condition=initial,
                nni=80, nnj=40,
                bc_list=[UserDefinedBC("udf-vortex-flow.lua"),
                         ExtrapolateOutBC(),
                         UserDefinedBC("udf-vortex-flow.lua"),
                         UserDefinedBC("udf-vortex-flow.lua")],
                label="Duct")

# Simulation-control information
gdata.flux_calc = ADAPTIVE
gdata.max_time = 20.0e-3
gdata.max_step = 6000
gdata.dt = 1.0e-6
gdata.dt_plot = 5.0e-3

# Some hints to scale and place the SVG layout figure.
sketch.xaxis(0.0, 1.5, 0.5, -0.1)
sketch.yaxis(0.0, 1.5, 0.5, -0.1)
sketch.window(0.0, 0.0, 1.5, 1.5, 0.05, 0.05, 0.17, 0.17)
```

25.2 Boundary condition file (.lua)

```
-- udf-vortex-flow.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedBC.
--
-- This particular example defines the inviscid flow field
-- of a compressible vortex.

Rgas  = 287      -- J/kg.K
g     = 1.4      -- ratio of specific heats
-- radial limits of flow domain
r_i   = 1.0      -- metres
r_o   = 1.384
-- Set flow properties ar the inner radius.
p_i   = 100.0e3                  -- Pa
M_i   = 2.25
```

```
rho_i = 1.0                        -- kg/m**3
T_i   = p_i / (Rgas * rho_i)       -- K
a_i   = math.sqrt(g * Rgas * T_i)      -- m/s
u_i   = M_i * a_i                  -- m/s

-- We'll use a bit of extra information to estimate
-- the locations of the ghost cells.
n = 40
dr = (r_o - r_i) / nnj

print("Set up inviscid vortex")
print("    p_i=", p_i, "M_i=", M_i, "rho_i=", rho_i,
      "T_i=", T_i, "a_i=", a_i, "u_i=", u_i)

function vortex_flow(r)
   u   = u_i * r_i / r
   t1  = r_i / r
   t2  = 1.0 + 0.5 * (g - 1.0) * M_i * M_i * (1.0 - t1 * t1)
   rho = rho_i * math.pow( t2, 1.0/(g - 1.0) )
   p = p_i * math.pow( rho/rho_i, g )
   T = p / (rho * Rgas)
   return u, p, T
end

function ghost_cell(args)
   -- Function that returns the flow states for a ghost cells.
   -- For use in the inviscid flux calculations.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   --
   -- We make an estimate of where the ghost cell is in space and
   -- then compute the vortex flow properties for that point.
   x = args.x
   y = args.y
   r = math.sqrt(x*x + y*y)
   theta = math.atan2(y, x)

   ghost1 = {}
   if which_boundary == NORTH then
      r_ghost1 = r + 0.5*dr
   else
      r_ghost1 = r - 0.5*dr
   end
   speed, p, T = vortex_flow(r_ghost1)
   ghost1.p = p
   ghost1.T = {}  -- temperatures as a table
   ghost1.T[0] = T
   ghost1.u = math.sin(theta) * speed
   ghost1.v = -math.cos(theta) * speed
   ghost1.w = 0.0
   ghost1.massf = {} -- mass fractions to be provided as a table
   ghost1.massf[0] = 1.0 -- mass fractions are indexed from 0 to nsp-1

   ghost2 = {}
   if which_boundary == NORTH then
      r_ghost2 = r + 1.5*dr
   else
      r_ghost2 = r - 1.5*dr
   end
   speed, p, T = vortex_flow(r_ghost2)
   ghost2.p = p
   ghost2.T = {}
   ghost2.T[0] = T
   ghost2.u = math.sin(theta) * speed
   ghost2.v = -math.cos(theta) * speed
   ghost2.w = 0.0
   ghost2.massf = {}
   ghost2.massf[0] = 1.0

   return ghost1, ghost2
end
```

```
function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   --
   -- args contains t, x, y, z, csX, csY, csZ, i, j, k, which_boundary
   x = args.x
   y = args.y
   r = math.sqrt(x*x + y*y)
   theta = math.atan2(y, x)
   speed, p, T = vortex(r)

   wall = {}
   wall.u = math.sin(theta) * speed
   wall.v = -math.cos(theta) * speed
   wall.w = 0.0
   wall.T_wall = T
   wall.massf = {}
   wall.massf[0] = 1.0
   return wall
end
```

25.3 Shell scripts

```
#! /bin/sh
# vtx_run.sh
e3prep.py --job=vtx --do-svg
time e3shared.exe --job=vtx --run
e3post.py --job=vtx --vtk-xml
```

```
#! /bin/sh
# vtx_plot.sh

# Generate the ideal profile.
awk -f make_profile.awk

# Extract the flow data 45 degrees around.
e3post.py --job=vtx --tindx=9999 --output-file=vtx_profile_45.dat \
    --slice-list="0,39,:,0"

# Extract the flow data 90 degrees around.
e3post.py --job=vtx --tindx=9999 --output-file=vtx_profile_90.dat \
    --slice-list="0,-1,:,0"

awk -f extract_radial.awk vtx_profile_45.dat > radial_profile_45.dat
awk -f extract_radial.awk vtx_profile_90.dat > radial_profile_90.dat

# Generate postscript plots of the radial profiles.
gnuplot radial_profile.gnu

echo At this point, we should have a plotted the solution
```

25.4 Notes

- This simulation reaches a final time of 20 ms in 2610 steps and, on an Intel Core 2 Duo CPU (E8400 @ 3.0 Ghz) system, this takes 2 min, 23 s.

- The plots were generated via the following scripts

```
# extract_radial.awk
# Extract the radial profile data from e3post.py generated files.
BEGIN{
   r_i = 1.0; p_i = 100.0e3; u_i = 841.87; T_i = 348.43;
}

$1 != "#" {
   x = $1; y = $2; p = $9; u = $6; v = $7; T = $20
   r = sqrt( x * x + y * y )
   speed = sqrt( u * u + v * v )
   print r/r_i, p/p_i, speed/u_i, 0.0, T/T_i
}
```

```
# radial_profile.gnu

set term postscript eps enhanced 20
set output "radial_profile_p.eps"
set title "Inviscid Vortex"
set xlabel "r/r_i"
set ylabel "p/p_i"
# set yrange [1.0:4.5]
set key 1.35, 2
plot "radial_profile_0.dat" using 1:2 title "exact" with lines, \
     "radial_profile_45.dat" using 1:2 title "45 degrees", \
     "radial_profile_90.dat" using 1:2 title "exit plane"

set term postscript eps enhanced 20
set output "radial_profile_u.eps"
set title "Inviscid Vortex"
set xlabel "r/r_i"
set ylabel "u/u_i"
# set yrange [0.7:1.0]
set key
plot "radial_profile_0.dat" using 1:3 title "exact" with lines, \
     "radial_profile_45.dat" using 1:3 title "45 degrees", \
     "radial_profile_90.dat" using 1:3 title "90 degrees"

set term postscript eps enhanced 20
set output "radial_profile_T.eps"
set title "Inviscid Vortex"
set xlabel "r/r_i"
set ylabel "T/T_i"
# set yrange [1.0:1.7]
set key 1.35, 1.2
plot "radial_profile_0.dat" using 1:5 title "exact" with lines, \
     "radial_profile_45.dat" using 1:5 title "45 degrees", \
     "radial_profile_90.dat" using 1:5 title "90 degrees"
```

```
# make_profile.awk
# Set up an inflow profile for the inviscid vortex case
# PJ, 20-Feb-01, 14-Dec-06 write 1.0 for mass-fraction[0]
#
function pow( base, exponent ) {
   # print base, exponent
   return exp( exponent * log(base) )
}

BEGIN {
   Rgas  = 287      # J/kg.K
   g     = 1.4      # ratio of specific heats
```

```
   n     = 40
   r_i   = 1.0      # metres
   r_o   = 1.384
   dr    = (r_o - r_i) / n

   # Set flow properties ar the inner radius.
   p_i   = 100.0e3                  # kPa
   M_i   = 2.25
   rho_i = 1.0                      # kg/m**3
   T_i   = p_i / (Rgas * rho_i)     # K
   a_i   = sqrt( g * Rgas * T_i )   # m/s
   u_i   = M_i * a_i                # m/s
   # print p_i, M_i, rho_i, T_i, a_i, u_i

   # Generate the profile along the radial direction.
   print n > "profile.dat"
   for ( i = 1; i <= n; ++i ) {
      r   = r_i + dr * (i - 0.5)
      # print "i= ", i, "r=", r
      u   = u_i * r_i / r
      t1  = r_i / r
      t2  = 1.0 + 0.5 * (g - 1.0) * M_i * M_i * (1.0 - t1 * t1)
      rho = rho_i * pow( t2, 1.0/(g - 1.0) );
      p   = p_i * pow( rho/rho_i, g )
      T   = p / (rho * Rgas)
      # print p, u, 0.0, T, 1.0 > "profile.dat"
      print r/r_i, p/p_i, u/u_i, 0.0, T/T_i, 1.0 > "radial_profile_0.dat"
   } # end for
}
```

26 Method of manufactured solutions – Euler flow

RJG's method of manufactured solutions as a code verification exercise. This shows a sophisticated use of the user-defined source terms to add the extra pieces required to model a known (manufactured) flow solution.

Figure 59: Density and pressure fields for the steady-state solution for the Method of Manufactured Solutions.

26.1 Input script (.py)

```
#
# This file can be used to simulate the
# Method of Manufactured Solutions test case.
#
# Author: Rowan J. Gollan
# Updated: 05-Feb-2008
#

gdata.title = "Method of Manufactured Solutions: Euler test case."
gdata.viscous_flag = 0
gdata.stringent_cfl = 1

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas',
                 species=['air'])

p0 = 1.0e5
u0 = 800.0
v0 = 800.0
T0 = p0 / 287.1

initial = FlowCondition(p=p0,  u=u0,    v=v0, T=T0,  massf=[1.0,])

a = Node(0.0, 0.0, label="a")
b = Node(1.0, 0.0, label="b")
c = Node(0.0, 1.0, label="c")
d = Node(1.0, 1.0, label="d")

ab = Line(a, b)
ac = Line(a, c)
cd = Line(c, d)
bd = Line(b, d)

nx = 16
ny = 16

blk_0 = Block2D(make_patch(cd, bd, ab, ac),
                nni=nx, nnj=ny,
                fill_condition=initial, label="blk-0")
blk_0.set_BC(NORTH, EXTRAPOLATE_OUT)
blk_0.set_BC(EAST, EXTRAPOLATE_OUT)
blk_0.set_BC(SOUTH, USER_DEFINED, filename="udf-bc.lua")
blk_0.set_BC(WEST, USER_DEFINED, filename="udf-bc.lua")

gdata.udf_file = "udf-source.lua"
gdata.udf_source_vector_flag = 1
gdata.flux_calc = AUSM
gdata.max_time = 20.0e-3
gdata.max_step = 2000
gdata.dt = 1.0e-6
gdata.fixed_time_step = False
gdata.cfl = 0.5
gdata.dt_plot = gdata.max_time/20.0
```

26.2 Boundary condition file (.lua)

```
-- Lua script for the south and west boundaries
-- of a Manufactured Solution which
-- treats Euler flow.
--
-- Author: Rowan J. Gollan
-- Date: 04-Feb-2008
```

```
M_PI = math.pi
cos = math.cos
sin = math.sin

L = 1.0
gam = 1.4

rho0 = 1.0
rhox = 0.15
rhoy = -0.1

uvel0 = 800.0
uvelx = 50.0
uvely = -30.0

vvel0 = 800.0
vvelx = -75.0
vvely = 40.0
wvel0 = 0.0

press0 = 1.0e5
pressx = 0.2e5
pressy = 0.5e5


function rho_function(x, y)
   rho = rho0 + rhox*sin((M_PI*x)/L) + rhoy*cos((M_PI*y)/(2.0*L))
   return rho;
end
function rho_south_bc(x) return rho_function(x, 0.0) end
function rho_west_bc(y) return rho_function(0.0, y) end


function pressure_function(x, y)
   p = press0 + pressx*cos((2.0*M_PI*x)/L) + pressy*sin((M_PI*y)/L)
   return p
end
function pressure_south_bc(x) return pressure_function(x, 0.0) end
function pressure_west_bc(y) return pressure_function(0.0, y) end


function u_function(x, y)
   u = uvel0 + uvelx*sin((3.0*M_PI*x)/(2.0*L)) + uvely*cos((3.0*M_PI*y)/(5.0*L))
   return u
end
function u_south_bc(x) return u_function(x, 0.0) end
function u_west_bc(y) return u_function(0.0, y) end


function v_function(x, y)
   v = vvel0 + vvelx*cos((M_PI*x)/(2.0*L)) + vvely*sin((2.0*M_PI*y)/(3.0*L))
   return v
end
function v_south_bc(x) return v_function(x, 0.0) end
function v_west_bc(y) return v_function(0.0, y) end


function ghost_cell(args)
   -- Function that returns the flow states for a ghost cells.
   -- For use in the inviscid flux calculations.
   --
   -- args contains {t, x, y, z, csX, csY, csZ, i, j, k, which_boundary}
   -- Set constant conditions across the whole boundary.
   x = args.x; y = args.y
   ghost = {}
   if args.which_boundary == SOUTH then
      ghost.p = pressure_south_bc(x) -- pressure, Pa
      rho = rho_south_bc(x)          -- density, kg/m^3
      ghost.u = u_south_bc(x)        -- x-velocity, m/s
      ghost.v = v_south_bc(x)        -- y-velocity, m/s
   else
      -- Assumed WEST and that we won't call this
```

```
      -- from any other boundary
      ghost.p = pressure_west_bc(y)  -- pressure, Pa
      rho = rho_west_bc(y)           -- density, kg/m^3
      ghost.u = u_west_bc(y)         -- x-velocity, m/s
      ghost.v = v_west_bc(y)         -- y-velocity, m/s
   end
   ghost.w = 0.0
   R = 287.1
   ghost.T = {}
   ghost.T[0] = ghost.p/(rho*R)      -- temperature, K
   ghost.massf = {}     -- mass fractions to be provided as a table
   ghost.massf[0] = 1.0 -- mass fractions are indexed from 0 to nsp-1
   ghost.Tvib = {}   -- vibrational temperatures also indexed from 0
   return ghost, ghost
end

function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   --
   -- args contains {t, x, y, z, csX, csY, csZ, i, j, k, which_boundary}
   x = args.x; y = args.y
   wall = {}
   if args.which_boundary == SOUTH then
      wall.u = u_south_bc(x)
      wall.v = v_south_bc(x)
      p = pressure_south_bc(x)
      rho = rho_south_bc(x)
   else
      wall.u = u_west_bc(y)
      wall.v = v_west_bc(y)
      p = pressure_west_bc(y)
      rho = rho_west_bc(y)
   end
   wall.w = 0.0
   R = 287.1
   wall.T = {}
   wall.T[0] = p/(rho*R)
   wall.massf = {}
   wall.massf[0] = 1.0
   return wall
end
```

26.3 Source term file (.lua)

The source terms were generated with the aid of the Maxima computer algebra system.

```
-- Lua script for the source terms
-- of a Manufactured Solution which
-- treats Euler flow.
--
-- Author: Rowan J. Gollan
-- Date: 04-Feb-2008

-- dummy functions to keep eilmer3 happy

function at_timestep_start(args) return nil end
function at_timestep_end(args) return nil end

M_PI = math.pi
cos = math.cos
sin = math.sin
pow = math.pow

L = 1.0
gam = 1.4
```

```
rho0 = 1.0
rhox = 0.15
rhoy = -0.1

uvel0 = 800.0
uvelx = 50.0
uvely = -30.0

vvel0 = 800.0
vvelx = -75.0
vvely = 40.0
wvel0 = 0.0

press0 = 1.0e5
pressx = 0.2e5
pressy = 0.5e5

function rho_source(x, y)
   f_m = (3*M_PI*uvelx*cos((3*M_PI*x)/(2.*L))*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) +
rhox*sin((M_PI*x)/L)))/(2.*L) + (2*M_PI*vvely*cos((2*M_PI*y)/(3.*L))*(rho0 + rhoy
*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L)))/(3.*L) + (M_PI*rhox*cos((M_PI*x)/L)
*(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L))))/L - (M_PI
*rhoy*sin((M_PI*y)/(2.*L))*(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI
*y)/(3.*L))))/(2.*L)
    return f_m
end

function xmom_source(x, y)
   f_x = (3*M_PI*uvelx*cos((3*M_PI*x)/(2.*L))*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) +
rhox*sin((M_PI*x)/L))*(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)
/(2.*L))))/L + (2*M_PI*vvely*cos((2*M_PI*y)/(3.*L))*(rho0 + rhoy*cos((M_PI*y)/(2.
*L)) + rhox*sin((M_PI*x)/L))*(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3
*M_PI*x)/(2.*L))))/(3.*L) + (M_PI*rhox*cos((M_PI*x)/L)*pow(uvel0 + uvely*cos((3*
M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)),2))/L - (2*M_PI*pressx*sin((2*M_PI
*x)/L))/L - (M_PI*rhoy*(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)
/(2.*L)))*sin((M_PI*y)/(2.*L))*(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*
M_PI*y)/(3.*L))))/(2.*L) - (3*M_PI*uvely*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*
sin((M_PI*x)/L))*sin((3*M_PI*y)/(5.*L))*(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) +
vvely*sin((2*M_PI*y)/(3.*L))))/(5.*L)
    return f_x
end

function ymom_source(x, y)
   f_y = (M_PI*pressy*cos((M_PI*y)/L))/L - (M_PI*vvelx*sin((M_PI*x)/(2.*L))*(rho0 +
rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))*(uvel0 + uvely*cos((3*M_PI*y)/(5.
*L)) + uvelx*sin((3*M_PI*x)/(2.*L))))/(2.*L) + (3*M_PI*uvelx*cos((3*M_PI*x)/(2.*L))
*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))*(vvel0 + vvelx*cos((M_PI
*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L))))/(2.*L) + (4*M_PI*vvely*cos((2*M_PI*y)/
(3.*L))*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))*(vvel0 + vvelx*
cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L))))/(3.*L) + (M_PI*rhox*cos((M_PI
*x)/L)*(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)))*(vvel0
+ vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L))))/L - (M_PI*rhoy*sin((
M_PI*y)/(2.*L))*pow(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L)
),2))/(2.*L)
    return f_y
end

function energy_source(x, y)
   f_e = (uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)))*((-2*
M_PI*pressx*sin((2*M_PI*x)/L))/L + (rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI
*x)/L))*((-2*M_PI*pressx*sin((2*M_PI*x)/L))/((-1 + gam)*L*(rho0 + rhoy*cos((M_PI*y)/
(2.*L)) + rhox*sin((M_PI*x)/L))) + ((3*M_PI*uvelx*cos((3*M_PI*x)/(2.*L))*(uvel0 +
uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L))))/L - (M_PI*vvelx*sin((
M_PI*x)/(2.*L))*(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L))))
/L)/2. - (M_PI*rhox*cos((M_PI*x)/L)*(press0 + pressx*cos((2*M_PI*x)/L) + pressy*sin(
(M_PI*y)/L)))/((-1 + gam)*L*pow(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)
/L),2))) + (M_PI*rhox*cos((M_PI*x)/L)*((pow(wvel0,2) + pow(uvel0 + uvely*cos((3*M_PI
*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)),2) + pow(vvel0 + vvelx*cos((M_PI*x)/(2.*L)
) + vvely*sin((2*M_PI*y)/(3.*L)),2))/2. + (press0 + pressx*cos((2*M_PI*x)/L) + pressy
*sin((M_PI*y)/L))/((-1 + gam)*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L
)))))/L) + (3*M_PI*uvelx*cos((3*M_PI*x)/(2.*L))*(press0 + pressx*cos((2*M_PI*x)/L) +
pressy*sin((M_PI*y)/L) + (rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))*
```

```
((pow(wvel0,2) + pow(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*
L)),2) + pow(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L)),2))/2.
+ (press0 + pressx*cos((2*M_PI*x)/L) + pressy*sin((M_PI*y)/L))/((-1 + gam)*(rho0 +
rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))))))/(2.*L) + (2*M_PI*vvely*cos((2*
M_PI*y)/(3.*L))*(press0 + pressx*cos((2*M_PI*x)/L) + pressy*sin((M_PI*y)/L) + (rho0 +
rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))*((pow(wvel0,2) + pow(uvel0 + uvely*
cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)),2) + pow(vvel0 + vvelx*cos((
M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L)),2))/2. + (press0 + pressx*cos((2*M_PI*
x)/L) + pressy*sin((M_PI*y)/L))/((-1 + gam)*(rho0 + rhoy*cos((M_PI*y)/(2.*L)) + rhox*
sin((M_PI*x)/L))))))/(3.*L) + (vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*
y)/(3.*L)))*((M_PI*pressy*cos((M_PI*y)/L))/L - (M_PI*rhoy*sin((M_PI*y)/(2.*L))*((pow(
wvel0,2) + pow(uvel0 + uvely*cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)),2)
+ pow(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*sin((2*M_PI*y)/(3.*L)),2))/2. +
(press0 + pressx*cos((2*M_PI*x)/L) + pressy*sin((M_PI*y)/L))/((-1 + gam)*(rho0 + rhoy
*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L)))))/(2.*L) + (rho0 + rhoy*cos((M_PI*y)/
(2.*L)) + rhox*sin((M_PI*x)/L))*((M_PI*pressy*cos((M_PI*y)/L))/((-1 + gam)*L*(rho0 +
rhoy*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L))) + ((-6*M_PI*uvely*(uvel0 + uvely*
cos((3*M_PI*y)/(5.*L)) + uvelx*sin((3*M_PI*x)/(2.*L)))*sin((3*M_PI*y)/(5.*L)))/(5.*L)
+ (4*M_PI*vvely*cos((2*M_PI*y)/(3.*L))*(vvel0 + vvelx*cos((M_PI*x)/(2.*L)) + vvely*
sin((2*M_PI*y)/(3.*L))))/(3.*L))/2. + (M_PI*rhoy*sin((M_PI*y)/(2.*L))*(press0 +
pressx*cos((2*M_PI*x)/L) + pressy*sin((M_PI*y)/L)))/(2.*(-1 + gam)*L*pow(rho0 + rhoy
*cos((M_PI*y)/(2.*L)) + rhox*sin((M_PI*x)/L),2))))
    return f_e
end

function source_vector(args, cell)
   src = {}
   src.mass = rho_source(cell.x, cell.y)
   src.momentum_x = xmom_source(cell.x, cell.y)
   src.momentum_y = ymom_source(cell.x, cell.y)
   src.momentum_z = 0.0
   src.total_energy = energy_source(cell.x, cell.y)
   src.species = {}
   src.species[0] = src.mass
   return src
end
```

26.4 Shell scripts

```
#!/bin/bash

e3prep.py --job=euler_manufactured
```

```
#!/bin/bash

time e3shared.exe --job=euler_manufactured --run
```

The postprocessing script shows features of the post-processor that allow one to compare one solution with another (in order to check convergence to steady state) and also to report the norms of the differences between the computed solution and a reference solution described by a Python file.

```
#!/bin/bash
echo "Check that simulation has converged by comparing solution instances:"
e3post.py --job=euler_manufactured --tindx=6 \
    --compare-job=euler_manufactured --compare-tindx=20
```

```
e3post.py --job=euler_manufactured --tindx=7 \
    --compare-job=euler_manufactured --compare-tindx=20

echo "----------------------------------------------------------------------"
echo "Check simulation against analytical data:"
e3post.py --job=euler_manufactured --tindx=20 \
    --ref-function=euler_wrapper.py \
    --per-block-norm-list="0,rho,L2;0,rho,L1" \
    --global-norm-list="rho,L2"

echo "----------------------------------------------------------------------"
echo "Generate VTK files for plotting:"
e3post.py --job=euler_manufactured --tindx=20 --vtk-xml
```

26.5 Python reference-function files

```
# euler_verify.py
from math import sin, cos, pi

R_air = 287.1

class EulerManufacturedSolution:
    def __init__(self, L,
                 rho0, rhox, rhoy, a_rhox, a_rhoy,
                 press0, pressx, pressy, a_pressx, a_pressy,
                 uvel0, uvelx, uvely, a_uvelx, a_uvely,
                 vvel0, vvelx, vvely, a_vvelx, a_vvely):
        self.L = L
        self.rho0 = rho0
        self.rhox = rhox
        self.rhoy = rhoy
        self.a_rhox = a_rhox
        self.a_rhoy = a_rhoy
        self.press0 = press0
        self.pressx = pressx
        self.pressy = pressy
        self.a_pressx = a_pressx
        self.a_pressy = a_pressy
        self.uvel0 = uvel0
        self.uvelx = uvelx
        self.uvely = uvely
        self.a_uvelx = a_uvelx
        self.a_uvely = a_uvely
        self.vvel0 = vvel0
        self.vvelx = vvelx
        self.vvely = vvely
        self.a_vvelx = a_vvelx
        self.a_vvely = a_vvely
        return

    def calculate_rho(self, x, y):
        rho = self.rho0
        rho += self.rhox*sin((self.a_rhox*pi*x)/self.L)
        rho += self.rhoy*cos((self.a_rhoy*pi*y)/self.L)
        return rho

    def calculate_p(self, x, y):
        p = self.press0
        p += self.pressx*cos((self.a_pressx*pi*x)/self.L)
        p += self.pressy*sin((self.a_pressy*pi*y)/self.L)
        return p

    def calculate_u(self, x, y):
        u = self.uvel0
        u += self.uvelx*sin((self.a_uvelx*pi*x)/self.L)
        u += self.uvely*cos((self.a_uvely*pi*y)/self.L)
```

```
        return u

    def calculate_v(self, x, y):
        v = self.vvel0
        v += self.vvelx*cos((self.a_vvelx*pi*x)/self.L)
        v += self.vvely*sin((self.a_vvely*pi*y)/self.L)
        return v
```

```
# euler_wrapper.py
from euler_verify import *

ev = EulerManufacturedSolution( 1.0,
                                1.0, 0.15, -0.1, 1.0, 0.5,
                                1.0e5, 0.2e5, 0.5e5, 2.0, 1.0,
                                800.0, 50.0, -30.0, 1.5, 0.6,
                                800.0, -75.0, 40.0, 0.5, 2.0/3.0)

def ref_function(x, y, z, t):
    rho = ev.calculate_rho(x, y)
    p = ev.calculate_p(x, y)
    T = p / (rho*R_air)
    u = ev.calculate_u(x, y)
    v = ev.calculate_v(x, y)
    return {"rho":rho, "p":p, "T":T, "vel.x":u, "vel.y":v}
```

26.6 Notes

- This simulation required 1 min, 18 sec on a single core of a Pentium 1.6 GHz processor to reach a final time of 20 ms in 1092 steps.

27 Oblique detonation wave

RJG's oblique detonation wave as a code verification exercise for the interaction of finite-rate chemistry and gas dynamics. The specified (nonlinear) shape of the ramp should result in a straight shock when the gas is reacting. This also shows a use of the user-defined source terms to activate the finite-rate chemical reactions.

Figure 60: Layout for the oblique detonation wave simulation.

Figure 61: Pressure and temperature-difference fields for the steady-state solution. The temperature difference is the computed flow temperature minus the analytic solution temperature.

27.1 Input script (.py)

```
# odw.py
#
# A Python input file to describe the oblique
# detonation wave used as a verification case.
#
# Thie Python file prepared by...
# Rowan J Gollan
# and adjusted for the new geometry spec by PJ (Aug-06)
# 17-May-2009: updated for Eilmer3 by RJG

gdata.title = "The oblique detonation wave verification case."
select_gas_model(fname="binary-gas.lua")

inflow = FlowCondition(p=86.1e3, u=964.302, v=0.0, T=[300.0], massf=[1.0, 0.0])
initial = FlowCondition(p=28.7e3, u=0.0, v=0.0, T=[300.0], massf=[1.0, 0.0])

#
# Geometry
xmin = -0.25
xmax = 1.75
ymin = 0.0
ymax = 2.0

nnx = 64
nny = 64

from oblique_detonation import *
from math import pi
od =  ObliqueDetonation( pi/4.0, 300.0, 3.0, 1.0)
wall = od.create_wall_spline(0.0, xmax, 1.0e-5)

a = Node( xmin, 0.0, label="a")
b = Node( 0.0, 0.0, label="b")
c = Node( wall.eval(1.0).x, wall.eval(1.0).y, label="c" )
d = Node( xmin, ymax, label="d")
e = Node( 0.0, ymax, label="e")
f = Node( xmax, ymax, label="f")
```

```
south0 = Line(a, b)
west0 = Line(a, d)
south1 = wall
# bc = Line(b, c) # straighten the wall
east0west1 = Line(b, e)
east1 = Line(c, f)
north0 = Line(d, e)
north1 = Line(e, f)

nnx0 = int( 0.125 * nnx )
nnx1 = nnx - int( 0.125 * nnx )

blk_0 = Block2D(
    psurf=make_patch(north0, east0west1, south0, west0),
    nni=nnx0, nnj=nny,
    bc_list=[ExtrapolateOutBC(), AdjacentBC(), SlipWallBC(), SupInBC(inflow)],
    fill_condition=inflow,
    label="blk-0"
    )
blk_1 = Block2D(
    psurf=make_patch(north1, east1, south1, east0west1),
    nni=nnx1, nnj=nny,
    bc_list=[ExtrapolateOutBC(), ExtrapolateOutBC(), SlipWallBC(), AdjacentBC()],
    fill_condition=inflow,
    label="blk-1"
    )
identify_block_connections()

# Simulate the reaction between reactants
# to form products by giving an appropriate
# user-defined source vector

gdata.udf_file = "udf-source.lua"
gdata.udf_source_vector_flag = 1

# Do a little more setting of global data.
gdata.flux_calc = AUSMDV
gdata.max_time = 1.0e-2  # seconds
gdata.max_step = 30000
gdata.dt = 1.0e-6
gdata.dt_plot = gdata.max_time/20.0
gdata.dt_history = 10.0e-5

# Values to make the SVG look good
sketch.xaxis(-0.25, 1.75, 0.25, -0.06)
sketch.yaxis(0.0, 2.0, 0.25, -0.06)
sketch.window(-0.25, 0.0, 1.75, 2.0, 0.05, 0.05, 0.17, 0.17)
```

27.2 gas-model file (binary-gas.lua)

```
-- Auto-generated by gasfile on: 17-May-2009 20:49:48
-- and edited manually by PJ, 21-Jan-2010
model = 'composite gas'
equation_of_state = 'perfect gas'
thermal_behaviour = 'constant specific heats'
sound_speed = 'equilibrium'
mixing_rule = 'Wilke'
diffusion_coefficients = 'hard sphere'
ignore_mole_fraction = 1.0e-15
species = {'A', 'B', }

A = {}
A.M = {
  value = 8.314/287.0,
  reference = "Powers and Aslam, artifical gas.",
  description = "molecular mass",
```

```
  units = "kg/mol",
}
A.gamma = {
  value = 1.2,
  reference = "Powers and Aslam, artificial gas.",
  description = "(ideal) ratio of specific heats at room temperature",
  units = "non-dimensional",
}
A.d = {
  value = 3.617e-10,
  reference = "Bird, Stewart and Lightfoot (2001), p. 864",
  description = "air value: equivalent hard-sphere diameter, sigma from L-J parameters",
  units = "m",
}
A.e_zero = {
  value = 0,
  description = "reference energy",
  units = "J/kg",
}
A.q = {
  value = 0,
  description = "heat release",
  units = "J/kg",
}
A.viscosity = {
  parameters = {
    T_ref = 273,
    ref = "Table 1-2, White (2006)",
    S = 111,
    mu_ref = 1.716e-05,
  },
  model = "Sutherland",
}
A.thermal_conductivity = {
  parameters = {
    S = 194,
    ref = "Table 1-3, White (2006)",
    k_ref = 0.0241,
    T_ref = 273,
  },
  model = "Sutherland",
}

B = {}
B.M = {
  value = 8.314/287.0,
  reference = "Powers and Aslam, artifical gas.",
  description = "molecular mass",
  units = "kg/mol",
}
B.gamma = {
  value = 1.2,
  reference = "Powers and Aslam, artificial gas.",
  description = "(ideal) ratio of specific heats at room temperature",
  units = "non-dimensional",
}
B.d = {
  value = 3.617e-10,
  reference = "Bird, Stewart and Lightfoot (2001), p. 864",
  description = "air value: equivalent hard-sphere diameter, sigma from L-J parameters",
  units = "m",
}
B.e_zero = {
  value = 0,
  description = "reference energy",
  units = "J/kg",
}
B.q = {
  value = 300000,
  description = "heat release",
  units = "J/kg",
}
B.viscosity = {
```

```
  parameters = {
    T_ref = 273,
    ref = "Table 1-2, White (2006)",
    S = 111,
    mu_ref = 1.716e-05,
  },
  model = "Sutherland",
}
B.thermal_conductivity = {
  parameters = {
    S = 194,
    ref = "Table 1-3, White (2006)",
    k_ref = 0.0241,
    T_ref = 273,
  },
  model = "Sutherland",
}
```

27.3 Source term file (.lua)

The source terms are used to activate the chemical reaction.

```
-- Lua script for the source terms
-- of a Manufactured Solution which
-- treats Euler flow.
--
-- Author: Rowan J. Gollan
-- Date: 04-Feb-2008

-- dummy functions to keep eilmer3 happy

function at_timestep_start(args) return nil end
function at_timestep_end(args) return nil end

local T_i = 362.58
local alpha = 1000

-- Heaviside step function
local function H(T)
   if T >= T_i then
      return 1
   else
      return 0
   end
end

function source_vector(args, cell)
   src = {}
   src.mass = 0
   src.momentum_x = 0
   src.momentum_y = 0
   src.momentum_z = 0
   src.total_energy = 0
   src.species = {}
   src.species[0] = -alpha*cell.rho*cell.massf[0]*H(cell.T[0])
   src.species[1] = alpha*cell.rho*cell.massf[0]*H(cell.T[0])
   return src
end
```

27.4 Shell scripts

```
#!/bin/bash
# prep_simulation.sh

e3prep.py --job=odw --do-svg
```

```
#!/bin/bash
# run_simulation.sh

time e3shared.exe --job=odw --run
```

The postprocessing script shows features of the post-processor that allow one to compare one solution with a reference solution described by a Python file.

```
#!/bin/bash

e3post.py --job=odw --ref-function=odw-analytical.py --tindx=all
```

27.5 Python reference function files

```
# odw_analytical.py
#
# Small script to help the mbcns_verify.py find
# the correct solution function.

from oblique_detonation import *
from math import pi

od = ObliqueDetonation( pi/4.0, 300.0, 3.0, 1.0)

def ref_function(x, y, z, t):
    x1, y1, rho, p, T, f, u, v, X, Y = od.solution(x, y)
    return {"rho":rho, "T[0]":T,
            "vel.x":u, "vel.y":v,
            "massf[0]":f[0], "massf[1]":f[1]}
```

```
#!/usr/bin/env python
# oblique_detonation.py
#
# This Python script contains a class
# which encapsulates the analytical
# solution for an oblique detonation wave.
#
# The analytical solution was originally published
# by Powers and Stewart (1992) and then re-presented
# as a verfication test case by Powers and Aslam (2006).
# The form of the solution is easier to interpret
# in the 2006 paper.
#
# References:
#
# 1. Powers, J.M. and Stewart, D.S. (1992)
#    Approximate solutions for oblique detonations
#    in the hypersonic limit.
#    AIAA Journal, 30:3 pp. 726--736
```

```
#
# 2. Powers, J.M and Aslam, T.D. (2006)
#    Exact solution for multidimensional compressible
#    reactive flow for verifying numerical algorithms
#    AIAA Journal, 44:2 pp. 337--344
#
# This Python script was created by...
# Rowan J Gollan
# 23-Jul-2006
#

from math import cos, sin, sqrt, pow, log, fabs
from zero_solvers import secant
from libprep3 import *

class ObliqueDetonation:

    def __init__(self, beta, T1, M1, rho1,
                 R=287.0, alpha=1000.0, gamma=6.0/5.0,
                 q=300000.0):

        self.beta = beta
        self.T1 = T1
        self.M1 = M1
        self.rho1 = rho1
        self.R = R
        self.alpha = 1000.0
        self.gamma = gamma
        self.q = q
        self.p1 = rho1*R*T1
        self.a1 = sqrt(gamma * R * T1)
        self.u1 = self.M1 * self.a1
        self.v1 = 0.0
        self.V = self.u1 * cos(self.beta)

    def get_V(self):
        return self.V

    def calculate_X(self, lmbda ):
        MsinBeta2 = (self.M1 * sin(self.beta))**2
        a1 = (1.0/ ((self.gamma + 1.0) * self.M1 * sin(self.beta))) * (self.a1 / self.alpha)
        a2 = 1.0 + self.gamma * MsinBeta2
        a3 = MsinBeta2 - 1.0
        a4 = ((2.0 * MsinBeta2) / (MsinBeta2 - 1)**2) * ((self.gamma**2 - 1.0) / self.gamma) \
            * ( self.q / (self.R*self.T1))

        OneMinusA4L = 1.0 - a4*lmbda
        OneMinusA4 = 1.0 - a4
        t1 = 2.0*a3*(sqrt(OneMinusA4L) - 1.0)
        t2 = pow( (1.0/(1.0 - lmbda)), a2)
        t3 = 1.0 - sqrt((OneMinusA4L)/(OneMinusA4))
        t4 = 1.0 + sqrt( 1.0 / OneMinusA4 )
        t5 = 1.0 + sqrt((OneMinusA4L)/(OneMinusA4))
        t6 = 1.0 - sqrt( 1.0 / OneMinusA4 )

        X = a1 * ( t1 + log( t2 * pow( (t3*t4) / (t5*t6) , a3*sqrt(OneMinusA4) ) ) )
        return X

    def calculate_rho(self, lmbda ):
        MsinBeta2 = (self.M1 * sin(self.beta))**2
        t1 = self.rho1 * (self.gamma + 1.0 ) * MsinBeta2
        t2 = 1.0 + self.gamma * MsinBeta2
        t3 = t2*t2
        t4 = (self.gamma + 1.0)*MsinBeta2
        t5 = ((self.gamma - 1.0)/self.gamma) * (2.0*lmbda*self.q / (self.R*self.T1))
        t6 = (self.gamma - 1.0)*MsinBeta2
        rho = t1 / ( t2 - sqrt( t3 - t4 * (2.0 + t5 + t6) ) )
        return rho

    def calculate_U(self, lmbda, rho ):
        U = self.rho1 * self.u1 * sin(self.beta) / rho
        return U
```

```
def calculate_T(self, lmbda, rho ):
    t1 = self.p1 / (rho*self.R)
    t2 = (self.rho1*self.u1*sin(self.beta))**2 / (rho*self.R)
    t3 = 1.0/self.rho1 - 1.0/rho
    T = t1 + t2*t3
    return T

def calculate_p(self, lmbda, rho ):
    t2 = (self.rho1*self.u1*sin(self.beta))**2
    t3 = 1.0/self.rho1 - 1.0/rho
    p = self.p1 + t2 * t3
    return p

def calculate_Yw(self, lmbda ):
    Yw = ( self.u1*cos(self.beta) / self.alpha ) * log( 1.0 / (1.0 - lmbda) )
    return Yw

def transform_xy_2_XY(self, x, y):
    X = x * sin(self.beta) - y * cos(self.beta)
    Y = x * cos(self.beta) + y * sin(self.beta)
    return (X, Y)

def transform_XY_2_xy(self, X, Y):
    x = X * sin(self.beta) + Y * cos(self.beta)
    y = Y * sin(self.beta) - X * cos(self.beta)
    return (x, y)

def transform_UV_2_uv(self, U, V):
    u = U * sin(self.beta) + V * cos(self.beta)
    v = V * sin(self.beta) - U * cos(self.beta)
    return (u, v)

def find_XYw_from_x(self, x):

    def f( lmbda ):
        X = self.calculate_X(lmbda)
        Yw = self.calculate_Yw(lmbda)
        (xg, yg) = self.transform_XY_2_xy(X, Yw)
        return (x - xg)

    lmbda = secant(f, 0.0, 0.999, limits=[0.0, 0.999])

    X = self.calculate_X(lmbda)
    Yw = self.calculate_Yw(lmbda)

    return (X, Yw)

def create_test_spline(self, xmin, xmax, no_points):
    dx = (xmax - xmin) / (no_points - 1.0)

    (X, Yw) = self.find_XYw_from_x( xmin )
    (x, y) = self.transform_XY_2_xy(X, Yw)
    points = [ Vector(x, y) ]

    for i in range(no_points-2):
        x = xmin + dx*(i+1)
        (X, Yw) = self.find_XYw_from_x( x )
        (x, y)  = self.transform_XY_2_xy(X, Yw)
        points.append( Vector(x, y) )


    (X, Yw) = self.find_XYw_from_x( xmax )
    (x, y) = self.transform_XY_2_xy(X, Yw)
    points.append( Vector(x, y) )

    return Spline(points)

def test_wall_spline(self, wall_spline):
    no_div = 2000
    dt = 1.0 / (no_div - 1.0)

    sp_point = wall_spline.eval( 0.0 )
    xs = sp_point.x
```

```
        ys = sp_point.y

        X, Yw = self.find_XYw_from_x(xs)
        xa, ya = self.transform_XY_2_xy(X, Yw)
        max_error = fabs(ya - ys)

        for i in range(1, no_div):
            t = dt*i
            sp_point = wall_spline.eval( t )
            xs = sp_point.x
            ys = sp_point.y

            X, Yw = self.find_XYw_from_x(xs)
            xa, ya = self.transform_XY_2_xy(X, Yw)
            error = fabs(ya - ys)
            if error > max_error:
                max_error = error

        return max_error

    def create_wall_spline(self, xmin, xmax, error_tol):

        no_points = 70
        error = 1.0
        while( error > error_tol ):
            spline = self.create_test_spline( xmin, xmax, no_points )
            error = self.test_wall_spline( spline )
            no_points += 1

        return spline

    def solution( self, x, y):
        (X, Y) = self.transform_xy_2_XY(x, y)

        if( X < 0.0 ):
            rho = self.rho1
            p = self.p1
            T = self.T1
            f = [1.0, 0.0]
            u = self.u1
            v = self.v1
        else:
            def f( lmbda ):
                X = self.calculate_X(lmbda)
                (xg, yg) = self.transform_XY_2_xy(X, Y)
                return (x - xg)

            lmbda = secant(f, 0.0, 0.999, limits=[0.0, 0.999])

            rho = self.calculate_rho( lmbda )
            p = self.calculate_p( lmbda, rho )
            T = self.calculate_T( lmbda, rho )
            U = self.calculate_U( lmbda, rho )
            V = self.V
            (u, v) = self.transform_UV_2_uv( U, V)
            f = [ 1.0 - lmbda, lmbda ]

        return (x, y, rho, p, T, f, u, v, X, Y)


if __name__ == '__main__':
    from math import pi
    obl = ObliqueDetonation( pi/4.0, 300.0, 3.0, 1.0)

    X = obl.calculate_X(0.1)
    Y = obl.calculate_Yw(0.1)
    (x, y) = obl.transform_XY_2_xy(X, Y)
    rho = obl.calculate_rho(0.1)
    p = obl.calculate_p(0.1, rho)
    T = obl.calculate_T(0.1, rho)
    U = obl.calculate_U(0.1, rho)
```

```
print "X(lmbda=0.1)= ", X
print "Yw(lmbda=0.1)= ", Y
print "x(lmbda=0.1)= ", x
print "y(lmbda=0.1)= ", y
print "rho(lmbda=0.1)= ", rho
print "p(lmbda=0.1)= ", p
print "T(lmbda=0.1)= ", T
print "U(lmbda=0.1)= ", U

(X, Yw) = obl.find_XYw_from_x(x)
print "X from x: ", X
print "Yw from x: ", Yw

#spline = obl.create_wall_spline(0.0, 1.75, 1.0e-5)

print "Solution at x=0.066116, y=0.035483..."
print obl.solution( 0.066116, 0.035483 )
print "Done."
```

27.6 Notes

- This simulation required 2 min, 20 sec on a single core of a pse-58 (HP workstation) to reach a final time of 10 ms in 871 steps. The cpu time on busemann (Toshiba L500 portable) was 3 min, 43 sec.

28 Subsonic compressor blade – sc10

Standard-condition 10 for a two-dimensional compressor blade with subsonic flow. The main objective is to provide a solution of a transonic flow for comparison with the solution produced by Paul Petrie-Repar's RPMTurbo code. The geometry for this example was set up in `mbcns2` by Hannes Wojciak and Paul Petrie-Repar. The UDF boundary conditions for a periodic boundary were later completed by PJ.

Figure 62: Overview of the flow geometry, showing the interior block boundaries and a number of anchor points. A number of the automatically-generated labels have been removed and others have been moved to make the diagram clearer.

28.1 Input script (.py)

```
"""
2D Compressor Blade Standard Condition 10

Hannes Wojciak, Paul Petrie-Repar
    February 2008: Original implementation
Peter J.
    March 2008: Clean-up and periodic boundary condition
    03-Sep-2008: Port to Eilmer3
Peter Blyton
    June 2011: Geometry cleaned up and simplified.

"""

# from cfpylib.geom.path import Polyline2, Spline2

# ---------------- First, set the global data ----------------------
gdata.title = "inviscid Euler for 2D-sc10"
gdata.dimensions = 2
```

Figure 63: A further-edited diagram showing the blade surface and the arrangement of the inner blocks. More of the anchor-points are labelled.

Figure 64: Mesh around the subsonic compressor blade.

Figure 65: Mach number field for flow over a subsonic compressor blade.

Figure 66: Pressure field for flow over a subsonic compressor blade.

Figure 67: Pressure around the blade surface; comparison with RPM-Turbo reference data.

```
# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
gdata.viscous_flag = 0 # inviscid simulation
gdata.t_order = 1
gdata.max_time = 0.300
gdata.max_step = 800000
gdata.dt_plot = 0.020
gdata.dt = 1.0e-7

# -----------flow conditions -------------------
p_tot = 100.0e3 # Pa
T_tot = 300.0 # degree K
gma = 1.4
Rgas = 287.0  # J/kg.K
a_tot = math.sqrt(gma*Rgas*T_tot)
M_exit = 0.45
T0_T = 1 + (gma-1.0)/2.0 * M_exit * M_exit
p0_p = T0_T**(gma/(gma-1.0))
print "p0_p=", p0_p, "T0_T=", T0_T
p_exit = p_tot / p0_p
T_exit = T_tot / T0_T
u_exit = M_exit * a_tot / math.sqrt(T0_T)
print "p_exit=", p_exit, "T_exit=", T_exit, "u_exit=", u_exit
initialCond = FlowCondition(p=p_exit, u=u_exit, T=T_exit)

# Mesh setup parameters
mrf = 6 # Mesh refinement factor, must be an even integer
clust_chord = RobertsClusterFunction(1, 1, 1.3) # clustering along chord
clust_blade_top = RobertsClusterFunction(1, 0, 1.05) # normal to chord, top
clust_blade_bottom = RobertsClusterFunction(0, 1, 1.05) # normal to chord, bottom
clust_LE_surface = RobertsClusterFunction(1, 0, 1.05) # along surface toward LE
clust_LE_chord = RobertsClusterFunction(0, 1, 1.02) # clustering toward LE in LE blocks

# Suction and pressure surfaces of blades defined using coordinate data.
profile_SS = Spline2("sc10_inner1.dat")
profile_Front_up = Spline2("sc10_inner2.dat")
profile_Front_down = Spline2("sc10_inner3.dat")
profile_Front_down.reverse()
profile_PS = Spline2("sc10_inner4.dat")
profile_PS.reverse()

# Nodes on and surrounding the blade surface
TE = profile_SS.eval(1.0)
TE_up = TE + Vector(-0.06, 0.12)
LE_up = Node(0.007375, 0.038160, label="LE_up")
LE_out_up = Node(-0.1, 0.07, label="LE_out_up")
LE = Node(0.0, 0.0, label="LE")
LE_out = Node(-0.05, -0.09, label="LE_out")
LE_down = Node(0.026541, 0.015230, label="LE_down")
LE_out_down = Node(0.09, -0.07, label="LE_out_down")
TE_down = Node(0.75, 0.6, label="TE_down")

# ---------------path definitions-------------------
SS = Node(0.18,0.44) # Node for spline at SS
spline_SS = Spline([LE_out_up,SS,TE_up]) # outer spline at SS of profile
Fup = Node(-0.1,-0.01) # Nodes for spline in front of profile
spline_Front_up = Spline([LE_out,Fup,LE_out_up]) # outer spline in front of profile
Fdown = Node(0.02,-0.1) # Nodes for spline in front of profile
spline_Front_down = Spline([LE_out,Fdown,LE_out_down]) # outer spline in front of profile
PS = Node(0.45,0.34) # Nodes for spline at PS of profile
spline_PS = Spline([LE_out_down,PS,TE_down]) # outer spline at PS of profile

# ---------------------- inner1 -------------------------
path_s = profile_SS
path_n = spline_SS
path_w = Line(LE_up,LE_out_up)
path_e = Line(TE,TE_up)

cflist = [clust_chord, clust_blade_top, clust_chord, clust_blade_top]
patch = make_patch(path_n, path_e, path_s, path_w)
inner1 = Block2D(label="inner1", nni=mrf*8, nnj=mrf, psurf=patch,
                 cf_list=cflist, fill_condition=initialCond)
```

```
# ---------------------- inner2 -------------------------
path_s = Line(LE_out,LE)
path_n = Line(LE_out_up,LE_up)
path_w = spline_Front_up
path_e = profile_Front_up

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [clust_blade_bottom, clust_LE_surface, clust_LE_chord, None]
inner2 = Block2D(label="inner2", nni=inner1.nnj, nnj=int(mrf*1.5), psurf=patch,
                cf_list=cflist, fill_condition=initialCond)

# ---------------------- inner3 -------------------------
path_s = spline_Front_down
path_n = profile_Front_down
path_w = Line(LE_out,LE)
path_e = Line(LE_out_down,LE_down)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [clust_LE_surface, clust_blade_bottom, None, clust_LE_chord]
inner3 = Block2D(label="inner3", nni=int(mrf*1.5), nnj=inner2.nni, psurf=patch,
                cf_list=cflist, fill_condition=initialCond)

# -------------------inner4 ------------
path_s = spline_PS
path_n = profile_PS
path_w = Line(LE_out_down,LE_down)
path_e = Line(TE_down,TE)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [clust_chord, clust_blade_bottom, clust_chord, clust_blade_bottom]
inner4 = Block2D(label="inner4", nni=mrf*7, nnj=inner3.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)

# -------------------- inflow1 -------------------
A = Node(-1.0, 0.15)
B = LE_out_up
C = Node(-0.3,0.5)
D = Node(-1.0,0.5)

path_s = Line(A,B)
path_n = Line(D,C)
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
in1 = Block2D(label="in1", nni=mrf*6, nnj=mrf*2, psurf=patch,
               fill_condition=initialCond)
in1.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")
in1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- inflow2 -------------------
A = Node(-1.0,-0.15)
B = LE_out
C = LE_out_up
D = Node(-1.0, 0.15)

path_s = Line(A,B)
path_n = Line(D,C)
path_w = Line(A,D)
path_e = spline_Front_up

patch = make_patch(path_n, path_e, path_s, path_w)
in2 = Block2D(label="in2", nni=in1.nni, nnj=inner2.nnj, psurf=patch,
               fill_condition=initialCond)
in2.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")

# -------------------- inflow3 -------------------
A = Node(-1.0,-0.5)
AB = Node(0.0,-0.5)
B = Node(0.05,-0.45)
C = LE_out
D = Node(-1.0,-0.15)
```

```
path_s = Polyline2([A,AB,B])
path_n = Line(D,C)
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
in3 = Block2D(label="in3", nni=in2.nni, nnj=mrf*2, psurf=patch,
               fill_condition=initialCond)
in3.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")
in3.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- outer1 -------------------
A = LE_out_up
B = TE_up
C = Node(0.6,1.1)
CD = Node(0.0,0.5)
D = Node(-0.3,0.5)

path_s = spline_SS
path_n = Polyline2([D,CD,C])
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [None, None, clust_chord, None]
outer1 = Block2D(label="outer1", nni=inner1.nni, nnj=in1.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outer1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- outer2 -------------------
A = Node(0.05,-0.45)
B = Node(0.2,-0.3)
C = LE_out_down
D = LE_out

path_s = Line(A,B)
path_n = spline_Front_down
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
outer2 = Block2D(label="outer2", nni=inner3.nni, nnj=in3.nnj, psurf=patch,
               fill_condition=initialCond)
outer2.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- outer3 -------------------
A = Node(0.2,-0.3)
AB = Node(0.707107,0.207107)
B = Node(0.9,0.207107)
C = TE_down
D = LE_out_down

path_s = Polyline2([A,AB,B])
path_n = spline_PS
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [clust_chord, None, None, None]
outer3 = Block2D(label="outer3", nni=inner4.nni, nnj=outer2.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outer3.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- outflow1 -------------------
A = TE_up
B = Node(1.707107,0.9)
C = Node(1.707107,1.207107)
CD = Node(0.707107,1.207107)
D = Node(0.6,1.1)

path_s = Line(A,B)
path_n = Polyline2([D,CD,C])
path_w = Line(A,D)
```

```
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
out1 = Block2D(label="out1", nni=mrf*8, nnj=outer1.nnj, psurf=patch,
               fill_condition=initialCond)
out1.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")
out1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

# -------------------- outflow2 -------------------
A = TE
B = Node(1.707107,0.707107,0.0)
C = Node(1.707107,0.9,0.0)
D = TE_up

path_s = Line(A,B)
path_n = Line(D,C)
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [None, None, None, clust_blade_top]
out2 = Block2D(label="out2", nni=out1.nni, nnj=inner1.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
out2.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")

# -------------------- outflow3 -------------------
A = TE_down
B = Node(1.707107,0.5,0.0)
C = Node(1.707107,0.707107,0.0)
D = TE

path_s = Line(A,B)
path_n = Line(D,C)
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)
cflist = [None, None, None, clust_blade_bottom]
out3 = Block2D(label="out3", nni=out2.nni, nnj=inner4.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
out3.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")

# -------------------- outflow4 -------------------
A = Node(0.9,0.207107,0.0)
B = Node(1.707107,0.207107,0.0)
C = Node(1.707107,0.5,0.0)
D = TE_down

path_s = Line(A,B)
path_n = Line(D,C)
path_w = Line(A,D)
path_e = Line(B,C)

patch = make_patch(path_n, path_e, path_s, path_w)

out4 = Block2D(label="out4", nni=out3.nni, nnj=outer3.nnj, psurf=patch,
               fill_condition=initialCond)
out4.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")
out4.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

identify_block_connections()

#------------------- Presentation -----------------
sketch.xaxis(-1.0, 2.0, 0.5, -0.1)
sketch.yaxis(-0.5, 1.5, 0.5, -0.1)
sketch.window(-1.0, -0.5, 2.0, 2.5, 0.02, 0.02, 0.20, 0.20)
```

28.2 Boundary-condition files (.lua)

```
-- udf-subsonic-sc10.lua
-- Lua script for the user-defined subsonic inflow for sc10 profile
-- called by the UserDefinedGhostCell BC.

-- input parameters:
T0 = 300                                    -- total temp in [K]
p0 = 100.0e3                                -- total pressure [Pa]
alpha = math.rad(55)                        -- inflow angle [rad]

-- constants and definitions:
R = 287.0                                   -- gas constant [J/(kg.K)]
g = 1.4                                     -- ratio of specific heats [-]
Cp = g*R/(g-1)                              -- specific-heat, constant volume [J/(kg.K)]
h0 = Cp*T0                                  -- total enthalpy [J/kg]

function ghost_cell(args)
   -- Function that returns the flow states for a ghost cells.
   -- For use in the inviscid flux calculations.
   -- Set constant conditions across the whole boundary.

   cell_flow = sample_flow(block_id, args.i, args.j, args.k) -- adjacent cell properties
   vel_sq = cell_flow.u^2+cell_flow.v^2      -- square of inflow velocity [m^2/s^2]
   vel = math.sqrt(vel_sq)                   -- inflow velocity [m/s]
   M_sq = vel_sq/(h0-0.5*vel_sq)/(g-1)       -- square of Mach number [-]
   ratio = 1+0.5*(g-1)*M_sq                  -- T0/T [-]

   ghost = {}
   ghost.p = p0/math.pow(ratio,(g/(g-1)))    -- pressure [Pa]
   ghost.T = {}
   ghost.T[0] = (h0-0.5*vel_sq)/Cp           -- temperature [K]
   ghost.u = vel*math.cos(alpha)             -- x-velocity [m/s]
   ghost.v = vel*math.sin(alpha)             -- y-velocity [m/s]
   ghost.w = 0.0
   ghost.massf = {}                          -- mass fractions to be provided as a table
   ghost.massf[0] = 1.0                      -- mass fractions are indexed from 0 to nsp-1
   return ghost, ghost
end


function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   return sample_flow(block_id, args.i, args.j, args.k)
end
```

```
-- udf-periodic-bc.lua
-- Lua script for the user-defined periodic BC
--
-- This particular example sets up peroidic boundary conditions
-- for the turbine-blade simulation.
-- When called, this boundary conditions looks up the flow data
-- in a cell that would overlay the ghost cell,
-- shifted by 1 period in the y-direction.
-- We will assume that the boundary blocks are approximately aligned
-- with the x,y-axes so that we simply add or subtract the y_period value.
--
-- PJ, 07-Mar-2008
--     03-Sep-2008 port to Elmer3

-- We will remember where we found the appropriate cells.
g1_src_blk = {}; g1_src_i = {}; g1_src_j = {}; g1_src_k = {}
g2_src_blk = {}; g2_src_i = {}; g2_src_j = {}; g2_src_k = {}
y_period = 1.0 -- as set by Hannes and Paul

function ghost_cell_position(xc, yc, xw, yw)
```

```
   -- c represents the cell-centre
   -- w represents the wall-interface position
   dx = xc - xw; dy = yc - yw
   return xc - 2*dx, yc - 2*dy
end

function ghost_cell(args)
   -- Function that returns the flow state for a ghost cell
   -- for use in the inviscid flux calculations.
   i = args.i; j = args.j; k = args.k
   x = args.x; y = args.y
-- old    indx = j*nnj + i
   indx = j * nni + i
   if g1_src_blk[indx] == nil then
      if args.which_boundary == NORTH then
         -- Search for the cell corresponding to the ghost-cell,
         -- offset by one period.
         c = sample_flow(block_id, i, j, k)
         xg, yg = ghost_cell_position(c.x, c.y, x, y)
         yg = yg - y_period
         g1_src_blk[indx], g1_src_i[indx], g1_src_j[indx], g1_src_k[indx] =
            locate_cell(xg, yg, 0.0)
         -- Locate cell corresponding to second ghost cell similarly.
         j = j - 1
         c = sample_flow(block_id, i, j, k)
         xg, yg = ghost_cell_position(c.x, c.y, x, y)
         yg = yg - y_period
         g2_src_blk[indx], g2_src_i[indx], g2_src_j[indx], g2_src_k[indx] =
            locate_cell(xg, yg, 0.0)
      elseif args.which_boundary == EAST then
         print("EAST boundary should not be periodic!")
      elseif args.which_boundary == SOUTH then
         -- Search for the cell corresponding to the ghost-cell,
         -- offset by one period.
         c = sample_flow(block_id, i, j, k)
         xg, yg = ghost_cell_position(c.x, c.y, x, y)
         yg = yg + y_period
         g1_src_blk[indx], g1_src_i[indx], g1_src_j[indx], g1_src_k[indx] =
            locate_cell(xg, yg, 0.0)
         -- Locate cell corresponding to second ghost cell similarly.
         j = j + 1
         c = sample_flow(block_id, i, j, k)
         xg, yg = ghost_cell_position(c.x, c.y, x, y)
         yg = yg + y_period
         g2_src_blk[indx], g2_src_i[indx], g2_src_j[indx], g2_src_k[indx] =
            locate_cell(xg, yg, 0.0)
      elseif args.which_boundary == WEST then
         print("WEST boundary should not be periodic!")
      end
      -- print("indx=", indx, "g1=", g1_src_blk[indx], g1_src_i[indx], g1_src_j[indx],
      --       "g2=", g2_src_blk[indx], g2_src_i[indx], g2_src_j[indx])
   end
   -- On subsequent calls, the array entries should be non-nil so
   -- we can immediately look up the flow data.
   cell1 = sample_flow(g1_src_blk[indx], g1_src_i[indx], g1_src_j[indx], g1_src_k[indx])
   cell2 = sample_flow(g2_src_blk[indx], g2_src_i[indx], g2_src_j[indx], g2_src_k[indx])
   return cell1, cell2
end


function interface(args)
   -- Function that returns the conditions at the boundary
   -- when viscous terms are active.
   return sample_flow(block_id, args.i, args.j, args.k)
end
```

28.3 Shell scripts

```
#! /bin/sh
# sc10_prep.sh
e3prep.py --job=sc10 --do-svg

# Extract the initial solution data and reformat.
e3post.py --job=sc10 --tindx=0 --vtk-xml

echo At this point, we should be ready to start the simulation.
```

```
#! /bin/sh
# sc10_run.sh

# Integrate the solution in time.
time e3shared.exe --job=sc10 --run

echo At this point, we should have a solution in sc10.flow.xxxx
```

```
#! /bin/sh
# sc10_post.sh
# 2D sc10 profile, extract data and plot it.

# Around the blade
e3post.py --job=sc10 --output-file=surface.dat --tindx=9999 \
    --slice-list="0,:,0,0;1,-1,:,0;2,:,-1,0;3,:,-1,0"
#                 0south  1east    2north   3north

# Extract the solution data over whole flow domain and reformat.
e3post.py --job=sc10 --vtk-xml --add-mach

# Calculate average flow properties at inlet and outlet
turbo_post.py sc10

gnuplot <<EOF
set term postscript eps 20
set output "surface_p.eps"
set title "Standard Condition 10 M=0.7"
set xlabel "x, m"
set ylabel "Pressure ratio p/p0"
#set xrange [0.05:0.8]
#set yrange [-0.6:1.0]
plot "surface.dat" using 1:(\$9/100000) title "Elmer3" with points, \
     "rpmTurboSc10SubsonicSteady-M0.7.txt" using 2:4 title "RPM-Turbo" with points
EOF

echo At this point, we should have a plotted data.
```

28.4 Notes

- Run time is approximately 11600 seconds for 276720 steps on a computer with an AMD Phenom 9650 2.7 GHz processor.

29 Subsonic compressor blade – PyFun version

This is the same flow specification as for the previous example but we directly use the functional form of the Standard Configuration 10 from http://rpmturbo.com/testcases/sc10/index.html.

In the input script, the blade surfaces are defined using `PyFunctionPath` objects that receive the functions `sc10_top_rotated` and `sc10_bottom_rotated`. These functions are created by rotating the original `sc10_top` and `sc10_bottom` functions by the blade stagger angle, using the `cfpylib.geom.transform_pyfunc` library. Shell scripts and user defined boundary condition files are the same as for the previous example.

Figure 68: Mesh around the subsonic compressor blade.

29.1 Input scripts (.py)

```
"""
2D Compressor Blade Standard Condition 10 parametric setup.

Hannes Wojciak, Paul Petrie-Repar
    February 2008: Original implementation.
Peter J.
    March 2008: Clean-up and periodic boundary condition.
    03-Sep-2008: Port to Eilmer3.
Peter Blyton
    March 2011: Blade profile defined using functions, sc10_blade_profile.py.
    April 2011: Block mesh set up to handle small perturbations.
    June 2011: Geometry cleaned up and simplified.

"""

from sc10_blade_profile import *
```

Figure 69: Mach number field for flow over a subsonic compressor blade.

Figure 70: Pressure field for flow over a subsonic compressor blade.

Figure 71: Pressure around the blade surface; comparison with RPM-Turbo reference data.

```
from cfpylib.geom.transform_pyfunc import rotate_pyfunc
from cfpylib.geom.path import Polyline2

#----------------------------------------------------
# Global data
#----------------------------------------------------
gdata.title = "Inviscid Euler Simulation for 2D sc10"
gdata.t_order = 1
gdata.max_time = 0.3
gdata.max_step = 800000
gdata.dt_plot = 0.020
gdata.dt = 1.0e-7

#----------------------------------------------------
# Flow conditions
#----------------------------------------------------
select_gas_model(model='ideal gas', species=['air'])
p_tot = 100.0e3 # Pa
T_tot = 300.0 # K
gma = 1.4
Rgas = 287.0  # J/kg.K
a_tot = math.sqrt(gma*Rgas*T_tot)
M_exit = 0.45
T0_T = 1 + (gma-1.0)/2.0 * M_exit * M_exit
p0_p = T0_T**(gma/(gma-1.0))
print "p0_p=", p0_p, "T0_T=", T0_T
p_exit = p_tot / p0_p
T_exit = T_tot / T0_T
u_exit = M_exit * a_tot / math.sqrt(T0_T)
print "p_exit=", p_exit, "T_exit=", T_exit, "u_exit=", u_exit
initialCond = FlowCondition(p=p_exit, u=u_exit, T=T_exit)

#----------------------------------------------------
# Geometric parameters
#----------------------------------------------------
STAGGER_ANGLE = math.pi/4.0
PITCH = 1.0

#----------------------------------------------------
# Mesh setup parameters
#----------------------------------------------------
mrf = 6 # Mesh refinement factor, must be an even integer
division = 0.03 # fraction of chord length for block division in C-mesh
clust_chord = RobertsClusterFunction(1, 1, 1.3) # clustering along chord
```

```
clust_blade_top = RobertsClusterFunction(1, 0, 1.05) # normal to chord, top
clust_blade_bottom = RobertsClusterFunction(0, 1, 1.05) # normal to chord, bottom
clust_LE_surface = RobertsClusterFunction(1, 0, 1.01) # along surface toward LE
clust_LE_chord = RobertsClusterFunction(0, 1, 1.01) # clustering toward LE in LE blocks

#---------------------------------------------------
# General path and node setup
#---------------------------------------------------
# Suction surface paths
sc10_top_rotated = rotate_pyfunc(sc10_top, "z", STAGGER_ANGLE)
profile_Front_up = PyFunctionPath(sc10_top_rotated, "", 0, division)
profile_SS = PyFunctionPath(sc10_top_rotated, "", division, 1)

# Pressure surface paths
sc10_bottom_rotated = rotate_pyfunc(sc10_bottom, "z", STAGGER_ANGLE)
profile_Front_down = PyFunctionPath(sc10_bottom_rotated, "", 0, division)
profile_PS = PyFunctionPath(sc10_bottom_rotated, "", division, 1)

# Nodes at leading and trailing edges
LE = profile_Front_up.eval(0.0)
LE_up = profile_SS.eval(0.0)
LE_down = profile_PS.eval(0.0)
TE = profile_SS.eval(1.0)
chord_normal = TE.clone().rotate_about_zaxis(math.pi/2.0)

# Nodes surrounding leading edge
LE_out = -0.1*TE
LE_out_up = LE_up + 0.1*chord_normal
LE_out_down = LE_down - 0.1*chord_normal

# Nodes surrounding trailing edge
TE_up = TE + Vector(-0.06, 0.12)
TE_down = TE + Vector(0.06, -0.12)

# Nodes bounding the flow domain
IN_top = Vector(-1.0, PITCH/2.0)
IN_bottom = Vector(-1.0, -PITCH/2.0)
LE_top = LE + Vector(0.0, PITCH/2.0)
LE_bottom = LE - Vector(0.0, PITCH/2.0)
TE_top = TE + Vector(0.0, PITCH/2.0)
TE_bottom = TE - Vector(0.0, PITCH/2.0)
OUT_top = TE_top + Vector(1.0, 0.0)
OUT_bottom = TE_bottom + Vector(1.0, 0.0)

# Spline above suction surface
SS = profile_SS.eval(0.5) + 0.1*chord_normal
spline_SS = Spline([LE_out_up, SS, TE_up])

# Splines in front of leading edge
Fup = LE + Vector(-0.1, 0)
spline_Front_up = Spline([LE_out, Fup, LE_out_up])
Fdown = LE + Vector(0, -0.1)
spline_Front_down = Spline([LE_out, Fdown, LE_out_down])

# Splines below pressure surface
PS = profile_PS.eval(0.5) - 0.12*chord_normal
spline_PS = Spline([LE_out_down, PS, TE_down])

#---------------------------------------------------
# inner1 block
#---------------------------------------------------
inner1_east = Line(TE, TE_up)
inner1_west = Line(LE_up, LE_out_up)
patch = make_patch(spline_SS, inner1_east, profile_SS, inner1_west, "A0")
cflist = [clust_chord, clust_blade_top, clust_chord, clust_blade_top]

inner1 = Block2D(label="inner1", nni=mrf*8, nnj=mrf, psurf=patch,
                cf_list=cflist, fill_condition=initialCond)

#---------------------------------------------------
# inner2 block
#---------------------------------------------------
inner2_south = Line(LE_out, LE)
```

```
patch = make_patch(inner1_west.reverse(), profile_Front_up, inner2_south, spline_Front_up)
cflist = [clust_blade_bottom, clust_LE_surface, clust_LE_chord, None]

inner2 = Block2D(label="inner2", nni=inner1.nnj, nnj=int(mrf*1.5), psurf=patch,
               cf_list=cflist, fill_condition=initialCond)

#---------------------------------------------------
# inner3 block
#---------------------------------------------------
inner3_east = Line(LE_out_down, LE_down)
patch = make_patch(profile_Front_down, inner3_east, spline_Front_down, inner2_south)
cflist = [clust_LE_surface, clust_blade_bottom, None, clust_LE_chord]

inner3 = Block2D(label="inner3", nni=int(mrf*1.5), nnj=inner2.nni, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)

#---------------------------------------------------
# inner4 block
#---------------------------------------------------
inner4_east = Line(TE_down, TE)
patch = make_patch(profile_PS, inner4_east, spline_PS, inner3_east)
cflist = [clust_chord, clust_blade_bottom, clust_chord, clust_blade_bottom]

inner4 = Block2D(label="inner4", nni=mrf*7, nnj=inner3.nnj, psurf=patch,
                cf_list=cflist, fill_condition=initialCond)

#---------------------------------------------------
# inflow1 block
#---------------------------------------------------
inflow1_lower_left = Vector(-1.0, 0.15)
inflow1_upper_right = Vector(-0.3,PITCH/2.0)
inflow1_north = Line(IN_top, inflow1_upper_right)
inflow1_east = Line(LE_out_up, inflow1_upper_right)
inflow1_south = Line(inflow1_lower_left, LE_out_up)
inflow1_west= Line(inflow1_lower_left, IN_top)
patch = make_patch(inflow1_north, inflow1_east, inflow1_south, inflow1_west)

inflow1 = Block2D(label="inflow1", nni=mrf*6, nnj=mrf*2, psurf=patch,
                fill_condition=initialCond)
inflow1.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")
inflow1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# inflow2 block
#---------------------------------------------------
inflow2_lower_left = Vector(-1.0, -0.15)
inflow2_south = Line(inflow2_lower_left, LE_out)
inflow2_west = Line(inflow2_lower_left, inflow1_lower_left)
patch = make_patch(inflow1_south, spline_Front_up, inflow2_south, inflow2_west)

inflow2 = Block2D(label="inflow2", nni=inflow1.nni, nnj=inner2.nnj, psurf=patch,
               fill_condition=initialCond)
inflow2.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")

#---------------------------------------------------
# inflow3 block
#---------------------------------------------------
inflow3_east = Line(LE_bottom, LE_out)
inflow3_south = Line(IN_bottom, LE_bottom)
inflow3_west = Line(IN_bottom, inflow2_lower_left)
patch = make_patch(inflow2_south, inflow3_east, inflow3_south, inflow3_west)

inflow3 = Block2D(label="inflow3", nni=inflow2.nni, nnj=mrf*2, psurf=patch,
               fill_condition=initialCond)
inflow3.set_BC(WEST, USER_DEFINED, filename="udf-subsonic-sc10.lua", label="INLET")
inflow3.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# outer1 block
#---------------------------------------------------
outer1_upper_right = LE_top + 0.8*TE
outer1_north = Polyline2([inflow1_upper_right, LE_top, outer1_upper_right])
outer1_east = Line(TE_up, outer1_upper_right)
```

```
patch = make_patch(outer1_north, outer1_east, spline_SS, inflow1_east)
cflist = [None, None, clust_chord, None]

outer1 = Block2D(label="outer1", nni=inner1.nni, nnj=inflow1.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outer1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# outer2 block
#---------------------------------------------------
outer2_lower_right = LE_bottom + 0.3*TE
outer2_east = Line(outer2_lower_right, LE_out_down)
outer2_south = Line(LE_bottom, outer2_lower_right)
patch = make_patch(spline_Front_down, outer2_east, outer2_south, inflow3_east)

outer2 = Block2D(label="outer2", nni=inner3.nni, nnj=inflow3.nnj,
               psurf=patch, fill_condition=initialCond)
outer2.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# outer3 block
#---------------------------------------------------
outer3_lower_right = TE_bottom + Vector(0.2, 0.0)
outer3_east = Line(outer3_lower_right, TE_down)
outer3_south = Polyline2([outer2_lower_right, TE_bottom, outer3_lower_right])
patch = make_patch(spline_PS, outer3_east, outer3_south, outer2_east)
cflist = [clust_chord, None, None, None]

outer3 = Block2D(label="outer3", nni=inner4.nni, nnj=outer2.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outer3.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# outflow1 block
#---------------------------------------------------
outflow1_lower_right = OUT_top - Vector(0, 0.3)
outflow1_north = Polyline2([outer1_upper_right ,TE_top, OUT_top])
outflow1_east = Line(outflow1_lower_right, OUT_top)
outflow1_south = Line(TE_up, outflow1_lower_right)
patch = make_patch(outflow1_north, outflow1_east, outflow1_south, outer1_east)

outflow1 = Block2D(label="outflow1", nni=mrf*8, nnj=outer1.nnj, psurf=patch,
               fill_condition=initialCond)
outflow1.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")
outflow1.set_BC(NORTH, USER_DEFINED, filename="udf-periodic-bc.lua")

#---------------------------------------------------
# outflow2 block
#---------------------------------------------------
outflow2_lower_right = OUT_top - Vector(0, PITCH/2.0)
outflow2_east = Line(outflow2_lower_right, outflow1_lower_right)
outflow2_south = Line(TE, outflow2_lower_right)
patch = make_patch(outflow1_south, outflow2_east, outflow2_south, inner1_east)
cflist = [None, None, None, clust_blade_top]

outflow2 = Block2D(label="outflow2", nni=outflow1.nni, nnj=inner1.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outflow2.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")

#---------------------------------------------------
# outflow3 block
#---------------------------------------------------
outflow3_lower_right = OUT_bottom + Vector(0.0, 0.3)
outflow3_east = Line(outflow3_lower_right, outflow2_lower_right)
outflow3_south = Line(TE_down, outflow3_lower_right)
patch = make_patch(outflow2_south, outflow3_east, outflow3_south, inner4_east, "AO")
cflist = [None, None, None, clust_blade_bottom]

outflow3 = Block2D(label="outflow3", nni=outflow2.nni, nnj=inner4.nnj, psurf=patch,
               cf_list=cflist, fill_condition=initialCond)
outflow3.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")

#---------------------------------------------------
```

```
# outflow4 block
#----------------------------------------------------
outflow4_east = Line(OUT_bottom, outflow3_lower_right)
outflow4_south = Line(outer3_lower_right, OUT_bottom)
patch = make_patch(outflow3_south, outflow4_east, outflow4_south, outer3_east)

outflow4 = Block2D(label="outflow4", nni=outflow3.nni, nnj=outer3.nnj,
               psurf=patch, fill_condition=initialCond)
outflow4.set_BC("EAST", "FIXED_P_OUT", Pout=p_exit, label="OUTLET")
outflow4.set_BC(SOUTH, USER_DEFINED, filename="udf-periodic-bc.lua")

identify_block_connections()

#----------------------------------------------------
# Presentation
#----------------------------------------------------
sketch.xaxis(-1.0, 2.0, 0.5, -0.1)
sketch.yaxis(-0.5, 1.5, 0.5, -0.1)
sketch.window(-1.0, -0.5, 2.2, 2.7, 0.02, 0.02, 0.20, 0.20)
```

```
"""
Standard Condition 10 blade profile and camber functions.

Peter Blyton
    March 2011: Blade profile defined using functional form.

"""

import math

def thickness(s):
    """
    Modified NACA0006 aerofoil thickness distribution.

    Standard NACA0006 aerofoil equation modified to give a zero thickness
    at the trailing edge. Return the full aerofoil thickness
    from top to bottom surface, not just centerline to top surface.
    Equation source: http://rpmturbo.com/testcases/sc10/index.html

    Arguments:
    s: (float) The distance along the chord of aerofoil [0 <= s <= 1].

    Return Value:
    (float) Full aerofoil thickness.

    """
    return 0.06*(2.969*s**0.5 - 1.26*s - 3.516*s**2 + 2.843*s**3 - 1.036*s**4)

def camber(s):
    """
    Standard Configuration 10 camber line.

    Equation for the circular arc for the camber line of the Standard
    Configuration 10 blade profile. This is the upper arc of a circle
    where camber(0) = camber(1) = 0. Return the y coordinate of the arc,
    and the angle that a tangent makes above the horizontal.
    Equation source: http://rpmturbo.com/testcases/sc10/index.html

    Arguments:
    s: (float) The distance along the chord of aerofoil [0 <= s <= 1].

    Return Value:
    (tuple(float, float)) The y coordinate and angle of a tangent line above
    the horizontal at the location "s".

    """
    # Camber line is equation of a circle.
    # (x - a)**2 + (y - b)**2 = r**2
    a = 0.5
```

```
    b = -2.475
    r = 2.525

    y = b + math.sqrt(r**2 - (s - a)**2)

    # First derivative of camber line.
    dy_ds = -(s - a)/(math.sqrt(r**2 - (s - a)**2))

    # Angle that tangent makes above horizontal
    phi = math.atan(dy_ds)

    return (y, phi)

def sc10_top(s):
    """
    Standard Configuration 10 upper surface.

    Returns a tuple coordinate along the surface to be used with PyFunctionPath.
    Equation source: http://rpmturbo.com/testcases/sc10/index.html

    Arguments:
    s: (float) The distance along the chord of aerofoil [0 <= s <= 1].

    """
    camber_data = camber(s)

    x = s - 0.5*thickness(s)*math.sin(camber_data[1])
    y = camber_data[0] + 0.5*thickness(s)*math.cos(camber_data[1])

    return (x, y, 0.0)

def sc10_bottom(s):
    """
    Standard Configuration 10 lower surface.

    Returns a tuple coordinate along the surface to be used with PyFunctionPath.
    Equation source: http://rpmturbo.com/testcases/sc10/index.html

    Arguments:
    s: (float) The distance along the chord of aerofoil [0 <= s <= 1].

    """
    camber_data = camber(s)

    x = s + 0.5*thickness(s)*math.sin(camber_data[1])
    y = camber_data[0] - 0.5*thickness(s)*math.cos(camber_data[1])

    return (x, y, 0.0)
```

29.2 Notes

- Run time is approximately 11700 seconds for 280460 steps on a computer with an AMD Phenom 9650 2.7 GHz processor.

Part III
Examples for 3D flow

Now, let's do it all again in three dimensions.

30 Mach 1.5 flow over a 10° ramp

This is a small (in both memory and run time) example that is useful for checking that the simulation and plotting programs have been built or installed correctly. Assuming that you have the program executable files built and accessible on your system's search `PATH`, try the following commands:

```
$ cd ~/cfcfd2/examples/eilmer3/3D/simple_ramp
$ ./simple_ramp_run.sh
```

And, within a couple of minutes, you should end up with a number of files containing the flow solution data. The grid and initial solution are created and the time-evolution of the flow field is computed for 5 ms (with 862 time steps being required). In the early stages of developing a new simulation, it may be best to run the commands manually because the main program writes information to the console and even more information to a log file. Although the shell script displayed in subsection 30.2 will run all stages of the simulation, each call to `e3shared.exe` will overwrite the log file from the previous call.

The flow domain shown in Figure 72 is essentially two-dimensional with all of the action happening in the (x, z)-plane. Hence, only a thin slice in the cross-stream (y) direction is defined. The free-stream conditions ($p_\infty = 95.84\,\text{kPa}$, $T_\infty = 1103\,\text{K}$ and $u_\infty = 1000\,\text{m/s}$) are related to the shock-over-ramp test problem in the original ICASE Report [2] for the two-dimensional flow simulation code MB_CNS and are set to give a Mach number of 1.5. From Chart 2 in Ref. [3], the expected steady-state shock wave angle is 57°

The postprocessing stage is the most variable part of the flow simulation process. Just what a user of the code wants to do in detail is often unclear at the start of a simulation exercise but visualizing the data is usually the first action in postprocessing. Using the visualization software, ParaView[16], one may view the transient development of the planar shock travelling over the ramp and establishing the steady-flow oblique shock seen in Fig. 72. Starting with VTK parallel file `simple_ramp.t0000.pvtu`, ParaView understands the time-stamp sequence numbering of the VTK outpout files and allows you to step back and forth in time and study the time development of the flow field.

Visualization is often followed by a more quantitative analysis. The Python program in section 30.3, for example, picks up the data and computes the pressure force on the inclined surface of the ramp. The end result is `force= Vector3(2209.07, 0, -12528.2) Newtons`. The `BlockGrid3D` class provides methods to read the grid and flow solution for any particular block and makes the data available as a multidimensional array. The

[16]The Parallel Visualization Application (http://www.paraview.org) developed by Kitware (http://www.kitware.com) is freely available for download.

Figure 72: Filled surface representation of the cells Colours representing pressure at $t = 5.0\,\mathrm{ms}$. The 10° slope on the ramp is seen running up to the right. Note that the shock propagating from the start of the ramp is nearly straight until it approaches the top surface of the simulation domain where it is reflected. This PDF figure was generated with Paraview from the final solution file.

Figure 73: Wireframe representation of the cells on the outer surfaces of the blocks. The grid is coloured, representing pressure at $t = 5.0\,\mathrm{ms}$.

libgeom2 module provides a number of geometric methods and these are used to compute the cell interface properties on the surface of the ramp. The final section of the postprocessing program computes the distances of the cell centres from the ramp surface for a strip of cells along the ramp. Such data might be useful when computing shear stress or heat flux, for example.

30.1 Input script (.py)

```
# A sample job description file ... is actually Python code.
# This is a fudged version of the cone20 case from mb_cns in 3D.
# It is now a ramp at 10 degrees rather than a conical surface.
# PJ, August 2004, Jan 2006, Jul 2006 (new thermochemistry module)
#     July 2008 Eilmer3 port by adding gdata.dimensions=3
# ------------------------------------------------------------------

# ---------------- First, set the global data ----------------------
# To see what parameters one can set, look up the class definition
# in the file e3prep.py.

gdata.title = "Ramp at 20 degrees."
gdata.dimensions = 3

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
gdata.viscous_flag = 0
gdata.max_time = 5.0e-3
gdata.max_step = 1000
gdata.reacting_flag = 0
# Set some of the other properties separately, just for fun.
gdata.t_order = 1
gdata.x_order = 2
# gdata.stringent_cfl = 1
gdata.dt_plot = 1.0e-3
gdata.dt_history = 1.0e-5

# ------------ Second, set up flow conditions -------------------
# These will be used for fill and boundary conditions.
initialCond = FlowCondition(p=5.955e3, u=0.0, T=304.0, massf=[1.0,])
inflowCond = FlowCondition(p=95.84e3, u=1000.0, T=1103.0, massf=[1.0,])

# ------------ Third, set up the blocks ---------------------
# These may explicitly reference previously defined flow conditions
# but, even if they don't, their setup implicitly references the
# first flow condition.

# Note that we can use the Python language to do some of our
# calculations.  Here are some handy definitions for later.

def toRadians(degrees):
    import math
    return degrees * math.pi / 180.0

def simpleBoxCorners(xPos=0.0, yPos=0.0, zPos=0.0, xSize=1.0, ySize=1.0, zSize=1.0):
    """\brief Creates a corner coordinate list for a simple box."""
    p0 = Node(xPos,       yPos,       zPos)
    p1 = Node(xPos+xSize, yPos,       zPos)
    p2 = Node(xPos+xSize, yPos+ySize, zPos)
    p3 = Node(xPos,       yPos+ySize, zPos)
    p4 = Node(xPos,       yPos,       zPos+zSize)
    p5 = Node(xPos+xSize, yPos,       zPos+zSize)
    p6 = Node(xPos+xSize, yPos+ySize, zPos+zSize)
    p7 = Node(xPos,       yPos+ySize, zPos+zSize)
    return [p0, p1, p2, p3, p4, p5, p6, p7]

def makeSimpleBox(p):
```

```
    return SimpleBoxVolume(p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7])

# ------------------------------------------------------------------
# First block is the region in front of the ramp. 10x40(x4)
pvolume = makeSimpleBox(simpleBoxCorners(xSize=0.2,ySize=0.1))
cluster_k = RobertsClusterFunction(1, 0, 1.2) # cluster down, toward the wedge surface
cflist = [None,]*8 + [cluster_k,]*4; # 12 edges is a full complement
blk0 = Block3D(label="first-block", nni=10, nnk=40,
               parametric_volume=pvolume,
               cf_list=cflist,
               fill_condition=initialCond)
blk0.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)

# For the grid over the ramp, start with a regular box... 30x40(x4)
blk1Corners = simpleBoxCorners(xPos=0.2,xSize=0.8,ySize=0.1)
# Now, raise the end of the ramp.
blk1Corners[1].z = 0.8 * math.tan(toRadians(10.0))
blk1Corners[2].z = blk1Corners[1].z
blk1 = Block3D(label="second-block", nni=30, nnk=40,
               parametric_volume=makeSimpleBox(blk1Corners),
               cf_list=cflist,
               fill_condition=initialCond,
               hcell_list=[(1,1,2),(20,1,1)])
blk1.set_BC("EAST", "SUP_OUT")
identify_block_connections()
```

30.2 Shell script

```
#! /bin/sh
# simple_ramp_run.sh
e3prep.py --job=simple_ramp
time e3shared.exe --job=simple_ramp --run --verbose
e3post.py --job=simple_ramp --vtk-xml --tindx=all
```

30.3 Postprocessing program

```
#! /usr/bin/env python
# \file estimate_ramp_force.py
#
# Example postprocessing script to look at the data along the ramp
# and compute some potentially useful information.

import sys, os, string
sys.path.append(os.path.expandvars("$HOME/e3bin")) # installation directory
sys.path.append("") # so that we can find user's scripts in working directory
from e3_grid import StructuredGrid
from e3_flow import StructuredGridFlow
from libprep3 import *
from gzip import GzipFile

print "\n\nEstimate force on the ramp surface."

fileName = 'grid/t0000/simple_ramp.grid.b0001.t0000.gz'
print "Read grid file:", fileName
fin = GzipFile(fileName, "rb")
grd = StructuredGrid()
grd.read(f=fin)
fin.close()
print "Read grid: ni=", grd.ni, "nj=", grd.nj, "nk=", grd.nk
```

```
fileName = 'flow/t0005/simple_ramp.flow.b0001.t0005.gz'
print "Read solution file:", fileName
fin = GzipFile(fileName, "rb")
soln = StructuredGridFlow()
soln.read(fin)
fin.close()
ni = soln.ni; nj = soln.nj; nk = soln.nk
print "Read solution: ni=", ni, "nj=", nj, "nk=", nk

# Integrate the pressure force over the BOTTOM surface of the block.
force = Vector(0.0, 0.0, 0.0)
k = 0
for i in range(ni):
    for j in range(nj):
        p0,p1,p2,p3,p4,p5,p6,p7 = grd.get_vertex_list_for_cell(i,j,k)
        # The bottom cell face has p0, p1, p2, p3 as corners.
        surface_centroid = quad_centroid(p0, p1, p2, p3)
        surface_normal = quad_normal(p0, p1, p2, p3)
        surface_area = quad_area(p0, p1, p2, p3)
        pressure = soln.data["p"][i][j][k] # average pressure in cell
        df = surface_area * pressure * surface_normal
        force -= df # negative because the unit normal of this cell face is into the volume
print "force=", force, "Newtons"

# Find the distance from the cell centre to the centroid of the cell face
# for a strip of cells along the ramp.  Although it is not of much use here,
# this information could be used to estimate the boundary-layer growth
# along the plate.  Katsu did this for his scramjet calculations.
fileName = "distances.txt"
fout = open(fileName, "w")
k = 0; j = 0
for i in range(ni):
    p0,p1,p2,p3,p4,p5,p6,p7 = grd.get_vertex_list_for_cell(i,j,k)
    # The bottom cell face has p0, p1, p2, p3 as corners.
    surface_centroid = quad_centroid(p0, p1, p2, p3)
    surface_normal = quad_normal(p0, p1, p2, p3)
    surface_area = quad_area(p0, p1, p2, p3)
    # We pull the cell-centre information out of the solution data.
    cell_centre = Vector(soln.data["pos.x"][i][j][k],
                         soln.data["pos.y"][i][j][k],
                         soln.data["pos.z"][i][j][k])
    distance1 = vabs(cell_centre - surface_centroid)
    # We compute the cell centroid from the grid vertices.
    cell_centre_2 = hexahedron_centroid(p0, p1, p2, p3, p4, p5, p6, p7)
    distance2 = vabs(cell_centre_2 - surface_centroid)
    fout.write("%d %e %e\n" % (i, distance1, distance2))
fout.close()

print "done."
```

30.4 Notes

- None.

31 Sod shock tube problem in 3D

This example shows the use of the Python functions to set up a very simple 3D flow geometry with a simple initial flow state. It's a long hexahedral box filled half with high-pressure and half with low-pressure gas. Run the case with the following commands:

```
$ cd ~/cfcfd2/examples/eilmer3/3D/sod/
$ ./sod_run_and_plot.sh
```

Figure 74: Flow properties along the duct for the Sod shock tube problem.

31.1 Input script (.py)

```
## \file sod.py
## \brief Test job-specification file for e3prep.py
## \author PJ, 08-Sep-2006 adapted from Tcl script to Python
##             11-Feb-2009 ported to Eilmer3 to demonstrate the use
##                         of user-supplied functions for geometry
##                         and flow conditions.
job_title = "One-dimensional shock tube with air driving air."
print job_title
gdata.dimensions = 3

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])

def tube_volume(r, s, t):
    """
    User-defined function for the parametric volume maps from
    parametric space to physical space.
    Note that a (Python) tuple of coordinates is returned.
    """
    # A simple hexahedron, one unit long in the i-direction.
    return (1.0*r, 0.1*s, 0.1*t)

def tube_gas(x, y, z):
    """
    User-defined function for the initial gas state
    works in physical space.
    Note that this function returns a dictionary of flow properties.
    """
    if x < 0.5:
        # Fill the left-half of the volume with high-pressure gas.
        p = 1.0e5; T = 348.4
    else:
        # and the right-half with low-pressure gas.
        p = 1.0e4; T = 278.8
    # We use the FlowCondition object to conveniently set all of
    # the relevant properties.
    return FlowCondition(p=p, u=0.0, v=0.0, T=T, add_to_list=0).to_dict()

# Define a single block for the tube.
Block3D(PyFunctionVolume(tube_volume),
        nni=100, nnj=2, nnk=2,
        fill_condition=tube_gas)

# We can set individual attributes of the global data object.
# These are often used to control the simulation process.
gdata.title = job_title
gdata.flux_calc = AUSMDV
gdata.max_time = 0.6e-3  # seconds
gdata.max_step = 600
gdata.dt = 1.0e-6
```

31.2 Shell script

```
# sod_run_and_plot.sh
# Sod's 1-D shock tube exercise as a 3D simulation overkill.
#
e3prep.py --job=sod
time e3shared.exe --job=sod --run
e3post.py --job=sod --output-file=sod_new.dat --slice-list="0:1,:,0,0"

gnuplot<<EOF
set term postscript eps
set output "sod_p.eps"
set title "One-D Shock Tube at t = 0.6ms"
set xlabel "x, m"
set ylabel "Pressure, Pa"
set xrange [0.0:1.0]
set yrange [0.0:120.0e3]
plot "sod_new.dat" using 1:9 with points 1 1, \
     "sod_old.dat" using 1:7 with points 1 2
EOF

gnuplot<<EOF
set term postscript eps
set output "sod_rho.eps"
set title "One-D Shock Tube at t = 0.6ms"
set xlabel "x, m"
set ylabel "Density, kg/m**3"
set xrange [0.0:1.0]
set yrange [0.0:1.2]
plot "sod_new.dat" using 1:5 with points 1 1, \
     "sod_old.dat" using 1:3 with points 1 2
EOF

gnuplot<<EOF
set term postscript eps
set output "sod_u.eps"
set title "One-D Shock Tube at t = 0.6ms"
set xlabel "x, m"
set ylabel "Velocity, m/s"
set xrange [0.0:1.0]
set yrange [0.0:500.0]
plot "sod_new.dat" using 1:6 with points 1 1, \
     "sod_old.dat" using 1:4 with points 1 2
EOF

gnuplot<<EOF
set term postscript eps
set output "sod_T.eps"
set title "One-D Shock Tube at t = 0.6ms"
set xlabel "x, m"
set ylabel "Temperature, K"
set xrange [0.0:1.0]
set yrange [0.0:500.0]
plot "sod_new.dat" using 1:20 with points 1 1, \
     "sod_old.dat" using 1:10 with points 1 2
EOF
```

31.3 Notes

- None

32 Injection of hydrogen into a nitrogen stream

Figure 75 shows half of a duct representing a simple scramjet combustor. Nitrogen flows through the duct, from the inflow plane to the outflow plane (in the x-direction), and hydrogen is injected normal to the main flow from a port in the bottom surface. The simulated flow domain represents half of the full scramjet duct which is symmetric about the $y = 0$ plane.

Figure 75: Wireframe representation of the duct with the hydrogen injection port shaded on the bottom surface. The main stream flow is from the closest (West) boundary to the furtherest (East) boundary.

The flow domain consists of 6 blocks filling the whole flow domain as shown in Figure 76. The plan form of the blocks is shown as an ASCII diagram in the middle of Python input script and has been arranged this way because each block face can accept only one boundary condition, be it a solid surface or an inflow/outflow surface. Thus the inflow of hydrogen is across the whole of the BOTTOM surface of block "10".

Figure 77 shows the pressure field and the distribution of the hydrogen jet at a point 3 ms into the simulation. With the flow properties selected, the hydrogen jet does not penetrate far into the main nitrogen stream but the pressure perturbations can can be seen right across the duct with reflections influencing the downstream part of the hydrogen plume. This figure was generated with Paraview version 3.8.0 by applying the following filters to the full data set:

- Cell Data to Point Data
- Group Data Sets
- Merge Blocks

and then the following to the Merge-Blocks data set:

- 3 Slice filters, one in each coordinate direction with its cutting plane somewhere near the boundary of the data. One of these shows massf1 at the exit plane and the others show the pressure field on the bottom surface and the symmetry plane through the injector.

- A contour (value 0.1) of massf1, coloured by p and set at an opacity of 0.6 so that the Slice planes show through it.

Figure 76: Wireframe representation of the surface grid on three surfaces, coloured by pressure on the bottom and symmetry surfaces and coloured by mass fraction on the outflow surface. Also plotted is a contour surface for a mass-fraction of hydrogen with a value 0.1, coloured by pressure and made partially transparent so that the planar surfaces show through.

Figure 77: Filled surface representation of pressure and mass-fraction for hydrogen at the final time.

32.1 Input script (.py)

```
# inject.py -- single discrete-hole injection.
# PJ
# Elmer2 original: Nov-2006
# Eilmer3 port: 06-Feb-2010

# ------------------------------------------------------------------
# Some handy definitions for later.
import math
# from cfpylib.geom.box3d import makeSimpleBox

# ---------------- First, set the global data ----------------------
gdata.title = "Single-hole injection."
gdata.dimensions = 3
gdata.dt = 1.0e-8
gdata.t_order = 1
gdata.max_time = 3.0e-3
gdata.max_step = 60000
gdata.reacting_flag = 0
gdata.dt_plot = 0.5e-3
gdata.dt_history = 1.0e-5

# ------------ Second, set up flow conditions -------------------
# These will be used for fill and boundary conditions.
species_list = select_gas_model(model='ideal gas', species=['N2', 'H2'])
initialCond = FlowCondition(p=5.955e3, u=0.0, T=304.0, massf={'N2':1.0})
inflowCond = FlowCondition(p=95.84e3, u=1000.0, T=1103.0, massf={'N2':1.0})
injectCond = FlowCondition(p=95.84e3, w=1000.0, T=300.0, massf={'H2':1.0})

# ------------ Third, set up the blocks ---------------------
# Parameters defining the duct...
L0 = 20.0e-2    # length of duct in flow direction
L1 = 5.0e-2     # distance from leading edge to injector
L2 = 1.0e-2     # streamwise length of injector
Whalf0 = 5.0e-2 # half-width of duct
```

```
Whalf1 = 5.0e-3 # half-width of injector
H = 5.0e-2      # height of duct

# Plan of blocks
#                  NORTH BNDRY
#          +--------+---+---------------+
#          |        |   |               |
#          |   01   | 11|      21       |
# inflow> |        |   |               | outflow>
# (WEST)  +--------+---+---------------+ (EAST)
#          |   00   | 10|      20       |
#          +--------+---+---------------+
#                  SOUTH BNDRY
#
#                     ^
#                  injector
cluster_k  = RobertsClusterFunction(1, 0, 1.2) # cluster down, toward the bottom surface
cluster_i0 = RobertsClusterFunction(0, 1, 1.2) # cluster streamwise toward injector
cluster_i2 = RobertsClusterFunction(1, 0, 1.2)
cluster_j1 = RobertsClusterFunction(1, 0, 1.2) # cluster cross-stream toward injector
# upstream pair of blocks
pv = makeSimpleBox(xPos=0.0, yPos=0.0, xSize=L1, ySize=Whalf1, zSize=H)
cflist = [cluster_i0,None,cluster_i0,None]*2 + [cluster_k,]*4;
# 12 edges is a full complement; see elmer_prep.py for the order of edges
blk00 = Block3D(nni=40, nnj=10, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)
pv = makeSimpleBox(xPos=0.0,yPos=Whalf1, xSize=L1,ySize=Whalf0-Whalf1, zSize=H)
cflist = [cluster_i0,cluster_j1,cluster_i0,cluster_j1]*2 + [cluster_k,]*4;
blk01 = Block3D(nni=40, nnj=30, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)

# injector and part of plate beside it
pv = makeSimpleBox(xPos=L1,yPos=0.0, xSize=L2, ySize=Whalf1, zSize=H)
cflist = [None,None,None,None]*2 + [cluster_k,]*4;
blk10 = Block3D(nni=10, nnj=10, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)
pv = makeSimpleBox(xPos=L1,yPos=Whalf1, xSize=L2, ySize=Whalf0-Whalf1, zSize=H)
cflist = [None,cluster_j1,None,cluster_j1]*2 + [cluster_k,]*4;
blk11 = Block3D(nni=10, nnj=30, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)

# blocks downstream of injector
pv = makeSimpleBox(xPos=L1+L2,yPos=0.0, xSize=L0-(L1+L2), ySize=Whalf1, zSize=H)
cflist = [cluster_i2,None,cluster_i2,None]*2 + [cluster_k,]*4;
blk20 = Block3D(nni=50, nnj=10, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)
pv = makeSimpleBox(xPos=L1+L2,yPos=Whalf1, xSize=L0-(L1+L2), ySize=Whalf0-Whalf1, zSize=H)
cflist = [cluster_i2,cluster_j1,cluster_i2,cluster_j1]*2 + [cluster_k,]*4;
blk21 = Block3D(nni=50, nnj=30, nnk=30, parametric_volume=pv,
                cf_list=cflist, fill_condition=initialCond)


identify_block_connections()
blk00.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
blk01.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
blk10.set_BC("BOTTOM", "SUP_IN", inflow_condition=injectCond)
blk20.set_BC("EAST", "SUP_OUT")
blk21.set_BC("EAST", "SUP_OUT")
```

32.2 Shell script

```
#! /bin/sh
# inject_run.sh
e3prep.py --job=inject
# time e3shared.exe --job=inject --run
mpirun -np 6 e3mpi.exe --job=inject --run
e3post.py --job=inject --vtk-xml
```

32.3 Notes

- The first part of the input script sets up an ideal-gas mixture model. This could have been done separately such that the thermochemistry files were already present at the preparation stage.
- For an ideal-gas model, the run time on 6 cores of *geyser* was 23,987 seconds for 21340 steps. This is about 1.1 μs-per-cell-per-update.

33 Flow of nitrogen over a cylinder of finite length

This example is relevant to Troy and Tim's X2 experiments on flows of weakly-ionizing nitrogen over cylinders of various length over diameter ratios. It exercises the three-dimensional flow solver with a strong bluff-body shock and a very sudden expansion over the end of the cylinder. The thermochemical module is also exercised with both near-equilibrium and frozen chemistry regions in the flow field and temperatures that rise above 20 000 K.

The flow domain shown is made up of 4 block-structured grids as shown in Figure 78 and number of the surface grids are are indicated in Figure 79 for a 15 mm diameter cylinder with $\frac{L}{D} = 2$. Note that only half of the length and only the upper-front quarter of the cylinder is in the simulation. Slip-wall boundary conditions are used (implicitly) along the planes of symmetry.

Figure 78: Left: full cylinder with the expected shock location scribed on the symmetry plane. Right: layout of finite-cylinder simulation with one-quarter of forward-facing half of the cylinder surface shown as wire-frame. Some of the edges of the flow domain are shown dashed and the labelled nodes correspond to those in the input script.

The free-stream conditions ($p_\infty = 2\,\text{kPa}$, $T_\infty = 3000\,\text{K}$ and $u_\infty = 10\,\text{km/s}$) correspond approximately to Troy's X2 experiments. These are representative of those produced by the X2 expansion tube and, for an ideal nitrogen test gas, the free stream Mach number is 8.96.

The script sets up the simulation to run for 30 flow-lengths ($30 * R_c/u_\infty$) and the final time reached is 22.5 μs The relieving effect on the shock is clear in both the pressure and temperature field (Figure 80). The temperature field also shows the influence of the finite-rate reactions with peak temperatures immediately behind the shock, followed by a relaxation as dissociation of the nitrogen molecules soaks up energy from within the shock layer.

Figure 79: A selection of surface grids from the finite-cylinder simulation, shown as wire-frame on the cylinder surface and coloured by pressure in the flow field. This PNG figure was generated with Paraview using block surfaces extrtacted from final solution file.

This case is quite difficult for both the flow solver and the finite-rate chemistry module and defects can be seen in the solution around the flat end of the cylinder and toward the outflow boundaries. These defects are quite obvious in the temperature field with a checker-board pattern of extreme high and low temperatures. However, the forebody flow looks to be reliably computed and the shock stand-off distance is 1.13 mm near the midplane of the cylinder.

Figure 80: Static temperature and mass fraction of nitrogen atoms in the flow field.

33.1 Input script (.py)

```
# \file cyl.py
#
# This geometry is a set of three blocks describing a quarter-cylinder
# of finite length in supersonic flow.
#
# PJ, 20-Jun-2005, 04-Dec-2005 increase number of blocks along cylinder axis
#     06-Feb-2006 new geometry objects
#     19-Aug-2009 Eilmer3 port
#     23-Jan-2010 SuperBlock3D and use of MPI code cor comparison
# RJG, 02-Apr-2007 new reacting gas spec.

gdata.dimensions = 3
D = 15.0e-3  # Diameter of cylinder, metres
L = 2.0 * D  # (axial) length of cylinder

# Gas model used in the simulation.
select_gas_model(model='thermally perfect gas', species=['N2','N','N2+','N+','e-'])
set_reaction_scheme("nitrogen-5sp-6r.lua",reacting_flag=1)
mf = {'N2':1.0}

# Free-stream properties
T_inf = 3000.0 # degrees K
p_inf = 2000.0 # Pa
u_inf = 10000.0 # m/s
gdata.title = "Cylinder L/D=%g in N2 at u=%g m/s." % (L/D, u_inf)
print "title=", gdata.title

# Flow conditions for fill and boundary conditions.
inflowCond = FlowCondition(p=p_inf, u=u_inf, v=0.0, T=T_inf, massf=mf)
initialCond = FlowCondition(p=p_inf/3.0, u=0.0, v=0.0, T=300.0, massf=mf)

# Geometry is built from the bottom up.
Rc = D/2.0  # cylinder radius

# Define a few key nodes.
a = Node(-Rc, 0.0, 0.0, label="a")  # stagnation point on the cylinder
b = Node( 0.0, Rc, 0.0, label="b")  # top of cylinder
c = Node( 0.0, 0.0, 0.0, label="c") # centre of curvature

# In order to have a grid that fits reasonably close the the shock,
# use Billig's shock shape correlation to generate
# a few sample points along the expected shock position.
from math import sqrt
from cfpylib.gasdyn.billig import x_from_y
# ideal N2 properties used for shock shape estimate
R_N2 = 296.8
gamma_N2 = 1.4
a_inf = sqrt(gamma_N2 * R_N2 * T_inf)
M_inf = u_inf / a_inf
print "M_inf=", M_inf
xys = []
for y in [0.0, 0.5, 1.0, 1.5, 2.0, 2.5]:
    x = x_from_y(y*Rc, M_inf, theta=0.0, axi=0, R_nose=Rc)
    xys.append((x,y*Rc)) # a new coordinate pair
    print "x=", x, "y=", y

# Scale the Billig distances, depending on the expected behaviour
# relative to the gamma=1.4 ideal gas.
if gdata.reacting_flag == 1:
    b_scale = 0.87  # for finite-rate chemistry
else:
    b_scale = 1.1   # for ideal (frozen-chemistry) gas
d = [] # will use a list to keep the nodes for the shock boundary
for x, y in xys:
    # the outer boundary should be a little further than the shock itself
    d.append(Node(-b_scale*x, b_scale*y, 0.0, label="d"))
print "front of grid: d[0]=", d[0]
```

```
# Extent of the cylinder in the z-direction to end face.
c2 = c.clone(); c2.translate(0.0, 0.0, L/2.0)
e = d[0].clone().translate(0.0, 0.0, L/2.0)
f = a.clone().translate(0.0, 0.0, L/2.0)
g = Node(-Rc/2.0, 0.0, L/2.0)
h = Node(0.0, Rc/2.0, L/2.0)
i = Node(0.0, Rc, L/2.0)
# the domain is extended beyond the end of the cylinder
j = e.clone().translate(0.0, 0.0, Rc)
k = f.clone().translate(0.0, 0.0, Rc)

# ...then lines, arcs, etc, that will make up the domain-end face.
xaxis = Line(d[0], a) # first-point of shock to nose of cylinder
cylinder = Arc(a, b, c)
shock = Spline(d)
outlet = Line(d[-1], b)  # top-point of shock to top of cylinder
domain_end_face = CoonsPatch(xaxis, outlet, shock, cylinder)

# ...lines along which we shall extrude the domain-end face
yaxis0 = Line(d[0], e)
yaxis1 = Line(e, j)

# End-face of cylinder
xaxis = Line(f, g)
cylinder = Arc(f, i, c2)
inner = Arc(g, h, c2)
outlet = Line(i, h)
cyl_end_face = CoonsPatch(xaxis, outlet, cylinder, inner)
yaxis2 = Line(f, k)

# Third, set up the blocks from the geometric and flow elements.
nr = 20            # radial discretization
nc = int(1.5 * nr) # circumferential discretization
na = int(L/D * nc) # axial discretization along the cylinder
na1 = nc           # axial discretization off the end of the cylinder
nr2 = int(nr/2)    # radial discretization toward the cylinder axis

# The volume constructor extrudes the end-face along the axis in the k-direction.
# We want to divide the over-cylinder block up to make reasonable use of the
# cluster computer.
blk0 = SuperBlock3D(label="over-cylinder", nni=nr, nnj=nc, nnk=na, nbk=int(L/D),
                    parametric_volume=WireFrameVolume(domain_end_face,yaxis0,"k"),
                    fill_condition=initialCond)
for blk in blk0.blks[0][0]:
    # We work along the line of blocks in the k-direction
    blk.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
    blk.set_BC("NORTH", "SUP_OUT")

blk1 = Block3D(label="outside-cylinder", nni=nr, nnj=nc, nnk=na1,
               parametric_volume=WireFrameVolume(domain_end_face,yaxis1,"k"),
               fill_condition=initialCond)
blk1.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
blk1.set_BC("NORTH", "SUP_OUT")

blk2 = Block3D(label="beside-cylinder", nni=nr2, nnj=nc, nnk=na1,
               parametric_volume=WireFrameVolume(cyl_end_face,yaxis2,"k"),
               fill_condition=initialCond)
blk2.set_BC("EAST", "SUP_OUT")
blk2.set_BC("NORTH", "SUP_OUT")

identify_block_connections()

# Finally, Other simulation control parameters. ----------
gdata.viscous_flag = 0
gdata.flux_calc = ADAPTIVE
gdata.interpolation_type = "pT"
gdata.t_order = 1
gdata.x_order = 2
gdata.max_time = Rc/u_inf * 30
gdata.max_step = 40000
gdata.dt = 1.0e-10
gdata.cfl = 0.5
gdata.dt_history = 1.0e-5
```

```
gdata.dt_plot = gdata.max_time/2

print "Total number of blocks=", len(blk0.blks)+2
```

33.2 Reaction scheme file (.lua)

```
-- Author: Rowan J. Gollan
-- Date: 11-Nov-2009
-- Place: Poquoson, Virginia, USA
--
-- Updated from the work by RJG and DFP as found in
-- lib/gas_models2/input_files/nitrogen/nitrogen-5sp-6r.py
--
-- Note: Based on Dan's comments, I've only included
-- the Goekcen rates at present.
--
-- Reference:
-- Goekcen (2004)
-- N2-CH4-Ar Chemical Kinetic Model for Simulations of
-- Atmospheric Entry to Titan
-- AIAA Paper 2004-2469
--

reaction{
   'N2 + N2 <=> N + N + N2',
   fr={'Arrhenius', A=7.0e21, n=-1.6, T_a=113200.0}
}

reaction{
   'N2 + N <=> N + N + N',
   fr={'Arrhenius', A=3.0e22, n=-1.6, T_a=113200.0}
}

reaction{
   'N2 + e- <=> N + N + e-',
   fr={'Arrhenius', A=3.0e24, n=-1.6, T_a=113200.0}
}

reaction{
   'N + N <=> N2+ + e-',
   fr={'Arrhenius', A=4.40e7, n=1.5, T_a=67500.0}
}

reaction{
   'N2 + N+ <=> N2+ + N',
   fr={'Arrhenius', A=1.0e12, n=0.5, T_a=12200.0}
}

reaction{
   'N + e- <=> N+ + e- + e-',
   fr={'Arrhenius', A=2.50e34, n=-3.82, T_a=168600.0}
}
```

33.3 Shell script

```
#!/bin/bash
# run_simulation.sh
#$ -S /bin/bash
#$ -N FiniteCyl
#$ -pe orte 4
#$ -cwd
#$ -V

job=cyl
np=4

echo "Start time: "; date
mpirun -np $np e3mpi.exe --job=$job --run
# e3shared.exe --job=$job --run
echo "Finish time: "; date
```

33.4 Postprocessing program

```
#!/bin/bash
# post_simulation.sh

# Create a VTK plot file of the steady full flow field.
e3post.py --job=cyl --tindx=9999 --vtk-xml

# Pull out the cylinder surfaces.
e3post.py --job=cyl --tindx=9999 --output-file=cylinder \
    --surface-list="0,east;1,east;3,bottom"

# Now pull out some block surfaces that show cross-sections of the flow field.
e3post.py --job=cyl --tindx=9999 --output-file=interior \
    --surface-list="0,bottom;1,bottom;0,north;1,north;2,north;3,north;0,south;1,south;2,south;3,south;3,ea

# Stagnation-line flow data
e3post.py --job=cyl --tindx=9999 --slice-list="0,:,0,0" \
    --output-file=stagnation-line.data
```

```
#! /usr/bin/env python
# \file locate_bow_shock.py
# PJ, 08-Nov-2009, updated for Eilmer3

import sys, os, gzip
sys.path.append(os.path.expandvars("$HOME/e3bin"))
from e3_flow import StructuredGridFlow

print "Locate a bow shock by its pressure jump."

# Block 0 contains the stagnation point.
fileName = 'flow/t9999/cyl.flow.b0000.t9999.gz'
fp = gzip.open(fileName, "r")
blockData = StructuredGridFlow()
blockData.read(fp)
fp.close()

# Since this is a 3D simulation, the shock is not expected
# to be flat in the k-direction (along the cylinder axis).
# Sample the shock layer in a few places near the stagnation line.
x_sum = 0.0
n_sample = 6
```

```
for k in range(n_sample):
    j = 0
    p_trigger = 10000.0  # Pa
    x_old = blockData.data['pos.x'][0,j,k]
    p_old = blockData.data['p'][0,j,k]
    for i in range(blockData.ni):
        x = blockData.data['pos.x'][i,j,k]
        p = blockData.data['p'][i,j,k]
        if p > p_trigger: break
        x_old = x
        p_old = p
    frac = (p_trigger - p_old) / (p - p_old)
    x_loc = x_old * (1.0 - frac) + x * frac
    print "shock at x=", x_loc, \
        "y=", blockData.data['pos.y'][0,j,k], \
        "z=", blockData.data['pos.z'][0,j,k]
    x_sum += x_loc
x_average = x_sum / n_sample
print "Average x-location=", x_average

print "Done."
```

33.5 Notes

- It is well worth the bother to run this simulation on multiple processors. The elapsed time for the run with 4 MPI processes is 9193 seconds on geyser, a Dell server with 4×4 AMD cores. Of course, our MPI job used only 4 of those cores.

- We tried a couple of reconstruction variations. The original simulation required 5999 steps using `rhoe` interpolation. Using `pT` interpolation (as shown in the script), the simulation required an elapsed computing time of 8492 seconds and 6025 steps on `geyser` in January 2010. In August 2010, the same calculation took 4315 seconds on 4 cores of the barrine cluster to do 6023 steps at final time.

- To double the grid resolution (as one might want to do for a convergence study), would require a factor of 8 increase in memory. If you are planning to do calculations of any reasonable complexity, it is worth your while to invest in learning to use the cluster computer and the parallel version of the code.

34 Spherically-blunted cone

An aeroshell-type model is shown in Figure 81. The surface of the aeroshell is constructed as a `RevolvedSurface` with a sphere blended into a cone. The construction `Path` is a `Polyline` consisting of an `Arc` and a straight `Line`. The outer (inflow) surface of the block is constructed by revolving a spline (approximating Billig's shock shape) about the x-axis. For specifying the flow domain, only subsections of these surfaces were used (as `MappedSurface` objects) as opposite sides of the single block grid. The remaining four surfaces were constructed by joining the edges and corners of these two original surfaces. During the development of this example, it was useful to view parts of the constructed paths and surfaces and a later section of the input script shows how the objects can be rendered to a Virtual Reality Markup Language (VRML) file. Although, not as convenient as a direct-manipulation graphical interface, this rendering facility does enable the debugging of fairly complex constructions.

Figure 81: Views of the pressure field around a spherically-blunted cone. The left figure is the cell-average data for the entire block rendered as a coloured surface. The view is from behind the aeroshell surface. Only one half of the `RevolvedSurface` was used in the simulation. The right figure shows two cutting planes through the block of data, coloured according to pressure, again. The surface mesh corresponds to the EAST boundary surface of the block and is shown with its mirror image in the (x,y)-plane.

For the $20 \times 20 \times 40$ grid and requested final time of 5 ms in this simulation, the run time was a fairly short 9m8.5s on geyser. (Compare with `Elmer2`, which used 3m57s on the LG LS70 laptop for the same exercise.) The grid generation phase takes a realtively long time because of the implied nested function calls required by the interpolation procedure when using mapped surfaces and paths defined on those mapped surfaces. For finer grids, the grid generation will become quite slow but it is a once-off cost.

34.1 Input script (.py)

```
# A job description file for a spherically-blunted cone.
# PJ
# Elmer2 original: 13-Feb-2006
# Eilmer3 port: 06-Feb-2010

from math import *
from cfpylib.gasdyn.billig import x_from_y

# First, set the global data
gdata.title = "Sphere-cone."
gdata.dimensions = 3
select_gas_model(model='ideal gas', species=['air'])
gdata.max_time = 5.0e-3
gdata.dt = 1.0e-7
gdata.max_step = 1000

# Second, set up flow conditions
initialCond = FlowCondition(p=1000.0, u=0.0, T=300.0)
M_inf = 4.0
u_inf = M_inf * initialCond.flow.gas.a
inflowCond  = FlowCondition(p=50.0e3, u=u_inf, T=300.0)

# Third, set up the block

# The vehicle surface is defined as a path that is revolved about the x-axis.
Rnose = 1.0                  # radius of spherical nose
Angle = 45.0 * pi / 180.0    # angle of cone wrt x-axis
Dmax = 4.0                   # diameter of base
# The conical section extends from nose to base radius.
Length = (Dmax / 2.0 - Rnose * sin(Angle)) / sin(Angle)
c = Vector(0.0, 0.0, 0.0)    # centre of radius
a = Vector(-Rnose, 0.0, 0.0) # tip of nose
b = Vector(-Rnose*cos(Angle), Rnose*sin(Angle), 0.0) # join between sphere and cone
d = b + Length * Vector(cos(Angle), sin(Angle), 0.0) # skirt of cone
path = Polyline([Arc(a,b,c), Line(b,d)])
surf1 = RevolvedSurface(path, "vehicle_surface")
# To put a mesh onto this revolved surface, we define a query surface with
# a better outline for the block grid.
L2 = Dmax / 2.0 / sqrt(2.0)
# We have made sure that our query surface is within the bounds of the original.
q0 = Vector3(0.0, -L2, L2)
q1 = Vector3(0.0, -L2, 0.0)
q2 = Vector3(0.0,  L2, 0.0)
q3 = Vector3(0.0,  L2, L2)
qsurf2 = CoonsPatch(q0, q1, q2, q3, "query_surface")
east = MappedSurface(qsurf2, surf1)

# The outer mesh surface is derived from Billig's shock-shape correlation.
# In preparation for defining nodes, generate a few sample points
# along the expected shock position.
e = [] # will use a list to keep the nodes for the shock boundary
for y in [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2]:
    y *= Dmax/2.0  # scale up to cover the base of the vehicle
    # Note that we lie about the cone angle.  Detached shock.
    x = x_from_y(y, M_inf, theta=20.0/180.0*pi, axi=1, R_nose=Rnose)
    # print "x=", x, "y=", y
    # the outer boundary should be a little further than the shock itself
    e.append( Vector(-1.2*x, 1.2*y, 0.0) )
shock = Spline(e)
# print "shock=", shock
surf2 = RevolvedSurface(shock, "shock_surface")
L3 = e[-1].y / sqrt(2.0)
qs0 = Vector3(0.0, -L3, L3)
qs1 = Vector3(0.0, -L3, 0.0)
qs2 = Vector3(0.0,  L3, 0.0)
qs3 = Vector3(0.0,  L3, L3)
qsurf2 = CoonsPatch(qs0, qs1, qs2, qs3, "query_surface_shock")
west = MappedSurface(qsurf2, surf2)
```

```
p0 = west.eval(0.0, 0.0)
p1 = east.eval(0.0, 0.0)
p2 = east.eval(1.0, 0.0)
p3 = west.eval(1.0, 0.0)
p4 = west.eval(0.0, 1.0)
p5 = east.eval(0.0, 1.0)
p6 = east.eval(1.0, 1.0)
p7 = west.eval(1.0, 1.0)
# print "p0=", p0, "p1=", p1

# We shall assemble the other surfaces as CoonsPatch surfaces with
# their relevant bounding edges lying on the shock and body surfaces.
c76 = Line(p7, p6)
c32 = Line(p3, p2)
c37 = PathOnSurface(west, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
c26 = PathOnSurface(east, LinearFunction(0.0,1.0), LinearFunction(1.0,0.0))
north = CoonsPatch(c32, c76, c37, c26, "symmetry-plane")

c45 = Line(p4, p5)
c01 = Line(p0, p1)
c04 = PathOnSurface(west, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
c15 = PathOnSurface(east, LinearFunction(0.0,0.0), LinearFunction(1.0,0.0))
south = CoonsPatch(c01, c45, c04, c15, "south-outflow")

c47 = PathOnSurface(west, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
c56 = PathOnSurface(east, LinearFunction(1.0,0.0), LinearFunction(0.0,1.0))
top = CoonsPatch(c45, c76, c47, c56)

c03 = PathOnSurface(west, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
c12 = PathOnSurface(east, LinearFunction(1.0,0.0), LinearFunction(0.0,0.0))
bottom = CoonsPatch(c01, c32, c03, c12)

if 0:
    # Here is a bit of debug...
    # You can look to see that the surfaces are reasonable and join at the edges.
    # print surf2
    # print "surf2.eval(0.25,0.75)=", surf2.eval(0.25,0.75)
    # print "west=", west
    # print "west.eval(0.25,0.75)=", west.eval(0.25,0.75)
    print "Render to VRML"
    outfile = open("sphere-cone.wrl", "w")
    outfile.write("#VRML V2.0 utf8\n")
    outfile.write(east.vrml_str() + "\n")
    outfile.write(west.vrml_str() + "\n")
    outfile.write(north.vrml_str() + "\n")
    outfile.write(south.vrml_str() + "\n")
    outfile.write(top.vrml_str() + "\n")
    outfile.write(bottom.vrml_str() + "\n")
    outfile.close()
    import sys; sys.exit()

# Assemble the surfaces into a volume.
pvolume = ParametricVolume(north, east, south, west, top, bottom, "Sphere-cone")

blk = Block3D(label="first-block", nni=20, nnj=20, nnk=40,
              parametric_volume=pvolume,
              fill_condition=initialCond)
blk.set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
blk.set_BC("SOUTH", "SUP_OUT")
blk.set_BC("TOP", "SUP_OUT")
blk.set_BC("BOTTOM", "SUP_OUT")
```

35 Katsu's scramjet combustor and nozzle

The core of Katsuyoshi Tanimizu's scramjet model is shown in Figure 82. It was designed in 1994 by Prof. Ray Stalker and consists of 6 scramjet ducts distributed around a centrebody.

Figure 82: The core section of the scramjet model with the cowl removed from the combustor and nozzle sections to show the individual scramjet ducts between the dividing walls. The inlets are at the upper-left of the image. This image was scanned from a document provided by Katsu.

The geometry for only half of one duct is set up in the job script. We use a TrianglePatch surface for the side wall (north surface) which is is somewhat angular; in the original model, it was milled from solid. The cowl and centre-body surfaces were originally cut on a lathe and so are curved about the streamwise axis. These surfaces (top and bottom) are approximated in sections as CoonsPatch surfaces and then faceted into TrianglePatch surfaces in order to join consistently with the side (north and south) walls. So that the volume is properly closed, the inflow(west) and outflow (east) surfaces are defined from paths along the edges of the other four surfaces.

The simulation is run for only 1000 steps and reaches this point in a little over 12 minutes on an Intel E2140 @ 1.6GHz (euler). Compare this with 4 minutes on an LG LS70 laptop computer; we really need to do some code profiling and optimization. The presure distribution across some of the surfaces is shown in Figure 84

Figure 83: Front-view of the wireframe representation of one scramjet duct. The labelling of key points corresponds to that used in the job input file with the exception that points 9 through 16 were moved to the centre-plane of the duct (south surface).

Figure 84: Pressure distribution on the inlet, combustor, nozzle, and cowl surfaces of the scramjet duct.

35.1 Input script (.py)

```
# A job description file for a simplified scramjet combustor and nozzle.
# PJ, Feb, Mar 2006
#     Apr 2007, updated for Elmer2
#     Feb 2010, updated for Eilmer3
# ------------------------------------------------------------------
from math import pi, sin, cos
def deg2rad(d):
    from math import pi
    return d/180.0*pi

# Define some geometric parameters that will be useful for specifying control points.
# See Katsu's sketch and workbook sketch on page 37 for labelling of points.
r1 = r2 = r3 = 34.0e-3
r4 = 41.5e-3
r5 = 14.6e-3
r6 = 18.0e-3
r7 = 28.0e-3
r8 = 30.0e-3
x1 = x8 = -95.981e-3
x2 = x7 = -65.981e-3
x3 = -60.564e-3
x4 = x5 = 9.019e-3
x6 = -15.858e-3
th1 = th8 = deg2rad(11.6)
th2 = th7 = deg2rad(14.0)
th3 = deg2rad(16.0)
th4 = th5 = deg2rad(29.0)
th6 = deg2rad(18.5)

# Create the collection of points for use in defining the surfaces.
p0  = Vector( 0.0, 0.0, 0.0  )
p1  = Vector( x1, r1*cos(th1), -r1*sin(th1) )
p2  = Vector( x2, r2*cos(th2), -r2*sin(th2) )
p3  = Vector( x3, r3*cos(th3), -r3*sin(th3) )
p4  = Vector( x4, r4*cos(th4), -r4*sin(th4) )
p5  = Vector( x5, r5*cos(th5), -r5*sin(th5) )
p6  = Vector( x6, r6*cos(th6), -r6*sin(th6) )
p7  = Vector( x7, r7*cos(th7), -r7*sin(th7) )
p8  = Vector( x8, r8*cos(th8), -r8*sin(th8) )
# Define the plane of symmetry
p9  = Vector( x1, r1, 0.0 )
p10 = Vector( x2, r2, 0.0 )
p11 = Vector( x3, r3, 0.0 )
p12 = Vector( x4, r4, 0.0 )
p13 = Vector( x5, r5, 0.0 )
p14 = Vector( x6, r6, 0.0 )
p15 = Vector( x7, r7, 0.0 )
p16 = Vector( x8, r8, 0.0 )
# A few more points along the x-axis for later generation of circular arcs.
p1_0 = Vector( x1, 0.0, 0.0 )
p2_0 = Vector( x2, 0.0, 0.0 )
p3_0 = Vector( x3, 0.0, 0.0 )
p4_0 = Vector( x4, 0.0, 0.0 )
p6_0 = Vector( x6, 0.0, 0.0 )

# North and south surfaces are defined directly as TrianglePatches
# In preparation, gather the control points into a single list.
# Note that a list is indexed from 0.
p = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10,
     p11, p12, p13, p14, p15, p16]
north = TrianglePatch(p, [1,8,7, 1,7,2, 2,7,3, 7,6,3, 3,6,4, 6,5,4],
                      [8,7,6,5], [1,2,3,4], [8,1], [5,4], "NORTH")
south = TrianglePatch(p, [9,16,15, 9,15,10, 10,15,11, 15,14,11, 11,14,12, 14,13,12],
                      [16,15,14,13], [9,10,11,12], [16,9], [13,12], "SOUTH")

# The top and bottom surfaces are somewhat curved.
# The surfaces in the physical model were cut on a lathe.
c9_1 = Arc(p9, p1, p1_0)
```

```
c10_2 = Arc(p10, p2, p2_0)
c9_10 = Line(p9, p10)
c1_2 = Line(p1, p2)
top = TrianglePatch(CoonsPatch(c9_10, c1_2, c9_1, c10_2), 1, 5, "COWL")
c11_3 = Arc(p11, p3, p3_0)
c10_11 = Line(p10, p11)
c2_3 = Line(p2, p3)
top.add(TrianglePatch(CoonsPatch(c10_11, c2_3, c10_2, c11_3), 1, 5))
c12_4 = Arc(p12, p4, p4_0)
c11_12 = Line(p11, p12)
c3_4 = Line(p3, p4)
top.add(TrianglePatch(CoonsPatch(c11_12, c3_4, c11_3, c12_4), 1, 5))

c16_8 = Arc(p16, p8, p1_0)
c15_7 = Arc(p15, p7, p2_0)
c16_15 = Line(p16, p15)
c8_7 = Line(p8, p7)
bottom = TrianglePatch(CoonsPatch(c16_15, c8_7, c16_8, c15_7), 1, 5, "CENTRE-BODY")
c14_6 = Arc(p14, p6, p6_0)
c15_14 = Line(p15, p14)
c7_6 = Line(p7, p6)
bottom.add(TrianglePatch(CoonsPatch(c15_14, c7_6, c15_7, c14_6), 1, 5))
c13_5 = Arc(p13, p5, p4_0)
c14_13 = Line(p14, p13)
c6_5 = Line(p6, p5)
bottom.add(TrianglePatch(CoonsPatch(c14_13, c6_5, c14_6, c13_5), 1, 5))

# The west and east faces are built to close the ends of the duct.
f_zero = LinearFunction(0.0, 0.0)
f_one = LinearFunction(0.0, 1.0)
f_linear = LinearFunction(1.0, 0.0)
cA = PathOnSurface(bottom, f_zero, f_linear)
cB = PathOnSurface(top, f_zero, f_linear)
cC = PathOnSurface(south, f_zero, f_linear)
cD = PathOnSurface(north, f_zero, f_linear)
west = CoonsPatch(cA, cB, cC, cD, "INLET")
cA = PathOnSurface(bottom, f_one, f_linear)
cB = PathOnSurface(top, f_one, f_linear)
cC = PathOnSurface(south, f_one, f_linear)
cD = PathOnSurface(north, f_one, f_linear)
east = CoonsPatch(cA, cB, cC, cD, "OUTLET")

# then assemble the surfaces into a volume.
pvolume = ParametricVolume([south, bottom, west, east, north, top],
                           "Simplified-scramjet")

# ------------------------------------------------------------------
# Now, to the flow part of the simulation definition...
gdata.title = "Simplified scramjet duct -- Katsu."
gdata.dimensions = 3
# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
gdata.max_time = 5.0e-3
gdata.dt = 1.0e-9
gdata.max_step = 1000
initialCond = FlowCondition(p=1000.0, u=0.0,    T=304.0)
inflowCond  = FlowCondition(p=50.0e3, u=2000.0, T=300.0)
nblocks = 3
blk = MultiBlock3D(label="duct",
                   parametric_volume=pvolume,
                   nbi=nblocks,
                   nni=40, nnj=20, nnk=20,
                   fill_condition=initialCond)
# Inlet to the nozzle is the first block.
blk.blks[0][0][0].set_BC("WEST", "SUP_IN", inflow_condition=inflowCond)
# Exit from the nozzle is in the last block.
blk.blks[nblocks-1][0][0].set_BC("EAST", "SUP_OUT")

# We are done with definitions; e3prep.py will do its work...
```

36 Titan aeroshell using imported grids

Another aeroshell model is shown in Figure 85. The grids were generated by Bianca Capra using ICEM-CFD grid generation software and written as Plot3D format file. These files were then converted to VTK files with the following script:

```
#!/bin/sh
# prepare_grid.sh

gzip -d icem_grid_plot3d.fmt
import_grid.py --input=icem_grid_plot3d.fmt \
               --output=icem_grid \
               --plot3dplanes
gzip icem_grid_plot3d.fmt

echo "Done."
```

The identification of the (seemingly random) orientation of each block was done manually by loading the VTK data into ParaView and examinng the grid planes as the indices were adjusted from one limit to another.

Figure 85: Views of the temperature and pressure fields around a Titan aeroshell. The surface grid is shown as a wire-frame rendering and a vertical slice (with solid colouring) is made through the flow field around the aeroshell.

Once in VTK format, the block meshes can be read in and used as `ParametricVolume` objects within the user's job script. We can then generate gride of arbitrary resolution from the original ICEM grids. Note that the bulk of the script is used to assign the boundary conditions to each block because the original information about boundary conditions (as might have been part of the ICEM database) is not available from the Plot3D file.

Because this exercise is only to show that complex grids can be imported, the simulation was run at low grid resolution and to a final time of 300 ms. This is long enough

for the Mach 7 flow to establish over the aeroshell and, on the `geyser` server, took just under one hour (3554 s) of CPU time and required 2251 steps.

36.1 Input script (.py)

```
# titan_x2_shell.py
# A job description file for Bianca's Titan Aeroshell used in X2.
# PJ
# 30-Oct-2006: Elmer2 original
# 07-Feb-2010: Eilmer3 port


# First, set the global data
gdata.title = "Titan Aeroshell used in X2."
gdata.dimensions = 3
# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])
gdata.max_time = 300.0e-3
gdata.dt = 1.0e-7
gdata.max_step = 5000
gdata.dt_plot = 30.0e-3

# Second, set up flow conditions
from math import pi, sin, cos
alpha = 20.0*pi/180.0 # angle of attack in radians
initialCond = FlowCondition(p=1000.0, u=0.0, T=300.0)
M_inf = 7.0
u_inf = M_inf * initialCond.flow.gas.a
inflowCond  = FlowCondition(p=50.0e3, u=-u_inf*cos(alpha),
                            v=u_inf*sin(alpha), T=300.0)

# Third, set up the blocks from the ICEM-generated grids.
# The discretization is just a fraction of the original ICEM grids.
# block      0  1  2  3  4  5  6  7  8  9 10 11 12
nni_list = [10,10,10,24,10,10,10,10,10, 7,10,10, 3]
nnj_list = [10,24,24,10,10,10,10, 7, 7,10, 3, 3,10]
nnk_list = [30,30,30,30,30,30,30,30,30,30,30,30,30]
pv_list = []
blk_list = []
for ib in range(13):
    pv_list.append( MeshVolume("icem_grid."+str(ib)+".g.vtk") )
    blk_list.append( Block3D(nni=nni_list[ib],
                             nnj=nnj_list[ib],
                             nnk=nnk_list[ib],
                             parametric_volume=pv_list[ib],
                             fill_condition=initialCond) )
identify_block_connections()

# Apply boundary conditions.
# The appropriate surfaces were determined by loading each block
# with MayaVi, then putting on a gridplane, and fiddling with the
# index directions to find out which surface was which.
blk_list[0].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[0].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[1].set_BC("BOTTOM", "SUP_IN", inflow_condition=inflowCond)
blk_list[1].set_BC("TOP", "FIXED_T", Twall=300.0)
blk_list[1].set_BC("SOUTH", "SUP_OUT")
blk_list[2].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[2].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[2].set_BC("SOUTH", "SUP_OUT")
blk_list[3].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[3].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[3].set_BC("WEST", "SUP_OUT")
blk_list[4].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[4].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[5].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[5].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
```

```
blk_list[6].set_BC("BOTTOM", "SUP_IN", inflow_condition=inflowCond)
blk_list[6].set_BC("TOP", "FIXED_T", Twall=300.0)
blk_list[7].set_BC("BOTTOM", "SUP_IN", inflow_condition=inflowCond)
blk_list[7].set_BC("TOP", "FIXED_T", Twall=300.0)
blk_list[8].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[8].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[9].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[9].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[10].set_BC("BOTTOM", "SUP_IN", inflow_condition=inflowCond)
blk_list[10].set_BC("TOP", "FIXED_T", Twall=300.0)
blk_list[11].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[11].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
blk_list[12].set_BC("BOTTOM", "FIXED_T", Twall=300.0)
blk_list[12].set_BC("TOP", "SUP_IN", inflow_condition=inflowCond)
```

Part IV
References and Appendices

References

[1] P. A. Jacobs, R. J. Gollan, A. J. Denman, B. T. O'Flaherty, D. F. Potter, P. J. Petrie-Repar, and I. A. Johnston. Eilmer's theory book: Basic models for gas dynamics and thermochemistry. Mechanical Engineering Report 2010/09, The University of Queensland, Brisbane, Australia, 2010.

[2] P. A. Jacobs. Single-block Navier-Stokes integrator. ICASE Interim Report 18, 1991.

[3] Ames Research Staff. Equations, tables and charts for compressible flow. NACA Report 1135, 1953.

[4] J. D. Anderson. *Hypersonic and High Temperature Gas Dynamics.* McGraw-Hill, New York, 1989.

[5] J. M. Kendall. Experiments on supersonic blunt-body flows. Progress Report 20-372, Jet Propulsopn Laboratory, California Institute of Technology, Pasadena, California., February 1959.

[6] L. H. Back, P. F. Massier, and H. L. Gier. Comparison of measured and predicted flows through conical supersonic nozzles, with emphasis on the transonic region. *A.I.A.A. Journal*, 3(9):1606–1614, 1965.

[7] K. Sawada and E. Dendou. Validation of hypersonic chemical equilibrium flow calculations using ballistic-range data. *Shock Waves*, 11:43–51, 2001.

[8] P. H. Rose and W. I. Stark. Stagnation point heat-transfer measurements in dissociated air. *Journal of the Aeronautical Sciences*, (February):86–97, 1958.

[9] N. H. Kemp, R. H. Rose, and R. W. Detra. Laminar heat transfer around blunt bodies in dissociated air. *Journal of the Aero/Space Sciences*, 26(7):421–430, 1959.

[10] H. F. Lehr. Experiments on shock induced combustion. *Astronautica Acta*, 17:589–597, 1972.

[11] G. J. Wilson. Computation of steady and unsteady shock-induced combustion over hypervelocity blunt bodies. PhD thesis, Stanford University, California., December 1991.

[12] A. Aftosmis, D. Gaitonde, and T. S. Tavares. Behaviour of linear reconstruction techniques on unstructured meshes. *A.I.A.A. Journal*, 33(11):2038–2049, 1995.

[13] Mark G. Sobell. *A Practical Guide to Linux Commands, Editors and Shell Programming.* Prentice Hall, Upper Saddle River, New Jersey, 2005.

[14] R. Ierusalimschy, L. H. de Figueiredo, and W.C. Filho. Lua - an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[15] S. Gordon and B. J. McBride. Computer program for calculation of complex chemical equilibrium compositions and applications. part 1: Analysis. Reference Publication 1311, NASA, 1994.

[16] B. J. McBride and S. Gordon. Computer program for calculation of complex chemical equilibrium compositions and applications. part 2: Users manual and program description. Reference Publication 1311, NASA, 1996.

[17] D. R. Mott. *New Quasi-Steady-State and Partial-Equilibrium Methods for Integrating Chemically Reacting Systems.* PhD thesis, University of Michigan, 1999.

[18] E. Fehlberg. Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems. Technical Report R-315, NASA, 1969.

A README.txt: Instructions for installation

The latest version of this file should be in the root directory of the package of source files. This loation is typically `$HOME/cfcfd2` on our development machines.

```
The Compressible-Flow CFD code collection
=========================================
:Author: Peter_Jacobs and Rowan_Gollan
:Email: peterj@mech.uq.edu.au


The codes
---------
.The main codes are found in the app subdirectory.
- *bos*: Background-oriented schlieren (current but needs maintenance; new numpy)
- *elmer*: The first version of our time-dependent 3D flow simulation code.
  C version built for LES simulations. (obsolete)
- *elmer2*: Second version using libgas2 thermochemsitry and written in C++.
  (obsolete)
- *eilmer3*: The current compresible-flow Navier-Stokes simulation code
  for 2D and 3D flows.
  This has the best bits of mbcns2 and elmer2, together with a rebuilt
  thermochemistry module.
  We have changed the name to Eilmer, the correct spelling of the name of the
  patron monk of CFD and aeronautics.
  See the wikipedia article http://en.wikipedia.org/wiki/Eilmer_of_Malmesbury
- *imoc*: interactive method-of-characteristics code, written in C+Tcl.
- *L1d*: Lagrangian simulation of quasi-one-dimensional flow. (obsolete)
- *L1d2*: second version of this code build with libgas2 thermochemistry.
  Written in C++. (current)
- *mb_cns*: Multiple-block compressible Navier-Stokes flow simulator.
  This is the code corresponding to the 1996 report. (obsolete)
- *mbcns2*: second version using libgas2 and written in C++. (obsolete)
- *newt*: Newtonian flow approximation for hypersonics flows.
- *octvce*: Joseph's fast Euler code with hierarchical adaptive grid for 3D flows.
- *poshax*: one-dimensional flow with complex thermochemistry.
- *sdas*: firmware for simple data acquisition system for student Zuni projects.
- *sf3d_v4.1*: Ian Johnston's shock-fitting code for 3D flow.
- *sm_3d_plus*: space-marching code for 3D flow.
  (obsolete now that block-sequencing is part of eilmer3)
- *tds*: tunnel data server, software for both the server and the client


Getting the simulation codes
----------------------------
The top-level repository doesn't have many files in itself.
Most of the interesting stuff is in the sub-repositories:
app, doc, examples and lib.
One may get a complete working copy via http-based checkout
(with read-only access).

[source,shell]
cd $HOME
svn checkout --username cfcfd http://triton.pselab.uq.edu.au/repos/cfcfd2 cfcfd2
cd cfcfd2
svn checkout http://triton.pselab.uq.edu.au/repos-cfcfd2/app app
svn checkout http://triton.pselab.uq.edu.au/repos-cfcfd2/lib lib
svn checkout http://triton.pselab.uq.edu.au/repos-cfcfd2/examples examples
svn checkout http://triton.pselab.uq.edu.au/repos-cfcfd2/doc doc

.Notes
- When prompted, the password is: cfd-socks
- There is a hyphen between repos and cfcfd2 for the second through fifth
checkouts.


Getting the simulation codes -- Developers
------------------------------------------
Developers should get the most up-to-date code using bazaar-ng.
```

```
To get (read-only) access via http, use the command:

[source,shell]
cd $HOME
bzr branch http://cfcfdlocal@triton.pselab.uq.edu.au/bzr-cfcfd2 cfcfd2

.Notes
- You will need a password for bzr access.  Please ask.
- To push code changes, you will need access via ssh.


Your build environment
----------------------
The code collection comes as source code only so,
to use any of them, you will need to compile and install them.

.To build and run the newer codes, you will need the following:
. a Unix-like system with GNU-make, C and C++ compilers
. popt (command-line parser) library and development files
. readline library (including the header files, libreadline5-dev on Ubuntu)
. Python + (with the numpy extension)
. SWIG
. Tcl/Tk + the BWidget library (to run the GUI program e3console.tcl)

We have been able to get the programs to build on Linux, MacOS-X
(with a recent Xcode development environment) and Cygwin 1.7 (on MS-Windows).

On MS-Windows, install the full kit of Cygwin (Python, X-Windows and all)
and be careful not to have another Python installed outside of Cygwin.
The multiple installations of Python seem not to play well together.

.Some other things that are useful:
. awk
. MetaPost (mpost) or, more recently, InkScape (for looking at and editing svg files)
. GNUplot
. Paraview or MayaVi or VisIt

.To a basic Fedora 16 installation, you should add the following packages:
. bzr
. gcc
. gcc-c++
. m4
. gcc-gfortran
. swig
. python-devel
. readline-devel (for Lua)
. popt-devel

.To a basic Ubuntu 10.04 installation, you should add the following packages and their dependencies:
. bzr
. bzrtools
. g++
. m4
. mpi-default-dev
. gfortran
. swig
. python-dev
. python-numpy
. libreadline-dev
. libpopt-dev
. tk
. bwidget
. gnuplot


Typical build and run procedures
--------------------------------

mbcns2 (2D flow simulation)
~~~~~~~~~~~~~~~~~~~~~~~~~~~
[source,shell]
cd $HOME/cfcfd2/app/mbcns2/build
make
```

```
make install
make clean
cd $HOME/cfcfd2/lib/plot/build
make
make install
make clean

You may need to add the installation directory to your system's
search path and to Lua's search path.
On a recent Linux system, this could be done by adding the lines

[source,shell]
export PATH=${PATH}:${HOME}/cfd_bin
export LUA_PATH=${HOME}/cfd_bin/?.lua
export LUA_CPATH=${HOME}/cfd_bin/?.so

to the .bash_profile file in your home directory.

Then, try out the cone20 example.

[source,shell]
mkdir $HOME/work; cd $HOME/work
mkdir cone20; cd cone20
cp $HOME/cfcfd2/examples/mbcns2/cone20/* .
./cone20_run.sh
./cone20_plot.sh

This should generate a few postscript figures of the starting flow
about a sharp 20-degree cone.

Elmer2
~~~~~~
This 3D-flow code is built similarly.
For running on the blackhole cluster:

[source,shell]
cd $HOME/cfcfd2/app/elmer2/build
module load openmpi/1.2-gnu-4.1
module load swig
module load lua
make TARGET=for_openmpi WITH_MPI=1 install

To run the flow over a cylinder simulation with dissociating nitrogen:

[source,shell]
mkdir $HOME/work/elmer2; cd $HOME/work/elmer2
mkdir tim1; cd tim1
cp $HOME/cfcfd2/examples/elmer2/tim1/* .
./prepare_thermo_files.sh
qsub cyl_run.sh

Eilmer3
~~~~~~~
The new 2D/3D code Eilmer3 is built much more like mbcns2 but
has a new installation directory $HOME/e3bin/.
A typical build procedure (using the default TARGET=for_gnu) might be:

[source,shell]
cd $HOME/cfcfd2/app/eilmer3/build
make install
make clean

Or, if you want the MPI version of the code built as well:

[source,shell]
cd $HOME/cfcfd2/app/eilmer3/build
make TARGET=for_openmpi install
make clean


You may need to add the installation directory to your system's
search path and to Lua's search path.
On a recent Linux system, this could be done by adding the lines
```

```
[source,sh]
export PATH=${PATH}:${HOME}/e3bin
export LUA_PATH=${HOME}/e3bin/?.lua
export LUA_CPATH=${HOME}/e3bin/?.so

to the .bash_profile or .profile file in your home directory.
(Note that it is not necessary to have the LUA_CPATH variable set
unless you want to access the Lua gas module from within
the user-defined (Lua) functions.
You don't need it to run Eilmer3 otherwise.)

For running on the blackhole cluster, also add the following:

[source,sh]
module load openmpi/1.2-gnu-4.1
module load swig

IMPORTANT: Do NOT load lua as was done in the example for Elmer2.

Then, try out the cone20-simple example.

[source,sh]
mkdir $HOME/work; cd $HOME/work; mkdir 2D; cd 2D
mkdir cone20-simple; cd cone20-simple
cp $HOME/cfcfd2/examples/eilmer3/2D/cone20-simple/* .
./cone20_run.sh  # exercise the shared-memory version of the code
       or
./cone20_run_mpi.sh  # exercise the MPI version of the code

This should generate a postscript figure of the drag coefficient history
about a sharp 20-degree cone and also put the VTK data file into the plot/
subdirectory.
It is not really necessary to make all of the subdirectories as shown above,
however, that arrangement reflects the directory tree that PJ uses.
If you want him to come and look at your simulation files when things go wrong,
use the same.
If not, use whatever hierarchy you like.


Other Notes
-----------
.On Xserver for Linux (especially Ubuntu)
If Paraview crashes on exporting a bitmap image, try adding the line

[source,shell]
Option "AIGLX" "false"

to the Section "ServerLayout" in /etc/X11/xorg.conf

.To use Paraview 3.6.1 on Ubuntu 9.04 or later,
it seems that we need to customize the look of the desktop
by turning off the Visual Effects.
This setting can be found in the System->Preferences->Appearance menu.

.To get Paraview Screenshot to behave,
uncheck "Use Offscreen Rendering for Screenshots" button
in the Edit->Settings ("Options") dialog.
You will find the checkbutton under "Render View"->General.
```

B Surviving the Linux Command Line

For running jobs on a Linux machine, it is worth knowing how to get around and do things in the *shell*, which is a command interpreter and programming language. Sobell's text [13] is a good source of information but here are a few notes to get you started.

A basic command is composed of a sequence of words, separated by spaces and has the usual form
`cmd [options] arguments`
where

- `cmd` is the name of the command or utility program that will do the work. Command names on Linux are often terse, 2 or 3 character names.

- `options` are words that are optionally included and are typically preceeded by one or two dashes. These modify the bahaviour of the command, if the default behaviour is not quite what you want.

- `arguments` are the things to work on. If these are file names, you can often use patterns with *wildcard* characters that may match more then one file at a time.

Commands often put their *standard output* to the console. If the amount of text output is overwhelming, it can be *redirected* to a file or *piped* through a paging filter. This latter option is an example of putting multiple command together so that the output from one command becomes the input for another. Once you understand the system, customised commands can be build rather simply in this way.

The following tables summarize a number of commands that you are likely to find useful while using `Eilmer3`.

Logging in and getting out

`ssh` *user*`@`*host*	Connect to computer named *host* as *user*.
Ctrl+d	Quit current session.
`exit`	Quit current session.

Getting help

`man` *cmd-name*	Display the manual page for the named command.
`man` *cmd-name* `| less`	Display the manual page through the paging filter.
`ls --help | less`	Look at the online help provided by the `ls` command.
`man -k` *keyword*	List `man` pages that contain *keyword*.
`apropos` *subject*	List `man` pages on `subject`.

Moving about and looking in your folders

`cd` *dir*	Change to directory *dir*.
`cd`	Change to home directory.
`cd ..`	Change to parent of current directory.
`pwd`	Print current (working) directory.
`pushd` *dir*	Change to new directory *dir*, putting the current directory onto a stack.
`popd`	Go back to the directory at the top of that stack.
`ls -l`	List the files in the current directory, long format.
`ls -a ..`	List the files in the directory above, including all hidden files.
`du -h` *dir*	Report the size of the directory and its subdirectories.
`df -h`	Report the capacities of the file systems and how much is used for each.
`mkdir` *dir*	Make new directory.
`rmdir` *dir*	Remove an empty directory.

Handling files

`cat` *file*	Displays the content of a text file.
`head -n 20` *file-to-show*	Display the first 20 lines of a text file.
`tail -f` *file-to-show*	Show the last few lines of a file and continue to show lines as that file changes.
`grep 'ideal gas' *.py`	Find the string `ideal gas` in all of the Python files in the current directory.
`mv` *src-file dest-file*	Renames the source file to the destination name.
`cp` *src-file dest-file*	Copy the content from the source file to the destination file.
`scp` *src-file user*`@`*host*`:`	Copy the file from the local computer to the home directory of *user* on the remote computer *host*.
`rm -r` *dir*	Remove a directory and all of its contents (recursively).
`gzip` *src-file*	Compresses the file, adding the extension `.gz` to its name.
`tar -zcf` *tarfile dir*	Pack all of the contents of *dir* into the *tarfile*.
`tar -zxf` *tarfile*	Unpack the contents of *tarfile* into the current directory.

Managing processes

`top`	Display information about all running processes. This is very handy for finding out which jobs are taking all of your workstation's CPU cycles and memory.
Ctrl+z	Stops the current command.
`bg`	Resumes a stopped job in the background.
`fg`	Brings most recent job to the foreground.
Ctrl+c	Halts current command.

Command-line editing

On most Linux systems, it seems that you can use the cursor keys to move about within the command line. Delete and backspace also seem to have suitable effect.

Ctrl+u	Erases whole command line.
`!!`	Repeats last command.
`history`	Shows command history.
`!`*n*	Repeats command *n*.

C Just enough Python to be dangerous

When `e3prep.py` is run, the first thing that happens is that a number of Eilmer3-specific modules are loaded and a number of classes are defined to assist with the definition of flow and geometry. The user's input file is then read in and executed by the Python interpreter in the context of these predefined classes and functions. Since the input file has to be valid Python code, it's worth knowing a little about the language itself. We will discuss the features of Python using examples from the periodic shear layer input file on page 133.

Python is a statement-based language where indentation is used to define the block structure of compound statements. One of the implications of this significant whitespace is that the first statement in the user's input file must start right at the beginning of the line. That is, it must not be indented. The first couple of lines in the periodic shear layer input file are:

```
# psl.py
gdata.title = "Periodic shear layer"
```

The comment line, starting with the sharp (or hash) character is ignored and the first statement assigns a string literal to the title.

Single statements, such as assignment statements, may extend over several lines if they are continued by one of:

- a backslash (\) at the end of each incomplete line;
- an open left parenthesis, bracket or brace without the corresponding closing parenthesis, bracket or brace; or
- an open triple quote that has indicated the start of a multiline string.

The second of these is quite commonly seen in the example files because many of the function calls and object constructors have a lot of arguments, some of which may be quite complex in themselves. The following assignment statement, from near the end of the periodic shear layer input file, calls the `SuperBlock2D` object constructor and then binds the resultant object to the name `superblk`.

```
superblk = SuperBlock2D( psurf=domain, nni=nnx, nnj=nny,
                         bc_list=[SlipWallBC(),]*4,
                         fill_condition=initial_gas,
                         nbi=nbi, nbj=nbj, label="blk")
```

On the selection of names to bind to returned data objects, the usual rules apply. Start the name with a letter from the alphabet and follow it with any number of letters, digits or underscores. Don't use any of the following reserved words for your own names:

```
and       del      from    None    try
as        elif     global  not     while
assert    else     if      or      with
break     except   import  pass    yield
class     exec     in      print
continue  finally  is      raise
def       for      lambda  return
```

And, if you want to see the list of names that are predefined for the environment in which your input file is interpreted, start `e3prep.py` with the `--show-names` option. There are conventions that leading and trailing underscores are reserved for system names and that names starting with an uppercase letter are class names.

Control flow statements such are implemented as compound statements. These start with an opening clause at the current indentation level. This clause will start with a keyword, such as `if`, `while`, or `for`, and end with a colon. The body of the compound statement will typically start on the following line, indented one level. All statements at that level of indentation or more form part of the body of that compound statement. There may be nested compounded statements and each level of indentation will be 4 spaces, by convention.

The definition of a function is itself a compound statement and an example can be seen in the periodic shear layer example (page 133) where the initial state of the gas is defined in the function `initial_gas(x, y, z))` in the user's input file. Collections of functions are typically available as modules in Python. These modules, or items from the modules may be "imported". By default, Python does not load a lot of modules so you will typically have to import math functions, for example.

As well as the simple numerical data types of integers and floats, there are strings and more complex, structured data types built into the language. These include tuples, lists and dictionaries. `e3prep.py` also makes use of `numpy` arrays.

You will make use of lists when defining collections of mass fractions and boundary conditions, for example. A list literal is denoted by square brackets, with items separated by commas. Lists are ordered collections of items that may be indexed, starting from zero. Negative index values count backward from the end of the list. The `for` loop is a convenient way of working through all the elements of a list. In the periodic shear layer example above, the boundary conditions are specified as a list of 4 `SlipWallBC` objects. The following code works through the list of blocks that had been returned by the SuperBlock2D constructor and makes the appropriate connections for a periodic domain.

```
for j in range(nbj):
    connect_blocks_2D(superblk.blks[-1][j], EAST, superblk.blks[0][j], WEST)
```

Here, the call to the function `range` returns a list of integer values starting with 0, going up to but not including the value bound to `nbj`.

Dictionaries are collections of named objects. They are a convenient way of setting species mass fractions, especially for a gas model that has many species. You may typically only have only one or a couple of species present in and particular inflow or initial gas condition as, for example, the literal dictionary {`'He':0.1, 'air':0.9`} is used to set the mass fraction of helium to 0.1 and the mass fraction of air to 0.9. Use of the dictionary has the benefit of making the input script somewhat self-documenting and you don't have to remember the order in which the gas species were defined in the call to the `select_gas_model` function, earlier in the input file.

More specialized data objects can be defined via classes, and `e3prep.py` does exactly that. The name `gdata` is bound to an instance of the `GlobalData` class and contains many attributes that set the configuration of the flow solver. The user input file will use the already defined gdata object but will typically create new instances of objects such as `Node` and `SuperBlock2D`. It is often convenient to bind the reference returned to the newly created object to a name in the input script so that it can be conveniently referenced in later statements. In the periodic shear layer case, the `Node` objects are bound to names that are then used to construct the `Line` objects that are, in turn, used to define the rectangular flow domain.

D Make your own debugging cube

E Notes on running MPI jobs on cluster computers

The details of running simulations on any cluster computer will be specific to the local configuration. The Blackhole cluster computer belongs to the Hypersonics Group at the University of Queensland and is a SUN Rack computer consisting of about 66 nodes with dual AMD Opteron processors. The Barrine cluster is run by the High-Performance Computing Unit at The University of Queensland and is a much larger machine, with a little over 3000 cores, running SUSE Enterprise Linux.

E.1 The Blackhole (SUN Rack) cluster

- Set the environment up by customizing your `.bash_profile` file.

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs
module load openmpi/1.2-gnu-4.1
module load swig
module load python/2.5.2
export PATH=${PATH}:${HOME}/bin:${HOME}/e3bin
export LUA_PATH=${HOME}/e3bin/?.lua
export LUA_CPATH=${HOME}/e3bin/?.so
unset USERNAME
```

- Get the source code tree onto Blackhole using the `rsync` program or whatever you find convenient to use.
  `rsync -av triton:cfcfd2 .`

- To compile the MPI-version of the code, use the command:
  `make TARGET=for_openmpi install`
  from the `cfcfd2/app/eilmer3/build/` directory.

- Optionally, clean up after the build.
  `make clean`

- To submit a job to Sun Grid Engine (SGE), which is the batch queue system on Blackhole, use the command:
  `qsub` *script_name.sh*

- An example of a shell script prepared for running on the Blackhole cluster.

```
#!/bin/bash
# run_simulation.sh
#$ -S /bin/bash
#$ -N FiniteCyl
#$ -pe orte 4
#$ -cwd
#$ -V

job=cyl
np=4

echo "Start time: "; date
mpirun -np $np e3mpi.exe --job=$job --run
# e3shared.exe --job=$job --run
echo "Finish time: "; date
```

- When running a job on the through the batch queue system, the job is identified by a number *nnnnn*. Output that would have gone to the console (as standard-output) for an interactive job will be collected in the file *job.onnnnn*. Error messages will accumulate in the file *job.ennnnn*.

- To see how the calculation is progressing by following the content of the output file, use the command
  `tail -f` *job.onnnnn*

- To put a hold on a job while waiting for another to finish, use the command
  `qsub --hold_jid` *nnnnn script_name.sh*

E.2 The barrine.hpcu.uq.edu.au SGI cluster

- Set up your environment by adding the following lines to your `.bashrc` file.
  ```
  module load python
  module load intel-cc-11
  module load intel-mpi/3.2.2.006
  export PATH=${PATH}:${HOME}/e3bin
  export LUA_PATH=${HOME}/e3bin/?.lua
  export LUA_CPATH=${HOME}/e3bin/?.so
  ```
  Note that we load a specific version of the MPI module.

- Get yourself an interactive shell on a compute node so that you don't hammer the login node while compiling. You won't make friends if you keep the login node excessively busy.
  `qsub -I -A uq-Jacobs`

- To compile the MPI-version of the code, use the command:
  `make TARGET=for_intel_mpi install`
  from the `cfcfd2/app/eilmer3/build/` directory.

- Optionally, clean up after the build.
  `make clean`

- To submit a job to PBS-Pro, which is the batch queue system on barrine, use the command:
  `qsub` *script_name.sh*

- An example of a shell script prepared for running on the Barrine cluster.

```
#!/bin/bash -l
#PBS -S /bin/bash
#PBS -N lehr
#PBS -q workq
#PBS -l select=3:ncpus=8:NodeType=medium:mpiprocs=8 -A uq-Jacobs
#PBS -l walltime=6:00:00
# Incantations to get bash to behave and the Intel MPI bits in place.
. /usr/share/modules/init/bash
module load intel-mpi/3.2.2.006
echo "Where are my nodes?"
echo $PBS_NODEFILE
cat $PBS_NODEFILE
echo "-------------------------------------------"
echo "Begin MPI job..."
date
cd $PBS_O_WORKDIR
mpirun -np 24 $HOME/e3bin/e3mpi.exe --job=lehr --run --max-wall-clock=20000 > LOGFILE
echo "End MPI job."
date
# As we leave the job, make sure that we leave no processes behind.
# (The following incantation is from Gerald Hartig.)
for i in $(cat $PBS_NODEFILE | grep -v `hostname` | sort -u); do
    ssh $i pkill -u `whoami`
done
killall -u `whoami` e3mpi.exe
```

  Here, we ask for 3 nodes with 8 cores each for a set of 24 MPI tasks. The medium nodes have 8 cores available, and we ask for all of them so that we are reasonably sure that our job will not be in competition with another job on the same nodes. Note the `-A` accounting option. You will have to use an appropriate group name and you can determine which groups you are part of with the `groups` command. Unlike SGE on Blackhole, we seem to need to change to the working directory before running the simulation code. Finally, we have redirected the standard output from the main simulation to the file `LOGFILE` so that we can monitor progress with the command
  `tail -f LOGFILE`

F cfpylib modules

There are a number of modules that are useful for the definition of flow simulations but are not part of the Eilmer code. These are available in a separate Python library that may be imported into the user's job definition script.

F.1 Billig shock shape correlation

```
## \file billig.py
## \brief Billig's correlations for hypersonic shock-wave shapes.
##
## These are a direct implementation of equations 5.36, 5.37 and 5.38
## from J.D. Anderson's text Hypersonic and High Temperature Gas Dynamics
##
## \version 19-June-2005 PJ

from math import exp, sqrt, pow, tan
from ideal_gas_flow import beta_obl

def delta_over_R(M, axi):
    """
    Calculates the normalised stand-off distance.
    """
    if axi == 1:
        # Spherical nose
        d_R = 0.143 * exp(3.24/(M*M))
    else:
        # Cylindrical nose
        d_R = 0.386 * exp(4.67/(M*M))
    return d_R


def Rc_over_R(M, axi):
    """
    Calculates the normalised radius of curvature of the shock.
    """
    if axi == 1:
        # Spherical nose
        Rc_R = 1.143 * exp(0.54/pow(M-1, 1.2))
    else:
        # Cylindrical nose
        Rc_R = 1.386 * exp(1.8/pow(M-1, 0.75))
    return Rc_R


def x_from_y(y, M, theta=0.0, axi=0, R_nose=1.0):
    """
    Determine the x-coordinate of a point on the shock wave.

    Input:
    y        : y-coordinate of the point on the shock wave
    M        : free-stream Mach number
    theta    : angle (in radians wrt free-stream direction)
               of the downstream surface
    axi == 0 : cylinder-wedge
        == 1 : sphere-cone
    R_nose   : radius of the forebody (either cylinder or sphere)

    It is assumed that, for the ideal gas, gamma=1.4.
    That's the only value relevant to the data used for
    Billig's correlations.
    """
    Rc = R_nose * Rc_over_R(M, axi)
    d = R_nose * delta_over_R(M, axi)
    beta = beta_obl(M, theta)
    tan_beta = tan(beta)
```

```
    cot_beta = 1.0/tan_beta
    x = R_nose + d - Rc * cot_beta**2 * (sqrt(1 + (y*tan_beta/Rc)**2) - 1)
    return x

def y_from_x(x, M, theta=0.0, axi=0, R_nose=1.0):
    """
    Determine the y-coordinate of a point on the shock wave.

    Input:
    x        : x-coordinate of the point on the shock wave
    M        : free-stream Mach number
    theta    : angle (in radians wrt free-stream direction)
               of the downstream surface
    axi == 0 : cylinder-wedge
        == 1 : sphere-cone
    R_nose   : radius of the forebody (either cylinder or sphere)

    It is assumed that, for the ideal gas, gamma=1.4.
    That's the only value relevant to the data used for
    Billig's correlations.
    """
    Rc = R_nose * Rc_over_R(M, axi)
    d = R_nose * delta_over_R(M, axi)
    beta = beta_obl(M, theta)
    tan_beta = tan(beta)
    cot_beta = 1.0/tan_beta
    tmpA = (x - R_nose - d)/(-Rc * cot_beta**2) + 1
    y = sqrt( ((tmpA**2 - 1) * Rc**2) / (tan_beta**2) )
    return y

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

if __name__ == '__main__':
    print "Begin demo of Billig's correlations."
    print "Compare with Fig 5.31 in Anderson's text."
    M_inf = 4.0
    for y in [0.0, 0.5, 1.0, 2.0]:
        print "x=", x_from_y(y, M_inf, 0.0, 1), "y=", y
    print "Done."
```

G Gas models: specification by configuration file

As explained in Section 5, most users can select a gas model by a call to `select_gas_model` using the `model` and `species` keyword arguments. For more advanced uses of the gas models, a configuration file created directly by the user is required. This section discusses the creation of that file.

The configuration file has a Lua-style format, meaning that the statements and expressions in the file must conform to proper Lua syntax. Do not be concerned with the need to learn Lua in order to build a configuration file: nearly all of the statements you will require can be taken from the following examples. Besides, Lua has been designed from the outset as an embedded language for configuration purposes, and, with that aim in mind, it has a simple syntax suitable for non-programmers [14]. The following subsections explain the requirements of configuration files for specific gas models.

G.1 User-defined gas model

The user may define her own gas model by providing callable functions that implement the desired behaviour. There is a minimal (read mandatory) set of functions that the user must provide in the configuration file. There are also some optional functions. When the optional functions are not provided, the underlying C++ code will provide a default implementation. For example, if the user does not provide a function `dT_dp_const_rho` then the default implementation will use a numerical differentiation technique to compute this value when required. In addition to providing some mandatory functions, the user's configuration file needs to set three global variables:

- `model`: set as `'user-defined'`
- `nsp`: the number of species in the gas model
- `nmodes`: the number of thermal modes in the gas model

Each of the functions which the user specifies has certain rules that they must conform to: they must accept a distinct set of arguments in the correct order; and they must return the expected number of results of the correct type and in correct order.[17] The job of the function will be to compute the required results based on the input arguments. The set of functions recognised by a 'user-defined' gas model, along with their arguments lists and return value lists, are given in Table 2. The mandatory functions are listed first, followed by the optional functions. A majority of the functions accept a `gas_data` table

[17] This statement about received function arguments is not strictly true. If the user is familiar with how Lua treats missing and or extra arguments, then (s)he will be aware that the implementation may still function even if not all arguments are present. In practice and for ease of understanding the code, it is best to stick to the documented function signatures.

as an argument and also return a `gas_data` table. As a pre-condition to the function calls, certain data members in the `gas_data` table may be assumed to be present and correct. As a post-condition to the function calls, certain data members in the `gas_data` table should be correctly set upon return. These pre- and post-conditions for the `gas_data` table are also shown in Table 2. Finally, Table 3 describes the entry in a `gas_data` table.

Table 2: User-defined functions for specification of gas model behaviour

Function	Arguments	Return values	Description
		Mandatory functions	
`eval_thermo_state_rhoe`	**Q**: `gas_data` *pre-conditions*: `rho`, `e` and `massf` are set	**Q**: `gas_data` *post-conditions*: set all other thermodynamic variables	Given density, the internal energy (vector) and mass fractions, compute the rest of the thermodynamic state.
`eval_transport_properties`	**Q**: `gas_data` *pre-conditions*: thermodynamic state is up-to-date	**Q**: `gas_data` *post-conditions*: transport data set, i.e., `mu` and `k`	Given an up-to-date thermodynamic state, compute the transport properties: viscosity and thermal conductivity (in all thermal modes as appropriate).
`eval_diffusion_coefficients`	**Q**: `gas_data` *pre-conditions*: thermodynamic state is up-to-date	**Q**: `gas_data` *post-conditions*: all binary diffusion coefficients set, `D_AB`	Given an up-to-date thermodynamic state, compute the binary diffusion coefficients for all species interaction pairs in the mixture.
`molecular_weight`	**isp**: species index, `integer`	**MW**: molecular weight of species *isp*, `float`	Returns the molecular weight, g/-mol, for species *isp*.
		Optional functions	
`eval_thermo_state_pT`	**Q**: `gas_data` *pre-conditions*: `p`, `T` and `massf` are set	**Q**: `gas_data` *post-conditions*: set all other thermodynamic variables	Given pressure, temperature(s), and mass fractions, compute the rest of the thermodynamic state.
`eval_thermo_state_rhoT`	**Q**: `gas_data` *pre-conditions*: `rho`, `T` and `massf` are set	**Q**: `gas_data` *post-conditions*: set all other thermodynamic variables	Given density, temperature(s), and mass fractions, compute the rest of the thermodynamic state.

Function	Arguments	Return values	Description
`eval_thermo_state_rhop`	**Q**: `gas_data` *pre-conditions*: `rho`, `p` and `massf` are set	**Q**: `gas_data` *post-conditions*: set all other thermodynamic variables	Given density, pressure, and mass fractions, compute the rest of the thermodynamic state.
`encode_conserved_energy`	**Q**: `gas_data` *pre-conditions*: `rho`, `e` and `massf` are set	**rhoe**: table of `float` values *post-conditions*: table has dimension `nmodes` and stores the conserved energy quantities	Given the primary variables, encode the conserved energy quantities.
`decode_conserved_energy`	**Q**: `gas_data` **rhoe**: table of `float` values *pre-conditions*: `rho` and `massf` are set	**Q**: `gas_data` *post-conditions*: the vector member `e` is up-to-date (decoded)	Given density and the vector of conserved energies, set the specific energies.
`update_massf_mode`	**Q**: `gas_data` *pre-condition*: `massf` are set	**Q**: `gas_data` *post-condition*: `massf_mode` are set	Given the mass fractions, update the vector `massf_mode` which stores the mass fraction associated with each thermal mode.
`dTdp_const_rho`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dTdp**: `float`	Return $\left(\frac{\partial T}{\partial p}\right)_\rho$.
`dTdrho_const_p`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dTdrho**: `float`	Return $\left(\frac{\partial T}{\partial \rho}\right)_p$.
`dpdrho_const_T`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dpdrho**: `float`	Return $\left(\frac{\partial p}{\partial \rho}\right)_T$.

Function	Arguments	Return values	Description
`dedT_const_v`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dedT**: `float`	Return $\left(\frac{\partial e}{\partial T}\right)_v$: C_v, the specific heat capacity at constant volume.
`dhdT_const_p`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dhdT**: `float`	Return $\left(\frac{\partial h}{\partial T}\right)_p$: C_p, the specific heat capacity at constant volume.
`gas_constant`	**Q**: `gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**R**: `float`	Return R, the specific gas constant.

Table 3: Description of fields in `gas_data` table

Field	Type	Description
	Thermodynamic properties	
`rho`	*float*	density, kg/m^3
`p`	*float*	pressure, Pa
`a`	*float*	sound speed, m/s
`e`	vector of *floats*	specific internal energies, J/kg
`T`	vector of *floats*	temperatures, K
	Transport properties	
`mu`	*float*	kinematic viscosity, Pa.s
`k`	vector of *floats*	thermal conductivities (for each mode), W/(m.K)
`D_AB`	matrix of *floats*	binary diffusion coefficients, m^2/s
	Composition	
`massf`	vector of *floats*	species mass fractions
`massf_mode`	vector of *floats*	mass fraction associated with specific thermal modes

G.1.1 An example minimal user-defined gas model

The following code listing shows the minimum requirements to specify a user-defined gas model. This is a concrete example to complement the abstract discussion presented previously. This particular example implements ideal air. This is a trivial example for the sake of demonstration, and one would not use the slow user-defined gas model for ideal air when an internally implemented model already exists. The intended use for the user-defined gas models is for more exotic gases or rapid prototyping of a new gas model.

```
-- Author: Rowan J. Gollan
-- Date: 08-Jul-2008
--
-- User-defined gas model
-- ----------------------
-- This is an example model which
-- implements ideal gas behaviour
-- for a single component gas.
--
-- This is a minimal implementation:
-- numerical techniques are used to
-- give the rest of the functionality.

-- Mandatory, set nsp and nmodes
model = 'user-defined'
nsp = 1
nmodes = 1

-- Local parameters for model
local R0 = 8.31451
local R = 287.1
local gamma = 1.4
local C_v = R / (gamma - 1)
local C_p = R + C_v
```

```
local mu0 = 1.716e-5
local T0_v = 273.0
local S_v = 111.0

local k0 = 0.0241
local T0_k = 273.0
local S_k = 194.0

-- Local helper functions
local sqrt, pow = math.sqrt, math.pow
local function sound_speed(gamma, R, T)
   return sqrt(gamma*R*T)
end

local function Sutherland_viscosity(T)
   return mu0 * pow(T/T0_v, 3/2) * (T0_v + S_v)/(T + S_v)
end

local function Sutherland_thermal_conductivity(T)
   return k0 * pow(T/T0_k, 3/2) * (T0_k + S_k)/(T + S_k)
end

-- Mandatory function:
function eval_thermo_state_rhoe(Q)
   -- Assume rho and e[1] are given, compute the
   -- remaining thermodynamic variables.
   -- Remember: we need to access the temperature
   -- and energy as the first value in an array
   -- of possible energies/temperatures.
   Q.T[1] = Q.e[1]/C_v
   Q.p = Q.rho*R*Q.T[1]
   Q.a = sound_speed(gamma, R, Q.T[1])
   -- Pass back the updated table
   return Q
end

function eval_transport_coefficients(Q)
   -- Assume that all pertinent values in Q are
   -- at the correct state.  In this particular
   -- model, viscosity and thermal conductivity
   -- are only dependent on temperature, ie. Q.T[1]
   Q.mu = Sutherland_viscosity(Q.T[1])
   Q.k[1] = Sutherland_thermal_conductivity(Q.T[1])
   return Q
end

function molecular_weight(isp)
   -- PJ added July 2010
   return R0/R
end

function eval_diffusion_coeficients(Q)
   -- PJ added July 2010
   Q.D_AB[1][1] = 0.0
   return Q
end
```

G.2 Equilibrium gas based on a look-up table

The properties of a gas mixture in thermochemical equilibrium can be computed using the CEA program [15, 16]. By pre-computing the properties for a range of densities and internal energies, a look-up table can be created. The use of a look-up table is much more efficient to use than calling out to the CEA program during simulation execution; there

is some small sacrifice in accuracy using the look-up table.

G.2.1 Selecting a look-up table for the gas model

A number of pre-built look-up tables are provided as par the code collection. After installing Elmer3, the pre-built look-up tables are provided in `$HOME/e3bin/cea-cases/`. A list of these tables and a description of what gas mixture they model is given in Table 4.

Table 4: Description of pre-built look-up tables distributed with Elmer3

Pre-built table	Description
`cea-lut-air.lua.gz`	Equilibrium air with ionisation. Useful for Earth reentry problems.
`cea-lua-CO2.lua.gz`	Pure carbon dioxide in equilibrium with ionisation.
`cea-lua-Jupiter-like.lua.gz`	A H_2-Ne mixture used to simulate the Jovian H_2-He atmosphere in expansion tube work.
`cea-lut-Kr.lua.gz`	Pure Kryton with ionisation allowed.
`cea-lut-Mars-basic.lua.gz`	A basic Martian atmosphere, without trace species. The included species are CO_2 (97% by weight) and N_2 (3% by weight). No ionisation is considered.
`cea-lut-Mars-trace.lua.gz`	A Martian atmosphere which includes the trace species O_2 and Ar. No ionisation is considered.
`cea-lut-Mars-trace-with-ions.lua.gz`	A Martian atmosphere incluing trace species and ionisation.
`cea-lut-N2.lua.gz`	Pure nitrogen in equilibrium with ionisation.
`cea-lut-Titan-like-no-ions.lua.gz`	A Titan-like atmosphere (N_2 and CH_4, no trace species). No ionisation is considered.
`cea-lut-Titan-like-with-ions.lua.gz`	A Titan-like atmosphere which includes ionisation.

The steps to using a look-up table in your simulation are:

1. Copy a pre-built table to your working directory.
2. Specify the name of thie pre-built table in your call to `select_gas_model` in your simulation setup script.

As an example, suppose that we wish to run a simulation with CO_2 in equilibrium. Then as per above the sequence of steps is (assuming your are in your working directory):

1. `cp $HOME/e3bin/cea-cases/cea-lut-CO2.lua.gz .`
2. Add the following function call to your script:
   `select_gas_model(fname='cea-lua-CO2.lua.gz')`

G.2.2 Building your own look-up table

Of course, you might have a gas mixture you wish to simulate that is not listed in Table 4. The tool `build-cea-lut` is provided as part of the code collection to aid in building a look-up table of the appropriate format. You will need access to the `cea2` program [15, 16], and have that setup in your working area [18].

The first step is to build a "template" file. This file specifies the composition of the gas model and some other options appropriate to the modelling. Here is an example template file to build a look-up table for a Mars-like atmosphere with no ionisation but includes trace species.

```
fname = 'cea-lut-Mars-trace.lua'
units = 'massf'
reactants = {CO2=0.9668,
             N2=0.0174,
             O2=0.0011,
             Ar=0.0147}

with_ions = false

only_list = "Ar C C2 CN CO CO2 N N2 NO O O2"
```

The list of expected parameters are given in Table 5

For more examples, the template files for all of the pre-built tables listed in Table 4 are available in `$HOME/e3bin/cea-cases/`.

Once you have created a template file, the `build-cea-lut` tool can be called. Supposing your template file is called `Venus-atmosphere.lua`, then issue the following command in your working directory:

```
> build-cea-lut --user-defined=Venus-atmosphere.lua
```

You may use the `--extrapolate` flag to fill in the edges of the look-up table by linear interpolation where no data is available, however, this option is not presently recommended. If this flag is omitted, a smaller table is built that does not extrapolate at the edges. Note that this flag only controls extrapolation during the *construction* of the table; in the actual use of the table during *simulation*, the values are always extrapolated if they exceed the limits of the table. Upon succesful execution of the `build-cea-lut`, you will have a compressed (gzipped) file in your working directory. This file (not the template file) can be used to select an equilibrium gas in the same manner as using a pre-built table, as was discussed in the preceding section.

[18] By setup, I mean that the `thermo.inp` and `trans.inp` files have been processed and the corresponding `.lib` files are available in the working directory. Also, the program `cea2` needs to be available as an executable in your `$PATH`.

Table 5: Description of parameters required to build a template file for `build-cea-lut`

Parameter	Type	Description
`fname`	*string*	A name for the look-up table. Usually just specifed as `some_name.lua`. The program will add a trailing `.gz` when it zips up the look-up table.
`units`	*string*	This denotes the units for specifying the relative amount of reactants. The choices are: • `'moles'` *(default if not specified)* to specify by molar composition • `'massf'` to specify by mass (weight) composition
`reactants`	*table*	This table lists the reactants in the mixture. The keys are the species names as recognised by CEA and the values are the composition.
`with_ions`	*boolean*	If true, allow ionisation (by specifying `'ions'` in the CEA input files. If false, ionisation is not allowed *(default)*.
`only_list`	*string*	This string restricts the allowable species in gas mixture computation. The format for the string is a space separated list of species. This item is optional: if omitted, the CEA program is free to choose which species are considered according to its own internal algorithm.

H Chemical reactions: specification by configuration file

The chemical reactions which may take place in a reacting flow simulation are described in a Lua input file. This input file, prepared by the user, is read directly by the main simulation code at run time. There is no pre-processing step for this input file. As the input file is Lua-based, the user has access to the full extent of the Lua scripting language when preparing her files. Do not be concerned if you do not know the Lua syntax; the instructions and examples given here should be ample to get you started building reaction schemes.[19]

Let's proceed by looking at an example input file and discussing the keywords and syntax. Listed here is an input file which describes the simple thermal dissociation of nitrogen. There are only two participating species, N_2 and N, and only two reactions.

```
reaction{
   'N2 + N2 <=> N + N + N2',
   fr={'Arrhenius', A=7.0e21,  n=-1.6, T_a=113200.0},
   br={'Arrhenius', A=1.09e16, n=-0.5, T_a=0.0}
}

reaction{
   'N2 + N <=> N + N + N',
   fr={'Arrhenius', A=3.0e22,  n=-1.6, T_a=113200.0},
   br={'Arrhenius', A=2.32e21, n=-1.5, T_a=0.0}
}
```

The first reaction is the dissocation of N_2 by collision with other N_2 molecules. The forward reaction rate coefficient is computed with a generalised Arrhenius model, and the parameters for that model are specified. Similarly the backward reaction rate coefficient is computed using the Arrhenius expression.

More generally, each reaction is specified within a `reaction` table. The table is delimited by the opening and closing braces ({ }). The first entry in the table is always a string. That string is the chemical equation for the reaction. The remaining items in the table are denoted by key-value pairs (of the form `key=val`), and may appear in any order. Each item in the table is separated by a comma.[20] This example file contained two `reaction` tables, hence two reactions are treated in the reaction scheme.

Some final notes before discussing the input file in further depth. There is no explicit mention of the participating species in the reaction file. The participating species are taken from the species that are present in the gas model file for the same flow simulation.

[19] If you are worried about needing to "learn Lua" just to get started, then don't be. First, you may just look at this as an input format for the chemistry, and forget that it has anything to do with Lua altogether. Second, Lua was designed with non-programmers in mind and so it uses a simple syntax, specifically so that those non-programmers could quickly use Lua as a configuration language.

[20] Lua also permits the use of semi-colons instead of commas to delimit table entries.

In other words, if you list species in the reaction scheme that are not present in the gas model, then you will get an error message.

H.1 Overview of input file format

By leveraging Lua as the input data description language, the input file is almost self-describing, in my opinion. This provides an excellent record of what modelling was used when you performed a simulation. A valid reaction input file will conform to the following rules.

1. Any legal Lua code is acceptable, but you must not rename the following the predefined functions:
   - `reaction`
   - `remove_reactions_by_label`
   - `remove_reactions_by_number`
   - `select_reactions_by_label`
   - `select_reactions_by_number`
2. Reactions are declared in `reaction` tables.
3. Comments in the file begin when two dashes (`--`) are encountered and proceed to the end of the line. (This is a repetition of Item 1 in that comments are legal Lua code.)

As the the reactions are listed, they are numbered internally beginning from 1. In some cases, it is convenient to list all reactions in a scheme but then only use some of the reactions. This is quite common if you wish to use a reduced mechanism or if you believe that one of the species is inert at your flow conditions of interest, and so would want to remove all reactions involving the transformation of that species. Two convenience functions are provided so that you do not have to hack into your input file to remove the unwanted reactions:

- `remove_reactions_by_label`
- `remove_reactions_by_number`

Both functions will take a single item or an array of items. An array is a special form of Lua table which is bracketed with braces ({ }). The first function accepts strings which correspond to the labels of reactions. The labelling of reactions is explained in the next section. The second function accepts integers which correspond to the internal numbering. The convenience functions must be called *after* the declaration of the associated reactions.

Typically, the user would place the calls to these functions at the end of his input file. Two examples follow.

```
remove_reactions_by_label({'r3', 'r5'})
```

This call would remove the reactions labelled `'r3'` and `'r5'`
from the list of participating reactions.

```
remove_reactions_by_number(13)
```

In this call, the 13th listed reaction is removed from the list (because we all know that 13 is unlucky, right?)[21]

Similarly, there are two complementary convenience functions that allow for the selection of only certain reactions from the full set:

- `select_reactions_by_label`
- `select_reactions_by_number`

They work in reverse to the `remove` functions: these functions will only select those reactions listed in their arguments for inclusion in the chemistry scheme.

Note, it is not advisable to mix and match the use of the `remove` and `select` functions in the one reaction script. The behaviour is untested. Now on to the details of the `reaction` table.

H.2 Details of the `reaction` table

The `reaction` table accepts a number of items; some are mandatory, most are not. The full list of items is shown here, and each item is described below.

```
reaction{
   'equation string',
   fr={...},
   br={...},
   ec='model name',
   efficiencies={...},
   label='r1'
}
```

[21] Actually, unlike the Americans and their buildings, you don't get rid of 13 that easily. If you have more than 13 reactions, the higher numbered reactions will shuffle up one spot so that the numbering remains continuous from 1. This all happens internally.

`'equation string'` *(mandatory)*

As mentioned earlier, this string must appear first in the table and has no key associated with it. This string represents the reaction equation. As an example, dissociation of nitrogen may be written as

```
'N2 + N2 <=> N + N + N2'
```

If the reaction involves a collision with a general third body, then this is strictly denoted as species `'M'`. For example, the formation of hydroperoxyl from oxygen and monatomic hydrogen requires the presence of a third body. This reaction is written as

```
'H + O2 + M <=> HO2 + M'.
```

The reactants and products are delimited by direction arrows. The use of `<=>` indicates that the reaction proceeds in both directions, while `=>` will mean that the reaction proceeds in the forward direction only (no backward rate of conversion will be computed).

`fr` *(optional, if* ***`br`*** *supplied)*

The `fr` key is used to specify the forward reaction rate coefficient and expects a table value. The format of the table is a string naming the model followed by key-value pairs giving the parameters for the model. The currently implemented reaction rate coefficient models are listed in Table 6, along with their input format.

`br` *(optional, if* ***`fr`*** *supplied)*

The `br` key is used to specify the backward reaction rate coefficient. It is used in the same manner as the forward rate key (`fr`).

`ec` *(optional)*

The `ec` key is used to specify the model for computing the equilibrium constant. It accepts a string naming the model. Currently, there is only one model implemented, `'from thermo'`, which calculates the equilibrium constant based on thermodynamic principles. For reversible reactions, if only one of `fr` or `br` is specified, then the use of the equilibrium constant is assumed and does not need to be declared.

`efficiencies` *(optional)*

If declaring a third body reaction, all species in the mixture are assumed to react with an efficiency of 1.0. The `efficiencies` key accepts a list of *exceptions* to that assumption of a value of 1.0. The list contains the key-value pairs of the type `species=efficiency_value`. For example, to denote that N_2 has a 6-fold efficiency value and O_2 a value of 3.5, the list would be:

Table 6: Reaction rate coefficient models input format

Model	Format
generalised Arrhenius $k = AT^n \exp(-T_a/T)$	{`'Arrhenius', A=..., n=..., T_a=...`} • `'Arrhenius'` appears first to name the model • `A=...` is the pre-exponential coefficient given in 'cgs' units (because they are most common in the chemistry reaction rate literature). • `n=...` is the non-dimensional power for T • `T_a=...` is the activation temperature in Kelvin. Do not get confused by the appearance of a negative sign in the formula; you are required to input the activation barrier temperature which in the majority of cases is a *positive* value. On occasion, the activation temperature is negative. This will be given in the reaction scheme you are following.

```
efficiencies={N2=6.0, O2=3.5}
```

Remember that all species are assumed to have a value of 1.0 unless otherwise noted in the list. If you have a species that does *not* participate as a third body, then be sure to set its efficiency value to 0.0 (e.g. `H=0.0`).

`label` *(optional)*

The `label` accepts a string allowing the user to give the reaction a name. This is useful if one wishes to later remove certain reactions based on their labels using the `remove_reactions_by_label` convenience function.

Note that if you specify all three of `fr`, `br` and `ec`, you have overspecified the modelling of reaction rate coefficients. In this case, no error is given. Instead, the `ec` model is ignored.

H.3 Extra control of the chemistry scheme

There are a number of details to do with solving the finite-rate chemistry problem that are set by default for the user. However, all of these parameters may be controlled by the user by setting values in the input file.

Let's first describe the `scheme` table. The user may set values in this table that pertain to the chemistry scheme as a whole. In the example input snippet below, the lower temperature limit is set to 300 K and the upper limit is set to 50 000 K. This value is used to control the temperature limits at which reactions are "turned off", that is, a chemistry increment is not computed in cells where the temperature is lower or higher thatn these limits, as the case may be. Note that these values are set as part of a subtable, `temperature_limits`.

```
scheme{
   temperature_limits = {lower = 300.0,
                         upper = 50000.0}
}
```

The `scheme` table is currently only a container for the temperature limits. In future implementations it is planned to contain other options. For example, the `scheme` table will contain options for setting parameters related to multi-temperature chemistry schemes.

The other table presently offered to the user is the `ode_solver` table which, unsuprisingly, contains parameters that allow the user to select details about the ODE method used to solve the chemistry system. Let's look at an example of its use.

```
ode_solver{
   step_routine = 'qss',
   max_step_attempts = 4,
   max_increase_factor = 1.15,
   max_decrease_factor = 0.01,
   decrease_factor = 0.333
}
```

The various parameters in the `ode_solver` table are described in the list below. As an aside, the values shown in the example above are actually the default values used when the user does not specify any `ode_solver` table.

`step_routine`

This string sets the ODE stepping method. The available stepping methods are:

`'qss'`	:	Mott's α-QSS method [17]
`'rkf'`	:	Runge-Kutta-Fehlberg method [18]
`'euler'`	:	Euler stepping

`max_step_attempts`

This integer value sets the maximum number of retry attempts the stepping routine will attempt on a single step if the ODE system indicates failure.

`max_increase_factor`

This value is used to control the maximum factor the chemistry timestep will increase when the step is successful. The `'qss'` and `'rkf'` methods can suggest their own timestep increase. However, the increase will be calculated as `MIN(suggestion, max_increase_factor)`.

`max_decrease_factor`

This value is used to control the maximum amount of decrease or reduction in timestep that occurs. It is computed as `MAX(suggestion, max_decrease_factor)`.

`decrease_factor`

Occasionally, the step fails and yet the step routines suggests using a *larger* timestep for the retry. In this case, the `decrease_factor` is used to reduce the timestep size for the retry attempt.

I User-defined functions for run-time customization

User-defined functions (UDFs) are callable for both specialized boundary conditions and for the addition of custom source terms.[22]

I.1 Boundary conditions

When the user's (Python) input script calls up a `UserDefinedBC()` boundary condition, a Lua file is specified. This file is run at the time of boundary-condition instantiation and it needs to define the Lua functions `ghost_cell(args)` and `interface(args)` at a minumum. These functions are later called, every time the boundary condition is applied during the simulation. As well as providing the *expected* functions, the Lua file may contain whatever else the user wishes. It may start up external processes, read data files, or any other suitable activity that sets up data for later use in the boundary condition functions.

The Lua execution environment provided to the file includes the following data:	
`block_id`	index of the current block. Boundary conditions exist in the context a block. This means that the information accessible from the UDFs is limited to that contained within the block plus a little bit of global data. This is particularly important for parallel (MPI) simulations because blocks exist is separate processes and the data in one block is not generally available in another.
`nsp`	number of species
`nmodes`	number of energy storage modes (and temperatures)
`nni,nnj,nnk`	number of cells in each index direction for the current block
`NORTH`	index of the "North" boundary. This index (and the following indices) will be handy for deciding which boundary we are working on when the `ghost_cell(args)` and `interface(args)` are called.
`EAST,SOUTH,WEST`	
`TOP,BOTTOM`	

As well as the data, there are a couple of functions that can be called to get more information about the flow at specific locations:

[22]Note that the following information is likely to become dated with code changes, so it is best to refer to the actual source code to see what is expected. Look in `bc_user_defined.cxx` for the boundary condition functions and `main.cxx` for the functions related to source terms.

`sample_flow(jb,i,j,k)`	a function that returns a table of the flow state for a particular cell. The data is the same as that listed for the `ghost_cell` tables with the addition of `vol`, the cell volume. This function is not likely to work for a MPI simulation, where only one block is visible to the curent process.
`locate_cell(x,y,z)`	a function that will search for the cell nearest the specified coordinates and return the cell indices and the index of the containing block. This function is not likely to work for a MPI simulation, where only one block is visible to the curent process.

There is also a function `compute_diffusion_coefficient` which can be called upon to get the actual diffusion coefficients at the boundary conditions.

On being called at run time, the function `ghost_cell(args)` returns two Lua tables. The first contains the flow state in the ghost cell nearest the boundary face, and the second contains the flow state for the ghost cell further away from the boundary face. Items to appear in the returned tables are:

`p`	gas pressure
`u,v,w`	velocity components in x,y,z-directions
`massf`	table of `nsp` mass fractions. The zero entry, at least, must be specified.
`T`	table of `nmodes` temperatures. The zero entry, at least, must be specified.
`tke`	turbulent kinetic energy
`omega`	ω for the $k-\omega$ turbulence model
`mu_t`	turbulence viscosity
`k_t`	turbulent heat conduction coefficient
`sigma_T`	variance of the local temperature (for Henrik's reacting flow)
`sigma_c`	variance of the local concentration (for Henrik's reacting flow)
`S`	shock-detector value (1 or 0)

and the input `args` table contains:

`t`	the current simulation time, in seconds
`x,y,z`	coordinates of the midpoint of the interface
`csX,csY,csZ`	direction cosines for the interface
`i,j,k`	indices of the cell adjacent to the interface
`which_boundary`	index of the boundary (`NORTH`,...)

If viscous effects are active, the Lua function `interface(args)` is called to get a few properties right at the bounding interface. These properties are to be returned in a table containing:

`T_wall`	static temperature at the interface, possibly a wall. All mode temperatures are set to this value.
`u,v,w`	flow velocity at the interface
`tke`	turbulent kinetic energy
`omega`	ω for the $k-\omega$ turbulence model

On entry to the function, `args` contains the same attributes as for the call to the `ghost_cell` function.

The functions are evaluated in the Lua interpreter environment that was set up when the boundary condition was instantiated so any data that was stored then is available to the functions now, possibly via global variables.

The user may also provide a function `flux(args)` that returns a table specifying the interface fluxes that can be used instead of the internally computed fluxes. The table of fluxes returned contains the following entries:

`mass`	mass flux per unit area of the interface
`momentum_x`	x-direction momentum flux per unit area
`momentum_y`	y-direction momentum flux per unit area
`momentum_z`	z-direction momentum flux per unit area
`total_energy`	flux of energy per unit area
`romega`	flux of ω for the $k-\omega$ turbulence model
`rtke`	flux of turbulent kinetic energy
`species`	table of `nsp` species mass fluxes. The zero entry, at least, must be specified.
`renergies`	table of `nmodes` energy fluxes. The zero entry, at least, must be specified.

and the input `args` table contains:

`t`	the current simulation time, in seconds
`x,y,z`	coordinates of the midpoint of the interface
`csX,csY,csZ`	direction cosines for the interface
`i,j,k`	indices of the cell adjacent to the interface
`which_boundary`	index of the boundary (`NORTH`,...)

I.2 Source terms

The Python input script can also specify the filename for a Lua file that contains functions that can be called to specify additional source terms for each step of the simulation. The functions *expected* to be defined are `source_vector(t, cell)`, `at_timestep_start(args)` and `at_timestep_end(args)`. If you don't have any useful work for the latter two, just define them to return `nil`. The Lua execution environment provided provided to the file includes the following data:

`nsp`	number of species
`nmodes`	number of energy storage modes (and temperatures)
`sample_flow`	a function that returns a table of the flow state for a particular cell
`locate_cell`	a function that will search for the cell nearest the specified coordinates and return the cell indices and the index of the containing block

When activated, the function `source_vector(t, cell)` will be called at each time step. The first argument, `t`, is the current simulation time, in seconds. The table `cell` contains:

`x,y,x`	coordinates of the cell centre
`vol`	cell volume
`p`	gas pressure
`rho`	gas density
`u,v,w`	gas velocity components
`a`	speed of sound in gas
`T`	table of `nmodes` temperatures
`massf`	table of `nsp` mass fractions

On return, the table of source terms should contain:

`mass`	rate of mass addition per unit volume
`momentum_x`	rate of x-momentum addition per unit volume
`momentum_y`	rate of y-momentum addition per unit volume
`momentum_z`	rate of z-momentum addition per unit volume
`total_energy`	rate of energy addition per unit volume
`romega`	$d\omega/dt$ addition per unit volume
`rtke`	rate of turbulent kinetic-energy addition per unit volume
`radiation`	rate of energy addition via radiation per unit volume
`species`	table of `nsp` values
`energies`	table of `nmodes` values

J Hints for Solution Visualisation with ParaView

J.1 Plotting Streamlines and Streamtubes

The following steps will guide the user through the process to visualise streamlines and streamtubes in ParaView.

1. Postprocess simulation results with the `--vtk-xml` flag as described in Section 3.5 to get the flow solution data into a form suitable for viewing in ParaView.

2. Open the Parallel (Partitioned) VTK Unstructured Data file (`.pvtu` file from the `plot` directory where the simulation was run) with ParaView and click `Apply` in the `Properties` tab of the `Object Inspector` panel.

3. Only streamlines of point data can be plotted, so the simulation results as cell data need to be converted to point data (at the cell nodes). This is achieved by selecting the menu `Filters > Alphabetical > Cell Data to Point Data` and once again clicking `Apply`.

4. Now the steamlines can be plotted by selecting the menu `Filters > Alphabetical > Stream Tracer` and once again clicking `Apply`.

5. These streamlines can be converted into streamtubes by selecting the menu `Filters > Alphabetical > Tube` and once again clicking `Apply`.

Streamtubes passing through the scramjet from Section 35 is illustrated in Figure 86.

Figure 86: Streamtubes passing through Katsu's scramjet combustor and nozzle.

J.2 Moving Blocks

Each block or collection of blocks visualised in ParaView can be translated, scaled or orientated. This may be useful when checking the operation of periodic boundary conditions, as illustrated in Figure 87. A block mesh can be moved by selecting it in the ParaView `Pipeline Browser` panel, then selecting the `Display` tab in the `Object Inspector` panel and making changes to the `Transform` section of this tab.

Figure 87: Standard Configuration 10 Mach field illustrating correct operation of periodic boundary condition.

Index

Billig, 239
block
  Block2D, 27
  Block3D, 33
  connect_blocks_2D, 32
  connect_blocks_3D, 35
  identify_block_connections, 32, 36
  MultiBlock2D, 32
  MultiBlock3D, 36
  SuperBlock2D, 31
  SuperBlock3D, 36
    example of use, 204
boundary conditions, 29
  AdiabaticBC, 30
  AdjacentBC, 29
  AdjacentPlusUDFBC, 30
  ExtrapolateOutBC, 30
  FixedPOutBC, 30
  FixedTBC, 30
  list of available, 29
  periodic, 37
    example of use, 127
  set_BC, 37
    example of use, 49
  setting individually, 37
  SlipWallBC, 30
  StaticProfileBC, 30
  SubsonicInBC, 30
  SupInBC, 29
  TransientUniBC, 30
  user defined, 259
  UserDefinedBC, 30
    example of use, 134, 148

chemical reaction, 20
  example of use, 107, 113, 155, 204
  reaction scheme file, 20, 251
    dissociating nitrogen, 110
    H2-air combustion, 119
    weakly-ionising nitrogen, 206
clustering
  See univariate function, 29
config file, 7, 39
configuration parameters, 39
control file, 7, 39
control parameters, 39

e3mpi.exe, 16
  example of use, 54, 207
e3post.py
  reference function, 152, 160
  report norms, 152
  using, 11
e3prep.py, 15
  interactive mode, 16
  using, 7
e3shared.exe, 16
  running a simulation, 9
ExistingSolution, 21, 22
  example of use, 100, 117

finish file, 10
finite-rate chemistry, *see* chemical reaction
FlowCondition, 20
  add_to_list parameter, 20
    example of use, 129

gas model
  change_ideal_gas_attribute, 19
    example of use, 125
  equilibrium chemistry, *see* look-up table
  gas-model.lua file, 18
  ideal, 17
  look-up table, 19, 247
    example of use, 89, 95

reacting, 19
thermally perfect, 17
example of use, 107, 113
user-defined, 19, 241
example of use, 155
minimal example, 246
geometric element
AOPatch, 25
Arc, 24
Arc3, 24
Bezier, 24
BezierPatch, 26
CoonsPatch, 25
example of use, 215
Helix, 24
Line, 24
MappedSurface, 26
example of use, 210
MeshPatch, 26
MeshVolume, 27
example of use, 218
Node, 23
Nurbs, 24
NurbsSurface, 26
ParametricSurface, 25
ParametricVolume, 26
Path, 24
PathOnSurface, 25
example of use, 215
PolarPath, 25
PolarSurface, 26
Polyline, 24
Polyline2, 24
PyFunctionPath, 25
example of use, 65
PyFunctionSurface, 26
example of use, 125
PyFunctionVolume, 27
RevolvedSurface, 26
example of use, 210
SimpleBoxVolume, 27
Spline, 24
example of use, 62
Spline2, 24
SurfaceThruVolume, 26
TrianglePatch, 26
example of use, 215
Vector, 23
Vector3, 23
WireFrameVolume, 27
grid
2D, 27
3D, 32
AO, 28
area orthogonality, 28
TFI, 28
transfinite interpolation, 28
gzip, 8

halting a simulation, 7
HeatZone, 38
history location, 9
extracting the data, 86
HistoryLocation, 39

import_grid_file_name, 28, 34
Installation, 223
Internet address, 5

Kirchhartz, 22

make_patch, 28
module
cfpylib, 239
e3_block.py, 27
e3_flow.py, 14
e3_grid.py, 14
libgas, 17
libgeom2, 23

postprocessing, 11
customized, 14
shock location, 207
preparation, 7

ReactionZone, 39
README, 223
restarting a simulation, 10

select_gas_model, 17
sketch, 43
source terms
user defined, 261
example of use, 147, 155
species
list of available, 18
SVG, 7

thermochemical models, 17
times file, 10
TurbulenceZone, 39

univariate function
LinearFunction, 29
example of use, 215
LinearFunction2, 29
RobertsClusterFunction, 29
example of use, 62, 100
ValliammaiFunction, 29

verification, 147, 155
VRML, 7, 43

xforce_list, 31
example of use, 54, 70