The Eilmer3 Code: User Guide and Example Book 2015 Edition

Mechanical Engineering Report 2015/07
Peter A. Jacobs* Rowan J. Gollan† Ingo Jahn‡ and Daniel F. Potter§

with contributions¶ from a cast of many, including:
Ghassan Al'Doori, Nikhil Banerji, Justin Beri, Peter Blyton, Daryl Bond, Arianna Bosco,
Djamel Boutamine, Laurie Brown, David Buttsworth, Wilson Chan, Sam Chiu,
Chris Craddock, Brian Cook, Jason Czapla, Kyle Damm, Andrew Dann,
Andrew Denman, Zac Denman, Luke Doherty, Elise Fahy, Antonia Flocco,
Delphine Francois, James Fuata, Nick Gibbons, David Gildfind, Richard Goozeé, Sangdi Gu,
Stefan Hess, Carolyn Jacobs, Chris James, Ian Johnston, Ojas Joshi, Xin Kang,
Rainer Kirchhartz, Sam Lamboo, Steven Lewis, Tom Marty, Matt McGilvray, David Mee,
Carlos de Miranda-Ventura, Luke Montgomery, Jan-Pieter Nap, Brendan O'Flaherty,
Andrew Pastrello, Paul Petrie-Repar, Jorge Sancho Ponce, Jason (Kan) Qin,
Deepak Ramanath, Andrew Rowlands, Michael Scott, Umar Sheikh, Sam Stennett,
Ben Stewart, Joseph Tang, Katsu Tanimizu, Pierpaolo Toniato,
Paul van der Laan, Tjarke van Jindelt, Anand Veeraragavan, Jaidev Vesudevan,
Han Wei, Mike Wendt, Brad (The Beast) Wheatley, Vince Wheatley,
Adriaan Window, Hannes Wojciak, Fabian Zander, Mengmeng Zhao

School of Mechanical and Mining Engineering,
The University of Queensland.

August 1, 2015

*peterj@mech.uq.edu.au
†r.gollan@uq.edu.au
‡i.jahn@uq.edu.au
§daniel.potter@uqconnect.edu.au
¶These contributions have come in the form of examples, debugging, proof-reading and constructive comments on the codes and this document, additions to this document and code for special cases.

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://cfcfd.mechmining.uq.edu.au/.

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.

Contents

Part I
Introduction

1 Compressible flow simulation and the Eilmer3 code

Eilmer3 code is an integrated collection of programs for the numerical simulation of transient, compressible gas flows in two and three dimensions. These programs answer the "What if ... ?" type of question where you will set up a flow situation by defining the spatial domain in which the gas moves, set an initial gas state throughout this domain, specify boundary condition constraints to the edges of the domain and then let the gas flow evolve according to the rules of gas dynamics.

The definition of the flow domain includes a mesh of finite-volume cells, together with boundary conditions such a solid no-slip walls, inflow surfaces and outflow surfaces. To help with the set up of this domain, the code collection includes a preparation program (`e3prep.py`) that can be used to set up a database of simulation parameters, a block-structured, body-fitted mesh defining the flow domain and an initial flow-field specification. This preparation program includes a mesh generator that can accept a description of the flow domain in terms of boundary surfaces and then generate the block-structured mesh of finite-volume cells. The mesh and initial flow state can 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.

If you wish to integrate CFD analysis in your design process, it is probably easiest to have in mind a family of domain shapes or inflow conditions, with variations defined by a small set of parameters. The Eilmer3 codes can then be used to run a number of simulations, answering the questions "What would the flow field do if we use *these* particular parameters?" This is essentially the process that we have followed when using the codes for the design of hypersonic nozzles [1] where the nozzle wall shape is adjusted to produce a uniform flow field toward the nozzle exit plane.

1.1 History of the codes

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]

1.2 More information

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://cfcfd.mechmining.uq.edu.au/. 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 [2] which covers the gas-dynamic formulation and the basic thermochemistry components.

1.3 Citing the user of Eilmer3

We hope that by using Eilmer you are able to produce some high quality simulations that aid your work. When it comes time to report the results of your Eilmer simulations to others, we ask that you acknowledge our work by citing our paper on the Eilmer code:

> Gollan, R.J. and Jacobs, P.A. (2013). About the formulation, verification and validation of the hypersonic flow solver Eilmer. *International Journal for Numerical Methods in Fluids* 73:19-57 (DOI: 10.1002/fld.3790)

Additionally, for those using the $k-\omega$ turbulence model, acknowledge our development work on that by citing:

> Chan W.Y.K., Jacobs P.A, and Mee D.J. (2011). Suitability of the k-omega turbulence model for scramjet flowfield simulations. *International Journal for Numerical Methods in Fluids* Vol 70, Issue 4, pages 493-514 (DOI: 10.1002/fld.2699)

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

Part II
User guide

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 reStructuredText file `eilmer3.rst` (Appendix A) or the corresponding HTML file from the web site[2] 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.

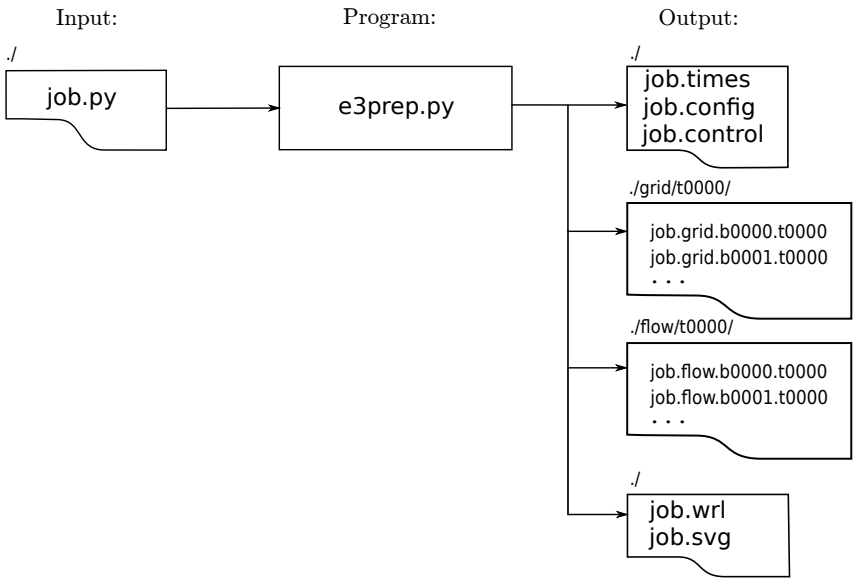
3 Reformat the flow solution data to produce files suitable for a data viewing program such as Paraview or GNU-Plot.

[2]The web site http://cfcfd.mechmining.uq.edu.au/ has a nicely formatted set of instructions, detailed API documents that have been extracted from the source code and a number of examples. It is regularly expanded and updated.

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 *nth* step, where n is given by the count value in the `control_count` parameter (default: 10). 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 10.

- *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.[3]

- 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.

3.2 Checking your grid

Before running the simulation code, it is worth checking that your grid has turned out as planned. Many a simulation has failed to start because its grid was flawed. Common problems include grids that are twisted or have adjoining blocks with edges that do not match where they are supposed to be joined. To get a set of plot files that can be loaded into `Paraview` for examination, use the post-processing program:

```
$ e3post.py --job=job --tindx=0 --vtk-xml
```

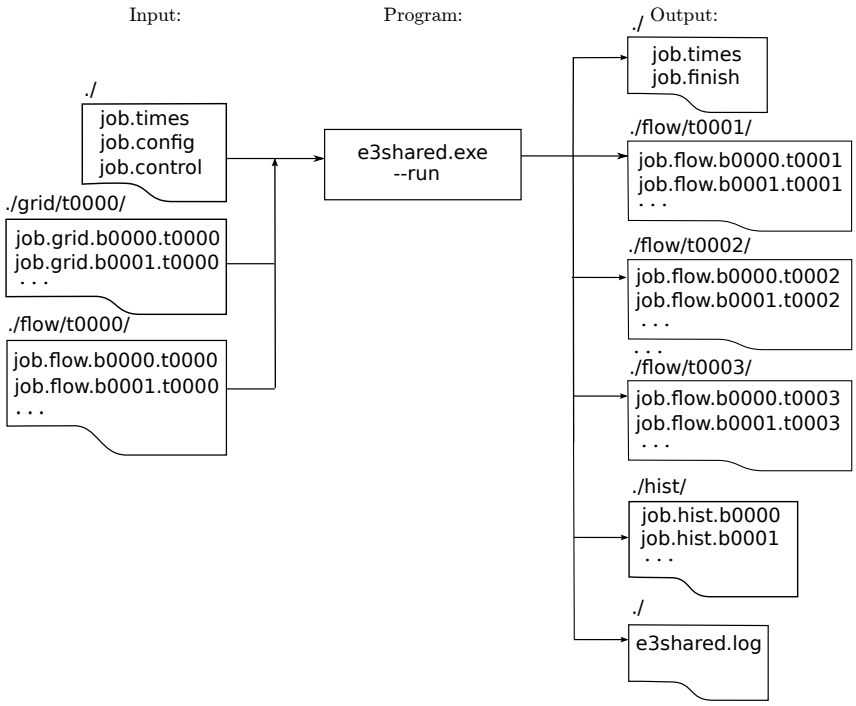
and then pick up the resulting files for inspection with `Paraview`. Look ahead to Sec. 3.7 for a more complete discussion of the postprocessing stage.

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

3.3 Running the simulation (with e3shared.exe)

Run the simulation code to produce flow data at subsequent times.[4]

```
$ 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/'' ```

[4] 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` if run with the `-q` option. See the `--heat-flux-list` option in Section 3.7 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.

For reference, here are the hints that are written out when the `--help` option is given on the command line:

```
$ e3shared.exe --help
Usage: e3shared.exe [OPTION...]
  -f, --job=<job_name>              job_name is typically the same as root_file_name
  -r, --run                         run the main simulation time-stepper
  -t, --tindx=<int>                 start with this set of flow data
  -z, --zip-files                   use gzipped flow and grid files
  -a, --no-zip-files                use ASCII (not gzipped) flow and grid files
  -q, --heat-flux-files             write heat-flux files
  -s, --surface-files               write surface files
  -v, --verbosity=<int>             set verbosity level for messages
  -w, --max-wall-clock=<seconds>    maximum wall-clock time in seconds
  -m, --mpimap=<mpimap_file>        use this specific MPI map of blocks to rank

Help options:
  -?, --help                        Show this help message
      --usage                       Display brief usage message
```

By default, the starting value for `tindx` will be zero, gzipped flow and grid files will be assumed, heat-flux and surface files will not be written, `verbosity` will be zero (*i.e.* at a minimum), and the wall-clock time will not be limited.

3.4 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[5] and runtime environment. The notes in Appendix A show how to build and run the Eilmer3 executable for Open-MPI.[6] To run Eilmer3 across multiple processors on a local machine use the following command

`$ mpirun -np` *n* `e3mpi.exe --job=name --run`

where n is the number of MPI processes to use. Note that when running the program

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

[6] These notes are also available in HTML form at the URL http://cfcfd.mechmining.uq.edu.au/.

with these options, one MPI process is assigned to each block; the number of MPI processes *must* match the number of blocks in the simulation. Each of these MPI processes is a separate program and you may run more than one per core or physical processor, however, if you want the shortest calculation time and you had lots of cores, you would probably run one per core. For simulations with many blocks, it is sometimes possible to achieve a better balance of computational load by assigning more than one block to a process. This is can be done with Eilmer3 by building a mapping file of blocks to MPI processes (using the `e3loadbalance.py` program), and then running `e3mpi.exe` with the `--mpimap=` option. The details of using Eilmer3 in this way are described in Appendix K.

3.5 Running a radiation transport calculation (e3rad.exe)

The user can build and run the shared-memory version of the radiation transport solver, `e3rad.exe`, on computers with the OpenMP API. The notes in Appendix A show how to build and run the Eilmer3 radiation transport solver executable for OpenMP. Note that you should first make the `e3shared` and `e3mpi`, then "make clean" and, finally, make `e3rad`.

You will almost certainly be running `e3rad` in the context of a partly-run flow solution.

```
$ e3rad.exe  --job=name --tindx=nnnn --run
```

where `nnnn` is the index of the flow solution for which `e3rad` will update the radiation source term. On output, `e3rad` will have incremented the `tindx` value and written a new set of data from which the flow solver can restart.

3.6 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[7] checking the state of the simulation at the end of run, then editing the *job*`.control` file and changing the parameters `dt`, `max_time` and `max_steps` to suitable values. Do not run `e3prep.py` again, else it will write all over the *job*.times file that you need to retain and 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 $k - \omega$

[7] To support old simulations that terminated with a 9999 solution frame, you can run the postprocessor with the command

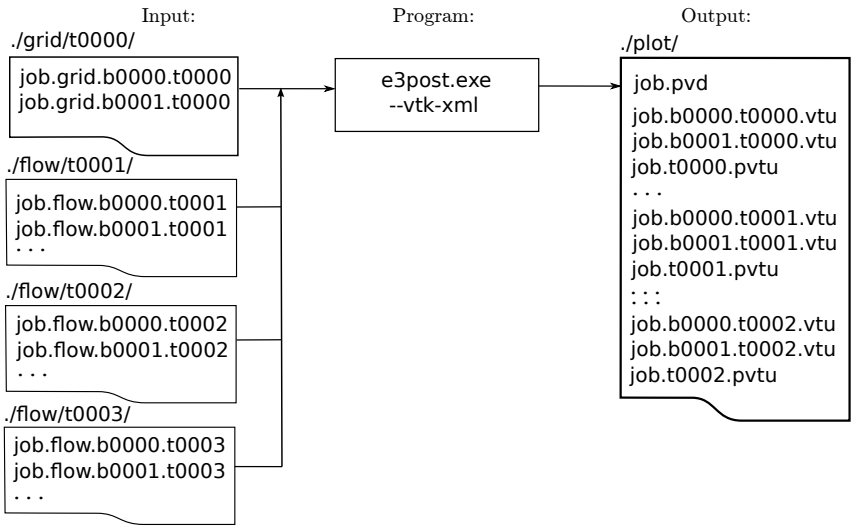
```
$ e3post.py --job=name --prepare-restart
```

This renames the 9999 flow files and tidies up the *job*`.times` file to reflect the changes.

model would lead to inconsistent data. It may be better to start a new job and use `ExistingSolution` objects (see Section 5.9) to pick up the old data. Note that your old and new soltions need to have consistent data, such as number of chemical species, etc. `ExistingSolution` works with the data available in the old solution and is not smart enough to fill in missing values.

3.7 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`[8].



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 of 1^{st} August 2015, this message is:

```
Begin e3post.py...
Source code revision string:  245f2e1c2af4+ 2320+ default tip
```

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

```
Usage:
e3post.py [--help] [--job=<jobFileName>] [--tindx=<index|9999|last|all|xxxx>]
          [--zip-files|--no-zip-files]
          [--moving-grid]
          [--omegaz="[omegaz0,omegaz1,...]"]

          [--add-pitot-p] [--add-total-p] [--add-mach] [--add-total-enthalpy]
          [--add-molef --gmodel-file="gas-model.lua"]
          [--add-transport-coeffs --gmodel-file="gas-model.lua"]
          [--add-user-computed-vars="user-script.py"]

          [--vtk-xml] [--binary-format] [--tecplot] [--plot3d] [--OpenFoam]

          [--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;..."]
          [--local-surface-list="blk,surface-name;..."]

          [--static-flow-profile="blk,face-name;..."]

          [--heat-flux-list="blk-range,surf-range,i-range,j-range,k-range;..."]
          [--bc-surface-list="blk-range,surf-range,i-range,j-range,k-range;..."]
          [--tangent-slab-list="blk-range,i-range,j-range,k-range;..."]

          [--probe="x,y,z;..."]

          [--report-norms]
          [--per-block-norm-list="jb,var-name,norm-name;..."
          [--global-norm-list="var-name,norm-name;..."
          [--ref-function=<python-script>]
          [--compare-job=<jobFileName> [--compare-tindx=<index>]]

          [--prepare-restart] [--prepare-fstc-restart]
          [--put-into-folders]

          [--verbosity=<int>]

For further information, see the online documentation, the Eilmer3 User Guide
and the source code.
```

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. These can be divided into a number of subsets. Data loading options:

- `--help` just prints the usage message. No other options are relevant.
- `--job=<jobFileName>` specifies the root name of the solution files
- `--tindx=<index|9999|last|all|xxxx>` You may pick up one solution time via its numeric index or you may specify all solution times via the keyword "`all`". The last solution frame written (and identified in the *job*.times file) can be specified by

giving the index as "last" or as "9999". If the simulation is run and a special solution frame was written at a particular time step, that solution frame not part of the standard sequence but accessible as index "xxxx".

- `--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.
- `--moving-grid` The default behaviour is to use a fixed grid (defined at `tindx=0`) for all solution frames. This flag indicates each solution frame has a dedicated grid that may change from the `tindx=0` grid.
- `--omegaz` Specify the angular velocities of the rotating-frame grids (if they any non-zero values).

Data addition options:

- `--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`.
- `--add-molef` Add species mole fractions to the data set.
- `--gmodel-file="gas-model.lua"` To add some of the mole-fractions, the gas model needs to be available. You can use this option to specify the correct gas model file if it is not the default name.

Whole-field output options:

- `--vtk-xml` The XML format for the Visualization Tool Kit (VTK) is readable by both `Paraview` and `Visit`. By default, the XML file will be simple text and probably quite large.
- `--binary-format` Write most of the data in the VTK file as appended binary records. This makes the files nonconforming XML files but it surely reduces the size of large data files and improves the speed of loading them into Paraview. For large 3D datasets, this is a good option.
- `--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.

- `--OpenFoam` This produces files usable in an OpenFOAM simulation. It may be convenient to use our grid preparation tools (e3prep with Python script input) to assemble a suitable set of grids and initial flow states but then run the simulation with the *other* CFD solver http://www.openfoam.org/ but, being biased, we wouldn't really like to talk about that here.

Data slicing and dicing options:

- `--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 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.

- `--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;..."` [9] 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`, `EAST`, `SOUTH`, `WEST`, `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.

Data manipulation and summary options:

- `--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 of a simulation.

Other house-keeping options for continuing old simulations:

- `--prepare-restart` does some house-keeping in the data files so that a simulation may be restarted cleanly. This is mainly dealing with the old `9999` file and adjusting the `.times` file. As of April 2013, the `9999` solution frame is no longer written.

- `--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. Again, this relates to a very old arrangement for the solution files.

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

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`. It is also worth noting that, by default, `e3post.py` does not write anything to the console while it it running successfully. If you want more commentary while it is doing its work, supply a nonzero integer to the option `--verbosity`. A value of 1 should give you a brief summary of the main activities whereas a value of 2 will prompt many more messages.

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. Some specific applications of writing a custom postprocessing script are:

- estimating the angle of the shock in the axisymmetric flow over a cone (Sec. 12)
- the estimation of surface force on the 10^o ramp case (Sec. 50) and
- finding the location of the bow shock for the finite cylinder simulation (Sec. 53).

3.8 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 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.

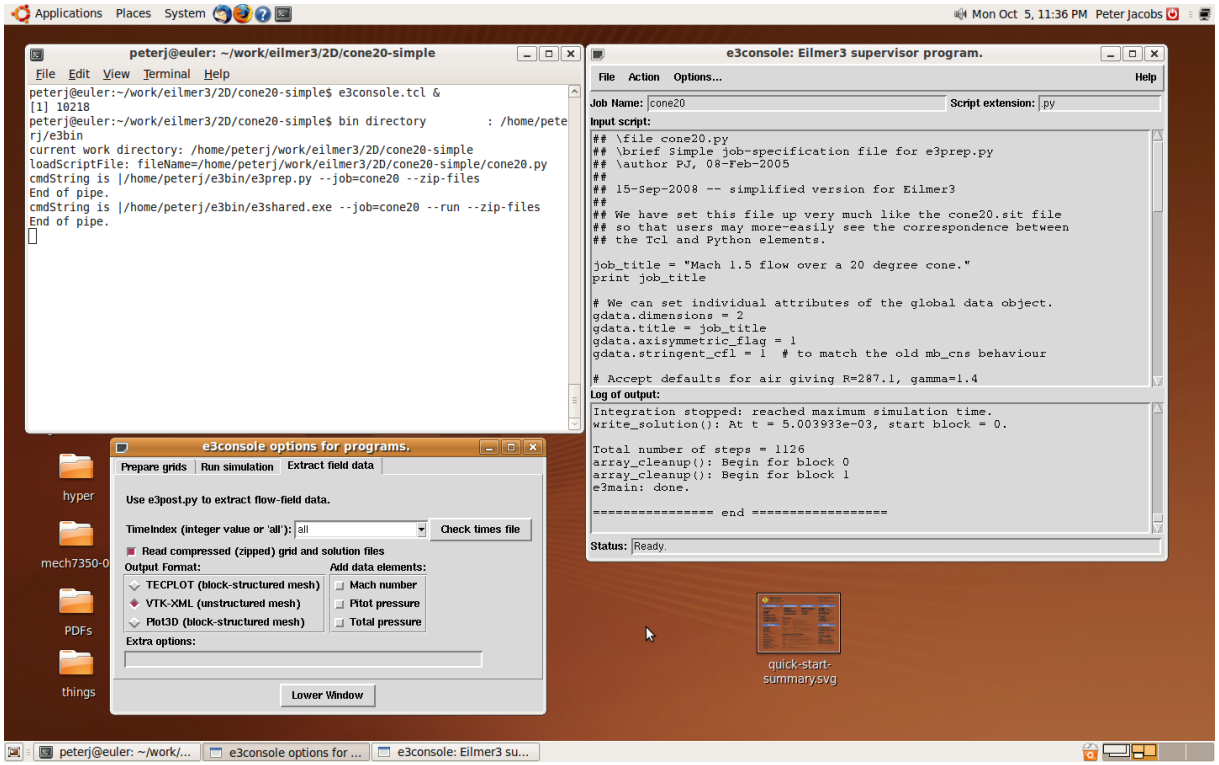


Figure 1: Screen shot of the `e3console.tcl` GUI running on PJ's workstation.

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 https://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`[10]. 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 35) uses a top-level Python script to coordinate a number of simulations with increasingly-refined grids as a crude multigrid simulation.

[10]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.

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.

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 32 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
- `'two temperature gas'`: a thermally perfect gas with two independent thermal modes: one temperature T_{tr} governs the heavy-particle translation and rotation modes, and another temperature T_{ve} governs the vibration, electronic and free-electron translation modes.
- `'real gas Bender'`: a gas with real behaviour, such as accurate thermodynamic property evaluation at high density and pressure near the saturation boundary and in the critical region. This model is based on the Bender p-v-T relationship.
- `'real gas MBWR'`: a gas with real behaviour, such as accurate thermodynamic property evaluation at high density and pressure near the saturation boundary and in the critical region. This model is based on the MBWR p-v-T relationship, which is more accurate than the Bender p-v-T relationship.
- `'real gas REFPROP'`: a gas with real behaviour, such as accurate thermodynamic property evaluation in all single and two phase regions. This model makes use of the REFPROP thermodynamic database and is more accurate than the MBWR gas model.

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 Specifying the gas model with `gasmodel.py`

Since the gas model module gets all its information about the gas from an external file (typically called `gas-model.lua`), it is reasonable to prepare the gas model specification external to your input script. To assist with this process, the program `gasmodel.py` is available. Running this program without specifying any options provided the following usage message:

```
$ gasmodel.py

Use this program to construct a simple or composite gas model for use with
the simulation codes Eilmer3 of L1d3.

Usage:
  gasmodel.py [--help] [--model=<modelName>] [--species=<speciesList|none>] \
              [--lut-file=<LUTFileName>] \
              [--output=<luaFile|gas-model.lua>]

Input parameters:
  model   : name of the gas model, may have embedded spaces.
  species : list of species names (space delimited) in a single string.
  output  : name of the gas-model file to be written.
  lut-file: name of the preexisting LUT-gas model file, if relevant.

Examples:
  $ gasmodel.py --model='thermally perfect gas' --species='N2 N' \
                --output='nitrogen.lua'

  $ gasmodel.py --model='ideal gas' --species='Ar He' \
                --lut-file='cea-lut-custom.lua.gz' \
                --output='LUT-plus-Ar-He.lua'

Notes:
  If you want a LUT-plus-composite gas model, set up the LUT table
  externally. Invoke this program, specifying the rest of the species
  for the composite gas model.  The LUT gas species is prepended to
  the composite gas species list.  You will need both gas model files
  in place to use the resulting LUT-plus-composite gas model.
```

Once you have your gas-model file generated, just give its file name to the `select_gas_model` function call in your input script using the `fname` keyword argument. This is explained further in the subsequent section.

5.4 10 minute version: the detail of gas model configuration

In the earlier 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 (such as a gas with multiple vibrational temperatures), 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 includes:

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

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

5.5 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 38. 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', MM)
```

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.

5.6 Specifying chemically reacting flow

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

```
set_reaction_scheme(config_file, reacting_flag=1, T_frozen=300.0)
```

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`. At low temperatures, it is unlikely that the reactions will proceed in any significant way so you may set value of temperature, `T_frozen`, below which the reaction updates will be

skipped. This is checked on a cell-by-cell basis.

Generally, you should use the `'thermally perfect gas'` mix for all reacting flow simulations. The enthalpies of formation are implicit in the enthalpy evaluation provided by the NASA Glen curves, thus providing the proper effect of heat release due to rearrangement of chemical bonds. Note that, at low temperatures, the ideal gas behaviour should be recovered so you shouldn't need to resort to using the `'ideal gas'` model.

An example of a reacting flow simulation is given in Section 36. The details of building a chemistry input file are provided in Appendix G.

5.7 Specifying thermal energy exchange mechanisms

For flow simulations where the number of thermal modes is greater than one (such as for the 'two temperature gas' model previously mentioned), energy exchange mechanisms can be defined that describe the exchange of thermal energy between modes due to particle collisions. If such energy exchange mechanisms wish to be modelled, the following function call is required:

```
set_energy_exchange_scheme(config_file, energy_exchange_flag=1,
                           T_frozen_energy=300.0)
```

where `config_file` is a string naming the Lua configuration file for the energy exchange scheme. This configuration file specifies all of the energy exchange mechanisms between the thermal modes due to thermal processes (i.e. particle collisions) and is built by hand by the user. Thermal energy exchange is, by default, turned on when the `set_energy_exchange_scheme(config_file)` function call is made, however, you may restrict the exchanges to the zones where chemical reactions are allowed and you can also set the temperature below which the exchanges will be skipped on a per-cell, per-timestep basis.

An example of a flow simulation with thermal energy exchange is given in Section 53. The details of building a thermal energy exchange input file are provided in Appendix H.

5.8 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[11] as:

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

```
my_flow = FlowCondition(p=1.0e5, u=0.0, v=0.0, w=0.0,
                        Bx=0.0, By=0.0, Bz=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.
- `Bx`: x-coordinate magnetic field in Tesla, default value 0.0.
- `By`: y-coordinate magnetic field in Tesla, default value 0.0.
- `Bz`: z-coordinate magnetic field in Tesla, 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.
  (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 5.10.
  (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 an extra input file describing the participating gas species and their reactions. Preparation of this file is described in Appendix G.

5.9 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. This steps through each cell and searches for closest corresponding cell centre in the old solution and inserts the flow data. 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. This assumes the same grid for the old and new solution and inserts flow data based on the i and j cell references.

- `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 new 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 10.

5.10 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 Radiation transport model

The selection of a radiation transport model and the definition of its parameters is done in a Lua file. The format for the Lua file describing the radiation transport model is given in Appendix L. A radiation model is brought into an Eilmer3 simulation via the `select_radiation_model` function:

```
select_radiation_model( input_file=None, update_frequency=1, scaling=True )
```

The input variables are:

`input_file` The name of the Lua file with the radiation transport and spectral model definitions (defaults to `None`)

`update_frequency` Number of time steps between re-calculation of the radiation solution (defaults to `1`)

`scaling` Flag to request scaling of stored radiation solution based on density and temperature for time steps where the radiation solution is not re-calculated due to the update frequency being greater than 1 (defaults to `True`)

For example, the following entry in the Eilmer input script requests the radiative source terms and heat fluxes to be recomputed every 100 time steps with scaling between recomputed solutions and directs `e3prep.py` to the file `rad-model.lua` for the details of the desired radiation transport and spectral modelling:

```
select_radiation_model( input_file="rad-model.lua", update_frequency=100,
scaling=True )
```

This setup of the radiation model would be appropriate for simulations that can be run on a single processor in reasonable time (i.e. `e3shared.exe` is used to run the simulation from beginning to end), or with radiation transport models that can be parallelised via OpenMPI (e.g. optically thin or tangent slab models). For more computationally intensive simulations, or when using the Monte–Carlo and Discrete Transfer models, it is desirable

to use the parallelised flowfield (`e3mpi.exe`) and radiation (`e3rad.exe`) solvers to enable faster run times[12]. In this situation, the update frequency would be set to zero:

```
select_radiation_model( input_file="rad-model.lua", update_frequency=0,
scaling=True )
```

and the recalculation of the radiation field coordinated via running `e3rad.exe` (see the description of the Rutowski hemisphere simulation in Sec. 48 for an example of this).

[12]The shuffling between `e3mpi.exe` and `e3rad.exe` is required for the Monte–Carlo and Discrete Transfer models as they are not implemented in `e3mpi.exe`

7 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 third 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.

7.1 Geometric elements

The most fundamental class of geometric object is the `Vector`[13] which represents a point in 3D space and has the usual behaviour of a geometric vector. This is in contrast to the behaviour of the `vector` collection class in C++ standard library. See, for example, the postprocessing program in the `simple_ramp` simulation (Section 50.3) which uses both `Vector` objects and lists of `Vector` objects. 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'`). When building models of 2D regions, you can omit the z-component value and it will default to zero.

It is possible to 'get' and 'set' values of attributes within a geometric element. For example, to create a node, extract the x value of that node, change the y value, or to use the geometry values for a new node, you could use the following commands.

```
a = Node(0.5,0.8,label='Node a')
x-value = a.x
a.y = 0.6
b = Node(a.x, a.y+0.2,label='Node b')
```

If you look into the file `cfcfd3/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)
```

[13] The `Vector` objects are actually `Vector3` objects, as defined in the C++ module `libgeom2`. Your Python input script may use either name.

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

7.1.1 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 $0 \leq t < 1$. `Path` is a base class and a number of derived types of paths are available. These 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^o 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.
- `Arc3seg`(a,b,c): a circular arc from a to b on the circular arc defined by points a, b and c. The point c is not on the path.
- `Ellipse`(a, b, c): an elliptical line from point a to point b. Point c defines the tip of the corner, where the two tangents of the ellipse from point a and b meet.
- `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. Sometimes the curve may have control points distributed such that the grid is not clustered in a good way. To fix this, it may be useful to specfy the Bezier curve to be parameterized by arc length. You need to specify all parameters, including the final `arc_length_p` parameter, *i.e.* `Bezier`($[b_0, b_1, ..., b_n]$, "label", 0.0, 1.0, 1). The 3rd and 4th parameters here specify that we want to use the full range of the Bezier curve. The final value of 1 is the `arc_length_p` parameter. A value of 0 recovers the original parametric distribution of the Bezier curve.
- `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 $0 \leq t < 1$. Just as for the `Bezier` path, short path segments mixed with large path segments may result in a grid that is not clustered in a good way. It may, therefore, be useful to specfy the Polyline to be fully parameterized by arc length. You need to specify all parameters, including the final `arc_length_p` parameter, *i.e.* `Polyline`($[p_0, p_1, ..., p_n]$, "label", 0.0, 1.0, 1). The 3rd and 4th parameters here specify that we want to use the full range of the Polyline path.

- `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.

Geometric objects can be copied with the `clone()` method and 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 `cfcfd3/lib/geometry2/source/`.

7.1.2 Surfaces

The `ParametricSurface` class represents two-dimensional objects which can be constructed from `Path` objects. These can be used as the `ParametricSurface` objects that are passed to the `Block2D` constructor (Sec. 7.2) or they can be used to form the bounding surfaces of a 3D `ParametricVolume` object (Sec. 7.1.3). Examples of the most commonly used surface patches are:

- `CoonsPatch`(p_S, p_N, p_W, p_E, `label="", r0=0.0, r1=1.0, s0=0.0, s1=1.0`): 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. The named arguments following the four paths each have default values so you will usually not need to think about them, however, it is occasionally convenient to specify a subsection of the full patch and the parameters `r0, r1, s0, s1` allow a subsection to be defined.

- `CoonsPatch`($p00, p10, p11, p01$, `label="", r0=0.0, r1=1.0, s0=0.0, s1=1.0`): a quadrilateral surface defined by it corners. Straight line segments (implicitly) join the corners. This is convenient for building simple regions that can be tiled with straight edged patches, since you don't need to explicitly generate Lines to form the edges of each patch. Note that the order for specifying the corners is counter-clockwise, starting with the South-West corner.

- `ChannelPatch`(p_S, p_N, `ruled=False, pure2D=False, label="", r0=0.0, r1=1.0, s0=0.0, s1=1.0`): an interpolated surface between two paths. By default, cubic Bezier curves are used to bridge the region between the defining curves, resulting in a grid that is orthogonal to those curves. Providing a `True` value for the `ruled` parameter results in the bridging curves being straight line segments. If you are integrating this patch into a larger construction, it may be convenient to obtain the WEST and EAST bounding curves by calling the Patch's `make_bridging_path` method with arguments of `0.0` and `1.0`, respectively. The argument for `pure2D` may be set to `True` to set all z-coordinate values to zero for purely two-dimensional constructions.

- `AOPatch`(p_S, p_N, p_W, p_E, `label="", nx=20, ny=20, r0=0.0, r1=1.0, s0=0.0, s1=1.0`):

an interpolated surface, bounded by four paths. When constructed, this surface sets up a background mesh with resolution specified by `nx` and `ny`. The construction method tries to keep the background mesh orthogonal near the edges and also tries to keep equal cell areas across the surface. The background mesh is retained within the `AOPatch` object and, if the `AOPatch` is later passed to the grid generator, the final grid is produced by interpolating within this background mesh. The default background mesh of 20×20 seems to work fairly well in simple situations. If the bounding paths have strong curvature, it may be beneficial to increase the resolution of the background mesh, so that the final interpolated mesh does not cut across the boundary paths. This is especially important if the final grid lines are clustered close to the boundaries. Setting a higher resolution of the background mesh will require more iterations to reach convergence and you may actually see a warning message that the iteration did not converge. Fortunately, an unconverged background mesh is usually fine for use, because it starts as `CoonsPatch` mesh and each iteration should just improve the metrics of orthogonality and distribution of cell areas.

- `AOPatch`($p00, p10, p11, p01$, `label="", nx=20, ny=20, r0=0.0, r1=1.0, s0=0.0, s1=1.0`): a quadrilateral surface defined by it corners. Straight line segments (implicitly) join the corners. Note that the order for specifying the corners is counter-clockwise, starting with the South-West corner. The difference with the corresponding CoonsPatch is that the background mesh, here, tries to be orthogonal to the edges and maintain equal cell areas across the surface.

- `make_patch`(p_N, p_E, p_S, p_W, `grid_type='TFI', tol=1.0e-6`): an interpolated surface, bounded by four paths. It actually returns either a `CoonsPatch` (TFI, by default) or an `AOPatch` object (`grid_type='AO'`). The convenience that it provides is in accepting the same order for the paths (*i.e.* N,E,S,W) as the other lists that are used as arguments when constructing a `Block2D` object. Look ahead to Sec.7.2.

- `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.

As listed below, there are more surface patch constructors, however, you will need to refer to their source code for documentation.

- `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.

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.

7.1.3 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 8.

- `WireFrameVolume`($surf, p$): consists of a surface $surf$ extruded along path p. The extrusion is actually done by 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 8).

- `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.

7.2 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,
                      transient_profile_faces=[],
                      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[14]:

- `psurf`: a region of 2D space bounded by 4 edges. Any flat `ParametricSurface` object (from Sec. 7.1.2) should work. This region is often constructed from 4 geometric paths via a call to `make_patch(north, east, south, west, 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.

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

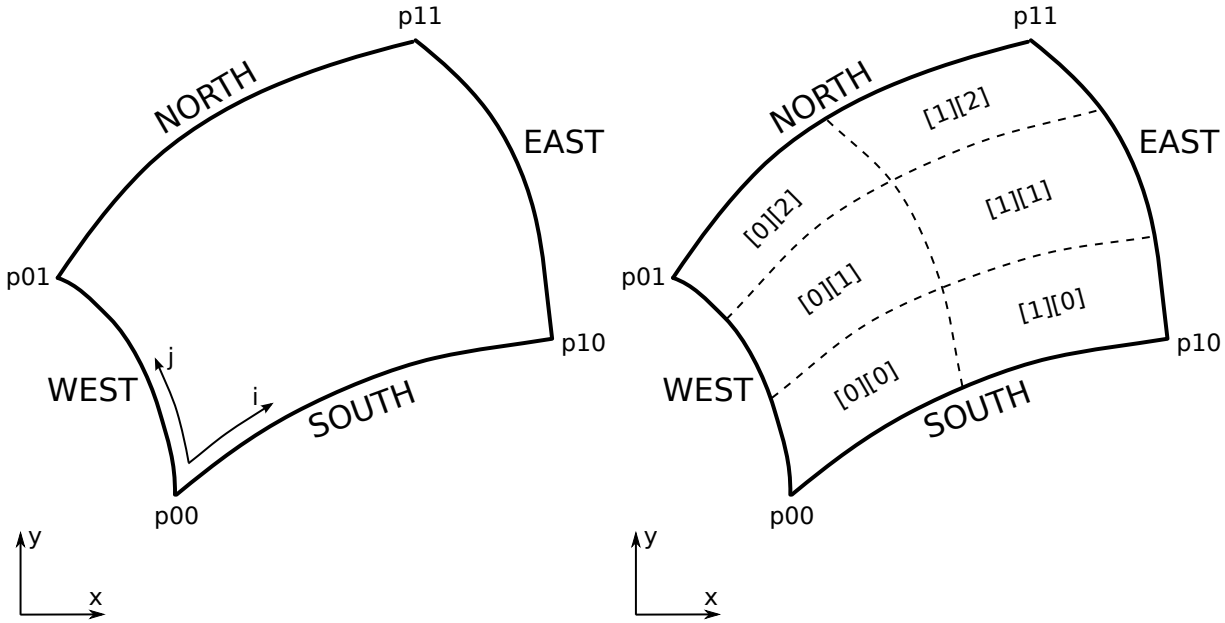


Figure 2: A two-dimensional patch containing the structured mesh for a Block2D object (left) and a collection of sub-blocks defined via a SuperBlock2D or MultiBlock2D constructor (right). The orientations of the bounding paths are important: WEST and EAST paths progress from SOUTH to NORTH; SOUTH and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

Sometimes, if the blocks are straight-sided quadrilaterals, it will be convenient to define them just with the corner points. For this case, constructing `CoonsPatch` and `AOPatch` objects directly from the corner points may be convenient. Providing a constructed `ParametricSurface` 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.
  - `HypertanClusterFunction(dL0,dL1)` sets the size of the first and last cell along a given line as a faction of the non-clustered cell size. Setting `dL0` $= 0.5$; `dL1` $= 1$ results in the first cell having half the width and the last cell having the same width as the non-clustered cell.
  - `ValliammaiFunction(dL0, dL1, L, n)` See Adriaan's source code for definitions.

  For a graphical representation of the effect of these functions, see Fig. 3. 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, as described in Section 8. You may omit this list completely, or pass `None` as any of the items. Omitted boundary conditions default to a solid, slip-wall condition. These boundary conditions may also be set at a later point in your input script, one at a time, either by assigning to individual elements of the the block's `cf_list` attribute or via the `set_BC()` method call described in the Section 8.1. Sometimes, this turns out to be handy.

- `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. See also the `HistoryLocation` object, described in Sec. 9, which may be specified separately from the the `Block2D` construction and is used to locate a history cell based on a Cartesion coordinate position in the domain rather than an i, j cell location.

- `transient_profile_faces` is an optional (unordered) list of block faces for which we want transient flow data to be written. The frequency of writing the data is the same as that for the history cells mentioned above. The particular faces may be identified by index or by string. For example, to have the flow data for the NORTH face to be written, we may specify `0`, `"north"` or `NORTH` as one of the entries in the list.

- `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^o cone test case (Section 12) 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` postprocessing 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, for situations where order is significant.

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=[],
                             transient_profile_faces=[], label="sblk")
```

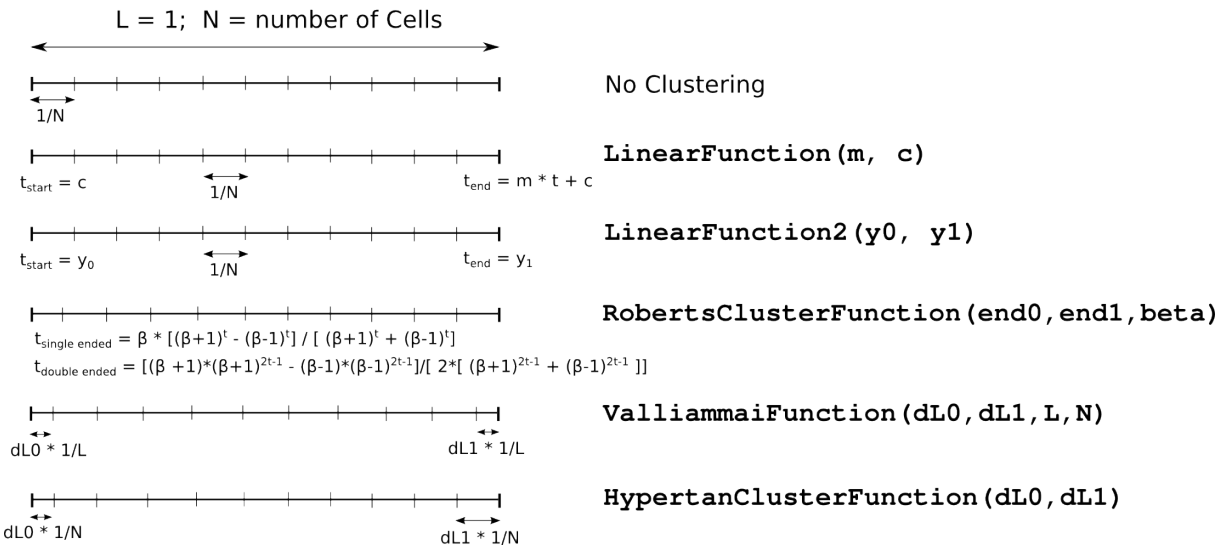


Figure 3: Graphical representation of effect the cluster functions on a one-dimensional grid.

which generates a single grid over `psurf` and then subdivides that grid into `nbi` × `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` × `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, sets_conv_flux=0, sets_visc_flux=0,
                  check_corner_locations=True,
                  reorient_vector_quantities=False,
                  nA=None, t1A=None, nB=None, t1B=None)
```

where `A` and `B` are references to the individual `Block2D` objects and `faceA` and `faceB` are their adjoining edges (`NORTH`, `EAST`, `SOUTH` or `WEST`).

By default, the function checks that the adjoining corners of the blocks do coincide in space. If they don't, a warning is issued. Usually, this is what you want, however, there are times when you really do wish to connect the flow for boundaries that are not actually coincident in space. For an example, see the periodic shear-layer in Sec. 39, where the ends of a periodic domain are manually connected. Setting `check_corner_locations=False` turns off the check on corner locations.

To handle connections where the boundaries are not aligned, you may specify `reorient_vector_quantities=True` and supply nominal vectors for each boundary's unit normal and first tangent. These nominal vector bases are used to define a rotational transformation for the vector flow quantities (*i.e.* velocity and magnetic field) that are exchanged between the boundaries. Such a transformation is useful for turbomachinery flows, where only a sector of the full flow field is being simulated and an assumed circumferential periodicity fills in the remaining detail.

Most of the time you can just ignore the default arguments associated with user-defined functions (*i.e.* `with_udf, filename, is_wall, sets_conv_flux, sets_visc_flux`). These are used to implement slowly-opening diaphragms and the like.

7.3 Putting a 2D description together

As a motivational example, especially for MECH4480 students of CFD, consider the construction of a grid around a bottle of *James Boag's Premium*. Figure 4 shows the final block arrangement with the bottle lying on its side. You can see the profile of the bottle in the curves from x=0 to x=0.2 metres. We model only the upper half plane, with the gas domain being the region around the bottle. Also, we'll do the modelling in stages, starting with a single block defining a limited subregion.

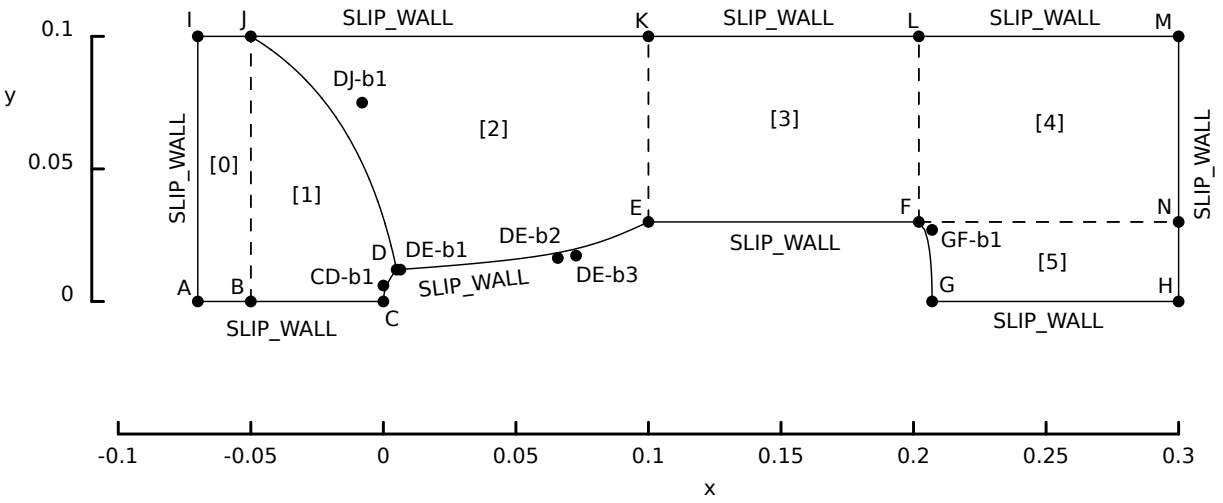


Figure 4: Schematic diagram of Ingo's beer bottle aligned with the x-axis. This PDF figure was generated from the SVG file with some edits to move the boundary labels to nicer positions.

Making a simple 2D grid

We start with just block [3] above the main body of the bottle and define just the 4 nodes E,F,K and L that mark the corners of our region of interest (Figure 5). These are created as `Nodes` with labels so that they show up in the generated SVG plot. A simple way to define the region is to make a patch with the four sides specified as staight-line paths. The `Block2D` is initialized with this patch, the number of cells in each direction and the initial gas state within the region.

```
# the_minimal_grid.py

select_gas_model(model='ideal gas', species=['air'])
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)

# Create the nodes that define key points for our geometry.
E = Node(0.1, 0.03, label="E"); F = Node(0.202, 0.03, label="F")
K = Node(0.1, 0.1, label="K"); L = Node(0.202, 0.1, label="L")

p = make_patch(Line(K,L), Line(F,L), Line(E,F), Line(E,K))
BL_3 = Block2D(p, nni=20, nnj=20, fill_condition=initial, label="[3]")

# Make a nicely-scaled SVG file at the end.
sketch.xaxis(0.1, 0.2, 0.05, -0.010)
sketch.yaxis(0.0, 0.1, 0.05, -0.030)
sketch.window(0.1, 0.0, 0.2, 0.1, 0.05, 0.05, 0.10, 0.10)
```

Making a multiblock grid

When making a flow domain that is reasonably complicated, it's probably best to make a collection of blocks where each block is roughly a quadrilateral, but with the bounding paths fitted to the curves of the object to be modelled. Figure 6 shows the resulting grid,

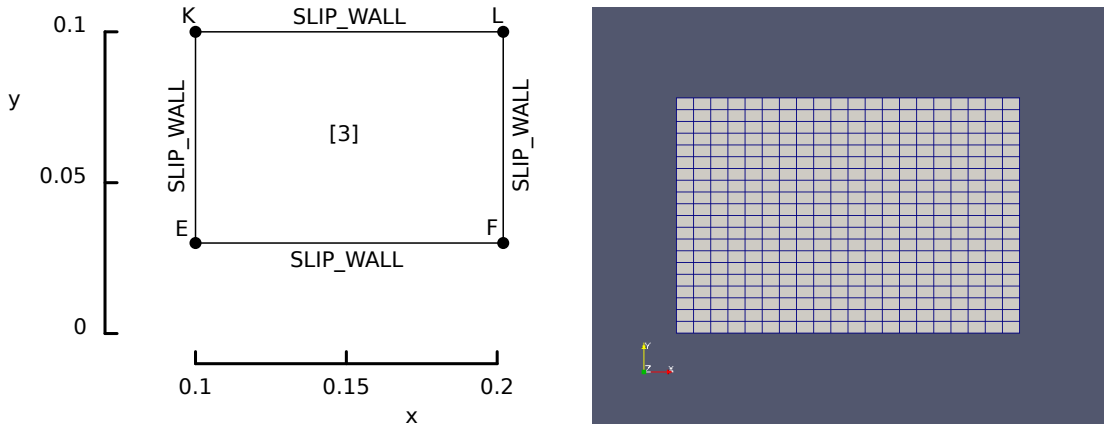


Figure 5: A single block for a simple subregion from the eventual model of the Ingo's beer bottle.

after dividing the full gas-flow region into 6 blocks.

```
# the_plain_bottle.py

select_gas_model(model='ideal gas', species=['air'])
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)

# Create the nodes that define key points for our geometry.
A = Node(-0.07, 0.0, label="A"); B = Node(-0.05, 0.0, label="B")
C = Node(0.0, 0.0, label="C"); D = Node(0.005, 0.012, label="D")
E = Node(0.1, 0.03, label="E"); F = Node(0.202, 0.03, label="F")
G = Node(0.207, 0.0, label="G"); H = Node(0.3, 0.0, label="H")
I = Node(-0.07, 0.1, label="I"); J = Node(-0.05, 0.1, label="J")
K = Node(0.1, 0.1, label="K"); L = Node(0.202, 0.1, label="L")
M = Node(0.3, 0.1, label="M"); N = Node(0.3, 0.03, label="N")

# Some interior Bezier control points
CD_b1 = Node(0.0, 0.006, label="CD-b1")
DJ_b1 = Node(-0.008, 0.075, label="DJ-b1")
GF_b1 = Node(0.207, 0.027, label="GF-b1")
DE_b1 = Node(0.0064, 0.012, label="DE-b1")
DE_b2 = Node(0.0658, 0.0164, label="DE-b2")
DE_b3 = Node(0.0727, 0.0173, label="DE-b3")

# Now, we join our nodes to create lines that will be used to form our blocks.
AB = Line(A, B); BC = Line(B, C); GH = Line(G,H) # lower boundary along x-axis
CD = Bezier([C, CD_b1, D]) # top of bottle
DE = Bezier([D, DE_b1, DE_b2, DE_b3, E]) # neck of bottle
EF = Line(E, F) # side of bottle
GF = Bezier([G, GF_b1, F],"GF",0.0,1.0,1) # bottom, with arc-length parameterization
# Upper boundary of domain
IJ = Line(I, J); JK = Line(J, K); KL = Line(K, L); LM = Line(L, M)
# Lines to divide the gas flow domain into blocks.
AI = Line(A, I); BJ = Line(B, J); DJ = Bezier([D, DJ_b1, J])
JD = DJ.copy(direction=-1); EK = Line(E, K); FL = Line(F, L);
NM = Line(N, M); HN = Line(H, N); FN = Line(F, N)

# Define the blocks, boundary conditions and set the discretisation.
n0 = 10; n1 = 4; n2 = 20; n3 = 20; n4 = 20; n5 = 12; n6 = 8

BL_0 = Block2D(make_patch(IJ, BJ, AB, AI), nni=n1, nnj=n0,
               fill_condition=initial, label="[0]")
BL_1 = Block2D(make_patch(JD, CD, BC, BJ), nni=n2, nnj=n0,
               fill_condition=initial, label="[1]")
BL_2 = Block2D(make_patch(JK, EK, DE, DJ), nni=n3, nnj=n2,
               fill_condition=initial, label="[2]")
BL_3 = Block2D(make_patch(KL, FL, EF, EK), nni=n4, nnj=n2,
```

```
                fill_condition=initial, label="[3]")
BL_4 = Block2D(make_patch(LM, NM, FN, FL), nni=n5, nnj=n2,
                fill_condition=initial, label="[4]")
BL_5 = Block2D(make_patch(FN, HN, GH, GF), nni=n5, nnj=n6,
                fill_condition=initial, label="[5]")
identify_block_connections()

# Make a nicely-scaled SVG file at the end.
sketch.xaxis(-0.1, 0.3, 0.05, -0.05)
sketch.yaxis(0.0, 0.10, 0.05, -0.01)
sketch.window(-0.1, 0.0, 0.3, 0.4, 0.02, 0.05, 0.20, 0.23)
```

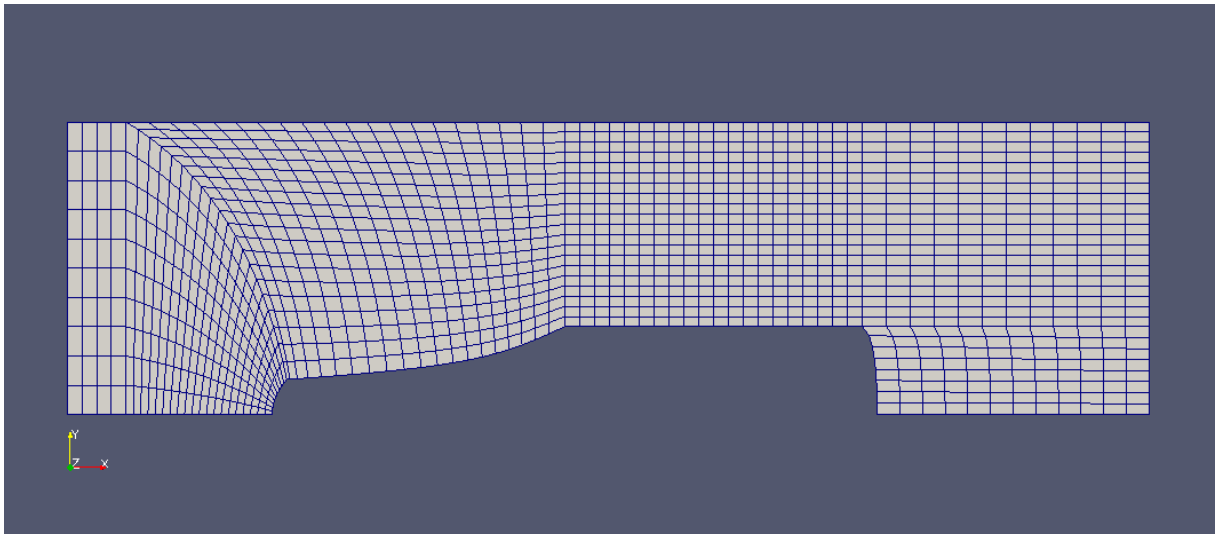


Figure 6: A multiple-block model of the region around Ingo's beer bottle.

Each of the blocks is generated independently of the others. It is your responsibility to ensure that the common defining edges are consistent and that the cell-discretization along each of these edges is consistent with the corresponding discretization of any adjacent edge of another block. The first constraint is easy to meet by defining each edge once only and reusing that path in the definition of different blocks. Sometimes, the orientation of a pair of blocks and the particular directions of the paths within each block means that one defining edge needs to be in the opposite sense to the original. In this case the `clone()` and `reverse()` methods may be useful. The script actually uses the equivalent `copy(direction=-1)` method call. For an example of this, note the orientation of blocks 1 and 2 in the script and study the patches that are used in their construction. Note that the line `DJ` has been copied and reversed and called `JD` so that it can serve as a `NORTH` boundary for block 1. The line `DJ` is oriented such that it can be used as a WEST boundary for block 2.

Improving the grid with clustering

We can now tweak the grid and improve the distribution and shape of the cells by adjusting the clustering of the points along each of the block edges. See Figure 7 for the result of the following script. The partular values used for the strength of the clustering are ad-hoc and some trial and error has been used to get these particular values.

Again, the distribution of points along each edge of each block is computed independently, so it is the responsibility of the user to ensure that the cells along the corresponding edges of adjoining blocks are aligned. This will require the use of matching clustering functions on these edges.

```
# the_clustered_bottle.py

select_gas_model(model='ideal gas', species=['air'])
initial = FlowCondition(p=5955.0,  u=0.0,    v=0.0, T=304.0)

# Create the nodes that define key points for our geometry.
A = Node(-0.07, 0.0, label="A"); B = Node(-0.05, 0.0, label="B")
C = Node(0.0, 0.0, label="C"); D = Node(0.005, 0.012, label="D")
E = Node(0.1, 0.03, label="E"); F = Node(0.202, 0.03, label="F")
G = Node(0.207, 0.0, label="G"); H = Node(0.3, 0.0, label="H")
I = Node(-0.07, 0.1, label="I"); J = Node(-0.05, 0.1, label="J")
K = Node(0.1, 0.1, label="K"); L = Node(0.202, 0.1, label="L")
M = Node(0.3, 0.1, label="M"); N = Node(0.3, 0.03, label="N")

# Some interior Bezier control points
CD_b1 = Node(0.0, 0.006, label="CD-b1")
DJ_b1 = Node(-0.008, 0.075, label="DJ-b1")
GF_b1 = Node(0.207, 0.027, label="GF-b1")
DE_b1 = Node(0.0064, 0.012, label="DE-b1")
DE_b2 = Node(0.0658, 0.0164, label="DE-b2")
DE_b3 = Node(0.0727, 0.0173, label="DE-b3")

# Now, we join our nodes to create lines that will be used to form our blocks.
AB = Line(A, B); BC = Line(B, C); GH = Line(G,H) # lower boundary along x-axis
CD = Bezier([C, CD_b1, D]) # top of bottle
DE = Bezier([D, DE_b1, DE_b2, DE_b3, E]) # neck of bottle
EF = Line(E, F) # side of bottle
GF = Bezier([G, GF_b1, F],"GF",0.0,1.0,1) # bottom, with arc-length parameterization
# Upper boundary of domain
IJ = Line(I, J); JK = Line(J, K); KL = Line(K, L); LM = Line(L, M)
# Lines to divide the gas flow domain into blocks.
AI = Line(A, I); BJ = Line(B, J); DJ = Bezier([D, DJ_b1, J])
JD = DJ.copy(direction=-1); EK = Line(E, K); FL = Line(F, L);
NM = Line(N, M); HN = Line(H, N); FN = Line(F, N)

# Define the blocks, boundary conditions and set the discretisation.
n0 = 10; n1 = 4; n2 = 20; n3 = 20; n4 = 20; n5 = 12; n6 = 8
rcfL = RobertsClusterFunction(1, 0, 1.2)
rcfR = RobertsClusterFunction(0, 1, 1.2)

BL_0 = Block2D(make_patch(IJ, BJ, AB, AI), nni=n1, nnj=n0,
               fill_condition=initial, label="[0]")
BL_1 = Block2D(make_patch(JD, CD, BC, BJ), nni=n2, nnj=n0,
               cf_list=[RobertsClusterFunction(0, 1, 1.1),None,rcfR,None],
               fill_condition=initial, label="[1]")
BL_2 = Block2D(make_patch(JK, EK, DE, DJ), nni=n3, nnj=n2,
               cf_list=[rcfR,None,None,RobertsClusterFunction(1, 0, 1.1)],
               fill_condition=initial, label="[2]")
BL_3 = Block2D(make_patch(KL, FL, EF, EK), nni=n4, nnj=n2,
               fill_condition=initial, label="[3]")
BL_4 = Block2D(make_patch(LM, NM, FN, FL), nni=n5, nnj=n2,
               cf_list=[rcfL,None,rcfL,None],
               fill_condition=initial, label="[4]")
```

```
BL_5 = Block2D(make_patch(FN, HN, GH, GF), nni=n5, nnj=n6,
               cf_list=[rcfL,None,rcfL,None],
               fill_condition=initial, label="[5]")
identify_block_connections()

# Make a nicely-scaled SVG file at the end.
sketch.xaxis(-0.1, 0.3, 0.05, -0.05)
sketch.yaxis(0.0, 0.10, 0.05, -0.01)
sketch.window(-0.1, 0.0, 0.3, 0.4, 0.02, 0.05, 0.20, 0.23)
```

Further improvement of the grid can be made by introducing a layer of blocks around the bottle surface, so that the cells near the surface can be made always nearly orthogonal and much more finely clustered toward the surface. The extra blocks add to the complexity of the input script but provide some decoupling with respect to cell number along block edges and allow the fine clustering of cells toward the bottle surface without greatly increasing the cell refinement in other parts of the gas-flow region. Such a grid would be suited to simulations of viscous flows.

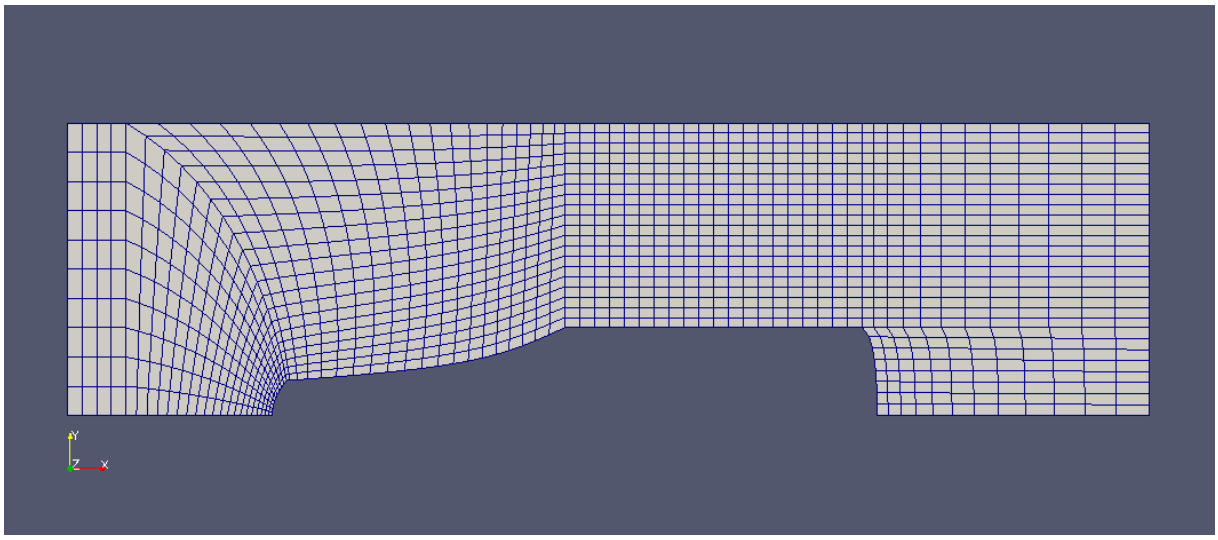


Figure 7: An improved multiple-block grid around Ingo's beer bottle.

7.4 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 8 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.[15] The corner vertices of the block are numbered 1 through 7 as shown. These points are used in the search to determine block

[15] 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 50.3), point 0 would be identified as $p000$, point 1 as $p100$, etc.

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.

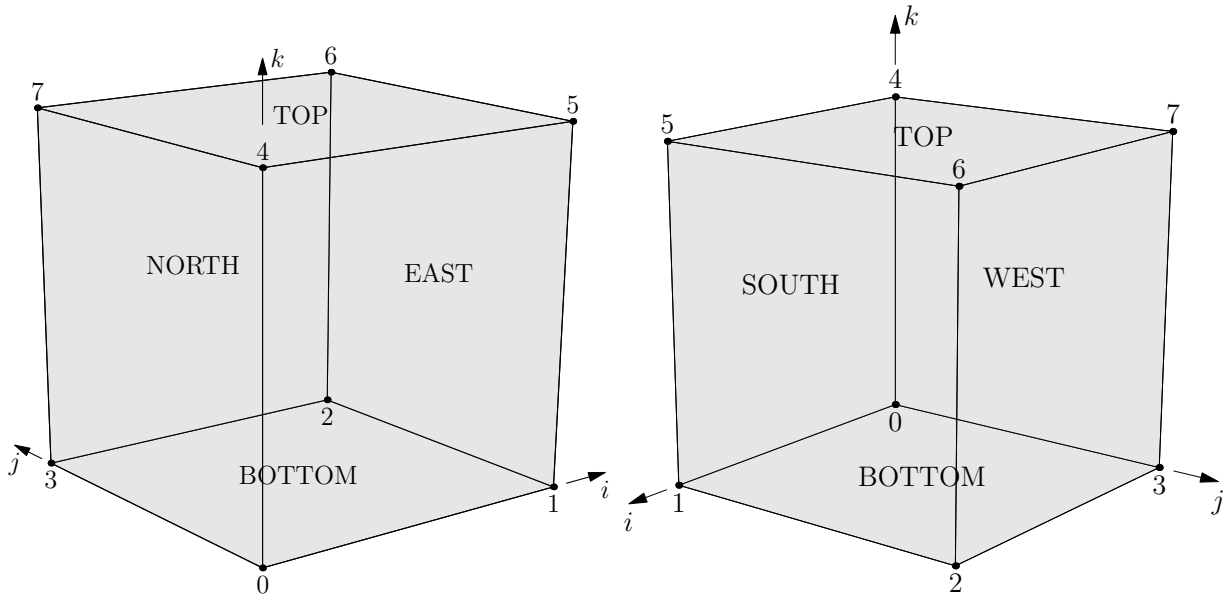


Figure 8: Two views of the hexahedral block containing the structured mesh. These figures are ambiguous but each is supposed to show a hollow box with the *far* surfaces in each view being labelled. The *near* surfaces are transparent and unlabelled. To get your hands on an unambiguous representation, 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,
                      transient_profile_faces=[], 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[16]:

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

- `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 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 8. 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 46 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 and is given in Section 8. Again, omitted conditions or those specified as `None` default to solid, no-slip walls.
- `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.

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

- `transient_profile_faces` is an optional (unordered) list of block faces for which we want transient flow data to be written. The frequency of writing the data is the same as that for the history cells mentioned above. The particular faces may be identified by index or by string. For example, to have the flow data for the NORTH face to be written, we may specify `0`, `"north"` or `NORTH` as one of the entries in the list.

- `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, sets_conv_flux=0, sets_visc_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,
                           hcell_list=None,
                           transient_profile_faces=[],
                           omegaz=0.0, label="sblk")
```

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 that 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 356) 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.

8 Specifying flow conditions at block boundaries

The preferred way to set boundary conditions is to assign specific `BoundaryCondition` objects to the `bc_list` within each constructed `Block2D` or `Block3D` object. Back in Sections 7.2 and 7.4, it was shown that the boundary conditions could be specified as a list of `BoundaryCondition` objects passed to the constructor of `Block2D` or `Block3D` objects, respectively. You have to provide a list with the correct number of entries, which is 4 for 2D blocks and 6 for 3D blocks. If you don't have a particular `BoundaryCondition` object for each element of the list, just specify `None` for the missing entries.

Alternatively, `BoundaryCondition` objects can be assigned individually to elements of the `bc_list` attribute after block construction. For example:

```
blk_0.bc_list[WEST] = SupInBC(inflow, label="inflow-boundary")
blk_1.bc_list[EAST] = ExtrapolateOutBC(label="outflow-boundary")
```

Available boundary condition classes include:

- `AdjacentBC(other_block=-1, other_face=-1, orientation=0, reorient_vector_quantities=False, Rmatrix=[1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0], label='')`
  Usually this boundary condition is applied implicitly, by calling the function `identify_block_connections()`, for cases where one block interfaces with another and the block boundaries are cleanly aligned, however, it can be applied manually for cases where you want the flow to be plumbed from on block face into another and the blocks are not geometrically aligned. A non-unity transformation matrix, `Rmatrix`, can be provided for cases where the flow vector quantities need to be reoriented when they are copied from the other boundary to this one.

- `SupInBC(inflow_condition, label='')` where we want to specify the inflow condition that gets copied 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(x_order=0, sponge_flag=0, label='')` where we want a (mostly supersonic) outflow condition. Flow data is effectively copied (`x_order=0`) or linearly-extrapolated (`x_order=1`) 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.

- `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. Note that we need to set `gdata.viscous_flag = 1` (see Section 10, Viscous effects) to make this boundary condition effective.

- `FixedTBC(Twall, label='')` where we want viscous effects to impose a no-slip velocity condition and a fixed wall temperature. As for the `AdiabaticBC`, we need to set `gdata.viscous_flag = 1` (see Section 10, Viscous effects) to make this boundary condition effective.

- `JumpWallBC(Twall, sigma, label='')` where we want viscous effects to impose a partial-slip velocity condition and temperature that approaches a specified wall temperature, subject to the limitation of rarefied-gas effects. This limitation is specified via the accommodation coefficient, `sigma`. As for the `FixedTBC`, we need to set `gdata.viscous_flag = 1` (see Section 10, Viscous effects) to make this boundary condition effective.

- `SubsonicInBC(stagnation_condition,`
  `mass_flux=0.0, relax_factor=0.05, p0_min=None, p0_max=None,`
  `direction_type='normal', direction_vector=[1.0,0.0,0.0],`
  `direction_alpha=0.0, direction_beta=0.0, label='')`
  The flow is assumed subsonic and we specify the stagnation pressure and temperature and a velocity *direction* at the boundary. When applied at each time step, the average local pressure across the block boundary is used with the stagnation conditions to compute a stream-flow condition. Depending on the value for `direction_type`, the computed velocity's direction can be set `'normal'` to the local boundary, `'uniform'` in direction and aligned with `direction_vector`, `'radial'` in through a cylindrical surface using flow angles `direction_alpha` and

direction_beta, or 'axial' in through a circular surface using the same flow angles. For the case with a nonzero value specified for mass_flux, the current mass flux (per unit area) across the block face is computed and the nominal stagnation pressure is incremented such that the mass flux across the boundary relaxes toward the specified value. The value for relax_factor adjusts the rate of convergence for this feedback mechanism. If this process drives the stagnation pressure to unreasonably small or large values while the flow is settling, you may set the limits p0_min and p0_max to indicate what is physically realizable for your modelling situation. Also note, that for multi-temperature simulations, all of the temperatures are set to the 0th entry in the temperature array. This should usually be a reasonable physical approximation because this boundary condition is typically used to simulate inflow from a reservoir, and stagnated flow in a reservoir has ample time to equilibriate at a common temperature. The implementation of this boundary condition may not be time accurate, particularly when large waves cross the boundary, however, it tends to work well in the steady-state limit.

- TransientUniBC(filename, label='') where we want to specify the time-history of the inflow condition. It is most likely used when feeding your flow simulation with inflow data produced by an earlier L1d3 simulation.

- StaticProfileBC(filename, n_profile=1, label='') where we want to apply a steady-state inflow which may vary in space.

- TransientProfBC(filename, label='') where we want to specify the time-history of the inflow condition that also has a varying profile across the block face. This boundary condition is most likely used in a simulation that takes it inflow data from an earlier simulation, which wrote its transient flow data via the transient_profile_faces option.

- FixedPOutBC(Pout, Tout=300.0, use_Tout=False, x_order=0, label='')
  is like ExtrapolateOutBC() but with a specified back pressure and, possibly, a temperature. 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, sets_conv_flux=0, sets_visc_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 sets_conv_flux indicates whether the user is supplying the convective fluxes

at the boundary interfaces (default 0), in which case the user-supplied file should contain a valid `convective_flux()` function. 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. Similar to the setting of convective fluxes, the flag `sets_visc_flux` indicates whether the user is supplying the viscous fluxes at the boundary interfaces (default 0). In this case, the user-supplied file should contain a valid `viscous_flux()` function. If not, the internal viscous derivatives are used to compute fluxes based on the supplied interface data. See Appendix I for the details of setting up this boundary condition.

- `AdjacentPlusUDFBC(other_block, other_face, orientation, filename, is_wall=0, sets_conv_flux=0, sets_visc_flux=0, reorient_vector_quantities=False, Rmatrix=None, 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.

- `MovingWallBC(r_omega=None, centre=None, v_trans=None, Twall_flag=False, Twall=None, label='')`: allows the user to specify a no-slip wall condition where the wall surface has a non-zero velocity. Note that this is only for *tangential* velocity at the wall and, to have any effect, needs to have `viscous_flag = 1`. Values for `r_omega`, `centre` and `v_trans` are specified as tuples of 3-components giving the angular-velocity, a point on the axis or rotation and a superimposed translational velocity. The actual velocity of a point on the wall is then given by the vector expression $\vec{\omega} \times (\vec{r} - \vec{c}) + \vec{v}_{trans}$, where $\vec{r}$ is the point on the wall, $\vec{c}$ is the point on the axis of rotation and $\vec{\omega}$ is the angular velocity. This combination allows the setting of planar and cylindrical moving surfaces. Optionally, the wall temperature may also be set. If not, the condition defaults to an adiabatic wall.

- `MappedCellBC(ghost_cell_trans_fn=lambda x, y, z: (x, y, z), reorient_vector_quantities=False, Rmatrix=[1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0], mapped_cell_list=[], label="")`: is something like the `AdjacentBC` but with an ad-hoc mapping of destination(ghost)-cell location to source-cell location.

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.

8.1 Setting conditions with setBC (deprecated)

This function is deprecated in favour of setting boundary conditions by direct interaction with the appropriate item in a `Block`'s `bc_list` as discussed in the previous section. 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, x_order=0, sponge_flag=None,
       Twall=None, Pout=None, Tout=300.0, use_Tout=False,
       r_omega=None, centre=None, v_trans=None,
       filename=None, n_profile=1,
       is_wall=0, sets_conv_flux=0, sets_visc_flux=0,
       Twall_flag=False,
       reorient_vector_quantities=False,
       Rmatrix=[1.0,0.0,0.0, 0.0,1.0,0.0, 0.0,0.0,1.0],
       assume_ideal=0, mdot=None, emissivity=None,
       Twall_i=None, Twall_f=None, t_i=None, t_f=None,
       mass_flux=0.0, p_init=100.0e3, relax_factor=0.05,
       direction_type="normal", direction_vector=[1.0,0.0,0.0],
       direction_alpha=0.0, direction_beta=0.0,
       ghost_cell_trans_fn=lambda x, y, z: (x, y, z),
       I_turb=0.0, u_turb_lam=1.0,
       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 copies or extrapolations 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 `sets_conv_flux` indicates whether the user is supplying the convective fluxes at the boundary interfaces (default 0). If not, the internal flux calculator is used together with the supplied ghost-cell data. The flag `sets_visc_flux` indicates whether the user is supplying the viscous fluxes at the boundary interfaces (default 0). If not, the internal viscous derivatives are used to compute fluxes based on the supplied interface data.
- `ADJACENT_PLUS_UDF`:
- `MOVING_WALL`: moving wall boundary condition is a solid boundary with no-slip but non-zero surface velocity. `r_omega` is a vector to set rotational speed, `centre` is used to set the rotational centre and `v_trans` is used to configure the translational velocity of surface. Initially, moving wall is a kind of adiabatic wall, if you want to set a fixed temperature condition, `Twall_flag` should be `True` temperature to and the temperature specified as `Twall`.
- `MAPPED_CELL`:

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

9 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 zone are given by the `Vector` values `point0` and `point1`. In a two-dimensional simulation, `point0` corresponds to `p00` in Figure 2 (on page 45) while `point1` corresponds to `p11`, at the opposite corner of the patch. In three-dimensional simulation, `point0` corresponds to `p0` in Figure 8 (on page 55) while `point1` corresponds to `p6` at the diagonally-opposite vertex of the hexahedral block.

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 `gdata.heat_time_stop` a positive non-zero value and `gdata.heat_time_start` and `gdata.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 viscosity 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.

An effective method to trigger chemical reactions is to use

```
IgnitionZone(Ti, point0, point1, label="")
```

A temperature, `Ti`, is set that controls the reaction rate used for chemical reactions, without effecting the gas temperature in the flow field. The rate-controlling temperature is typically set to an artificially inflated value to promote ignition (e.g. a value at 2000 K is effective in igniting certain compositions of a methane/air mixture). The rate-controlling temperature is used to evaluate the chemical reaction rates only within the in the physical extents of the `IgnitionZone`. This zone is in effect between `gdata.ignition_time_start` (default value is 0) and `gdata.ignition_time_stop` and then "switching-off" subsequently. In this manner, the reaction rates within the zone are artificially inflated, which is able to ignite reactions in a short time.

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 the Cartesian coordinate location specified by x, y and z.

10 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[17] which is accessible to the user's input script. Grouped by theme, the possible attributes include[18]:

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

[17] 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.

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

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. If you are sure that your geometric and boundary descriptions but your simulation fails for no clear reason, try setting the initial time step to a very small value. For some simulations of viscous hypersonic flow on fine grids, it is not unusual to require time steps to be as small as a nanosecond.

- `dt_max`‡: Maximum allowable time step (in seconds), default value 1.0e-3. Sometimes, especially when strong source terms are at play, the CFL-based time-step determination does not suitably limit the size of the allowable time step. This parameter allows the user to limit the maximum time step directly.

- `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.

- `gasdynamic_update_scheme`‡: one of: `'euler'`, `'pc'`, `'predictor-corrector'`, `'midpoint'`, `'classic-rk3'`, `'tvd-rk3'`, `'denman-rk3'`.
  Default value is `'predictor-corrector'`.
  Note that `'pc'` is equivalent to `'predictor-corrector'`. If you want time-accurate solutions, use a two- or three-stage stepping scheme, otherwise, Euler stepping has less computational expense but you may get less accuracy and the code will not be as robust for the same CFL value. For example the shock front in the Sod shock tube example is quite noisy for Euler stepping at CFL=0.85 but is quite neat with any of the two- or three-stage stepping schemes at the same value of CFL. The midpoint and predictor-corrector schemes produce a tidy shock up to CFL = 1.0 and the rk3 schemes still look tidy up to CFL = 1.2.

- `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.

- `viscous_signal_factor`‡: 1.0=full viscous effect for the signal calculation within the time-step calculation. It has been suggested that the full viscous effect may not be needed to ensure stable calculations for highly-resolved viscous calculations. A value of 0.0 will completely suppress the viscous contribution to the signal speed calculation but you may end up with unstable stepping. It's a matter of "try a value and see" if you get a larger time-step while retaining a stable simulation.

- `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 reconstruction limiter, default value 1.
- `extrema_clipping_flag`: 1=do extrema clipping at end of 1D scalar reconstruction, default value 1. A value of 0 suppresses clipping.
- `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"`. Options are:
  - `"riemann"`: An *exact* flux calculator that iteratively solves the Riemann sub-problem and then constructs the fluxes from the hypothetical interface state. It's expensive and doesn't behave any better than the much cheaper AUSMDV scheme but it does have very little diffusion. The lack of diffusion can cause problems [3] and it is not recommended for use.

- `"ausm"`: A cheap, effective, but sometimes noisy scheme from Ref. [4].
- `"efm"`: A cheap and very diffusive scheme by Pullin and Macrossan [5, 6]. For most hypersonic flows, it is too diffusive to be used for the whole flow field but it does work very nicely in conjunction with AUSMDV, especially for example, in the shock layer of a blunt-body flow.
- `"ausmdv"` A good all-round scheme low-diffusion for supersonic flows.[7].
- `"adaptive"` A blend [3] of the low-dissipation AUSMDV scheme for the regions away from shocks with the much more diffusive EFM used for cell interfaces near shocks. It seems to work quite reliably for hypersonic flows that are a mix of very strong shocks with mixed regions of subsomic and supersonic flow. The blend is controlled by the parameters `compression_tolerance` and `shear_tolerance` that are described below.
- `"ausm_plus_up"`: Implemented from Ref. [8]. It should be accurate and robust for all speed regimes. It is the flux calculator of choice for very low Mach number flows, where the fluid behaviour approaches the incompressible limit. For best results, you should set the value of `M_inf`.
- `"hlle"` The MHD version of the HLLE scheme.

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.

- `M_inf`: representative Mach number for the free stream. Used by the `AUSM_PLUS_UP` flux calculator. The default value is 0.01.

Viscous effects

- `viscous_flag`: 1=viscous terms are active, 0=inviscid simulation, default value 0.
- `separate_update_for_viscous_flag`‡: 1=the update for the viscous transport terms are done separately to the convective terms, 0=the viscous-term updates are initegrated with the explicit update of the convective terms, 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_model`: string specifying which model to use, "none", "k_omega", "baldwin_lomax", default "none".
- `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.

Thermo-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.
- `T_frozen`: temperature (in degrees K) below which reactions are frozen. The default value is 300.0 since most reaction schemes seem to be valid for temperatures above this, however, you may have good reasons to set it higher or lower. (May also be set in the call to `set_reaction_scheme()`.)

- `T_frozen_energy`: temperature (in degrees K) below which the energy exchange is skipped. The default value is 300.0, however, you may have good reasons to set it higher or lower. (May also be set in the call to `set_energy_exchange_scheme()`.)

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. If 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.
- `control_count`‡: number of time steps between re-parsing the `.control` file. If the `.control` has been edited, then the new values are used after re-parsing, default value 20.
- `heat_time_start`: default value 0.0, in seconds. For a description of HeatZones, see Section 9.
- `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.
- `mhd_flag`: 1=make MHD physics active. default value 0.
- `electric_field_work_flag`: 1=make $\vec{u} \cdot \nabla p_e$ source term in the electron energy equation active. default value 0.

11 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 point (xmin,ymin) to upper-right point (xmax,ymax) in the simulation space to the corresponding points on a display page, as shown in Figure 9.

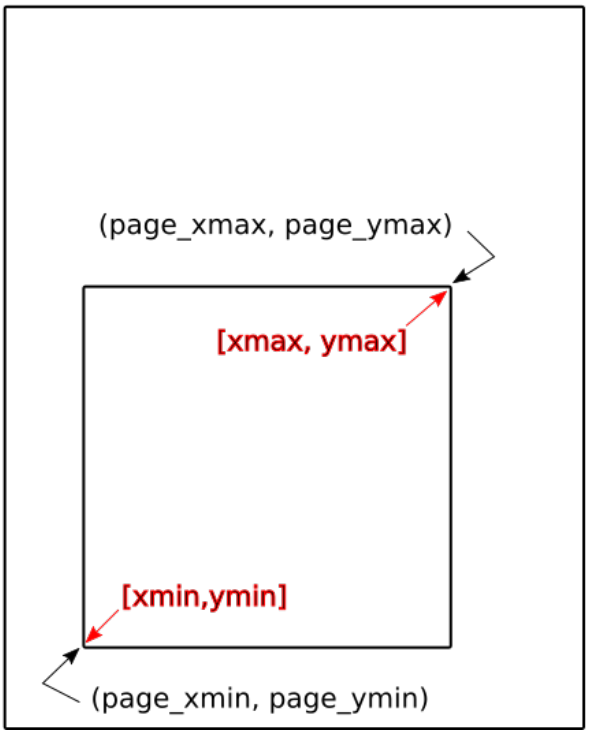


Figure 9: Parameters defining the sketch window placed on a display page. Parameters shown in red are coordinate values in the simulation domain, while those shown in black are coordinate values on the displayed page. All values are in metres.

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. Note that these axis parameters are specified in the coordinate system of the simulation space and that all values are in metres. Figure 10 shows the axes as they are placed in the 20-degree cone example on page 77.

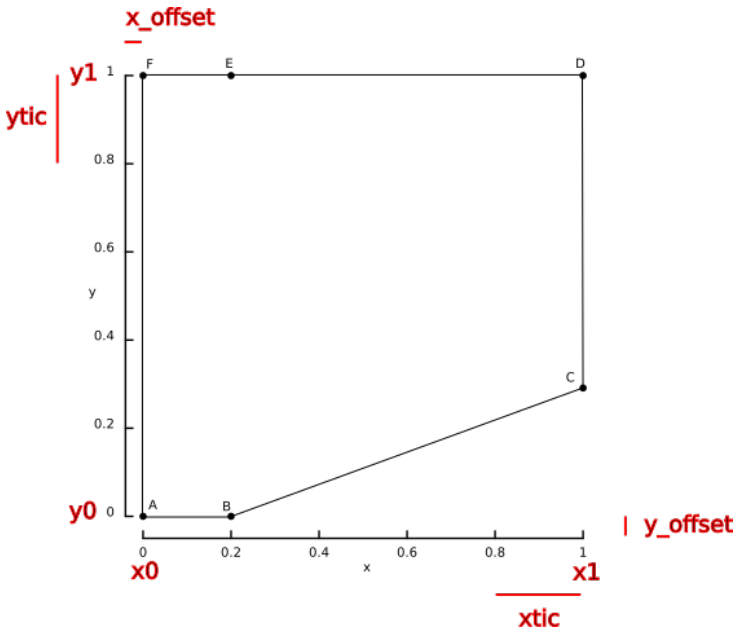


Figure 10: Parameters specifying the arrangement of the axes for the SVG sketch.

Part III
A tutorial example

The first example, of ideal, inviscid flow over a cone (Sec. 12), is a simple flow situation but the description provided here goes into fair detail on setting up the simulation and then on extracting interesting flow quantities to help in the interpretation of the results. It is recommended reading for all beginning users. Once you have run and mastered this particular example, pick whichever example most closely matches your flow of interest and have a go at building your own simulation.

Later examples also use more of Python's capabilities. The input script for the heat-transfer to a sphere (Sec, 35), 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.

12 Mach 1.5 flow over a 20-degree cone

Let's start with a simple-to-imagine flow of ideal air over a sharp-nose of a supersonic projectile. Figure 11 is a reproduction of Fig. 3 from Maccoll's 1937 paper [9] and shows a shadowgraph image of a two-pounder projectile, in flight at Mach 1.576. We'll restrict our simulation to just the gas flow coming onto and moving up the conical surface of the projectile and work in a frame of reference attached to the projectile. Further, we will assume that all of the interesting features of the three-dimensional flow can be characterized in a two-dimensional plane. The red lines mark out the region of our gas flow simulation, assuming axial symmetry about the centreline of the projectile.

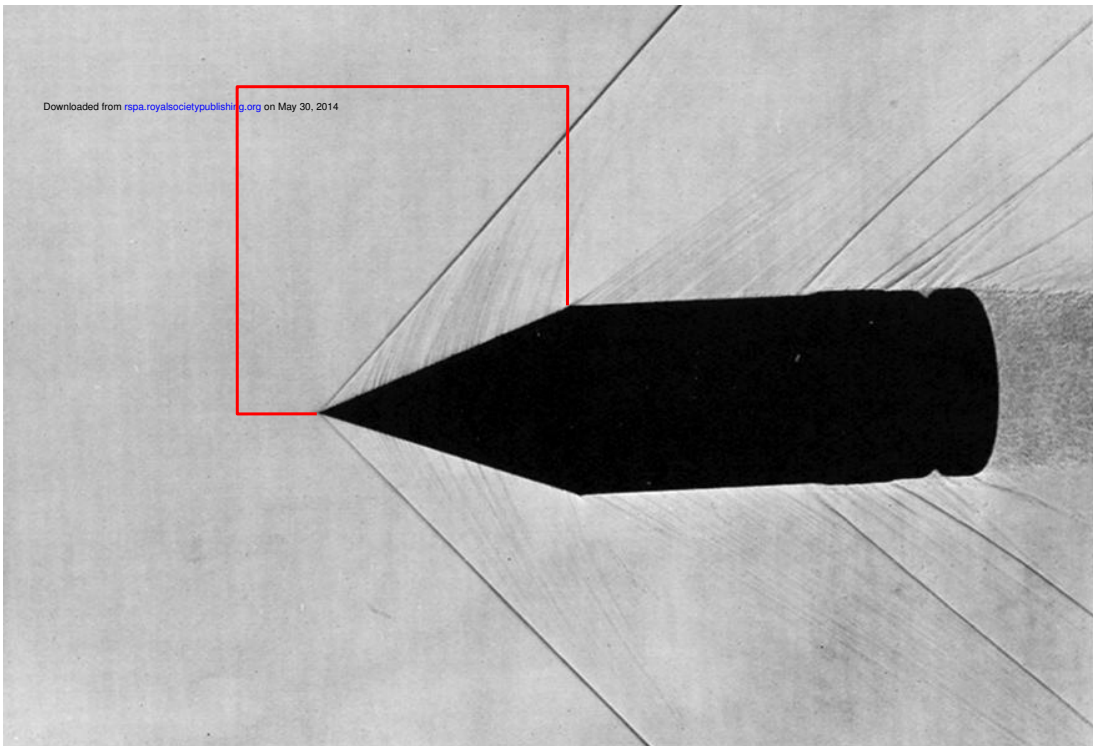


Figure 11: A two-pound projectile in flight. A conical shock is attached to the sharp nose of the projectile. This photograph was published by Maccoll in 1937. The red lines have been added to demark the region of gas flow for which we will set up our simulation.

The resulting flow, in the steady-state limit, should have a single shock that is straight in this 2D meridional plane (but conical in the original 3D space). The angle of this shock can be checked against Taylor and Maccoll's gas-dynamic theory and, since the simulation demands few computational resources (in both memory and run time), it is useful for checking that the simulation and plotting programs have been built and installed correctly.

13 The simulation

To build our simulation, we abstract the boxed region from Figure 11 and consider the axisymmetric flow of an ideal, inviscid gas over a sharp-nosed cone with 20 degree half-angle. The constraint of axisymmetry implies zero angle of incidence for the original 3D flow.

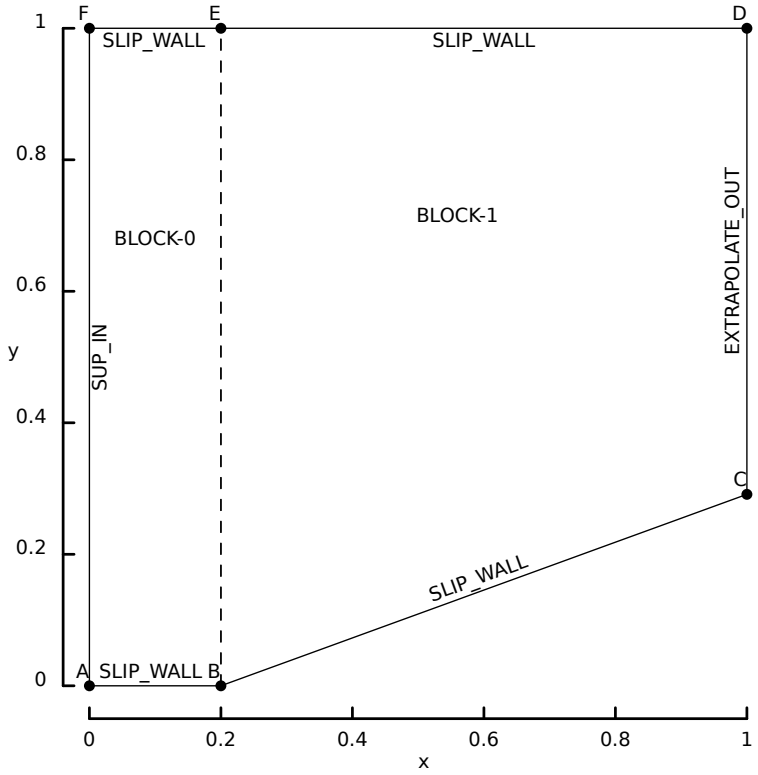


Figure 12: Schematic diagram of the geometry for a cone with 20 degree half-angle. This PDF figure was generated from the SVG file with some edits to move the boundary labels to nicer positions.

Despite Figure 11 being a good motivator for this simulation, the free-stream conditions of $p_\infty = 95.84\,\text{kPa}$, $T_\infty = 1103\,\text{K}$ and $u_\infty = 1000\,\text{m/s}$ are actually related to the shock-over-ramp test problem in the original ICASE Report [10] and are set to give a Mach number of 1.5. It is left as an exercise for the reader to run a simulation at Maccoll's value of Mach number and check that the simulation closely matches the shadowgraph image.

13.1 Input script (.py)

```
# cone20.py
# Simple job-specification file for e3prep.py
# PJ, 08-Feb-2005
#     15-Sep-2008 -- simplified version for Eilmer3
```

```
#     29-May-2014 -- discard old way of setting BCs

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

# 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 by first defining
# the corner nodes, then the lines between those corners.
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, with particular 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])

# Set boundary conditions.
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(inflow, label="inflow-boundary")
blk_1.bc_list[EAST] = ExtrapolateOutBC(label="outflow-boundary")

# Do a little more setting of global data.
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)
```

13.2 Running the simulation

Assuming that you have the program executable files built and accessible on your system's search `PATH`, as described in Appendix A, try the following commands:

```
$ cd ~/cfcfd3/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 862 time steps being required). The commands invoke the shell script shown below. This script, less the commands to generate the plot, could be used as a template for your own simulation 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."
```

Note that long-format command-line options start with two dashes. These double dashes are a little hard to distinguish in the shell scripts.

14 Results and Postprocessing

Figure 13 shows the flow field 5 milliseconds after flow start. This has been long enough for the flow to reach a steady state, with the shock being essentially straight. The plots have been produced with `Paraview`, picking up the `plot/cone20.pvd` file. The time stamp in the lower left corner has been added as an `Annotate Time Filter`, selected from the main `Filters` menu. Also, the pressure field has been plotted as a coloured `surface`, while the temperature field has been plotted as a `surface with edges` to clearly show the computational grid. The distortion of the grid in the right-hand block is a result of the area-orthogonality (AO) grid generator making the compromises required to achieve a reasonably-orthogonal mesh at the edges of the block. The default transfinite grid generator would have produced a mesh that appears less distorted overall but would have individual cells that are more sheared for this particular block. For the rectangular block on the left, both generators would produce the same mesh.

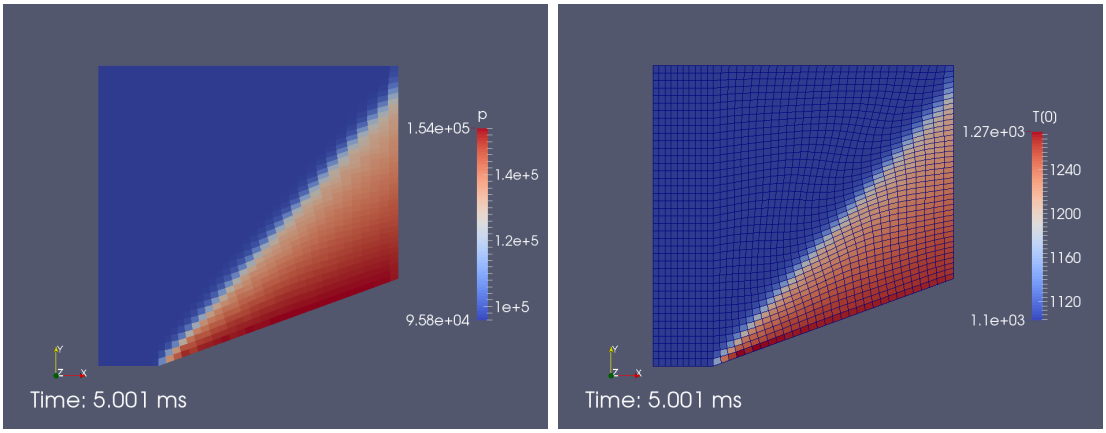


Figure 13: Pressure and temperature fields for a low-resolution simulation of flow over a cone with 20 degree half-angle. The temperature field plot also included the mesh.

The shock displayed in the pressure field shows features that are characteristic of a flow solution produced by a "shock-capturing" code such as `Eilmer3`. With the coarse grid, the shock has a stair-case appearance. This is accentuated by the plotting program which was set to display the cell-average value as a uniform colour within each cell.[19] Also, when following a line that crosses the shock, a small number of cells to be counted before the full pressure jump has been reached. In an ideal, inviscid simulation, the shock should be a zero-thickness transition. This can be approached by increasing the mesh resolution, as seen in Figure 14. The high-resolution solution is looking clean but the computational cost, in terms of calculation time, has gone up from a few seconds to nearly 2 hours.

Since Eilmer3 is a simulation program, it starts with some initial (but possibly variable)

[19] If you want a smoother appearance, you can use the Paraview filter `Cell Data to Point Data`.

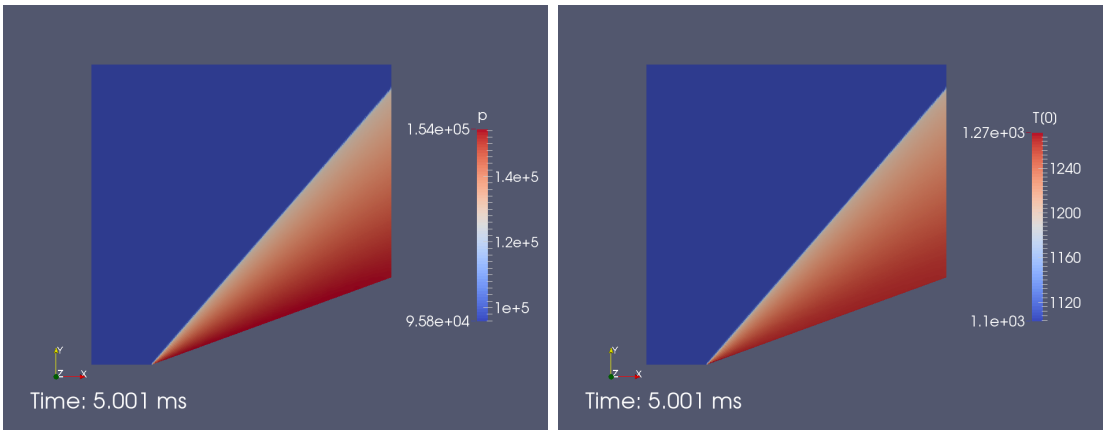


Figure 14: Pressure and temperature fields for a mesh with 8 times more resolution in each direction.

flow state across the whole simulation domain and then, subject to the applied boundary conditions, integrates the conservation equations forward in a *time-accurate* manner. In this case of a constant free stream flow coming onto a sharp cone, the flow field evolves toward a steady state. Figure 15 shows the pressure field at a number of times through the simulation. The time increment, in seconds, between the frames was specified in the input script as `gdata.dt_plot = 1.5e-3`.

Although not obvious in the figure, a lot of detailed flow structure has passed through the flow domain even before the 1.5 milliseconds frame. From then until the final time of 5.0 milliseconds, not a lot seems to be happening. It would be tempting to terminate the simulation at 3.0 milliseconds, however, depending on how accurately you need to report flow quantities, you may need to run much longer to achieve a sufficiently steady flow.

A key flow parameter of interest might be the drag on the cone and we can get `Eilmer3` to occasionally write out the integrated forces on the cone surface with the `xforce_list = [0,0,1,0]` argument used when constructing the second block. This causes `Eilmer3` to write the integrated forces to the log file at the same frequency as history files are written. We then use an Awk program (`cp.awk`) to filter the log file, extracting lines that have the x-force data of interest. New users might like to use an equivalent program written in Python, however, the Awk language is very convenient for writing filter programs.

```
# 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
```

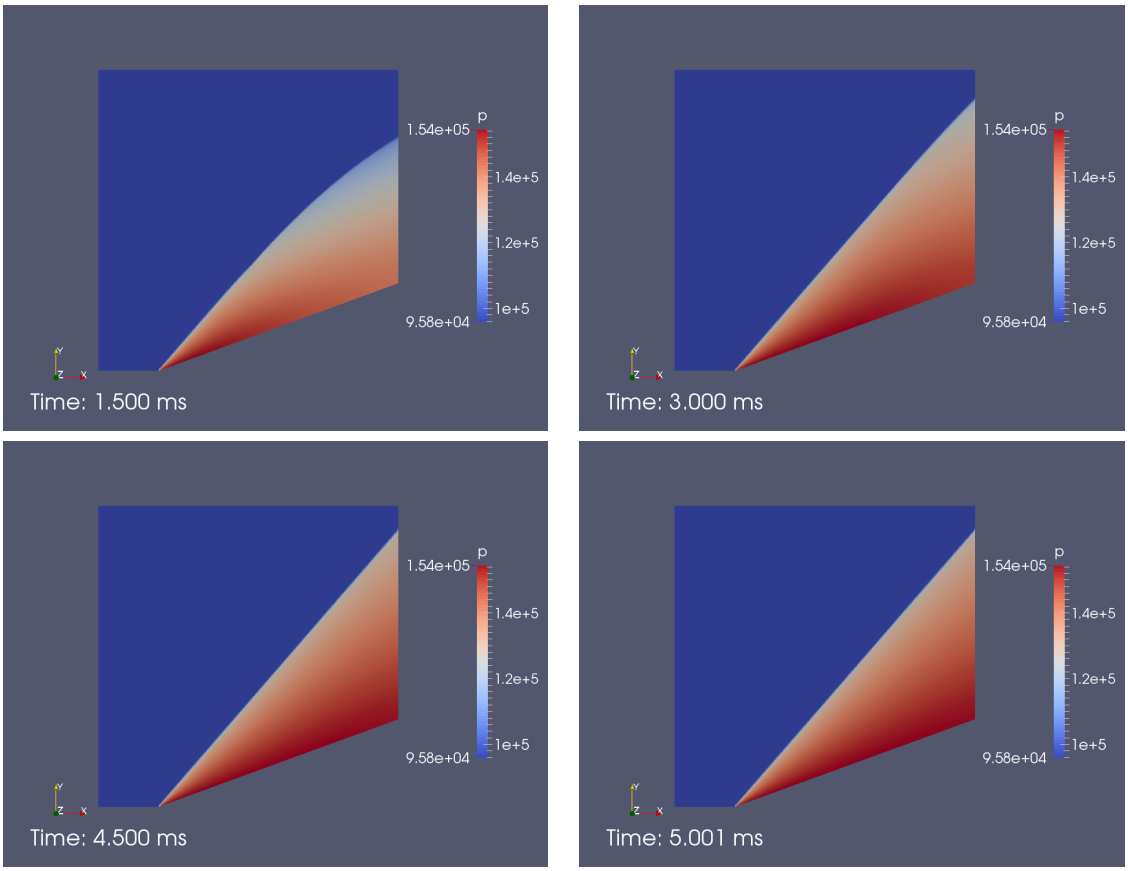


Figure 15: Evolution of the pressure field, times as indicated.

```
    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;
}
```

Before plotting the drag force history, it is convenient to normalize it into a history of drag coefficient. From Chart 5 in Ref. [11], the expected steady-state shock wave angle is 49^o and, from Chart 6, the pressure coefficient 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 16 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. Note the sudden rise as the shock structure driven by the free-stream flow arrives at the cone surface. There is a more gradual rise after this initial jump as the conical flow region fills out and becomes steady. You can now see the motivation for choosing 5.0 milliseconds as the end time for the simulation.

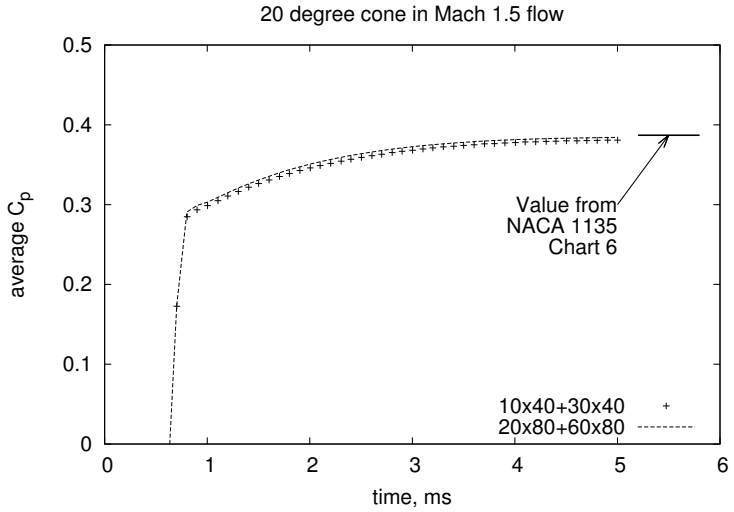


Figure 16: Evolution of the axial (drag) force for flow over a cone with 20 degree half-angle for two mesh resolutions.

15 Accessing the field data for specialized postprocessing

Beyond the usual slice-and-dice type of postprocessing that is provided by `e3post.py`, it may be useful to do specialized calculations on the flow data. In this flow, the shock is expected to be straight and we can compute that it should have an angle of $\beta = 48.96^o$, with respect to the free-stream direction, using one of the gas-dynamic functions[20] from `cfpylib`

```
from cfpylib.gasdyn.ideal_gas_flow import beta_cone
from math import degrees, radians
beta = beta_cone(V1=1000.0, p1=95.84e3, T1=1103.0, theta=radians(20.0))
print "beta=", degrees(beta), "degrees"
```

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.17

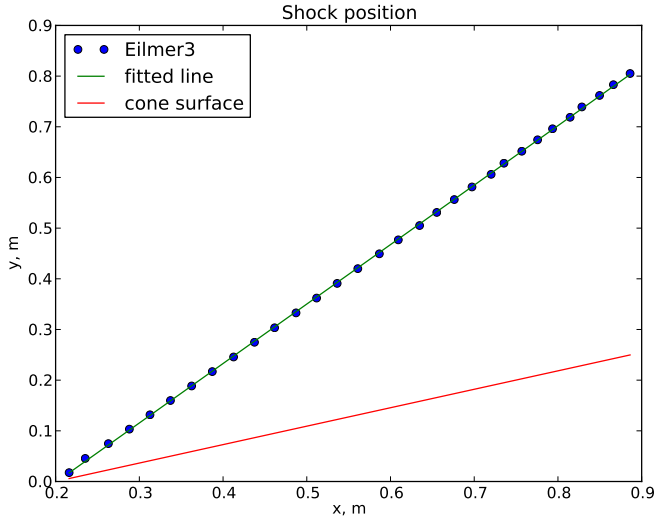


Figure 17: Shock shape for Mach 1.5 flow over the 20-degree cone.

[20] For an overview the functions in `cfpylib`, see Appendix E then, for more information on each specific function, look at the web site `http://cfcfd.mechmining.uq.edu.au/` under the heading `Libraries`.

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
#     17-Apr-2013 change accommodate no 9999 file at end

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, tindx, nbi, nbj):
    """
    Reads flow blocks and returns the coordinates of the shock front.

    Input:
    jobName: string name used to construct file names
    tindx: integer index of the target solution
    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/t%04d/%s.flow.b%04d.t%04d.gz' % \
                (tindx, jobName, blkindx, tindx)
            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", 4, 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."
```

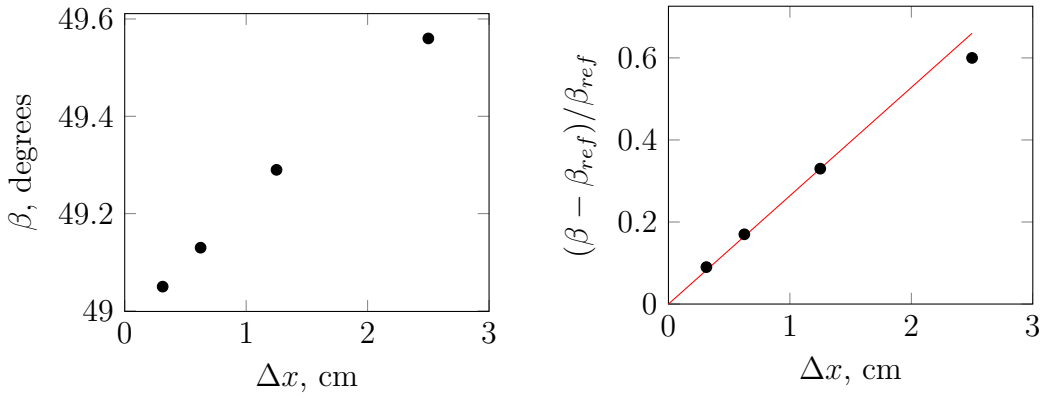


Figure 18: Convergence of the shock angle and its relative error with mesh refinement. $\beta_{ref} = 48.96^o$.

16 Grid convergence

Determining a single value for some parameter is only part of the complete job. Usually, you must provide some guide as to the reliability of that value and this is often done with a grid convergence study. For our estimate of shock wave angle, we could follow the initial simulation run with a number of runs on successively finer meshes and check that the estimated values converge in the limit of cell size going to zero.

Since this example is not very demanding for a low-resolution grid, it is easy to double the grid resolution a couple of times over and get data over a good range of cell sizes. Figure 18 shows the raw shock angle estimates converging nicely to a value of 49^o. In general, this is usually the end point for our analysis. Since we have a reference value computed via the Taylor-Maccoll theory, we can also look at the convergence to the *true* value and, given sufficient computational resource, it looks at though we can get as close as we wish.

17 Other notes on this first example

- Run time for the simple cone20 simulation is approximately 17 seconds for 862 steps on a computer with an AMD Phenom II X4 840, 800 MHz processor. Of course, the shared-memory code does not make use of the other 4 processor cores, 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 *".

- The script `cone20_run_mpi.sh` is available for running the simulation with the parallel version of the code on a machine with OpenMPI 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`.

18 Parametric modelling using Python

Let's rework the simulation to explore the gas-dynamics a little more and also make use of the parametric capabilities of the Python input script. We'll first parameterize the descriptions of the flow and the geometric description of the flow domain by replacing some of the literal numeric values of the original script with variables and simple algebraic expressions.

Specifically, let's introduce a variable, M, for the Mach number of the in-flow stream and then compute the velocity from that value and the estimated sound-speed of that in-coming stream. This gives us a convenient way of specifying a sample Mach number so we can explore the response of the simulated flow field to a range of inflow Mach numbers. We'll also describe the cone by its half-angle and axial length. From these items, we can compute the base radius. For the remaining key items defining the flow domain, we need to know where the apex of the cone is placed with respect to the inflow boundary and we need to say how far away the top-edge of the flow domain is from the axis. Finally, to make the grid generation a little more convenient as we change the boundaries of the flow domain, we'll define a cell size as length dx, and determine numbers of cells within each block as an overall length-scale of each dimension of the block divided by this cell size.

18.1 Input script (.py)

```
# conep.py
# Simple job-specification making use of parametric capabilities.
# PJ, 25-Jul-2015 -- adapted from the classic cone20

# We can set individual attributes of the global data object.
gdata.dimensions = 2
gdata.axisymmetric_flag = 1

# 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)
Tinf = 1103.0
a = sqrt(1.4*287.1*Tinf) # sound speed in free stream
M = 1.5
ux = M * a
print "ux=", ux
inflow  = FlowCondition(p=95.84e3, u=ux, v=0.0, T=Tinf)

# Define cone/flow-domain geometry
theta = 20 # cone half-angle, degrees
L = 0.8 # axial length of cone, metres
rbase =  L * math.tan(math.radians(theta))
x0 = 0.2 # upstream distance to cone tip
H = 1.0 # height of flow domain, metres

gdata.title = "Mach %.1f flow over a %.1f-degree cone." % (M, theta)
print gdata.title

# Set up two quadrilaterals in the (x,y)-plane by first defining
# the corner nodes, then the lines between those corners.
a = Node(0.0, 0.0, label="A")
```

```
b = Node(x0, 0.0, label="B")
c = Node(x0+L, rbase, label="C")
d = Node(x0+L, H, label="D")
e = Node(x0, H, label="E")
f = Node(0.0, H, 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, with particular discretisation.
dx = 1.0/40
nx0 = int(x0/dx); nx1 = int(L/dx); ny = int(H/dx)
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])

# Set boundary conditions.
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(inflow, label="inflow-boundary")
blk_1.bc_list[EAST] = ExtrapolateOutBC(label="outflow-boundary")

# Do a little more setting of global data.
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)
```

19 Exploring the gas dynamics

Except for a small difference in the number of cells in the x-direction of each block, Figure 19 shows the same flow field 5 milliseconds after flow start as Figure 13. It has the same straight, attached shock and same range of pressures displayed.

Looking up the conical shock charts in NACA-1135 [11], we can see that a 32 degree cone falls outside the shock-polar for a free-stream Mach number of 1.5 and so should have a detached shock. Let's try that by changing the value of `theta` from 20 to 32. That's all that needs to be done before re-running the prepration program and main simulation program, with the calculations to get the appropriate velocity already encoded within the user input script. Figure 20 shows the resulting pressure field at 5 ms.

The result is not quite as expected because the flow has choked between the conical surface and the upper edge of the domain, with its default `SlipWallBC` boundary condition, that acts as a smooth inside wall of a slippery pipe. The obvious fix to attempt is to increase the height of the flow domain by setting H to a larger value. Figure 21 shows the resulting pressure field at 5 ms for an inflow Mach number of 1.5, which should have a detached shock, and for a free-stream Mach number of 1.6, which should have an attached shock, according to the inviscid flow theory.

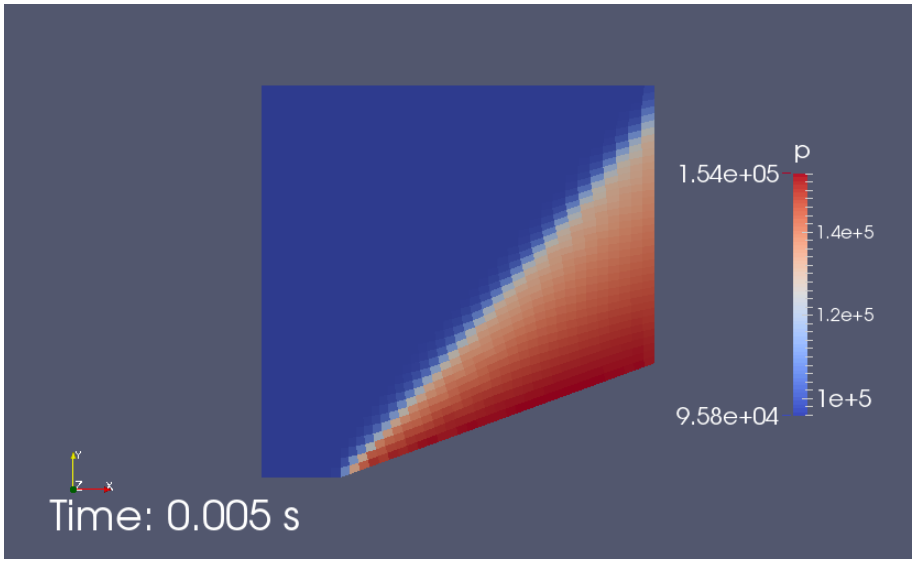


Figure 19: Pressure field for the low-resolution simulation of Mach 1.5 flow over a cone with 20 degree half-angle.

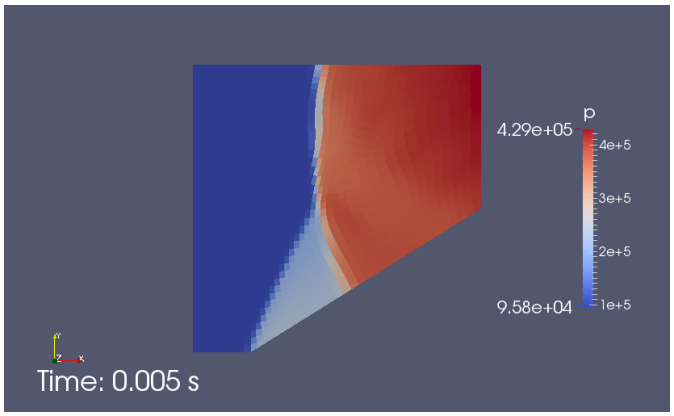
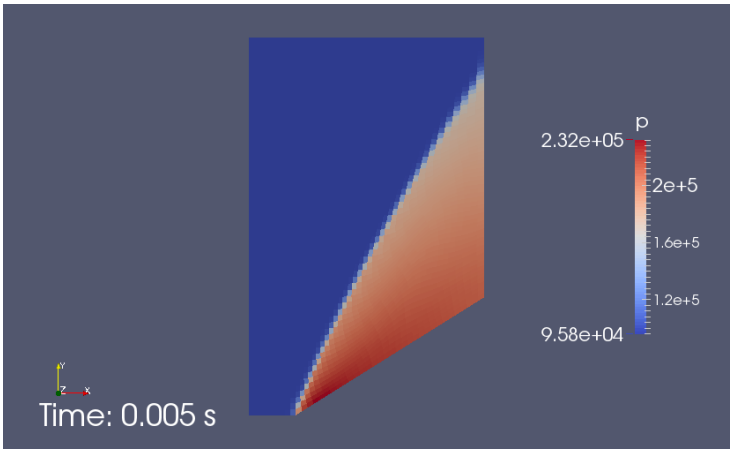
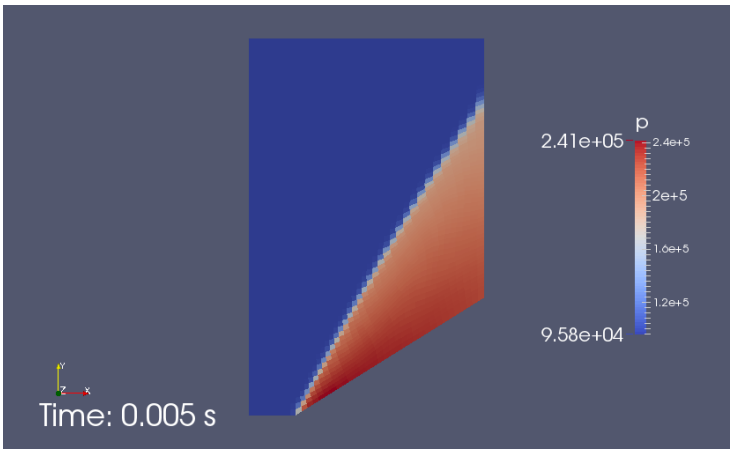


Figure 20: Pressure field for the low-resolution simulation at 5 ms of Mach 1.5 flow over a cone with 32 degree half-angle.



(a) Mach 1.5 inflow

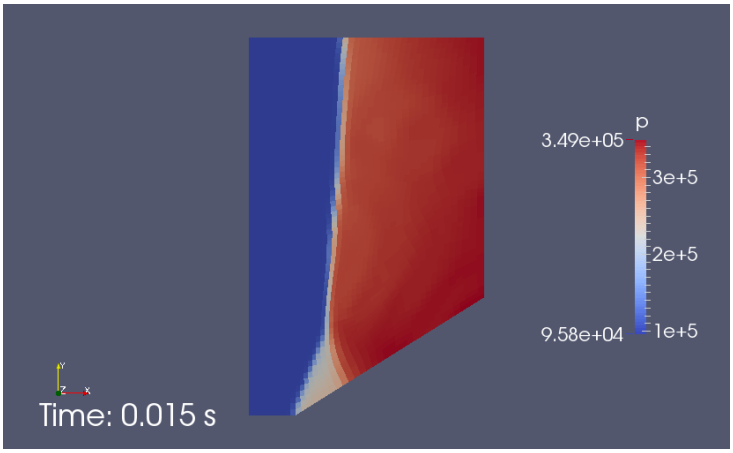


(b) Mach 1.6 inflow

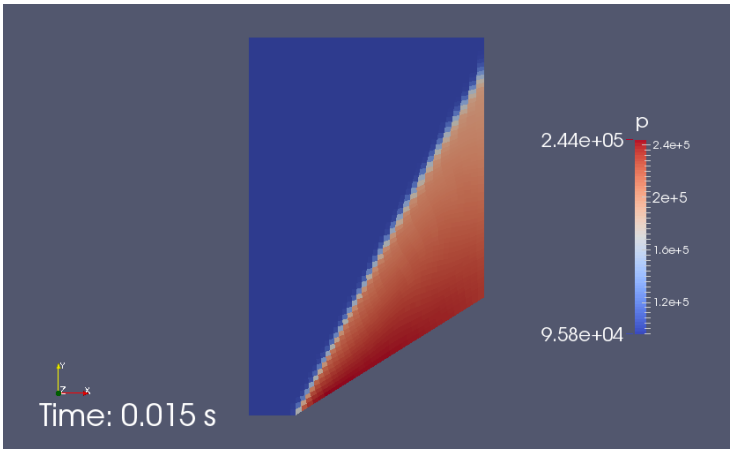
Figure 21: Pressure field for the low-resolution simulation at 5 ms of flow over a cone with 32 degree half-angle in a larger flow domain, $H = 1.6$.

Now, the results are looking better, with the shocks looking quite orderly in each simulation. The Mach 1.6 flow has a straighter shock and a cleaner start at the tip of the cone, such that it looks attached in this fairly low resolution simulation. At this point, we could be tempted to declare victory and head to the Red Room to study the generation of high quality multi-block grids as shown in Figure 7. However, we want to be good students of CFD and shall confirm that the flows really have reached steady state by running the simulations for a longer time. Besides, the simulations are being done in less than a minute each so how much extra effort can it be?

Approximately 5 minutes later, you see the results shown in Figure 22 and you wish that you had left for the Red Room some time earlier. The Mach 1.6 shock looks good and a little straighter, as it should, but the Mach 1.5 case is not showing the desired result. Why, with such a small difference in inflow specification should there be such a big difference? And, why does that difference seem to come from downstream?



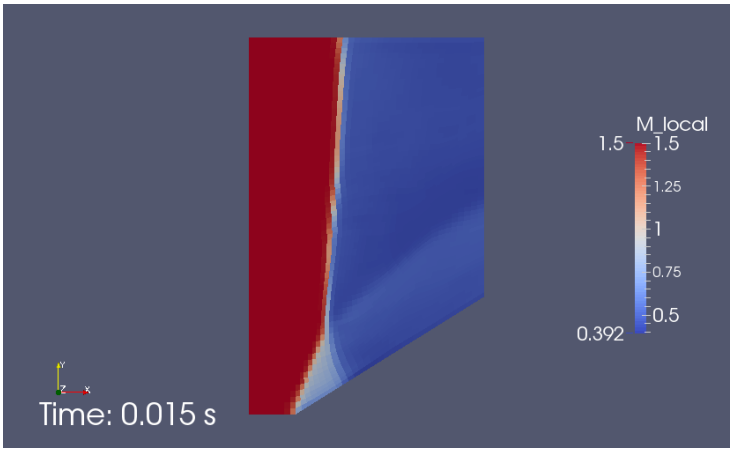
(a) Mach 1.5 inflow



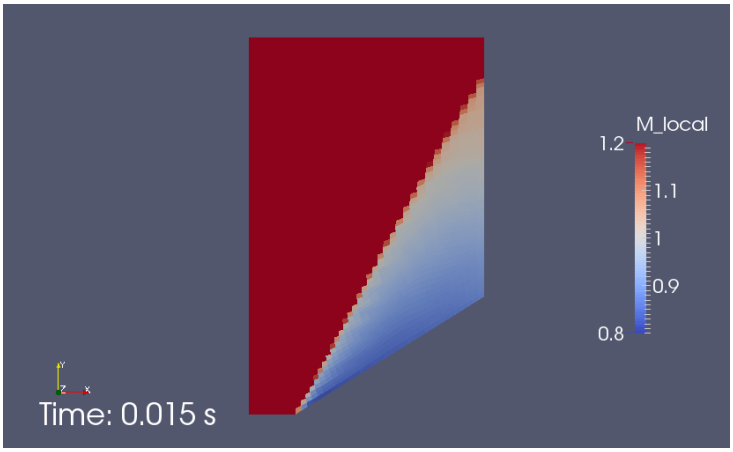
(b) Mach 1.6 inflow

Figure 22: Pressure field for the low-resolution simulation at 15 ms of flow over a cone with 32 degree half-angle in a larger flow domain, $H = 1.6$.

If you ask a tutor at this point, you are likely to be asked: "What does the Mach number look like, especially at the outflow boundary?" In preparation, you use the `--add-mach` option to your `e3post.py` command and produce the plots shown in Figure 23. The Mach number approaching the exit plane for the Mach 1.6 inflow is transonic but the Mach numbers for the Mach 1.5 inflow are very low for the near-normal shock processed flow but, even for the little bit of flow processed by the oblique shock, they are looking to be well below sonic conditions. The simple `ExtrapolateOutBC` applied to the outflow boundary does not handle subsonic flow across it very well at all, and results in the whole simulation not being a good representation of the physical situation. The fix is to alter the flow domain, so that the outflow is mostly supersonic.



(a) Mach 1.5 inflow, The full rnage of M_local is shown.



(b) Mach 1.6 inflow. Note that a partial range of M_local is displayed so as to show the transonic region more clearly.

Figure 23: Mach number field for the low-resolution simulation at 15 ms of flow over a cone with 32 degree half-angle in a larger flow domain, $H = 1.6$.

20 Building a more robust simulation

The fix is very much as you should do in a physical experiment. If a boundary effect is messing with your flow, move that boundary away. Fortunately, this is (usually) easy to do in a numerical simulation. Here, we will add another block to the downstream edge of the original domain and effectively move the outflow further downstream. This extra block (`blk_2` in the following input script) allows the flow to regain supersonic flow conditions before crossing the outflow boundary.

20.1 Input script (.py)

```
# conepe.py
# Simple job-specification making use of parametric capabilities.
# PJ, 25-Jul-2015 -- adapted from the classic cone20, extended

# We can set individual attributes of the global data object.
gdata.dimensions = 2
gdata.axisymmetric_flag = 1

# 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)
Tinf = 1103.0
a = sqrt(1.4*287.1*Tinf) # sound speed in free stream
M = 1.5
ux = M * a
print "ux=", ux
inflow  = FlowCondition(p=95.84e3, u=ux, v=0.0, T=Tinf)

# Define cone/flow-domain geometry
theta = 32 # cone half-angle, degrees
L = 0.8 # axial length of cone, metres
rbase =  L * math.tan(math.radians(theta))
x0 = 0.2 # upstream distance to cone tip
H = 3.0 # height of flow domain, metres

gdata.title = "Mach %.1f flow over a %.1f-degree cone." % (M, theta)
print gdata.title

# Set up two quadrilaterals in the (x,y)-plane by first defining
# the corner nodes, then the lines between those corners.
a = Node(0.0, 0.0, label="A")
b = Node(x0, 0.0, label="B")
c = Node(x0+L, rbase, label="C")
d = Node(x0+L, H, label="D")
e = Node(x0, H, label="E")
f = Node(0.0, H, 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, with particular discretisation.
dx = 1.0/40
nx0 = int(x0/dx); nx1 = int(L/dx); ny = int(H/dx)
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])
# Extend the flow domain
xend = x0+2*L
```

```
blk_2 = Block2D(CoonsPatch(c,Vector(xend,rbase/2),Vector(xend,H),d),
                nni=nx1, nnj=ny,
                fill_condition=initial, label="BLOCK-2")

# Set boundary conditions.
identify_block_connections()
blk_0.bc_list[WEST] = SupInBC(inflow, label="inflow-boundary")
blk_2.bc_list[EAST] = ExtrapolateOutBC(label="outflow-boundary")

# Do a little more setting of global data.
gdata.max_time = 30.0e-3  # seconds
gdata.max_step = 15000
gdata.dt = 1.0e-6
gdata.dt_plot = 1.5e-3
gdata.dt_history = 10.0e-5

sketch.xaxis(0.0, 2.0, 0.5, -0.05)
sketch.yaxis(0.0, 2.0, 0.5, -0.04)
sketch.window(0.0, 0.0, 1.0, 1.0, 0.05, 0.05, 0.17, 0.17)
```

20.2 Final results

For a domain height $H = 2$, Figure 24 shows the Mach number field at the simulation time of 30 milliseconds. This is double the time shown in the short-domain simulations, where the flow was clearly choked. The slightly detached shock from the cone tip is much cleaner but the upper boundary is still showing a strong effect with a near-normal shock processing the upper part of the inflow. The slightly-subsonic values of mach number immediately behind the detached shock are clearly shown in light blue.

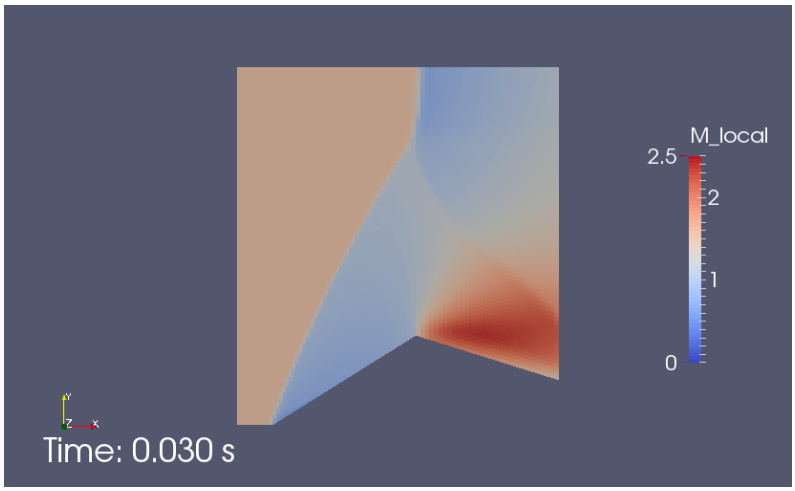


Figure 24: Mach field for the low-resolution simulation at 30 ms of Mach 1.5 flow over a cone with 32 degree half-angle. Flow domain height $H = 2$.

Since we've made all this effort at getting the downstream boundary condition behaving well, we should take advantage of the parametric modelling once more and finish the job by raising the flow domain height simply by setting $H = 3$ and running the simulation

again. This time, the flow field in Figure 25 appears to be clean and mostly free from obvious boundary induced problems. The `ExtrapolateOutBC` boundary has mostly a clear supersonic flow crossing it and can probably be trusted to behave well. This would be the correct time to declare victory, however, the tutor now points out that the expansion radiating from the corner at the end of the conical surface is probably affecting the whole of the subsonic region behind the curved shock.

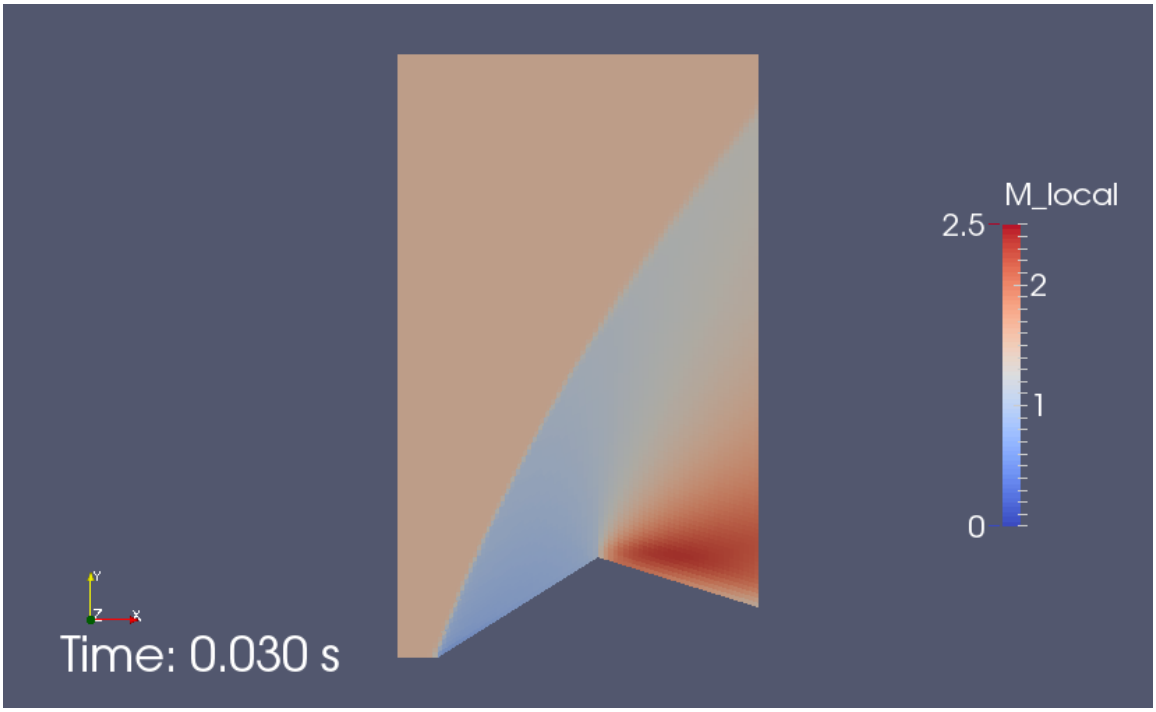


Figure 25: Mach field for the low-resolution simulation at 30 ms of Mach 1.5 flow over a cone with 32 degree half-angle. Flow domain height $H = 3$.

Part IV
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-species 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. The Rutowski simulation (Sec.48) is probably the most sophisticated example with respect to phemonological models. It exercises just about every capability the code has, including radiation energy exchange and thermal nonequilibrium, in a simulation of a radiating flow of argon over a sphere.

21 Oblique shock boundary layer interaction.

With some confidence that the code is working correctly and a knowledge of the manual postprocessing arrangements shown in the previous example, you are ready to try to simulate a flow that has is a bit more "realistic".

This is an example that introduces viscous effects but retains a very simple geometric arrangement for the flow boundaries. It is simple to model but immediately shows the computational demands that result from requesting an increase in "flow fidelity". Consider the Mach 2 flow of ideal air over a flat plate, as shown below in Figure 26. This flow image was taken as part of an experimental campaign [12] in a continuous flow wind tunnel at MIT. The flow is from left to right in the image and the plate with the boundary layer of interest in the lower boundary and there is a viscous-interaction shock propagating from the sharp edge of the plate (bottom left of the image) and across the flow. There is another plate at a small angle of attack forming the upper surface of the test region. The leading-edge of this shock-generator plate is out of view but the generated shock is seen entering the field of view at the top-left of the image and reflection from the bottom plate at approximately 49 mm from the leading edge. The shock reflection results in an overall pressure ratio of 1.4 across the interaction region. The boundary layer on the plate can be seen thickening to the point of intersection with the reflected shock and then thinning again past the interaction point. The case for a pressure ratio of 1.4 was chosen for simulation because, as noted in the original report [12], shear-stress data indicated that the boundary layer remained laminar after the interaction.

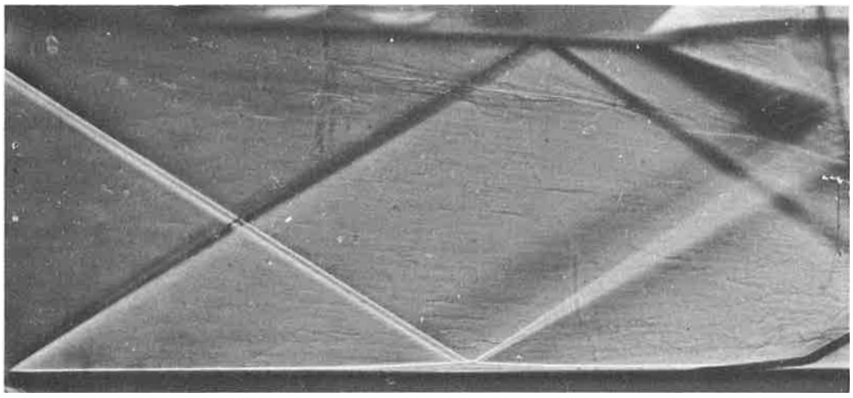


Figure 26: Schlieren image of the Mach 2 flow over a flat plate taken from Fig.6b in Reference [12].

Although the behaviour of laminar compressible-flow boundary layers on flat plates are well predicted via simple theories, the addition of an impinging shock significantly more difficult to analyse manually. The flow complexity significantly while the defining flow geometry remains very simple.

21.1 Input script (.py)

Figure 27 shows the region, as modelled for simulation. The flat plate is truncated at the length seen in the experimental flow image even though the actual plate extended for 8 inches in the experiment. Also the shock generator plate is modelled as an idealized, inviscid wall, even though the real shock generator would have had a boundary layer and associated viscous interaction at its leading edge. It has been convenient to apply a slip-wall boundary condition at the shock generator surface so that the calculation to estimate the deflection angle for the specified pressure rise across the reflected shock uses just the usual oblique-shock relations for an ideal gas. Using the cfpylib ideal-flow functions in the following script and a minute of trial and error fiddling, the shock generator deflection angle can be estimated as being 3.09^o.

```
# double_oblique_shock.py
"""
Estimate pressure rise across a reflected oblique shock.
PJ, 01-May-2013
"""
print "Begin..."
from cfpylib.gasdyn.ideal_gas_flow import *
import math
M1 = 2.0
p1 = 1.0
g = 1.4
print "First shock: ",
delta1 = 3.09 * math.pi/180.0
beta1 = beta_obl(M1,delta1,g)
p2 = p2_p1_obl(M1,beta1,g)
M2 = M2_obl(M1,beta1,delta1,g)
print "beta1=", beta1, "p2=", p2, "M2=", M2

print "Reflected shock:",
delta2 = delta1
beta2 = beta_obl(M2,delta2,g)
p3 = p2 * p2_p1_obl(M2,beta2,g)
M3 = M2_obl(M2,beta2,delta2,g)
print "beta2=", beta2, "p3=", p3, "M3=", M3

print "Done."
```

In the input script, geometric dimensions of the flow region and plate are simply scaled from the flow image and the shock location identified in the associated pressure and skin-friction plot. The flow region is modelled as a box with straight-line boundary segments and, although the geometry is particularly simple, we use a three SuperBlock2D objects to split the region into 20 individual blocks. This is done so that these blocks may be assigned to several processors of a multicore machine and we don't have to wait quite so long for our simulation to run.

Using data in the original report [12], the free-stream conditions for Fig.6b with $Re_{x-shock} = 2.96 \times 10^5$, can be estimated to be $p_\infty = 6.205\,\text{kPa}$, $T_\infty = 164.4\,\text{K}$ and $u_\infty = 514\,\text{m/s}$ for ideal air with R_{gas}=287 J/kg·K and γ=1.4.

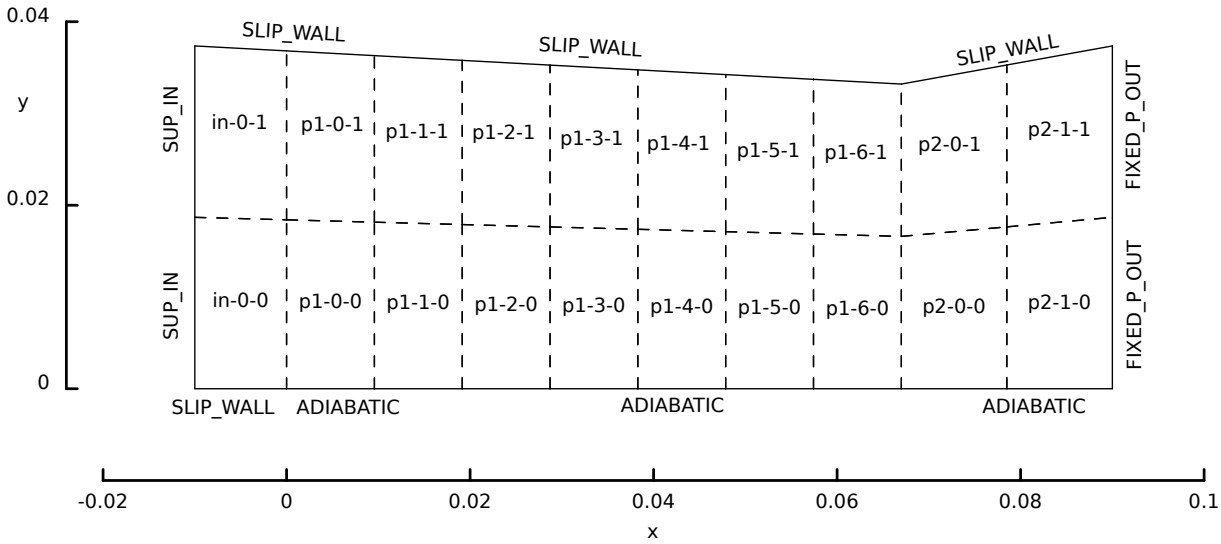


Figure 27: Schematic view of the simulated flow region for the shock-wave interaction with a laminar boundary layer.

```
# swlbli.py
# PJ, 01-May-2013
# Model of Hakkinen et al's 1959 experiment.

gdata.title = "Shock-wave laminar-boundary-layer interaction."
print gdata.title
# Conditions to match those of Figure 6: pf/p0=1.4, Re_shock=2.96e5
p_inf = 6205.0 # Pa
u_inf = 514.0 # m/s
T_inf = 164.4 # degree K

# Accept defaults giving perfect air (R=287 J/kg.K, gamma=1.4)
select_gas_model(model='ideal gas', species=['air'])
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf)

mm = 1.0e-3 # metres per mm
L1 = 10.0*mm; L2 = 90.0*mm; L3 = 67*mm
H1 = 37.36*mm
alpha = 3.09*math.pi/180.0 # angle of inviscid shock generator
tan_alpha = math.tan(alpha)
a0 = Vector(-L1, 0.0); a1 = a0+Vector(0.0,H1) # leading edge of shock generator
b0 = Vector(0.0, 0.0); b1 = b0+Vector(0.0,H1-L1*tan_alpha) # start plate
c0 = Vector(L3, 0.0); c1 = c0+Vector(0.0,H1-(L1+L3)*tan_alpha) # end shock generator
d0 = Vector(L2, 0.0); d1 = d0+Vector(0.0,H1) # end plate

# The following lists are in order [N, E, S, W]
rcf = RobertsClusterFunction(1,1,1.1)
ni0 = 20; nj0 = 80 # We'll scale discretization off these values
factor = 4
ni0 *= factor; nj0 *= factor

inlet = SuperBlock2D(CoonsPatch(a0,b0,b1,a1),
                     nni=ni0, nnj=nj0, nbi=1, nbj=2,
                     bc_list=[SlipWallBC(),None,SlipWallBC(),SupInBC(inflow)],
                     cf_list=[None,rcf,None,rcf],
                     fill_condition=inflow, label="in")
plate1 = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                      nni=ni0*7, nnj=nj0, nbi=7, nbj=2,
                      bc_list=[SlipWallBC(),None,AdiabaticBC(),None],
                      cf_list=[None,rcf,None,rcf],
                      fill_condition=inflow, label="p1")
plate2 = SuperBlock2D(CoonsPatch(c0,d0,d1,c1),
                      nni=ni0*2, nnj=nj0, nbi=2, nbj=2,
                      bc_list=[SlipWallBC(),FixedPOutBC(6205.0),AdiabaticBC(),None],
                      cf_list=[None,rcf,None,rcf],
```

```
                    fill_condition=inflow, label="p2")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
gdata.max_time = 5.0*L2/u_inf  # in flow lengths
gdata.max_step = 200000
gdata.dt = 1.0e-8
gdata.dt_plot = gdata.max_time/10

sketch.xaxis(-0.020, 0.100, 0.020, -0.010)
sketch.yaxis(0.000, 0.040, 0.020, -0.004)
sketch.window(-0.02, 0.0, 0.10, 0.12, 0.05, 0.05, 0.25, 0.25)
```

21.2 Running the simulation

To get the simulation started, try the following commands:

```
$ cd ~/cfcfd3/examples/eilmer3/2D/hakkinen-SWLBLI/
$ ./prep.sh
$ ./run.sh
$ tail -f run.transcript
```

You should see the usual console output of a simulation proceeding to take time steps and reporting it's progress toward reaching a final time. If you are working on a 4-core machine, go and have dinner and return in about 5 hours to check the state of the simulation. The grid and initial solution are created with the `prep` script and the time-evolution of the flow field is then computed for about 876 μs (with 11186 time steps being required). At the end of this pass of the simulation, it turns out that the separation region is still slightly evolving as indicated by small movements of the waves propagating from that region. We restart the calculation and run it to twice the original value of `max_time`. This is achieved by manually editing the `swlbli.control` file as described in Section 3.6 and setting `max_time = 1.750972e-03` and `dt = 8.000000e-08` then running the command:

```
$ ./run-2.sh
```

The commands invoke the shell scripts displayed in subsection 21.4.

21.3 Results

Figure 28 shows some of the flow field data at t=1.75 ms after flow start. The magnitude of the gradients of density (Fig. 28c) are also shown as an approximation to the schlieren

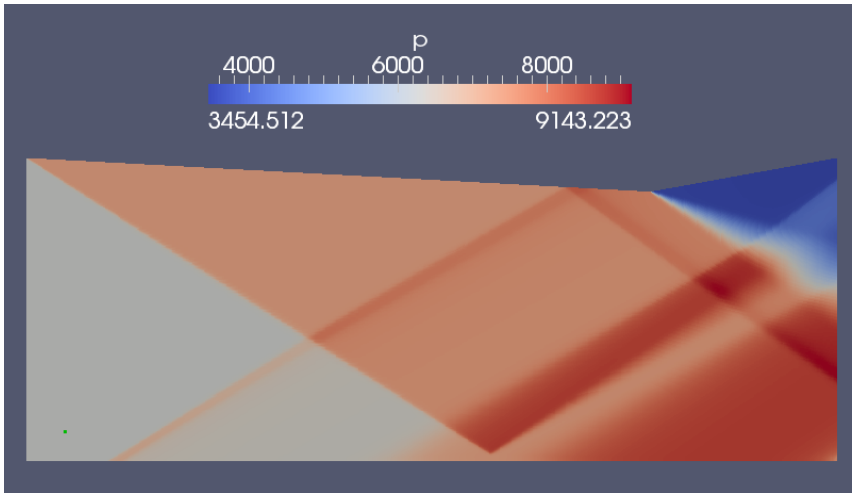
image of Figure 26. The image of the pressure clearly shows the waves propagating from the leading-edge viscous interaction and their reflection from the shock generator. As expected, the boundary layer is not directly evident in the pressure field but shows up clearly in the temperature field. The more gradual compression, as the boundary layer approaches the incident shock, is evident as a much as a much broader band in the pressure field. This is followed by an expansion and then a recompression. All of these waves are most clearly shown in the gradient of density field. The shock, expansion and recomression shock from the leading-edge viscous interaction are displayed more distinctly and the convergence of the gradual compressions becomes clear. The structure of expansion fans also appears more clearly in this gradient field than in the pressure or temperature fields.

The real proof of success is in comparison with the experimental data. Figure 29 shows the pressure and shear-stress along the plate. The simulation has done a reasonable job of estimating the pressure distribution right through the separation zone. Features that look a little wrong include the viscous interaction region at x=0, which is a bit extended because of lack of resolution at the start of the boundary layer, however, doubling the grid resolution (factor=8) tightens up solution in this region. Also, there is an artificial drop in pressure at the right end of the simulation domain where the boundary layer exits the flow domain but this is of no concern because the flat plate used in the experiment was more than twice the length of this simulated version. This behaviour is grid independent.

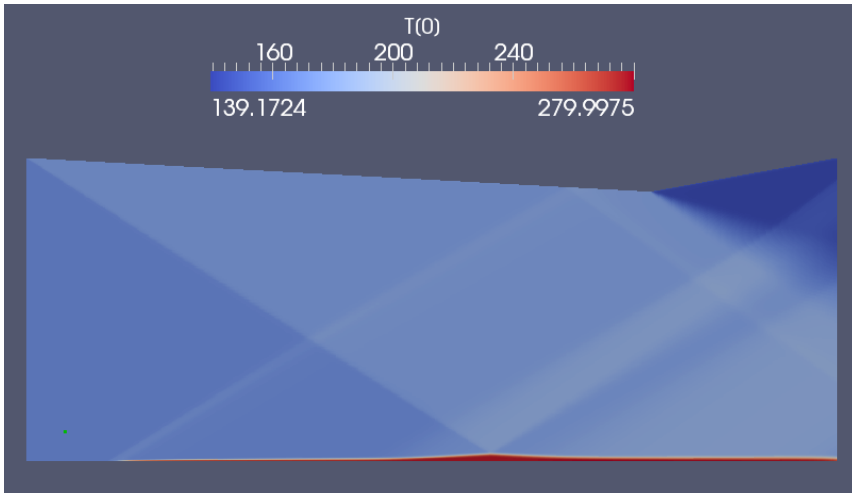
The simulation has done a reasonable job on the shear stress, which has been computed from the field data using the script in Section 21.5. This quantity is difficult to compute and difficult to measure so it is reassuring that both sets of data line up nicely with the Blasius value in the boundary layer leading into the interaction region. After the interaction region, the computed values recover to the Blasius level just before rising toward the end of the flow domain. This is, again, the interaction with the outflow boundary condition and would be removed from view if the full length of the plate was simulated. The only discernible difference with increasing grid resolution (from factor=4 to factor=8) is that the early development of the boundary layer moves a little closer to the Blasius behaviour.

21.4 Shell scripts

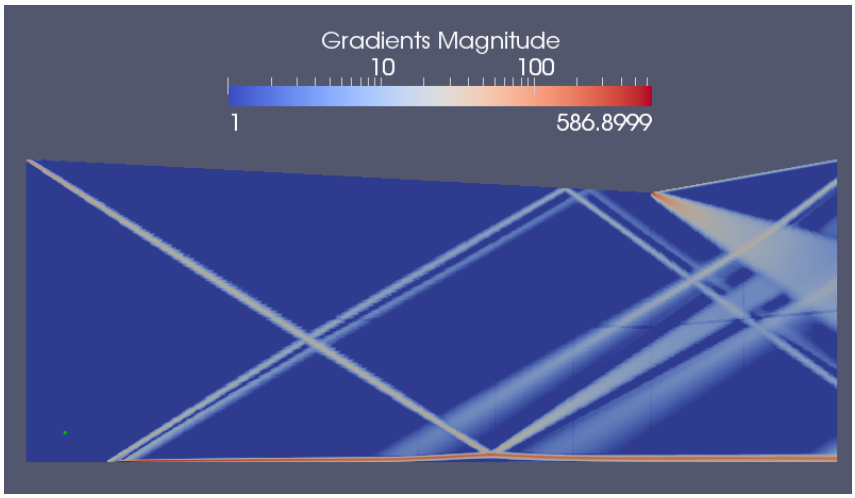
```
#! /bin/bash
# prep.sh
e3prep.py --job=swlbli --do-svg
e3loadbalance.py --job=swlbli -n 4
e3post.py --job=swlbli --tindx=0 --vtk-xml

echo "At this point, we should have a grid."
echo "Use run.sh next"
```



(a) Pressure field.

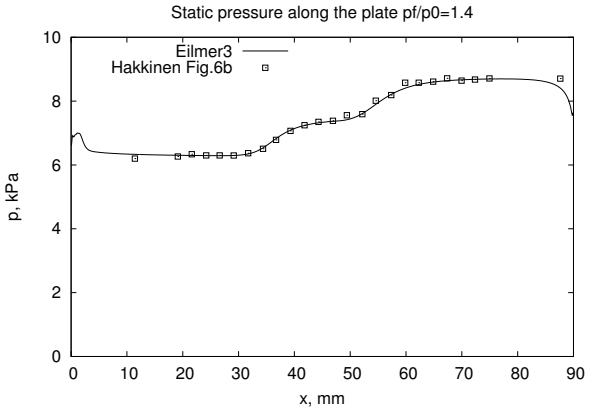


(b) Temperature field.

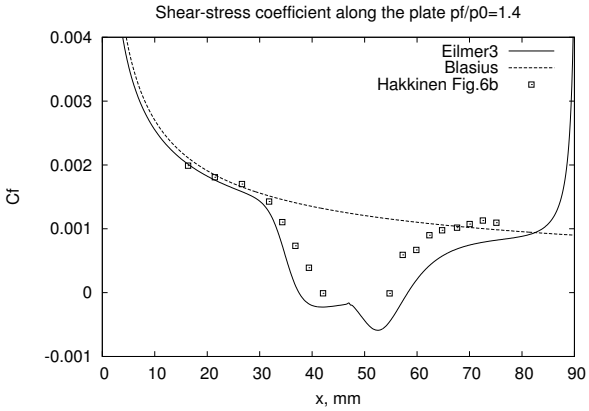


(c) Gradient of density field.

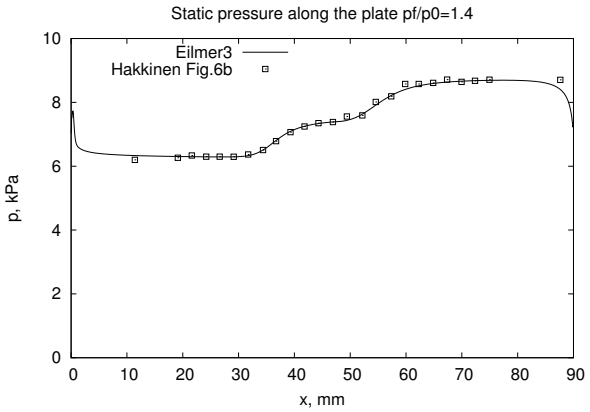
Figure 28: Computed flow field at t=1.75 ms.



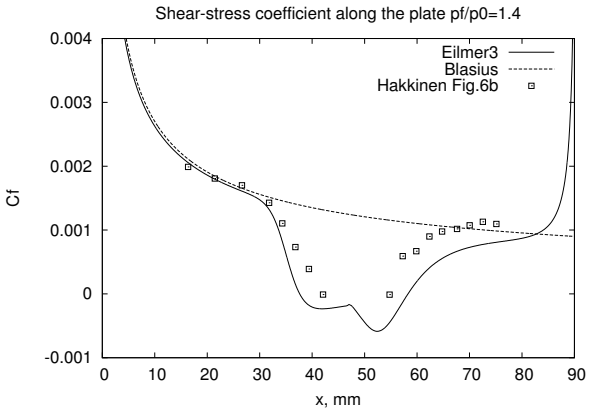
(a) Pressure (factor=4).



(b) Shear stress (factor=4).



(c) Pressure (factor=8).



(d) Shear stress (factor=8).

Figure 29: Distribution of pressure and shear along the plate at t=1.75 ms.

```
#! /bin/bash
# run.sh
module load openmpi-x86_64
date
mpirun -np 4 e3mpi.exe --job=swlbli --mpimap=swlbli.mpimap --run > run.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

```
#! /bin/bash
# run-2.sh
# restart the calculation where run.sh left it
module load openmpi-x86_64
date
mpirun -np 4 e3mpi.exe --job=swlbli --mpimap=swlbli.mpimap --run --tindx=10 > run-2.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

```
#! /bin/bash
# post.sh
e3post.py --job=swlbli --tindx=all --vtk-xml --add-mach

echo "At this point, we should have data to view"
```

```
#! /bin/bash
# plot.sh
e3post.py --job=swlbli --tindx=last --add-mach --output-file=bl.data \
    --slice-list="2,:,0,0;4,:,0,0;6,:,0,0;8,:,0,0;10,:,0,0;12,:,0,0;14,:,0,0;16,:,0,0;18,:,0,0"

gnuplot pressure.gnuplot
gnuplot shear.gnuplot
echo "At this point, we should have pictures to view"
```

21.5 Postprocessing for shear stress

The script below uses the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. It extracts the data for the cell nearest to the flat plate and uses that data to compute the expected shear stress on the plate.

```
#! /usr/bin/env python
# compute_shear.py
#
# Pick up the simulation data at the last simulated time
# and compute an estimate of the shear-stress coefficient.
#
# PJ, 08-May-2013
```

```
import sys, os
from math import sqrt
from e3_flow import read_all_blocks
job = "swlbli"
nb = 20
pick_list = [2, 4, 6, 8, 10, 12, 14, 16, 18] # blocks against plate
rho_inf = 0.1315 # kg/m**3
u_inf =  514.0 # m/s
T_inf = 164.4 # K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'Air')

print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx
print "Begin: Pick up data."
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute shear stress for cell-centres along plate surface"
outfile = open("shear.data", "w")
outfile.write("# x(m)  tau_w(Pa)  Cf   y_plus\n")
for ib in pick_list:
    j = 0 # plate is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k];
        y = flow[ib].data['pos.y'][i,j,k]
        rho = flow[ib].data['rho'][i,j,k];
        u1 = flow[ib].data['vel.x'][i,j,k]
        mu = flow[ib].data['mu'][i,j,k]
        dudy = (u1 - 0.0) / y # Assuming that the wall is straight down at y=0
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * y * rho / mu
        Rex = rho_inf * u_inf * x / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        outfile.write("%f %f %f %f %f\n" % (x, tau_w, Cf, Cf_blasius, y_plus))
        print "x=", x, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus
outfile.close()
print "Done"
```

21.6 Notes

- The influence of the flat plate boundary layer on the pressure in the region near the plate is small but measureable. With a free-stream pressure of 6.205 kPa specified at the inflow plane, we see 6.28 kPa in the pressure data leading into the shock-interaction region. For the free-stream conditions used, the displacement thickness of a simple flat-plate boundary layer would be expected to be approximately 0.112 mm at 25 mm from the leading edge of the plate. If this displacement effect could be modelled as a straight wedge deflecting the inviscid free-stream, the corresponding oblique shock would have a static pressure ratio of 1.0146. This gives an expected pressure of 6.295 kPa in the boundary-layer external flow leading into the shock interaction, quite close to the simulation value.

22 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[10] to verify the implementation of the viscous and axisymmetric terms in the code.

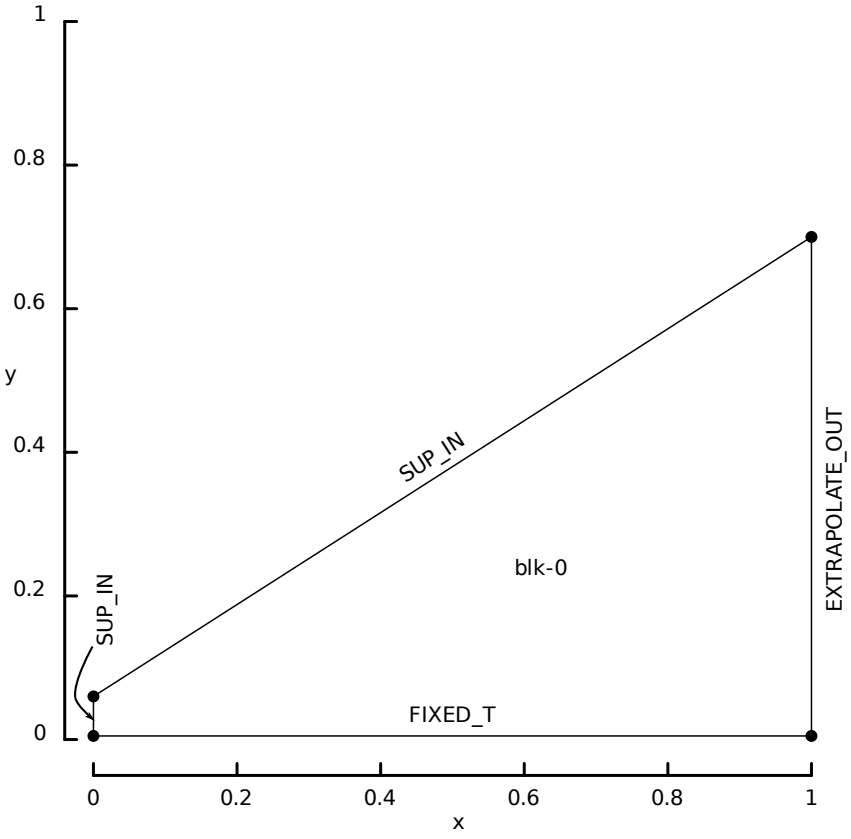


Figure 30: Flow domain for viscous flow along a cylinder.

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 30 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 31 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.

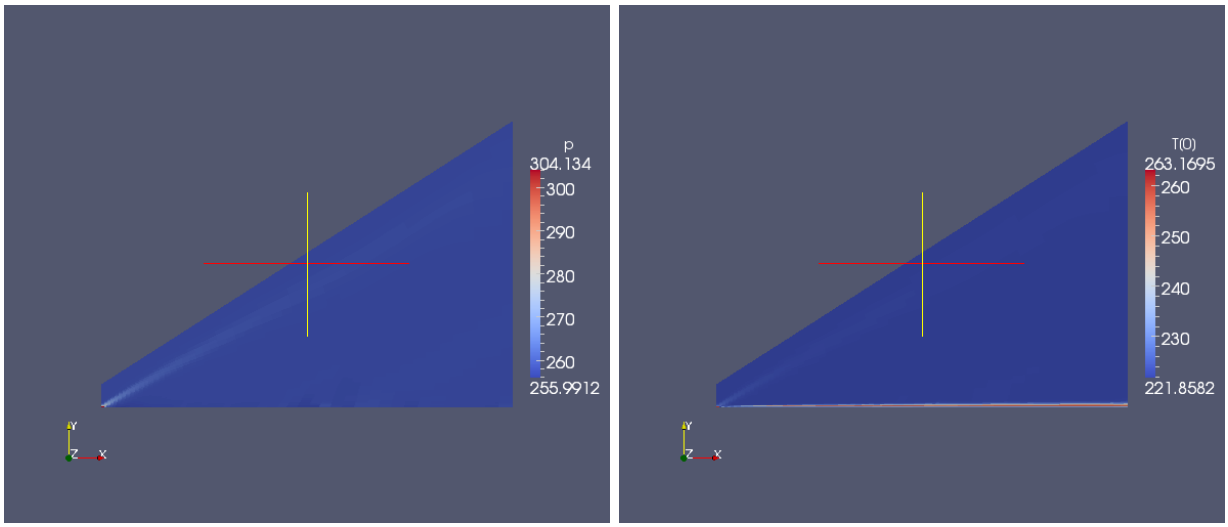


Figure 31: Pressure and temperature fields for viscous flow along a cylinder.

Figure 32 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.

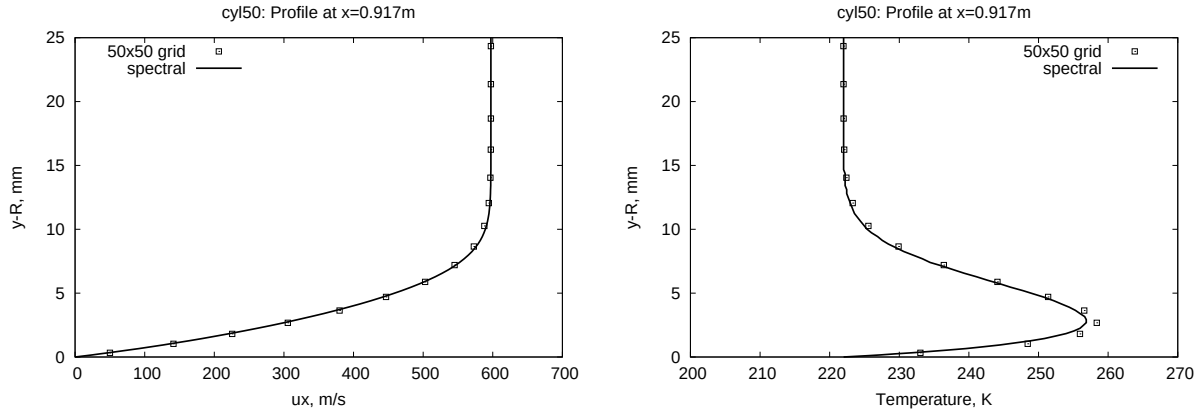


Figure 32: Velocity and temperature profiles at $x = 0.916\,\mathrm{m}$ for viscous flow along a cylinder.

22.1 Input script (.py)

```
## \file cyl50.py
## \author PJ
## \version 14-Aug-2006 updated from Tcl script
## \version 18-Jan-2010 updated for Eilmer3
## \version 14-Apr-2013 use MPI with 4 procs to speed things up.

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)]
rcfns = RobertsClusterFunction(1,0,1.1)
rcfew = RobertsClusterFunction(1,0,1.01)

# Assemble the block from the geometry, discretization and boundary data.
blk = SuperBlock2D(psurf=make_patch(north, east, south, west, grid_type="AO"),
                   nni=50, nnj=50, nbi=2, nbj=2,
                   bc_list=bndry_list, cf_list=[rcfns, rcfew, rcfns, rcfew],
                   fill_condition=inflow)

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.2
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)
```

22.2 Shell scripts

```
#! /bin/sh
# cyl50_run.sh
e3prep.py --job=cyl50 --do-svg
time mpirun -np 4 e3mpi.exe --job=cyl50 --run

echo "At this point, we should have a new solution"
echo "Run cyl50_post.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 pt 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 pt 4, \
     "cyl50_dimensional.dat" using (\$3):(\$1-0.005)*1000.0 title "spectral" with lines ls 1
EOF
```

22.3 Notes

- None

23 Hypersonic flow over a concave surface.

This is one of the two hypersonic flows studied by Mohammadian [13] in the Imperial College gun tunnel, more than 4 decades ago. The gun tunnel was operated with a total pressure of 1600 psia (11.03 MPa) and a total temperature of 1300 K with a Mach 12.25 contoured nozzle.

Since we don't have full information on the tunnel and its operating condition, we have to make a few assumptions. First, we assume that the nozzle produced a parallel and uniform flow of ideal air with free stream conditions $p = 63.43$ Pa, T=41.92 K and $u = 1.59$ km/s. Using Sutherland's expression for the viscosity of air, we estimate the viscosity of the free stream to be $\mu = 2.593 \times 10^{-6}$ Pa.s. Taking a length scale $L = 1$ inch, we compute a Reynolds number $Re_L = 0.86 \times 10^5$, which is the same as that reported in the original paper [13].

23.1 Input script (.py)

Figure 33 shows the flow region, as modelled for simulation. The region is very simple but we have divided it into 22 blocks so that the computational load can be shared across a number of CPU cores.

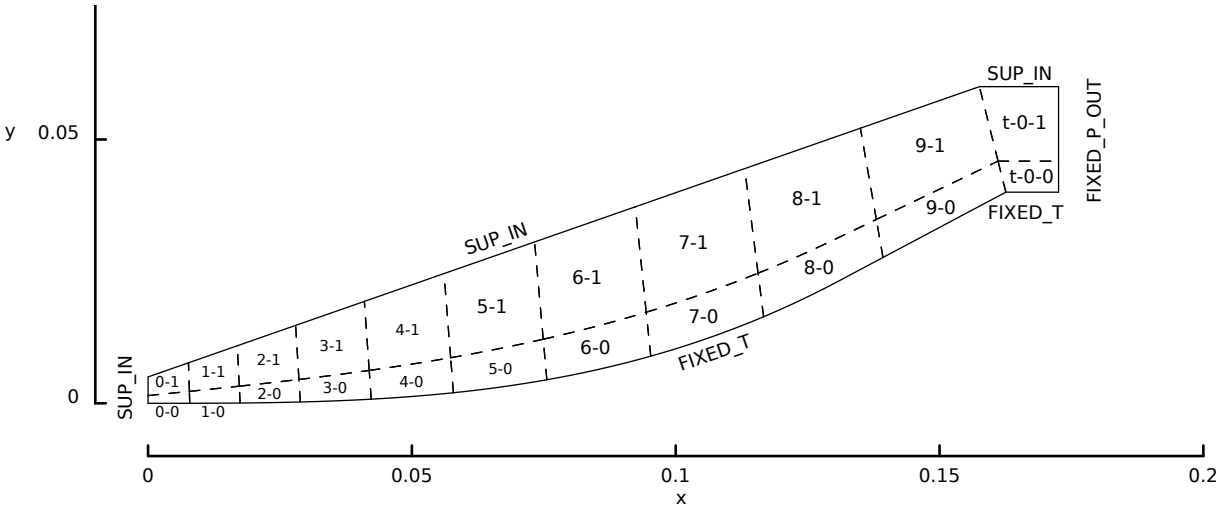


Figure 33: Schematic view of the simulated flow region for the hypersonic flow over Mohammadian's concave ramp.

The concave ramp is defined by

$$y = \frac{1}{150} x^3$$

where x is in inches and the surface angle is less than 28^o. Beyond that point, the ramp surface is assumed straight until the corner at $x = 6.4$ inches. The ramp surface temperature was assumed to be a constant 296 K.

```
# cubic-ramp.py
# PJ, 10-Aug-2013
# Model of Mohammadian's concave surface experiment.

gdata.title = "Mohammadian cubic ramp."
print gdata.title
# Conditions to match those reported in JFM paper.
p_inf = 66.43 # Pa
u_inf = 1589.8 # m/s
T_inf = 41.92 # degree K
select_gas_model(model='ideal gas', species=['air'])
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf)
initial = FlowCondition(p=p_inf/5, u=0, T=T_inf)
T_wall = 296.0 # degree K --assumed cold-wall temperature

def ramp(t):
    """
    Parametric definition of ramp profile in xy-plane.
    """
    m_per_inch = 25.4e-3 # metres per inch
    alpha = 28.0*math.pi/180.0 # angle of final straight section
    tan_alpha = math.tan(alpha)
    x_join_inch = math.sqrt(50.0*tan_alpha)
    y_join_inch = x_join_inch**3 / 150.0
    x_inch = 6.4 * t
    if x_inch < x_join_inch:
        y_inch = x_inch**3 / 150.0
    else:
        y_inch = y_join_inch + (x_inch - x_join_inch) * tan_alpha
    return (x_inch*m_per_inch, y_inch*m_per_inch, 0.0)

mm = 1.0e-3 # metres per mm
x,y,z = ramp(0.0); a0 = Vector(x,y,z); a1 = a0+Vector(0.0,5*mm) # leading edge
x,y,z = ramp(1.0); b0 = Vector(x,y,z); b1 = b0+Vector(-5*mm,20*mm) # downstream end
c0 = b0+Vector(10*mm,0.0); c1 = Vector(c0.x,b1.y) # end of model

rcfx = RobertsClusterFunction(1,0,1.2)
rcfy = RobertsClusterFunction(1,0,1.1)
ni0 = 200; nj0 = 40 # We'll scale discretization off these values
factor = 2.0
ni0 = int(ni0*factor); nj0 = int(nj0*factor)

wedge = SuperBlock2D(make_patch(Line(a1,b1),Line(b0,b1),
                                PyFunctionPath(ramp),Line(a0,a1)),
                     nni=ni0, nnj=nj0, nbi=10, nbj=2,
                     bc_list=[SupInBC(inflow),None,
                              FixedTBC(T_wall),SupInBC(inflow)],
                     cf_list=[rcfx,rcfy,rcfx,rcfy],
                     fill_condition=inflow, label="wedge")
tail = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                    nni=int(ni0/10), nnj=nj0, nbi=1, nbj=2,
                    bc_list=[SupInBC(inflow),FixedPOutBC(p_inf/5),
                             FixedTBC(T_wall),None],
                    cf_list=[None,rcfy,None,rcfy],
                    fill_condition=initial, label="tail")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
gdata.max_time = 1.0e-3 # long enough for several flow lengths
gdata.max_step = 2000000
gdata.dt = 1.0e-9
gdata.dt_plot = 0.1e-3

sketch.xaxis(0.0, 0.20, 0.05, -0.010)
sketch.yaxis(0.0, 0.20, 0.05, -0.010)
sketch.window(0.0, 0.0, 0.20, 0.20, 0.05, 0.05, 0.25, 0.25)
```

23.2 Running the simulation

In terms of required computer time, this simulation is fairly demanding, taking more than 12 hours on a 4-core workstation. The job scipts submitted to the batch system are shown below. Note that the preparation script sets up the mapping of the full set of 22 blocks to fit onto 4 MPI tasks.

```
#! /bin/bash
# prep.sh
e3prep.py --job=cubic-ramp --do-svg
e3loadbalance.py --job=cubic-ramp -n 4
e3post.py --job=cubic-ramp --tindx=0 --vtk-xml

echo "At this point, we should have a grid."
echo "Use run.sh next"
```

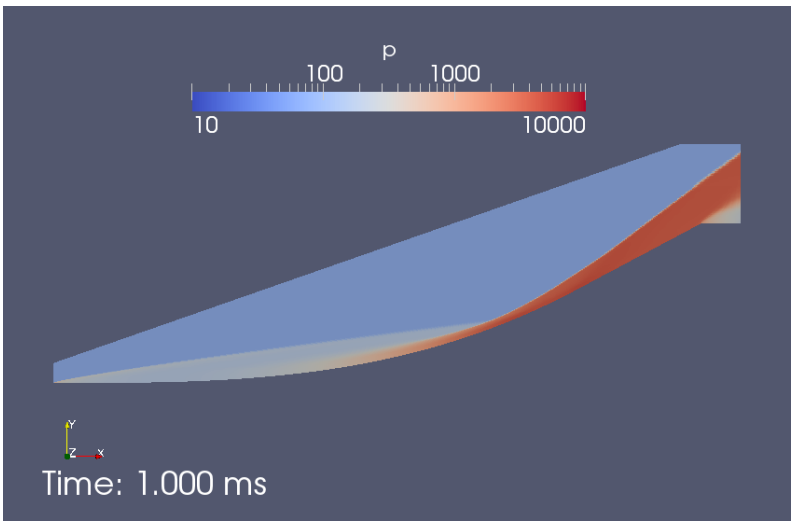
```
#! /bin/bash
# run.sh
# module load openmpi-x86_64
date
mpirun -np 4 e3mpi.exe --job=cubic-ramp --mpimap=cubic-ramp.mpimap --run > run.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

23.3 Results

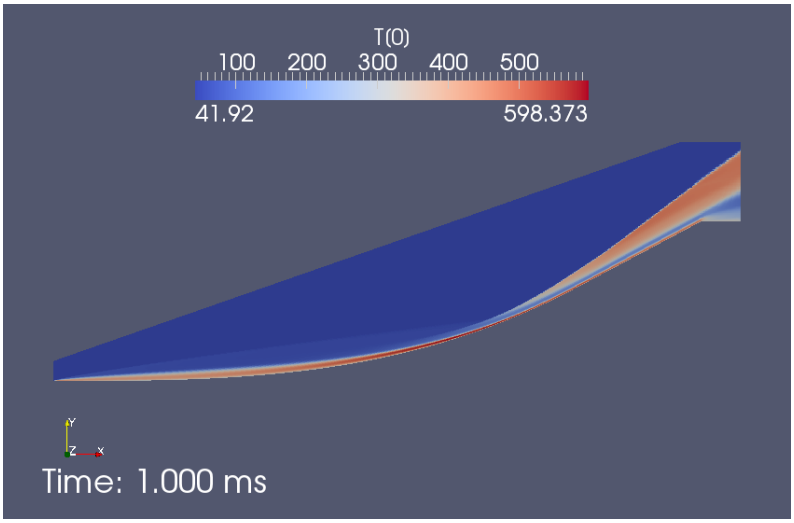
Figure 34 shows some of the flow field data at t=1 ms after flow start. This is sufficient time for the flow to reach steady state.

The logarithmic pressure field shows the leading-edge interaction shock propagating into the free-stream flow but then intersecting the ramp about two-thirds of the way along its length. The gradually increasing pressure can also be seen in the near-surface region, leading up to this intersection.

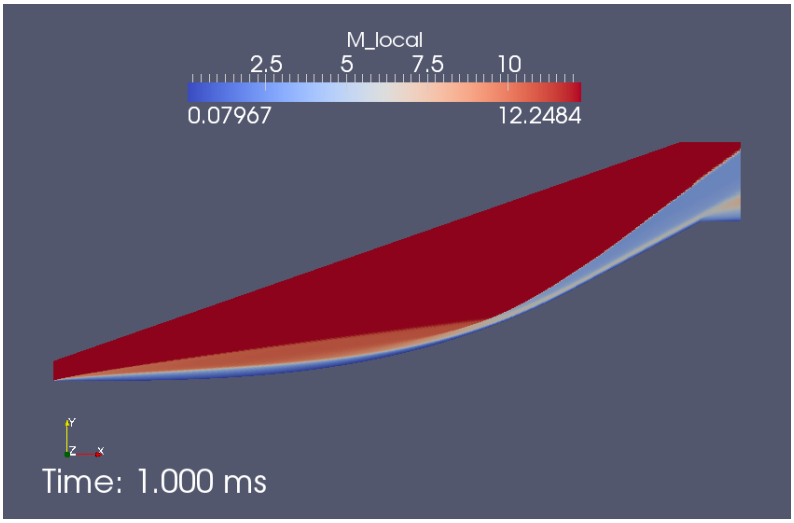
The temperature field shows more clearly the boundary layer that grows along the concave surface. It also shows the compression of the boundary-layer flow as the surface curves up. The oblique shock propagating up over the top of the ramp is a combination of the reflected lead-edge interaction shock and the coalesced compression waves from the curved ramp.



(a) Pressure field.



(b) Temperature field.



(c) Mach number.

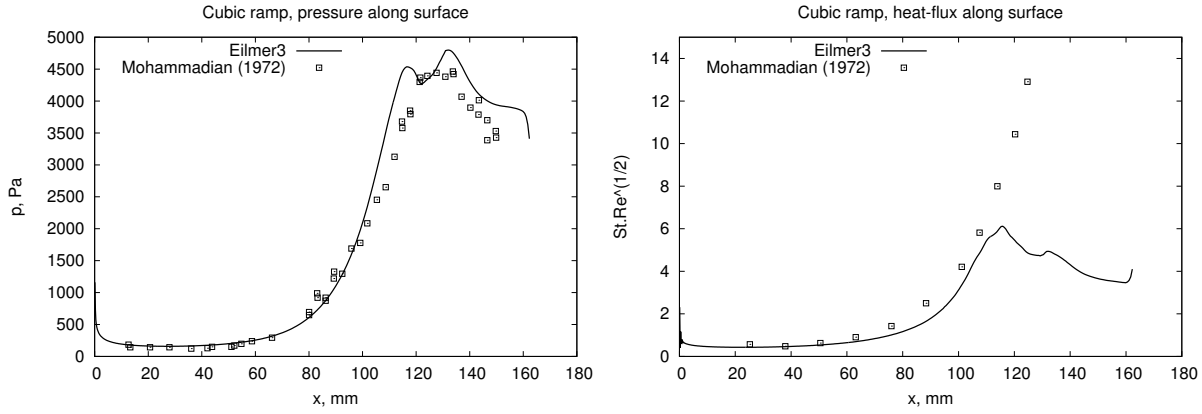
Figure 34: Computed flow field at t=1 ms.

Although the computed flow field looks plausible, the real proof of success of the simulation is in comparison with the experimental data. Figure 35 shows the pressure and heat-transfer along the surface of the ramp. The simulation has done a good job of estimating the pressure distribution for the early part of the ramp, and does a fair job all the way up to the sharp corner at the top of the ramp, although there appears to be a mismatch in the location of this corner (indicated by the sharp drop in pressure at the right end of each data set).

The simulation has also done a good job on the heat transfer estimate, which has been computed from the field data using the script in Section 23.4. Mohammadian has not provided dimensional data in the original paper so we have normalized the simulation data is the same way as can be best guessed (with the assistance of Andrew Knisely, University of Illinois). First, we compute local heat transfer from the normal gradient of temperature at the ramp surface, followed by Stanton number as

$$St = \frac{q}{\rho U C_p (T_0 - T_w)}$$

and then plot the combination $St.Re_x^{\frac{1}{2}}$, to remove the singularity in heat transfer at the leading edge of the ramp. Agreement for the early part of the plate to 110 mm is excellent, so we can have some confidence in the codes ability to model hypersonic viscous interactions, however, what happens beyond that point is not captured by this purely two-dimensional simulation.



(a) Pressure.

(b) Heat transfer.

Figure 35: Distribution of pressure and heat transfer along the concave ramp. Simulation data is recorded at t=1 ms into the simulation. Experimental data is from Ref. [13].

The boundary-layer thickness, as identified by the outer edge of the maximum density gradient seen in a schlieren image was also provided by Mohammadian. Figure 36 shows the density gradient field of the simulated flow and, from this image, corresponding points

were measured manually (with the assistance of the `g3data` program). These boundary-layer thickness data are shown in Figure 37, together with the experimental values from Ref.[13]. Although there is some scatter in the simulation-derived data, comparison is good.

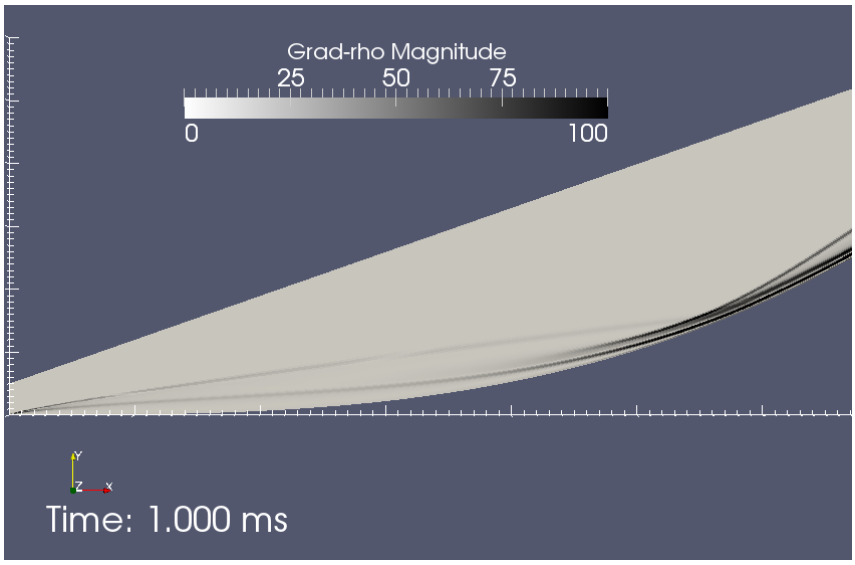


Figure 36: Computed density gradient field.

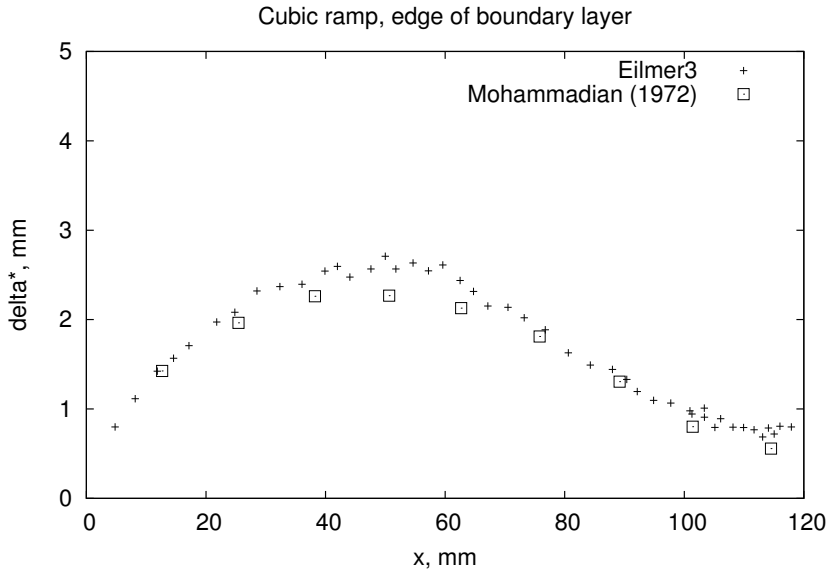


Figure 37: Estimates of the boundary-layer displacement thickness along the ramp.

23.4 Postprocessing to get heat transfer

The scripts below use the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. The first extracts the data for the cell nearest to the ramp

surface and uses that data to compute the expected shear stress and heat transfer at the surface.

```
#! /usr/bin/env python
# surface_properties.py
#
# Pick up the simulation data at the last simulated time
# compute an estimate of the shear-stress coefficient and
# output both shear and pressure along the cubic surface.
#
# PJ, 11-Aug-2013
#     14-Aug-2013 heat transfer normalised as St.sqrt(Re_x)

import sys, os
job = "cubic-ramp"
print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx

print "Begin: Pick up data for tindx=", tindx
from libprep3 import Vector, cross, dot, vabs
from e3_flow import read_all_blocks
from math import sqrt
#
nb = 22
pick_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18] # blocks against cubic surface
rho_inf = 5.521e-3 # kg/m**3
p_inf = 66.43 # Pa
u_inf =  1589.8 # m/s
T_inf = 41.92 # K
T_wall = 296.0 # K
T_0 = 1300.0 # K
specific_heat = 1004.5 # J/kg.K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'Air')
mm = 0.001  # metres
#
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute shear stress for cell-centres along the surface"
outfile = open("surface.data", "w")
outfile.write("# x(m) tau_w(Pa) Cf Cf_blasius y_plus p(Pa) Cp q(W/m**2) St.Re^0.5\n")
for ib in pick_list:
    j = 0 # surface is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k]
        y = flow[ib].data['pos.y'][i,j,k]
        ctr = Vector(x, y)
        # Get vertices on surface, for this cell.
        x = grid[ib].x[i,j,k]
        y = grid[ib].y[i,j,k]
        vtx0 = Vector(x, y)
        x = grid[ib].x[i+1,j,k]
        y = grid[ib].y[i+1,j,k]
        vtx1 = Vector(x, y)
        t1 = (vtx1-vtx0)
        t1.norm() # tangent vector for surface
        midpoint = 0.5*(vtx0+vtx1) # on surface
        normal = cross(Vector(0,0,1),t1)
        normal.norm()
        # Surface to cell-centre distance.
        dy = dot(normal, ctr-midpoint)
        # Cell-centre flow data.
        rho = flow[ib].data['rho'][i,j,k]
        ux = flow[ib].data['vel.x'][i,j,k]
        uy = flow[ib].data['vel.y'][i,j,k]
        v = Vector(ux, uy)
```

```
        vt = dot(v,t1) # velocity component tangent to surface
        mu = flow[ib].data['mu'][i,j,k]
        kgas = flow[ib].data['k[0]'][i,j,k]
        p = flow[ib].data['p'][i,j,k]
        Cp = (p-p_inf)/(0.5*rho_inf*u_inf*u_inf)
        T = flow[ib].data['T[0]'][i,j,k]
        # Shear stress
        dudy = (vt - 0.0) / dy # no-slip wall
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * dy * rho / mu
        Rex = rho_inf * u_inf * midpoint.x / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        # Heat flux
        dTdy = (T - T_wall) / dy # conductive heat flux at the wall
        q = kgas * dTdy
        St = q / (rho_inf*u_inf*specific_heat*(T_0-T_wall)) # Stanton number
        #
        outfile.write("%f %f %f %f %f %f %f %f %f\n" %
                      (midpoint.x, tau_w, Cf, Cf_blasius,
                       y_plus, p, Cp, q, St*sqrt(Rex)))
        print "x=", midpoint.x, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus, \
            "p=", p, "Cp=", Cp, "q=", q, "St.Rex^0.5=", St*sqrt(Rex)
outfile.close()
print "Done"
```

23.5 Notes

- Plotting was done with the following GNUPlot scripts.

```
# surface-pressure.gnuplot
set term postscript eps 20
set output 'surface-pressure.eps'
set title 'Cubic ramp, pressure along surface'
set ylabel 'p, Pa'
set xlabel 'x, mm'
set key top left
plot './surface.data' using ($1*1000):($6) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/mohammadian-figure-9-p_p_inf.data' \
     using ($1*25.4):($2*66.43) \
     title 'Mohammadian (1972)' with points pt 4

# surface-heat-transfer.gnuplot
set term postscript eps 20
set output 'surface-heat-transfer.eps'
set title 'Cubic ramp, heat-flux along surface'
set ylabel 'St.Re^(1/2)'
set xlabel 'x, mm'
set yrange [0:15]
set key top left
plot './surface.data' using ($1*1000):($9) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/mohammadian-figure-10-stanton.data' \
     using ($1*25.4):($2) \
     title 'Mohammadian (1972)' with points pt 4
```

24 Hypersonic flow over a convex ramp.

This is the second of the two hypersonic flows studied by Mohammadian [13] in the Imperial College gun tunnel. We use the same free-stream conditions as in Sec. 23 along with the slightly more difficult-to-describe convex ramp geometry. The favourable pressure gradient should make this an easier test flow to simulate.

24.1 Input script (.py)

Figure 38 shows the flow region, as modelled for simulation. The region is very simple but, this time, we have divided it into 28 blocks so that the computational load can be shared across a number of CPU cores.

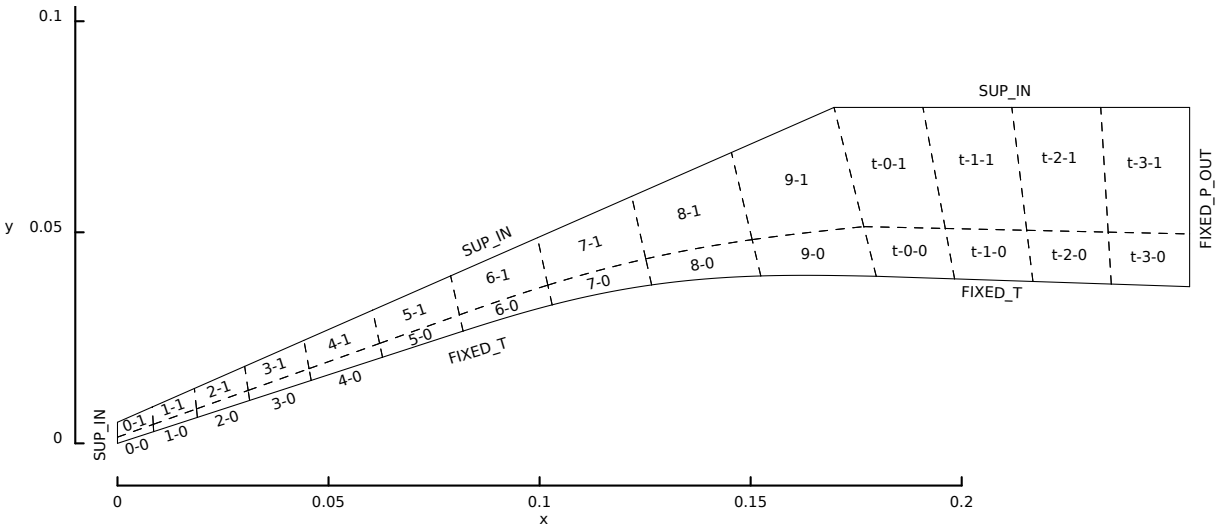


Figure 38: Schematic view of the simulated flow region for the hypersonic flow over Mohammadian's convex ramp.

The convex ramp is initially straight at 18^o, until $x = 3$ inches, followed by a faired section defined by

$$g = 0.0026\, s^4 - 0.0211\, s^3$$

where s and g are the local coordinates, in inches, rotated 18^o to the x, y coordinates. Following the faired section, there is a final straight section, continuing on at the same slope as the faired section at that joining point. The second derivative of the fairing is zero at both jouning points and this occurs at $s = 0$ and $s = 4.058$. The angle of this final straight section is computed as -1.90^o in the x, y-plane, that is, slightly away from the free-stream flow direction. Again, the ramp surface temperature was assumed to be a constant 296 K.

```
# convex-ramp.py
# PJ, 15-Aug-2013
# Model of Mohammadian's convex-ramp experiment.
# Revised 26-Aug-2013 to take his polynomial at face value.

gdata.title = "Mohammadian convex ramp."
print gdata.title
# Conditions to match those reported in JFM paper.
p_inf = 66.43 # Pa
u_inf = 1589.8 # m/s
T_inf = 41.92 # degree K
select_gas_model(model='ideal gas', species=['air'])
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf)
initial = FlowCondition(p=p_inf/5, u=0, T=T_inf)
T_wall = 296.0 # degree K --assumed cold-wall temperature

# Mohammadian used the inch as his length scale.
m_per_inch = 0.0254
mm = 1.0e-3 # metres per mm

def ramp(t):
    """
    Parametric definition of ramp profile in xy-plane.

    Here, we join the initial straight 18-degree ramp to the polynomial.
    """
    alpha = 18.0*math.pi/180.0 # angle of initial straight section
    sin18 = math.sin(alpha)
    cos18 = math.cos(alpha)
    tan18 = math.tan(alpha)
    x_join_inch = 3.0
    y_join_inch = x_join_inch * tan18
    L1 = x_join_inch/cos18 # length of initial straight section
    L2 = 4.14677 # length of fairing (computed via maxima)
    t2 = (L1+L2) * t
    if t2 < L1:
        x_inch = t2 * cos18
        y_inch = t2 * sin18
    else:
        s = (t2 - L1)/L2 * 4.0577
        g = 0.0026 * s**4 - 0.0211 * s**3
        x_inch = x_join_inch + s * cos18 - g * sin18
        y_inch = y_join_inch + s * sin18 + g * cos18
    return (x_inch*m_per_inch, y_inch*m_per_inch, 0.0)

# leading edge of ramp
x,y,z = ramp(0.0); a0 = Vector(x,y,z); a1 = a0+Vector(0.0,5*mm)
# downstream end of transition curve
x,y,z = ramp(1.0); b0 = Vector(x,y,z); b1 = b0+Vector(-10.0*mm,40*mm)
# For the final straight section, angle continues at final angle of transition.
x_length = 10*m_per_inch - b0.x
beta = -1.90*math.pi/180.0
# end of model
c0 = Vector(b0.x+x_length,b0.y+x_length*math.tan(beta))
c1 = Vector(c0.x,b1.y)

rcfx = RobertsClusterFunction(1,0,1.2)
rcfy = RobertsClusterFunction(1,0,1.1)
ni0 = 200; nj0 = 40 # We'll scale discretization off these values
factor = 2.0
ni0 = int(ni0*factor); nj0 = int(nj0*factor)

wedge = SuperBlock2D(make_patch(Line(a1,b1),Line(b0,b1),
                                PyFunctionPath(ramp),Line(a0,a1)),
                     nni=ni0, nnj=nj0, nbi=10, nbj=2,
                     bc_list=[SupInBC(inflow),None,
                              FixedTBC(T_wall),SupInBC(inflow)],
                     cf_list=[rcfx,rcfy,rcfx,rcfy],
                     fill_condition=inflow, label="wedge")
tail = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                    nni=int(ni0/4), nnj=nj0, nbi=4, nbj=2,
                    bc_list=[SupInBC(inflow),FixedPOutBC(p_inf/5),
                             FixedTBC(T_wall),None],
```

```
                    cf_list=[None,rcfy,None,rcfy],
                    fill_condition=initial, label="tail")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
gdata.max_time = 1.0e-3 # long enough for several flow lengths
gdata.max_step = 2000000
gdata.dt = 1.0e-9
gdata.dt_plot = 0.1e-3

sketch.xaxis(0.0, 0.20, 0.05, -0.010)
sketch.yaxis(0.0, 0.20, 0.05, -0.010)
sketch.window(0.0, 0.0, 0.20, 0.20, 0.05, 0.05, 0.25, 0.25)
```

24.2 Running the simulation

In terms of required computer time, this simulation is fairly demanding, taking more than 12 hours on a 4-core workstation. The job scipts submitted to the batch system are shown below. Note that the preparation script sets up the mapping of the full set of 28 blocks to fit onto 4 MPI tasks.

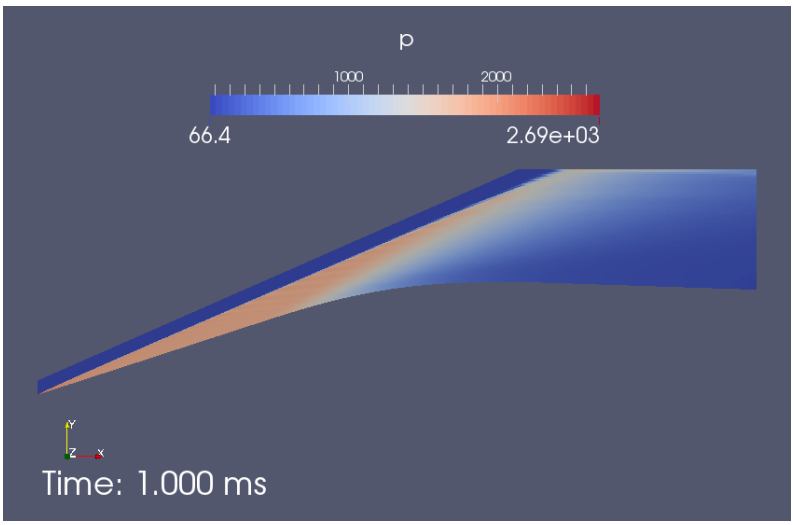
```
#! /bin/bash
# prep.sh
e3prep.py --job=convex-ramp --do-svg
e3loadbalance.py --job=convex-ramp -n 4
e3post.py --job=convex-ramp --tindx=0 --vtk-xml

echo "At this point, we should have a grid."
echo "Use run.sh next"
```

```
#! /bin/bash
# run.sh
# module load openmpi-x86_64
date
mpirun -np 4 e3mpi.exe --job=convex-ramp --mpimap=convex-ramp.mpimap --run > run.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

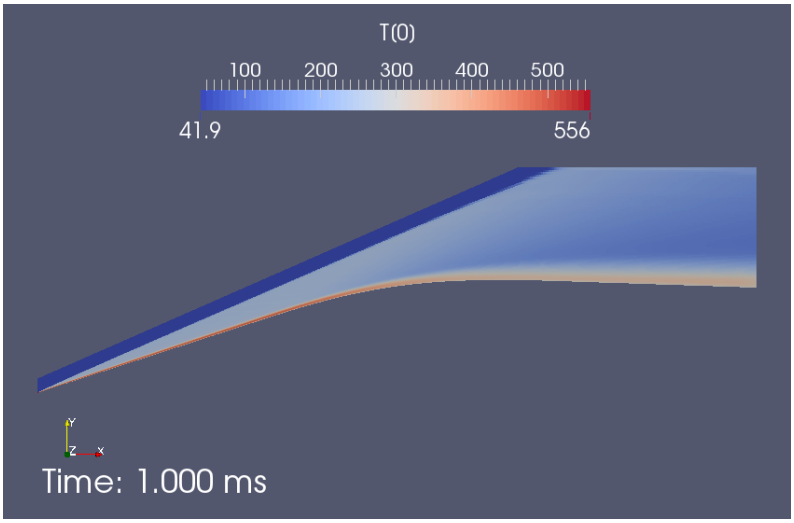
24.3 Results

Figure 39 shows some of the flow field data at t=1 ms after flow start. This is sufficient time for the flow to reach steady state.

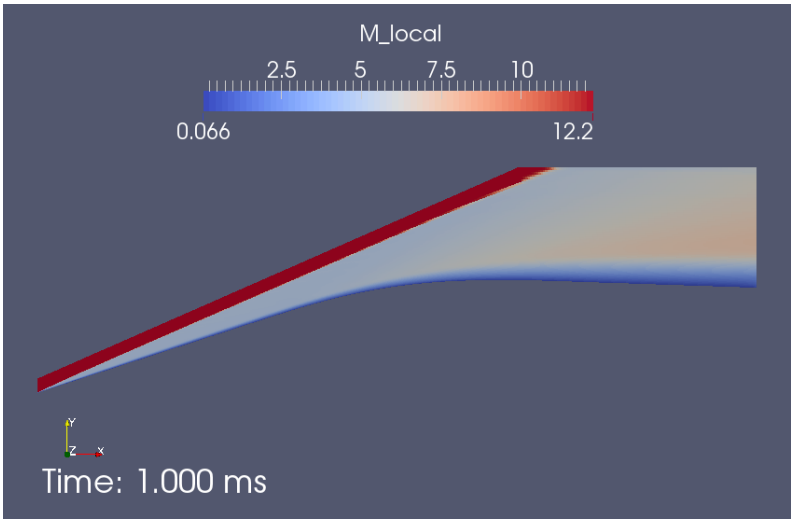
The pressure field shows a nice, straight shock propagating into the free-stream, with an almost constant pressure region between the shock and the straight ramp surface. The



(a) Pressure field.



(b) Temperature field.

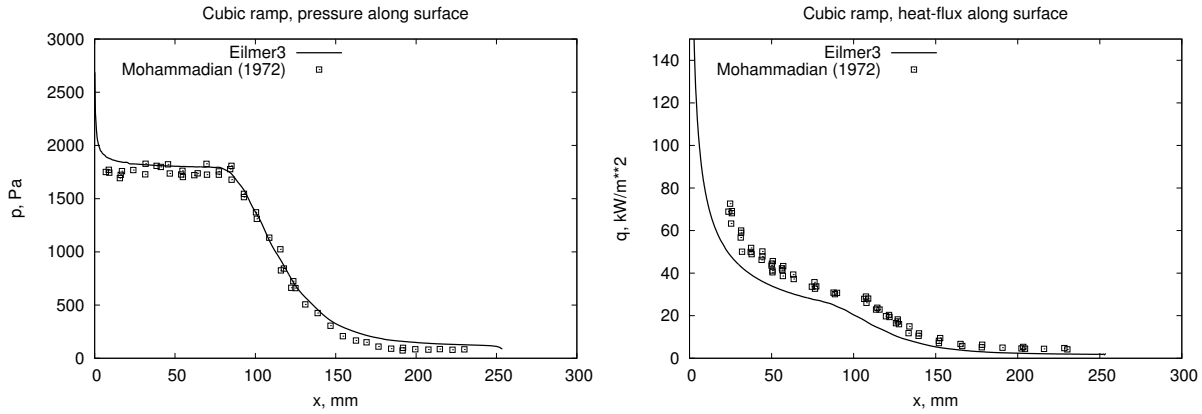


(c) Mach number.

Figure 39: Computed flow field at t=1 ms.

faired section produces a smoothly decreasing pressure and, true to boundary layer theory, the pressure gradient through the boundary layer to the ramp surface is essentially zero. The temperature field, however, shows clearly the boundary layer that grows along the ramp surface.

Although the computed flow field looks plausible, the real proof of success of the simulation is in comparison with the experimental data. Figure 40 shows the pressure and heat-transfer along the surface of the ramp. The simulation has done a good job of estimating the pressure distribution over the full ramp, with a mismatch in magnitude only after passing over the faired section to reach the very low pressure conditions. The simulation has also done a reasonable job on the heat transfer estimate, which has been computed from the field data using the script in Section 24.4. For this case Mohammadian has provided dimensional data in the original paper so we have plotted that directly, after converting to SI units. Agreement is good in form but only fair in magnitude. This will be further exploted in the following example, where a thermal-nonequilibrium model for air is tried.



(a) Pressure.

(b) Heat transfer.

Figure 40: Distribution of pressure and heat transfer along the concave ramp. Simulation data is recorded at t=1 ms into the simulation. Experimental data is from Ref. [13].

24.4 Postprocessing to get heat transfer

The scripts below use the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. The first extracts the data for the cell nearest to the ramp surface and uses that data to compute the expected shear stress and heat transfer at the surface.

```
#! /usr/bin/env python
# surface_properties.py
```

```
#
# Pick up the simulation data at the last simulated time
# compute an estimate of the shear-stress coefficient and
# output both shear and pressure along the convex surface.
#
# PJ, 11-Aug-2013
#     14-Aug-2013 heat transfer normalised as St.sqrt(Re_x)

import sys, os
job = "convex-ramp"
print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx

print "Begin: Pick up data for tindx=", tindx
from libprep3 import Vector, cross, dot, vabs
from e3_flow import read_all_blocks
from math import sqrt
#
nb = 28
pick_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26] # blocks against surface
rho_inf = 5.521e-3 # kg/m**3
p_inf = 66.43 # Pa
u_inf =  1589.8 # m/s
T_inf = 41.92 # K
T_wall = 296.0 # K
T_0 = 1300.0 # K
specific_heat = 1004.5 # J/kg.K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'Air')
mm = 0.001  # metres
#
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute shear stress for cell-centres along the surface"
outfile = open("surface.data", "w")
outfile.write("# x(m) tau_w(Pa) Cf Cf_blasius y_plus p(Pa) Cp q(W/m**2) St.Re^0.5\n")
for ib in pick_list:
    j = 0 # surface is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k]
        y = flow[ib].data['pos.y'][i,j,k]
        ctr = Vector(x, y)
        # Get vertices on surface, for this cell.
        x = grid[ib].x[i,j,k]
        y = grid[ib].y[i,j,k]
        vtx0 = Vector(x, y)
        x = grid[ib].x[i+1,j,k]
        y = grid[ib].y[i+1,j,k]
        vtx1 = Vector(x, y)
        t1 = (vtx1-vtx0)
        t1.norm() # tangent vector for surface
        midpoint = 0.5*(vtx0+vtx1) # on surface
        normal = cross(Vector(0,0,1),t1)
        normal.norm()
        # Surface to cell-centre distance.
        dy = dot(normal, ctr-midpoint)
        # Cell-centre flow data.
        rho = flow[ib].data['rho'][i,j,k]
        ux = flow[ib].data['vel.x'][i,j,k]
        uy = flow[ib].data['vel.y'][i,j,k]
        v = Vector(ux, uy)
        vt = dot(v,t1) # velocity component tangent to surface
        mu = flow[ib].data['mu'][i,j,k]
        kgas = flow[ib].data['k[0]'][i,j,k]
        p = flow[ib].data['p'][i,j,k]
        Cp = (p-p_inf)/(0.5*rho_inf*u_inf*u_inf)
        T = flow[ib].data['T[0]'][i,j,k]
        # Shear stress
        dudy = (vt - 0.0) / dy # no-slip wall
```

```
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * dy * rho / mu
        Rex = rho_inf * u_inf * midpoint.x / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        # Heat flux
        dTdy = (T - T_wall) / dy # conductive heat flux at the wall
        q = kgas * dTdy
        St = q / (rho_inf*u_inf*specific_heat*(T_0-T_wall)) # Stanton number
        #
        outfile.write("%f %f %f %f %f %f %f %f %f\n" %
                      (midpoint.x, tau_w, Cf, Cf_blasius,
                       y_plus, p, Cp, q, St*sqrt(Rex)))
        print "x=", midpoint.x, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus, \
            "p=", p, "Cp=", Cp, "q=", q, "St.Rex^0.5=", St*sqrt(Rex)
outfile.close()
print "Done"
```

24.5 Notes

- Plotting was done with the following GNUPlot scripts.

```
# surface-pressure.gnuplot
set term postscript eps 20
set output 'surface-pressure.eps'
set title 'Cubic ramp, pressure along surface'
set ylabel 'p, Pa'
set xlabel 'x, mm'
set key top left
plot './surface.data' using ($1*1000):($6) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/mohammadian-figure-12-p_p_inf.data' \
     using ($1*25.4):($2*66.43) \
     title 'Mohammadian (1972)' with points pt 4

# surface-heat-transfer.gnuplot
set term postscript eps 20
set output 'surface-heat-transfer.eps'
set title 'Cubic ramp, heat-flux along surface'
set ylabel 'q, kW/m**2'
set xlabel 'x, mm'
set yrange [0:150]
set key top left
plot './surface.data' using ($1*1000):($8/1000) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/mohammadian-figure-13-heat-flux.data' \
     using ($1*25.4):($2*11.4) \
     title 'Mohammadian (1972)' with points pt 4
```

25 Hypersonic, nonequilibrium flow over a convex ramp.

This is a variation on the convex-ramp hypersonic flow studied by Mohammadian [13] in the Imperial College gun tunnel, bringing in a thermal-nonequilibrium model for the air. We use the same static free-stream conditions as in Sec. 23 but now assume that the vibrational temperature of the molecules is frozen at a temperature not far below the stagnation temperature. The hope is that the extra vibrational energy will be lead to an extra bit of heat flux at the ramp surface.

25.1 Input script (.py)

The user input script now needs to specify the gas model as a mixture of N2 and O2 molecules and their vibrational temperatures, when specifying the flow conditions. Also, it needs to specify the thermal nonequilibrium energy exchange scheme (`N2-O2-TV.lua`).

```
# convex-ramp.py
# PJ, Dan and Rowan, 15-Aug-2013
# Model of Mohammadian's convex-ramp experiment with thermal nonequilibrium.
# Revised 26-Aug-2013 to take his polynomial at face value.

gdata.title = "Mohammadian convex ramp, 2T thermo."
print gdata.title

# Gas-model
species = select_gas_model(model='two temperature gas', species=['N2','O2'])
gm = get_gas_model_ptr()
nsp = gm.get_number_of_species()
ntm = gm.get_number_of_modes()
# Energy exchange model (only if there are nonequilibrium temperatures)
if ntm > 1:
    set_energy_exchange_update("N2-O2-TV.lua")
# Conditions to match those reported in JFM paper with a guess for Tvib.
p_inf = 66.43 # Pa
u_inf = 1589.8 # m/s
# Temperatures
T_inf = [ 0.0 ] * ntm
T_inf[0] = 41.92 # gas static temperature: degree K
Tv_inf = 1000.0 # NOTE: freestream vibrational temperature closer to stagnation T
for itm in range(1,ntm):
    T_inf[itm] = Tv_inf # nonequilibrium temperature: degree K
# Mass-fractions
massf_inf = [ 0.0 ] * nsp
massf_inf[species.index("N2")] = 0.767 # standard air
massf_inf[species.index("O2")] = 0.233 # standard air
#
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf, massf=massf_inf)
initial = FlowCondition(p=p_inf/5, u=0, T=T_inf, massf=massf_inf)
#
T_wall = 296.0 # degree K --assumed cold-wall temperature

# Mohammadian used the inch as his length scale.
m_per_inch = 0.0254
mm = 1.0e-3 # metres per mm

def ramp(t):
    """
    Parametric definition of ramp profile in xy-plane.
```

```
    Here, we join the initial straight 18-degree ramp to the polynomial.
    """
    alpha = 18.0*math.pi/180.0 # angle of initial straight section
    sin18 = math.sin(alpha)
    cos18 = math.cos(alpha)
    tan18 = math.tan(alpha)
    x_join_inch = 3.0
    y_join_inch = x_join_inch * tan18
    L1 = x_join_inch/cos18 # length of initial straight section
    L2 = 4.14677 # length of fairing (computed via maxima)
    t2 = (L1+L2) * t
    if t2 < L1:
        x_inch = t2 * cos18
        y_inch = t2 * sin18
    else:
        s = (t2 - L1)/L2 * 4.0577
        g = 0.0026 * s**4 - 0.0211 * s**3
        x_inch = x_join_inch + s * cos18 - g * sin18
        y_inch = y_join_inch + s * sin18 + g * cos18
    return (x_inch*m_per_inch, y_inch*m_per_inch, 0.0)

# leading edge of ramp
x,y,z = ramp(0.0); a0 = Vector(x,y,z); a1 = a0+Vector(0.0,5*mm)
# downstream end of transition curve
x,y,z = ramp(1.0); b0 = Vector(x,y,z); b1 = b0+Vector(-10.0*mm,40*mm)
# For the final straight section, angle continues at final angle of transition.
x_length = 10*m_per_inch - b0.x
beta = -1.90*math.pi/180.0
# end of model
c0 = Vector(b0.x+x_length,b0.y+x_length*math.tan(beta))
c1 = Vector(c0.x,b1.y)

rcfx = RobertsClusterFunction(1,0,1.2)
rcfy = RobertsClusterFunction(1,0,1.1)
ni0 = 200; nj0 = 40 # We'll scale discretization off these values
factor = 2.0
ni0 = int(ni0*factor); nj0 = int(nj0*factor)

wedge = SuperBlock2D(make_patch(Line(a1,b1),Line(b0,b1),
                                PyFunctionPath(ramp),Line(a0,a1)),
                     nni=ni0, nnj=nj0, nbi=10, nbj=2,
                     bc_list=[SupInBC(inflow),None,
                              FixedTBC(T_wall),SupInBC(inflow)],
                     cf_list=[rcfx,rcfy,rcfx,rcfy],
                     fill_condition=inflow, label="wedge")
tail = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                    nni=int(ni0/4), nnj=nj0, nbi=4, nbj=2,
                    bc_list=[SupInBC(inflow),FixedPOutBC(p_inf/5),
                             FixedTBC(T_wall),None],
                    cf_list=[None,rcfy,None,rcfy],
                    fill_condition=initial, label="tail")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
gdata.max_time = 1.0e-3 # long enough for several flow lengths
gdata.max_step = 2000000
gdata.dt = 1.0e-9
gdata.dt_plot = 0.1e-3

sketch.xaxis(0.0, 0.20, 0.05, -0.010)
sketch.yaxis(0.0, 0.20, 0.05, -0.010)
sketch.window(0.0, 0.0, 0.20, 0.20, 0.05, 0.05, 0.25, 0.25)
```

```
scheme_t = {
    update = "energy exchange ODE",
```

```
    temperature_limits = {
        lower = 20.0,
        upper = 100000.0
    },
    error_tolerance = 0.000001
}

ode_t = {
    step_routine = 'rkf',
    max_step_attempts = 4,
    max_increase_factor = 1.15,
    max_decrease_factor = 0.01,
    decrease_factor = 0.333
}

mechanism{
   'N2 ~~ ( N2, O2 ) : V-T',
   rt={'Millikan-White' }
}

mechanism{
   'O2 ~~ ( N2, O2 ) : V-T',
   rt={'Millikan-White' }
}
```

25.2 Running the simulation

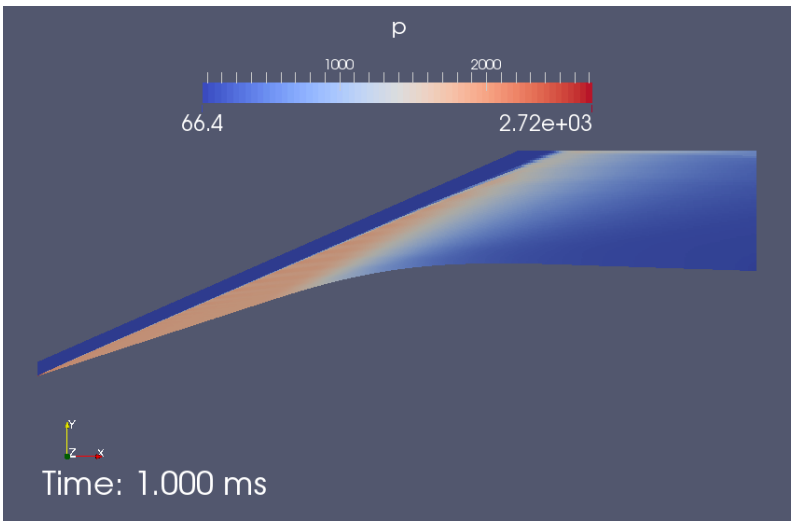
In terms of required computer time, this simulation is significantly more demanding than the ideal-air simulation, taking more than 30 hours on a 4-core workstation (up from 12 hours). The job scipts are essentially the same as for the ideal-air case.

25.3 Results

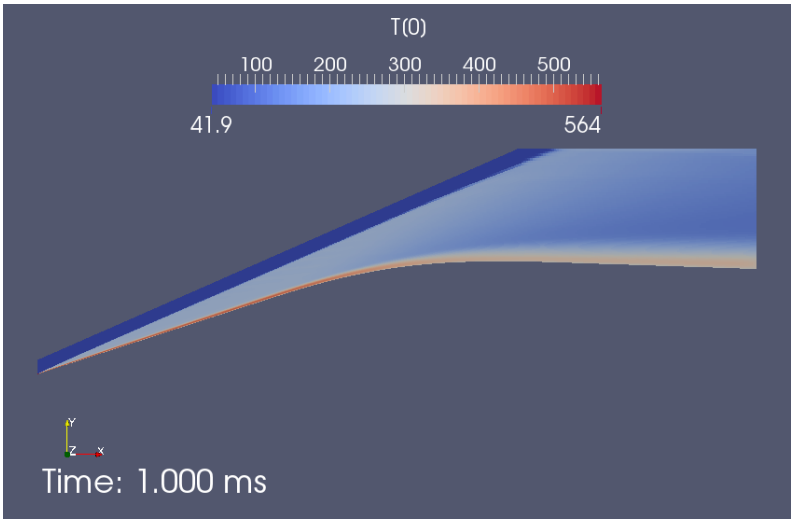
Figure 41 shows some of the flow field data at t=1 ms after flow start. This is sufficient time for the flow to reach steady state.

Again, the pressure field shows a nice, straight shock propagating into the free-stream, with an almost constant pressure region between the shock and the straight ramp surface. The static temperature field, again, shows clearly the boundary layer that grows along the ramp surface. The first vibrational temperature shows a relaxation toward the static temperature as the gas approaches the ramp surface.

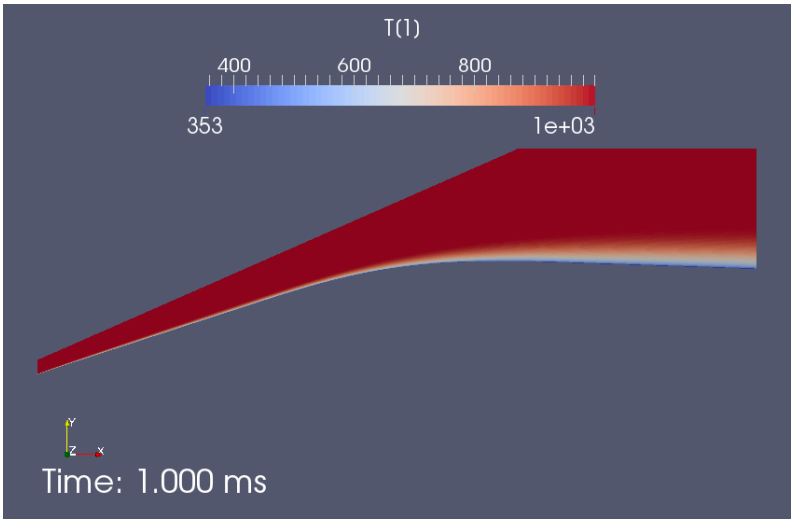
Although the computed flow field looks plausible, again, the real proof of success of the simulation is in comparison with the experimental data. Figure 42 shows the pressure and heat-transfer along the surface of the ramp. The simulation has done essentially the same good job of estimating the pressure distribution over the full ramp, with a mismatch in magnitude only after passing over the faired section to reach the very low pressure conditions. This pressure distribution is indistinguishable from the distribution for the ideal air simulation. The simulation has again done a reasonable job on the heat transfer estimate, with a small improvement over that computed in the ideal air simulation. That



(a) Pressure field.



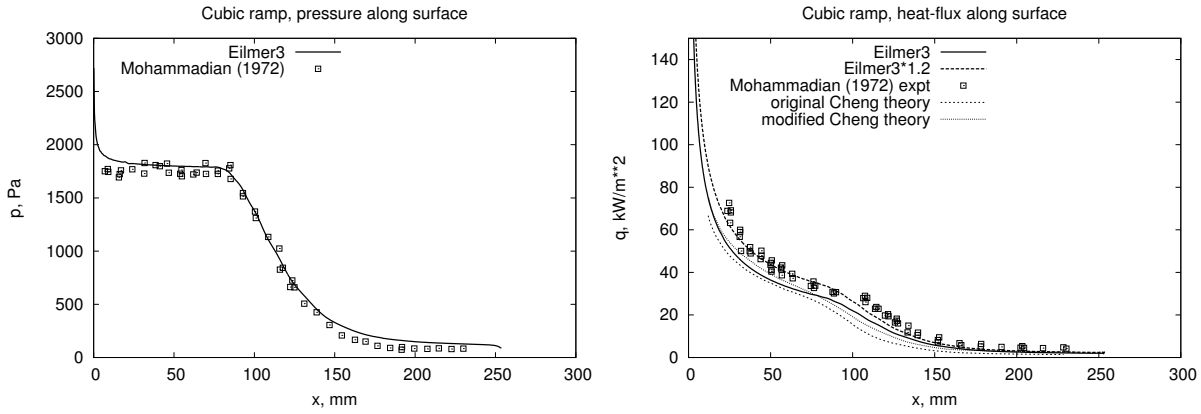
(b) Static temperature field.



(c) Vibrational temperature field.

Figure 41: Computed flow field at t=1 ms.

agreement is good in form but only fair in magnitude is now emphasised by scaling the Eilmer3 result by 1.2 and seeing that it falls very nicely onto the experimental data. What is the correct answer remains unclear since the original Cheng and modified Cheng theories, as used by Mohammadian fall closer to the unscaled Eilmer3 result. Sigh...



(a) Pressure.

(b) Heat transfer.

Figure 42: Distribution of pressure and heat transfer along the concave ramp. Simulation data is recorded at t=1 ms into the simulation. Experimental data is from Ref. [13].

25.4 Postprocessing to get heat transfer

The scripts below use the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. The first extracts the data for the cell nearest to the ramp surface and uses that data to compute the expected shear stress and heat transfer at the surface.

```
#! /usr/bin/env python
# surface_properties.py
#
# Pick up the simulation data at the last simulated time
# compute an estimate of the shear-stress coefficient and
# output both shear and pressure along the convex surface.
#
# PJ, 11-Aug-2013
#     14-Aug-2013 heat transfer normalised as St.sqrt(Re_x)

import sys, os
job = "convex-ramp"
print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx

print "Begin: Pick up data for tindx=", tindx
from libprep3 import Vector, cross, dot, vabs
from e3_flow import read_all_blocks
from math import sqrt
#
nb = 28
pick_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26] # blocks against surface
```

```
rho_inf = 5.521e-3 # kg/m**3
p_inf = 66.43 # Pa
u_inf =  1589.8 # m/s
T_inf = 41.92 # K
T_wall = 296.0 # K
T_0 = 1300.0 # K
specific_heat = 1004.5 # J/kg.K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'Air')
mm = 0.001  # metres
#
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute shear stress for cell-centres along the surface"
outfile = open("surface.data", "w")
outfile.write("# x(m) tau_w(Pa) Cf Cf_blasius y_plus p(Pa) Cp q(W/m**2) St.Re^0.5\n")
for ib in pick_list:
    j = 0 # surface is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k]
        y = flow[ib].data['pos.y'][i,j,k]
        ctr = Vector(x, y)
        # Get vertices on surface, for this cell.
        x = grid[ib].x[i,j,k]
        y = grid[ib].y[i,j,k]
        vtx0 = Vector(x, y)
        x = grid[ib].x[i+1,j,k]
        y = grid[ib].y[i+1,j,k]
        vtx1 = Vector(x, y)
        t1 = (vtx1-vtx0)
        t1.norm() # tangent vector for surface
        midpoint = 0.5*(vtx0+vtx1) # on surface
        normal = cross(Vector(0,0,1),t1)
        normal.norm()
        # Surface to cell-centre distance.
        dy = dot(normal, ctr-midpoint)
        # Cell-centre flow data.
        rho = flow[ib].data['rho'][i,j,k]
        ux = flow[ib].data['vel.x'][i,j,k]
        uy = flow[ib].data['vel.y'][i,j,k]
        v = Vector(ux, uy)
        vt = dot(v,t1) # velocity component tangent to surface
        mu = flow[ib].data['mu'][i,j,k]
        kgas = flow[ib].data['k[0]'][i,j,k]
        p = flow[ib].data['p'][i,j,k]
        Cp = (p-p_inf)/(0.5*rho_inf*u_inf*u_inf)
        T = flow[ib].data['T[0]'][i,j,k]
        # Shear stress
        dudy = (vt - 0.0) / dy # no-slip wall
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * dy * rho / mu
        Rex = rho_inf * u_inf * midpoint.x / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        # Heat flux
        dTdy = (T - T_wall) / dy # conductive heat flux at the wall
        q = kgas * dTdy
        St = q / (rho_inf*u_inf*specific_heat*(T_0-T_wall)) # Stanton number
        #
        outfile.write("%f %f %f %f %f %f %f %f %f\n" %
                      (midpoint.x, tau_w, Cf, Cf_blasius,
                       y_plus, p, Cp, q, St*sqrt(Rex)))
        print "x=", midpoint.x, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus, \
            "p=", p, "Cp=", Cp, "q=", q, "St.Rex^0.5=", St*sqrt(Rex)
outfile.close()
print "Done"
```

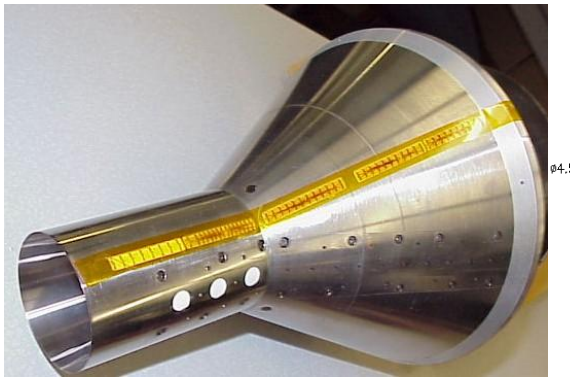
25.5 Notes

- Plotting was done with the following GNUPlot scripts.

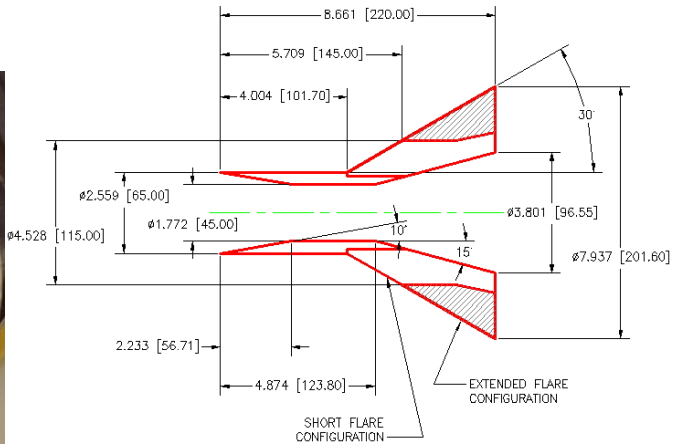
```
# surface-pressure.gnuplot
set term postscript eps 20
set output 'surface-pressure.eps'
set title 'Cubic ramp, pressure along surface'
set ylabel 'p, Pa'
set xlabel 'x, mm'
set key top left
plot './surface.data' using ($1*1000):($6) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/mohammadian-figure-12-p_p_inf.data' \
     using ($1*25.4):($2*66.43) \
     title 'Mohammadian (1972)' with points pt 4

# surface-heat-transfer.gnuplot
set term postscript eps 20
set output 'surface-heat-transfer.eps'
set title 'Cubic ramp, heat-flux along surface'
set ylabel 'q, kW/m**2'
set xlabel 'x, mm'
set yrange [0:150]
set key top left
plot './my-surface.data' using ($10*1000):($2/1000) with lines \
     lw 3.0 title 'Eilmer3', \
     './my-surface.data' using ($10*1000):($2/1000*1.2) with lines \
     lw 3.0 lt 2 title 'Eilmer3*1.2', \
     './notes/mohammadian-figure-13-heat-flux.data' \
     using ($1*25.4):($2*11.4) \
     title 'Mohammadian (1972) expt' with points pt 4, \
     './notes/mohammadian-figure-13-original-cheng-theory.data' \
     using ($1*25.4):($2*11.4) \
     title 'original Cheng theory' with lines lw 1.5 lt 3, \
     './notes/mohammadian-figure-13-modified-cheng-theory.data' \
     using ($1*25.4):($2*11.4) \
     title 'modified Cheng theory' with lines lw 1.5 lt 4
```

26 Hypersonic flow over a hollow cylinder with flare.

This is one of the hypersonic test flows provided by the Calspan-University of Buffalo Research Center (CUBRC). As for the previous flows, it is an example that retains a very simple geometric arrangement for the flow boundaries but inclusion of the viscous effects leads to a very challenging flow. The experimental facility provides a Mach 10.3 flow of nitrogen along the cylinder with flare shown below in Figure 43. This model was used as part of an experimental campaign [14, 15] in the LENS shock tunnel at CUBRC. The very high Mach number free stream produces a fairly strong leading-edge interaction region at the sharp leading edge of the cylinder and a boundary layer develops along the cylinder surface. The flare deflects the flow, inducing a strong shock, but the viscous effects lead to a separation bubble forming in the boundary layer on the cylinder surface. The leading edge of the separation bubble also deflects the flow and forms another shock that merges with the leading-edge interaction shock. This combined shock happens to impinge on the flare surface and interact strongly with the boundary layer on the flare and the downstream end of the separation bubble. These features interact strongly but the overall flow eventually settles to a steady state and the boundary layer remains laminar.



(a) Physical model.



(b) Geometric definition from [16]. Dimensions in inches and mm.

Figure 43: Hollow cylinder with extended flare used in the CUBRC experiments.

26.1 Input script (.py)

In setting up this exercise, we follow the details provided by MacLean [16] and concentrate on the CUBRC Run 14 experiment. We assume an ideal nitrogen free stream, with conditions $p = 31.88\,\text{Pa}$, ρ=0.881 g/m^3, $u = 2.304\,\text{km/s}$ and a static temperature T=120.4 K. The actual nitrogen flow in the shock tunnel nozzle was far from ideal and had an estimated vibrational temperature of 2467 K. However, for the simulation reported here, this

vibrational energy is assumed frozen and thus ignored. The model surface temperature was a constant 295.2 K.

```
# cyl-flare.py
# PJ, 11-May-2013, 29-May-2013
# Model of the CUBRC hollow cylinder with extended-flare experiment.

gdata.title = "Hollow cylinder with extended flare."
print gdata.title
gdata.axisymmetric_flag = 1
# Conditions to match those reported for CUBRC Run 14
p_inf = 31.88 # Pa
u_inf = 2304.0 # m/s
T_inf = 120.4 # degree K
T_vib = 2467.0 # degrees K (but we will ignore for ideal-gas)
select_gas_model(model='ideal gas', species=['N2'])
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf)
initial = FlowCondition(p=p_inf/5, u=0, T=T_inf)
T_wall = 295.2 # degree K

mm = 1.0e-3 # metres per mm
L1 = 101.7*mm # cylinder length
L2 = 220.0*mm # distance to end of flare
R1 = 32.5*mm
alpha = 30.0*math.pi/180.0 # angle of flare
tan_alpha = math.tan(alpha)
a0 = Vector(0.0, R1); a1 = a0+Vector(0.0,5*mm) # leading edge of cylinder
b0 = Vector(L1, R1); b1 = b0+Vector(-5*mm,20*mm) # start flare
c0 = Vector(L2, R1+tan_alpha*(L2-L1)); c1 = c0+Vector(0.0,25*mm) # end flare

rcfx = RobertsClusterFunction(1,0,1.2)
rcfy = RobertsClusterFunction(1,0,1.1)
ni0 = 200; nj0 = 80 # We'll scale discretization off these values
factor = 1
ni0 *= factor; nj0 *= factor

cyl = SuperBlock2D(CoonsPatch(a0,b0,b1,a1),
                   nni=ni0, nnj=nj0, nbi=6, nbj=2,
                   bc_list=[SupInBC(inflow),None,
                            FixedTBC(T_wall),SupInBC(inflow)],
                   cf_list=[rcfx,rcfy,rcfx,rcfy],
                   fill_condition=inflow, label="cyl")
flare = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                     nni=ni0, nnj=nj0, nbi=6, nbj=2,
                     bc_list=[SupInBC(inflow),FixedPOutBC(p_inf/5),
                              FixedTBC(T_wall),None],
                     cf_list=[None,rcfy,None,rcfy],
                     fill_condition=initial, label="fl")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
# The settling of the separation bubble will probably dominate.
gdata.max_time = 2.5e-3 # long enough, looking at earlier simulations
gdata.max_step = 2000000
gdata.dt = 1.0e-9
gdata.dt_plot = 0.25e-3

sketch.xaxis(0.0, 0.250, 0.05, -0.010)
sketch.yaxis(0.0, 0.250, 0.05, -0.010)
sketch.window(0.0, 0.0, 0.250, 0.250, 0.05, 0.05, 0.25, 0.25)
```

26.2 Running the simulation

Figure 44 shows the flow region, as modelled for simulation. The region is very simple but we have divided it into 24 blocks so that the computational load can be shared across a number of CPU cores.

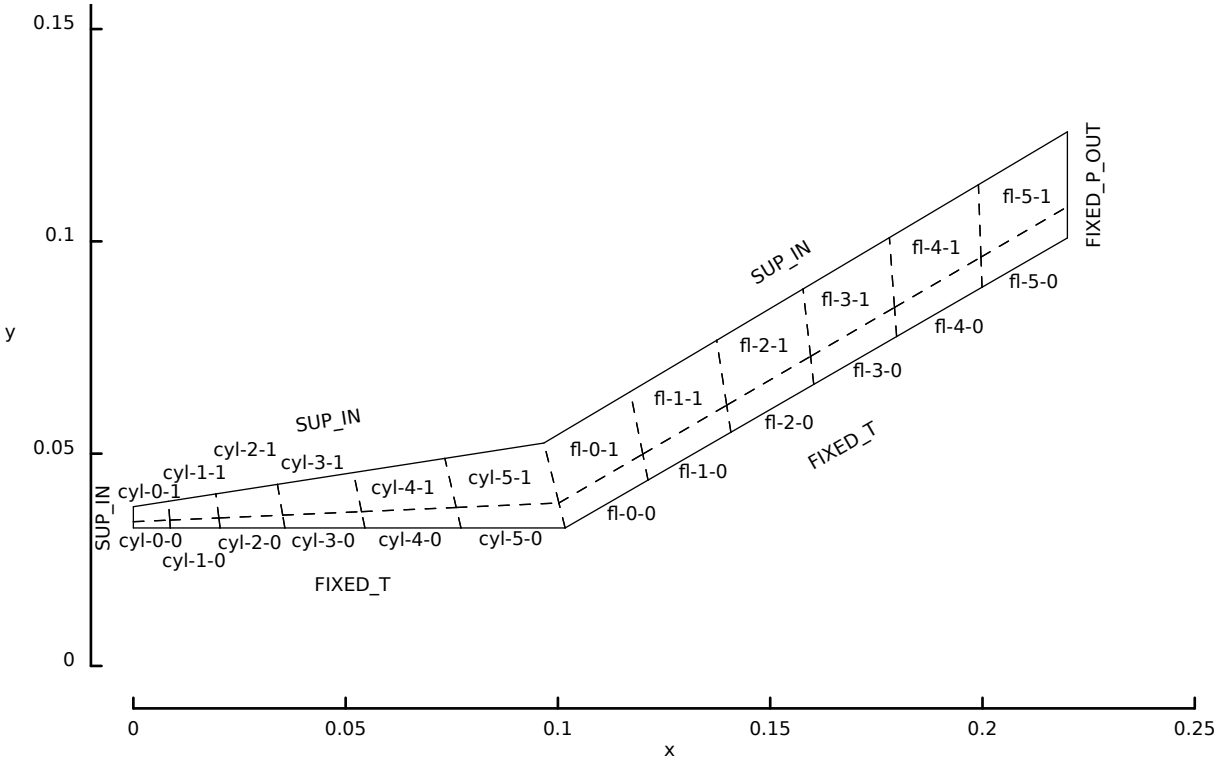


Figure 44: Schematic view of the simulated flow region for the hypersonic flow over a cylinder with flare.

In terms of required computer time, this simulation is a demanding. Unless you are extremely patient, you are advised to run it on a cluster computer, as was done for the results shown here. The job scipts submitted to the batch system are shown below. The preparation of the grids and initial flow-state files was done on a local workstation, these files transferred to the cluster computer file system and then the simulation was done in two stages, 0–2.5 ms and 2.5 ms–5 ms. In between the simulation stages, the `cyl-flare.control` file was edited by hand to extend the maximum time from 2.5 ms to 5.0 ms. Note that the 24 blocks have been grouped, via the mpimap file (that was generated by the e3loadbalance program), to 12 MPI tasks. Each of the simulation stages required a little less than a day on 12 cores of a Dell cluster with 2.2 GHz Xeon processors. The particular cluster was called "arcus" and was located at the Oxford e-Research Centre.

```
#! /bin/bash
# prep.sh
e3prep.py --job=cyl-flare --do-svg
e3loadbalance.py --job=cyl-flare -n 12
e3post.py --job=cyl-flare --tindx=0 --vtk-xml
```

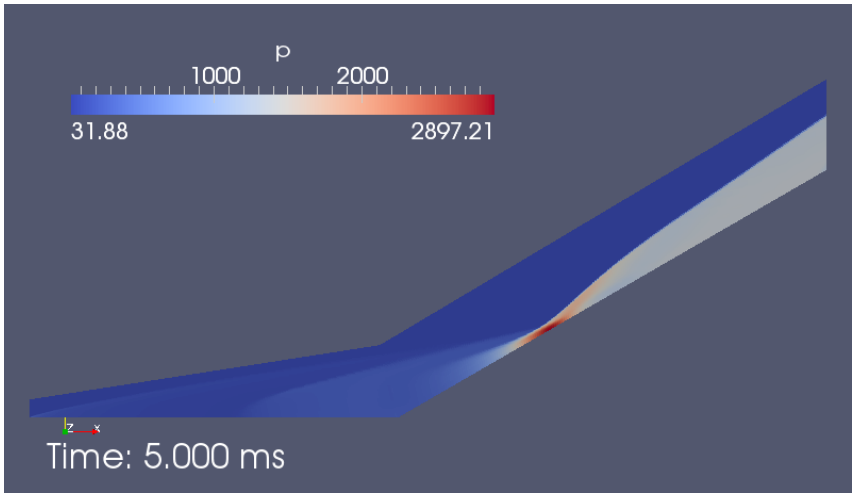
```
echo "At this point, we should have a grid."
echo "Use run.sh next"
```

```
#! /bin/bash
# run-arcus.sh
#PBS -l select=1:mpiprocs=12
#PBS -l walltime=51:00:00
#PBS -N cyl-flare
#PBS -m bea
#PBS -M peter.jacobs@eng.ox.ac.uk
#PBS -V
cd $PBS_O_WORKDIR
date
mpirun -np 12 -machinefile $PBS_NODEFILE $DATA/e3bin/e3mpi.exe \
    --job=cyl-flare --mpimap=cyl-flare.mpimap --run \
    --max-wall-clock=180000 > run-arcus.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

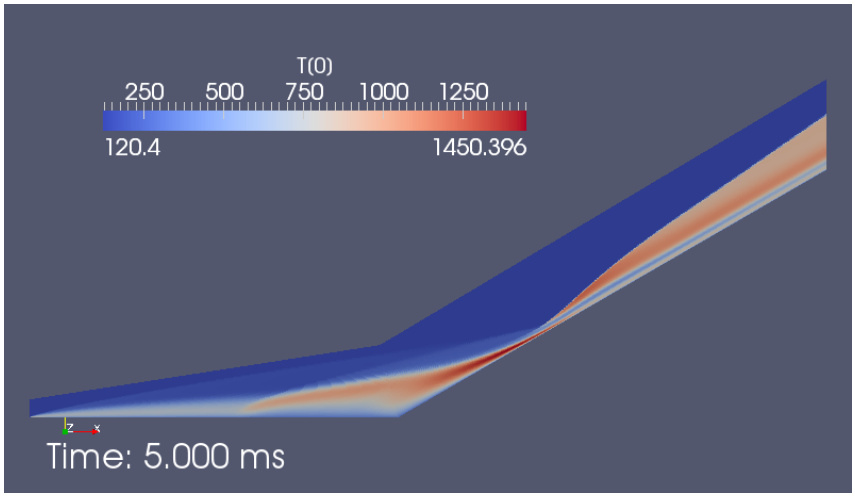
```
#! /bin/bash
# run-arcus.sh
#PBS -l select=1:mpiprocs=12
#PBS -l walltime=31:00:00
#PBS -N cyl-flare
#PBS -m bea
#PBS -M peter.jacobs@eng.ox.ac.uk
#PBS -V
cd $PBS_O_WORKDIR
date
mpirun -np 12 -machinefile $PBS_NODEFILE $DATA/e3bin/e3mpi.exe \
    --job=cyl-flare --mpimap=cyl-flare.mpimap --run \
    --tindx=10 --max-wall-clock=108000 > run-arcus-2.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

26.3 Results

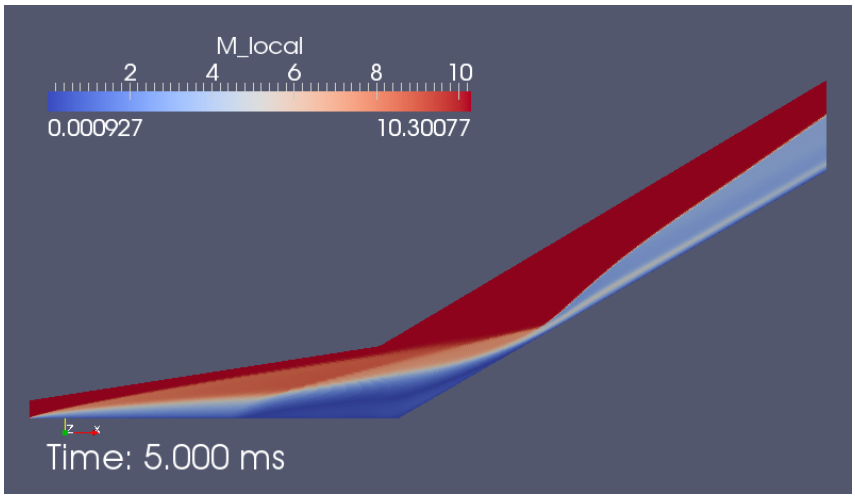
Figure 45 shows some of the flow field data at t=5 ms after flow start. The leading-edge interaction shock and the shock caused by the boundary-layer separation are both clearly defined. The leading-edge interaction shock starts strong but weakens with the immediately following expansion fan. The shock from the start of the separation region only becomes clear a small distance above the surface, near the outer edge of the boundary layer. The two shocks merge, shortly before impinging on the flare surface. By the time shown, the flow has settled to a steady-state configuration, as confirmed by the history of the separation point location on the cylinder surface, plotted in Figure 46.



(a) Pressure field.



(b) Temperature field.



(c) Mach number.

Figure 45: Computed flow field at t=5 ms.

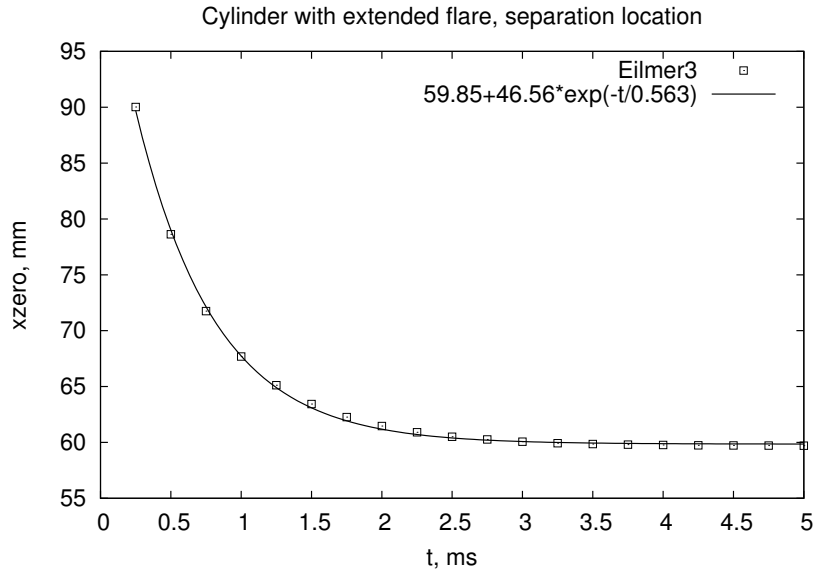
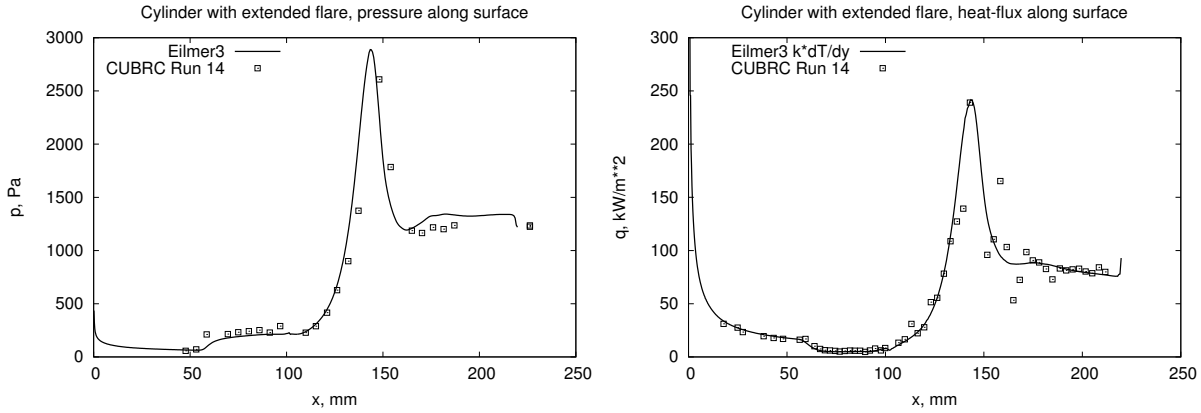


Figure 46: History of the separation location along the cylinder surface.

The real proof of success is in comparison with the experimental data. Figure 47 shows the pressure and heat-transfer along the surface of the cylinder and the flare. The simulation has done a good job of estimating the pressure distribution right through the separation zone and the shock-interaction zone on the flare. There is a sudden drop in pressure (and a corresponding rise in heat transfer) at the right end of the simulation domain where the boundary layer thins. This is expected because the expansion off the trailing edge of the flare would propagate upstream a little, through the subsonic part of the boundary layer.

The simulation has also done a good job on the heat transfer estimate, which has been computed from the field data using the script in Section 26.4. This quantity required lots of resolution to compute accurately and difficult to measure so it is reassuring that both sets of data line up nicely in the boundary layer leading into the sepration region, through the separation, and also after the interaction region on the flare surface. The separation bubble appears to be well captured in position and extent.



(a) Pressure.

(b) Heat transfer.

Figure 47: Distribution of pressure and heat transfer along the cylinder and flare. Simulation data is recorded at t=5 ms into the simulation. Experimental data is for Run 14 of the CUBRC experiment [14].

26.4 Postprocessing heat transfer and separation-point tracking

The scripts below use the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. The first extracts the data for the cell nearest to the cylinder and flare surface and uses that data to compute the expected shear stress and heat transfer at the surface. The second looks at the x-component of the velocity of the first cell above the cylinder surface to identify the location of the start of the separation region for all frames of the solution. After writing the location data to a file, it uses the

SciPy optimization module to fit a simple function to that data, in order to estimate the asymptotic position of the separation point for large times.

```
#! /usr/bin/env python
# surface_properties.py
#
# Pick up the simulation data at the last simulated time
# compute an estimate of the shear-stress coefficient and
# output both shear and pressure along the cylinder and flare.
#
# PJ, 06-June-2013

import sys, os
job = "cyl-flare"
print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx

print "Begin: Pick up data for tindx=", tindx
from libprep3 import Vector, cross, dot, vabs
from e3_flow import read_all_blocks
from math import sqrt
#
nb = 24
pick_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22] # blocks against surface
rho_inf = 8.81e-4 # kg/m**3
p_inf = 31.88 # Pa
u_inf =  2304.0 # m/s
T_inf = 120.4 # K
T_wall = 295.2 # K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'N2')
mm = 0.001  # metres
R = 32.5*mm
xcorner = 101.7*mm
corner = Vector(xcorner,R)
#
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute shear stress for cell-centres along the surface"
outfile = open("surface.data", "w")
outfile.write("# x(m) s(m) tau_w(Pa) Cf Cf_blasius y_plus p(Pa) Cp q(W/m**2) Ch\n")
for ib in pick_list:
    j = 0 # surface is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k]
        y = flow[ib].data['pos.y'][i,j,k]
        ctr = Vector(x, y)
        # Get vertices on surface, for this cell.
        x = grid[ib].x[i,j,k]
        y = grid[ib].y[i,j,k]
        vtx0 = Vector(x, y)
        x = grid[ib].x[i+1,j,k]
        y = grid[ib].y[i+1,j,k]
        vtx1 = Vector(x, y)
        t1 = (vtx1-vtx0)
        t1.norm() # tangent vector for surface
        midpoint = 0.5*(vtx0+vtx1) # on surface
        normal = cross(Vector(0,0,1),t1)
        normal.norm()
        # Surface to cell-centre distance.
        dy = dot(normal, ctr-midpoint)
        # Distance along surface
        if midpoint.x <= xcorner:
            # Along the cylinder surface.
            s = midpoint.x
        else:
```

```
            # Up the flare surface.
            s = vabs(midpoint-corner) + xcorner
        # Cell-centre flow data.
        rho = flow[ib].data['rho'][i,j,k]
        ux = flow[ib].data['vel.x'][i,j,k]
        uy = flow[ib].data['vel.y'][i,j,k]
        v = Vector(ux, uy)
        vt = dot(v,t1) # velocity component tangent to surface
        mu = flow[ib].data['mu'][i,j,k]
        kgas = flow[ib].data['k[0]'][i,j,k]
        p = flow[ib].data['p'][i,j,k]
        Cp = (p-p_inf)/(0.5*rho_inf*u_inf*u_inf)
        T = flow[ib].data['T[0]'][i,j,k]
        # Shear stress
        dudy = (vt - 0.0) / dy # no-slip wall
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * dy * rho / mu
        Rex = rho_inf * u_inf * s / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        # Heat flux
        dTdy = (T - T_wall) / dy # conductive heat flux at the wall
        q = kgas * dTdy
        Ch = q / (0.5*rho_inf*u_inf*u_inf*u_inf)
        #
        outfile.write("%f %f %f %f %f %f %f %f %f %f\n" %
                      (midpoint.x, s, tau_w, Cf, Cf_blasius,
                       y_plus, p, Cp, q, Ch))
        print "s=", s, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus, \
            "p=", p, "Cp=", Cp, "q=", q, "Ch=", Ch
outfile.close()
print "Done"
```

```
#! /usr/bin/env python
# separation_point.py
#
# Pick up the simulation data at all time frames.
# Search for the zero-crossing of ux to identify the separation point
# on the cylinder surface.
#
# PJ, 07-June-2013
print "Begin..."
import sys, os
from e3_flow import read_all_blocks
#
nb = 24
pick_list = [0, 2, 4, 6, 8, 10] # blocks against cylinder only
job = "cyl-flare"
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
times = []; xzero = []
for item in lines:
    items = item.strip().split()
    if items[0] == '#': continue
    tindx = int(items[0])
    if tindx == 0: continue
    t = float(items[1])
    print "Begin: Pick up data for tindx=", tindx, "t=", t
    grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
    x = []; y = []; ux = []
    for ib in pick_list:
        j = 0 # surface is along the South boundary
        k = 0 # of a 2D grid
        for i in range(flow[ib].ni):
            # Cell closest to surface
            x.append(flow[ib].data['pos.x'][i,j,k])
            ux.append(flow[ib].data['vel.x'][i,j,k])
    # Find the zero-crossing interval,
    # assuming that we start with positive velocity.
```

```
    # For no zero-crossing we run to the end.
    i = 0
    while ux[i] >= 0.0 and i < len(ux)-1: i += 1
    # Linearly interpolate the zero-crossing point.
    frac = ux[i-1]/(ux[i-1]-ux[i])
    xzero.append((1.0-frac)*x[i-1] + frac*x[i])
    times.append(t)
    print "t=", t, "xzero=", xzero[-1]

outfile = open("separation-location.data", "w")
outfile.write("# t(s) x(m)\n")
for i in range(len(xzero)):
    outfile.write("%f %f\n" % (times[i], xzero[i]))
outfile.close()

outfile = open("separation-velocity.data", "w")
outfile.write("# t(s) -dx/dt(m/s)\n")
for i in range(1,len(xzero)):
    outfile.write("%f %f\n" % (times[i], -(xzero[i]-xzero[i-1])/(times[i]-times[i-1])))
outfile.close()

print "Fit an asymptotic function to the location data."
import numpy
x = numpy.array(xzero) * 1000.0 # to get units of mm
t = numpy.array(times) * 1000.0 # to get units of ms
def f(t, xf, dx, tau):
    return xf + dx * numpy.exp(-t/tau)
from scipy.optimize import curve_fit
popt, pcov = curve_fit(f, t, x, [60.0, 30.0, 0.8])
print "Fitted parameters:"
print "xf=", popt[0], "mm"
print "dx=", popt[1], "mm"
print "tau=", popt[2], "ms"
print "pcov=", pcov
print "Done"
```

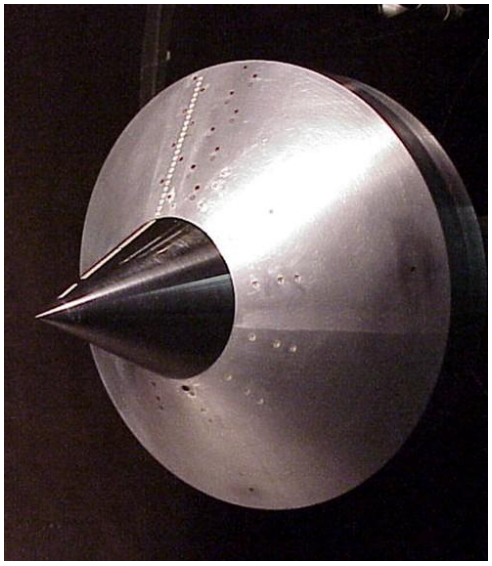
26.5 Notes

- The experimental data has come from a spreadsheet, kindly provided by Dr Matthew MacLean of CUBRC. Plotting was done with the following GNUPlot scripts.

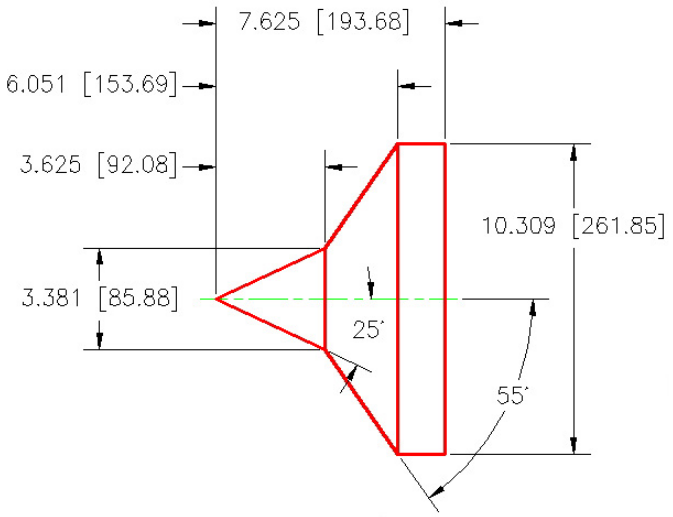
```
# surface-pressure.gnuplot
set term postscript eps 20
set output 'surface-pressure.eps'
set title 'Cylinder with extended flare, pressure along surface'
set ylabel 'p, Pa'
set xlabel 'x, mm'
set key top left
plot './surface.data' using ($1*1000):($7) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/cylinder-extended-flare-pressure.data' \
     using ($2*101.7):($10*6894.8) \
     title 'CUBRC Run 14' with points pt 4

# surface-heat-transfer.gnuplot
set term postscript eps 20
set output 'surface-heat-transfer.eps'
set title 'Cylinder with extended flare, heat-flux along surface'
set ylabel 'q, kW/m**2'
set xlabel 'x, mm'
set yrange [0:300]
set key top left
plot './surface.data' using ($1*1000):($9/1000) with lines \
     lw 3.0 title 'Eilmer3 k*dT/dy', \
     './notes/cylinder-extended-flare-heat-transfer.data' \
     using ($2*101.7):($10*11.377) \
     title 'CUBRC Run 14' with points pt 4
```

27 Hypersonic flow over a double-cone.

This is another of the hypersonic test flows provided by the Calspan-University of Buffalo Research Center (CUBRC) as part of an experimental campaign [14, 15]. The experimental facility provides a Mach 12.49 flow of nitrogen onto the double cone shown below in Figure 48. As for the hollow-cylinder with flare case in Section 26, it is an example that retains a very simple geometric arrangement for the flow boundaries, however, the stronger shock interaction with the steeper cone surface produces a more complex flow in this case. Despite this complexity, the overall flow eventually settles to a steady state and we again assume that the boundary layer remains laminar.



(a) Physical model.



(b) Geometric definition from [16]. Dimensions in inches and mm.

Figure 48: Double-cone with sharp nose used in the CUBRC experiments.

27.1 Input script (.py)

In setting up this exercise, we follow the details provided by MacLean [16] and concentrate on the CUBRC Run 35 experiment. We assume an ideal nitrogen free stream, with conditions $p = 18.55\,\text{Pa}$, ρ=0.6081 g/m^3, $u = 2.576\,\text{km/s}$ and a static temperature T=102.2 K. The actual nitrogen flow in the shock tunnel nozzle was far from ideal and had an estimated vibrational temperature of 2711 K. However, for the simulation reported here, this vibrational energy is assumed frozen and thus ignored. The model surface temperature was a constant 295.8 K.

```
# dbl-cone.py
# PJ, 12-June-2013
# Model of the CUBRC double-cone with sharp nose.
```

```
gdata.title = "Double-cone, sharp nose."
print gdata.title
gdata.axisymmetric_flag = 1
# Conditions to match those reported for CUBRC Run 35
p_inf = 18.55 # Pa
u_inf = 2576.0 # m/s
T_inf = 102.2 # degree K
T_vib = 2711.0 # degrees K (but we will ignore for ideal-gas)
select_gas_model(model='ideal gas', species=['N2'])
inflow  = FlowCondition(p=p_inf, u=u_inf, T=T_inf)
initial = FlowCondition(p=p_inf/5, u=0, T=T_inf)
T_wall = 295.8 # degree K

mm = 1.0e-3 # metres per mm
a0 = Vector(0.0, 0.0)
a1 = Vector(0.0,5*mm) # leading edge of domain
b0 = Vector(92.08*mm,42.94*mm) # junction between cones
b1 = Vector(76*mm,61*mm) # out in the free stream
c0 = Vector(153.69*mm,130.925*mm) # downstream-edge of second cone
c1 = Vector(124*mm,181*mm) # out in the free stream
d0 = Vector(193.68*mm,130.925*mm) # down-stream edge of domain
d1 = Vector(193.68*mm,181*mm)

rcfx = RobertsClusterFunction(1,0,1.2)
rcfy = RobertsClusterFunction(1,0,1.1)
ni0 = 120; nj0 = 40 # We'll scale discretization off these values
factor = 2
ni0 *= factor; nj0 *= factor

cone1 = SuperBlock2D(CoonsPatch(a0,b0,b1,a1),
                     nni=ni0, nnj=nj0, nbi=6, nbj=2,
                     bc_list=[SupInBC(inflow),None,
                              FixedTBC(T_wall),SupInBC(inflow)],
                     cf_list=[rcfx,rcfy,rcfx,rcfy],
                     fill_condition=inflow, label="cone1")
cone2 = SuperBlock2D(CoonsPatch(b0,c0,c1,b1),
                     nni=ni0, nnj=nj0, nbi=6, nbj=2,
                     bc_list=[SupInBC(inflow),None,
                              FixedTBC(T_wall),None],
                     cf_list=[None,rcfy,None,rcfy],
                     fill_condition=initial, label="cone2")
cone3 = SuperBlock2D(CoonsPatch(c0,d0,d1,c1),
                     nni=int(ni0/2), nnj=nj0, nbi=2, nbj=2,
                     bc_list=[SupInBC(inflow),FixedPOutBC(p_inf/5),
                              FixedTBC(T_wall),None],
                     cf_list=[None,rcfy,None,rcfy],
                     fill_condition=initial, label="out")
identify_block_connections()

# Do a little more setting of global data.
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 1.0
# The settling of the separation bubble will probably dominate.
gdata.max_time = 5.0e-3 # long enough, maybe
gdata.max_step = 4000000
gdata.dt = 1.0e-9
gdata.dt_plot = 0.25e-3

sketch.xaxis(0.0, 0.250, 0.05, -0.010)
sketch.yaxis(0.0, 0.250, 0.05, -0.010)
sketch.window(0.0, 0.0, 0.250, 0.250, 0.05, 0.05, 0.25, 0.25)
```

27.2 Running the simulation

Figure 49 shows the flow region, as modelled for simulation. The region is very simple but we have divided it into 28 blocks so that the computational load can be shared across a number of CPU cores.

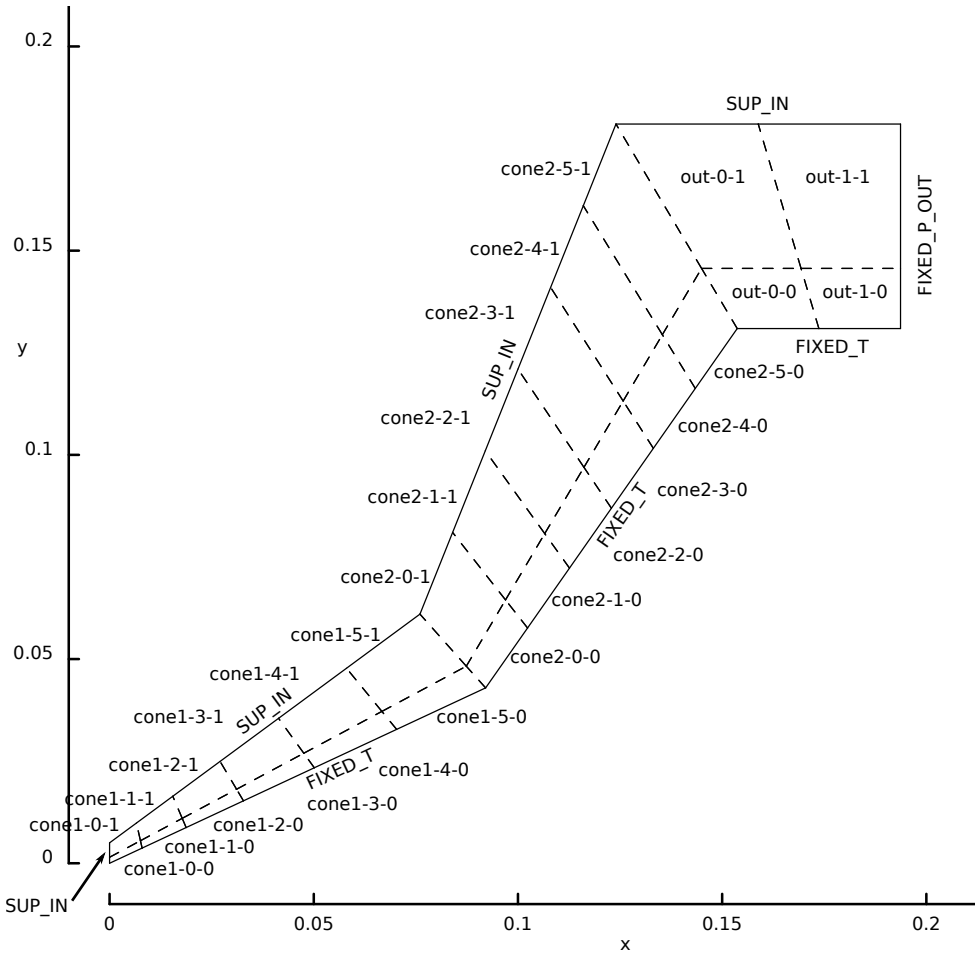


Figure 49: Schematic view of the simulated flow region for the hypersonic flow over a double-cone with sharp nose.

In terms of required computer time, this simulation is more demanding than the cylinder-flare example. The job scipts submitted to the batch system are shown below. The preparation of the grids and initial flow-state files was done on a local workstation, these files transferred to the cluster computer file system ("arcus", located at the Oxford e-Research Centre) and then the simulation was done in two stages, 0–1.75 ms and 1.75 ms–3 ms. Note that the 28 blocks have been grouped, via the mpimap file (that was generated by the e3loadbalance program), to 14 MPI tasks. The first simulation stage required just over two days on 14 cores of the arcus cluster, with the `max-wall-clock` option being used to terminate the simulation cleanly after 50 hours has elapsed. Before restarting the simulation, the `.control` file was edited to set `max_time` to 3 ms. The second-stage run script (`run-arcus-2.sh`) then restarted the simulation from solution frame 7 (*i.e.*

1.75 ms).

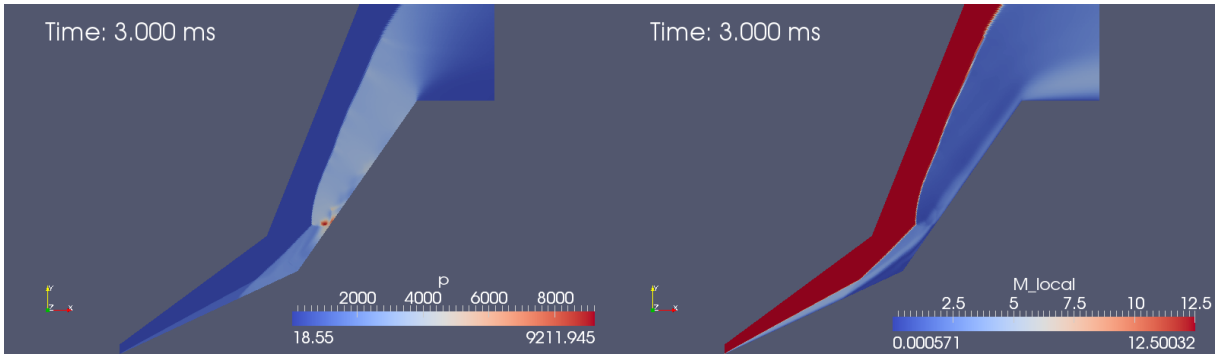
```
#! /bin/bash
# prep.sh
e3prep.py --job=dbl-cone --do-svg
e3loadbalance.py --job=dbl-cone -n 14
e3post.py --job=dbl-cone --tindx=0 --vtk-xml

echo "At this point, we should have a grid."
echo "Use run.sh next"
```

```
#! /bin/bash
# run-arcus.sh
#PBS -l select=1:mpiprocs=14
#PBS -l walltime=51:00:00
#PBS -N dbl-cone
#PBS -m bea
#PBS -M peter.jacobs@eng.ox.ac.uk
#PBS -V
cd $PBS_O_WORKDIR
date
mpirun -np 14 -machinefile $PBS_NODEFILE $DATA/e3bin/e3mpi.exe \
    --job=dbl-cone --mpimap=dbl-cone.mpimap --run \
    --max-wall-clock=180000 > run-arcus.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

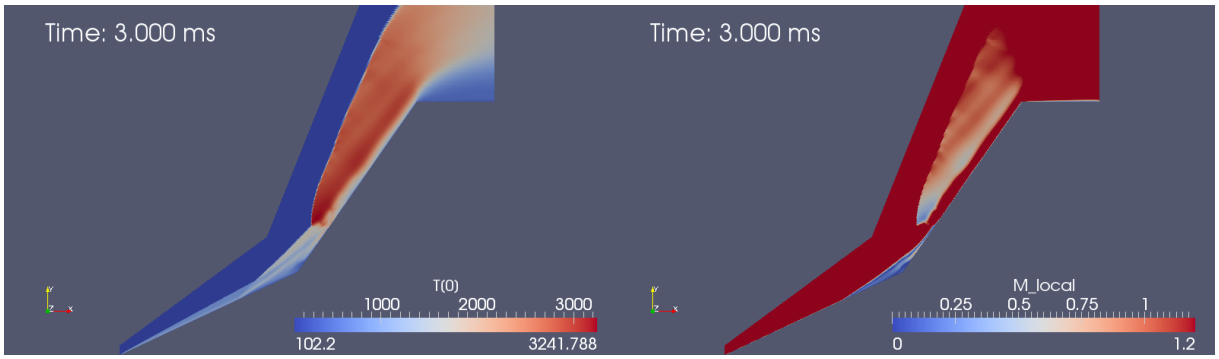
```
#! /bin/bash
# run-arcus-2.sh
#PBS -l select=1:mpiprocs=14
#PBS -l walltime=51:00:00
#PBS -N dbl-cone
#PBS -m bea
#PBS -M peter.jacobs@eng.ox.ac.uk
#PBS -V
cd $PBS_O_WORKDIR
date
mpirun -np 14 -machinefile $PBS_NODEFILE $DATA/e3bin/e3mpi.exe \
    --job=dbl-cone --mpimap=dbl-cone.mpimap --tindx=7 --run \
    --max-wall-clock=180000 > run-arcus-3.transcript
date

echo "At this point, we should have a flow solution"
echo "Use post.sh next"
```

27.3 Results

Figure 50 shows some of the flow field data at t=3 ms after flow start. In the pressure field, the attached shock from the sharp tip of the cone and the shock caused by the boundary-layer separation are both clearly defined and can be seen to merge about three-quarters of the way along the first cone. This combined shock interacts strongly with the flow up the second cone surface and a Mach stem is formed on top af a supersonic jet running up along the cone surface. The Mach number field, rescaled to highlight the subsonic regions (Figure 50d), shows clearly the separation region in the junction between the conical surfaces, the large subsonic region behind the detached shock over the 55^o cone, and the supersonic jet up the surface of that cone.



(a) Pressure field. (b) Mach number.

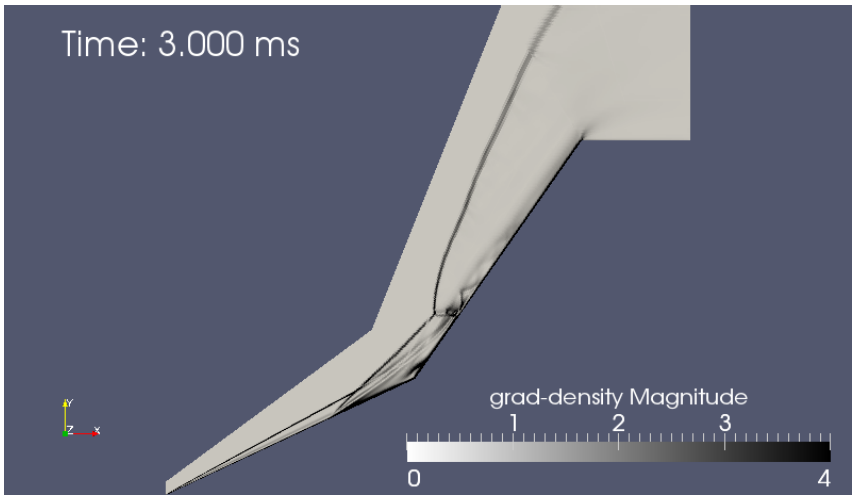


(c) Temperature field. (d) Mach number rescaled.

Figure 50: Computed flow field at t=3 ms.

The details of the separation, Mach stem and subsequent jet stream are quite complex and some features, such as shear layers and the shocks within the supersonic jet, are more clearly shown by visualizing the gradient of density, as shown in Figure 51a. This flow has been more carefully studied in Ref.[17] so read that paper if you want to learn more about the physics of this flow. Here, we are interested only in demonstration how to set up a the simulation with `Eilmer` and that the code does indeed produce correct results.

By the 3 ms time shown in Figures 50 and 51, the flow has settled to a steady-state configuration, as confirmed by the history of the separation point location on the first



(a) Gradient of density field.

Figure 51: Computed flow field at t=3 ms.

conical surface. The data, plotted in Figure 52, shows a close approach to the asymptotic value of 62.1 mm by a time of 3 ms. The separation point was detected simply as a reversal of the x-velocity, as seen in the first postprocessing script in Section 27.4.

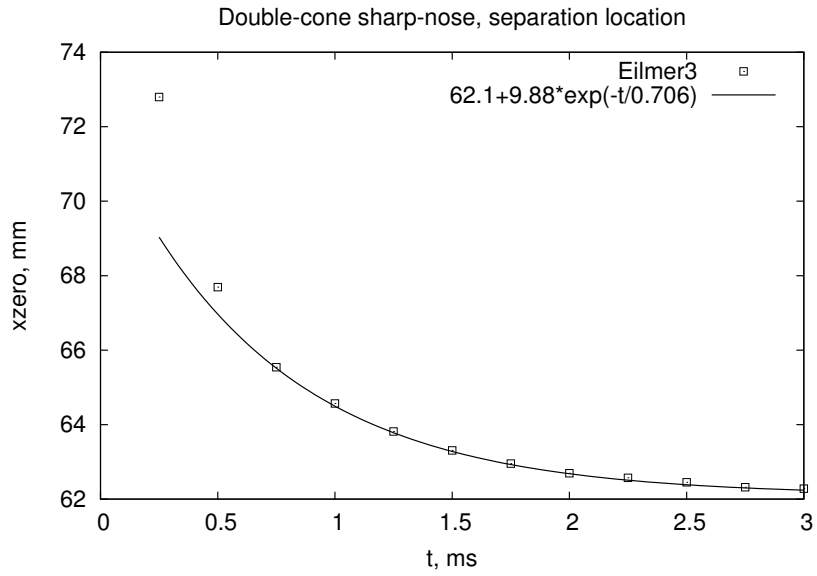
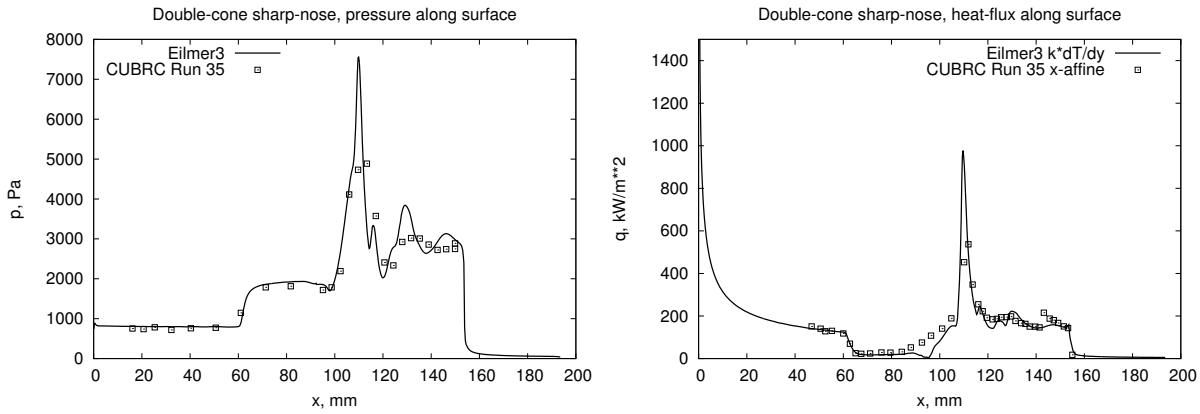


Figure 52: History of the separation location along the first conical surface.

As another validation case, the real proof of success is in comparison with the experimental data. Figure 53 shows the pressure and heat-transfer along the surface of both cones. The plot uses the model axial-coordinate rather than distance along the surface to match the presentation by MacLean [16] and the spreadsheet record of data from the experiments [14, 15]. The simulation has done a good job of estimating the pressure distribution right through the separation zone and the shock-interaction zone on the second cone's surface. The separation bubble appears to be well captured in position and extent.



(a) Pressure.

(b) Heat transfer.

Figure 53: Distribution of pressure and heat transfer along the double cone with sharp nose. Simulation data is recorded at t=3 ms into the simulation. Experimental data is for Run 35 of the CUBRC experiment [14].

The simulation has also done a good job on the heat transfer estimate, which has been computed from the field data using the second script in Section 27.4. It is reassuring that the simulation has accruately captured the heat transfer in the boundary layer leading into the sepration region, through the separation, and also after the interaction region on the flare surface.

27.4 Postprocessing heat transfer and separation-point tracking

The scripts below use the functions imported from `e3_flow.py` at a slightly higher level than in the cone20 example. The first looks at the x-component of the velocity of the first cell above the conical surface to identify the location of the start of the separation region for all frames of the solution. After writing the location data to a file, it uses the SciPy optimization module to fit a simple function to that data, in order to estimate the asymptotic position of the separation point for large times. The second extracts the data for the cell nearest to the cone surface and uses that data to compute the expected shear stress and heat transfer at the surface.

```
#! /usr/bin/env python
# dbl_cone_separation_point.py
#
# Pick up the simulation data at all time frames.
# Search for the zero-crossing of ux to identify the separation point
# on the cone surface.
#
# PJ, 25-June-2013, adapted from cylinder-flare case.
print "Begin..."
import sys, os
from e3_flow import read_all_blocks
#
nb = 28
pick_list = [0, 2, 4, 6, 8, 10] # blocks against cylinder only
job = "dbl-cone"
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
times = []; xzero = []
for item in lines:
    items = item.strip().split()
    if items[0] == '#': continue
    tindx = int(items[0])
    if tindx == 0: continue
    t = float(items[1])
    print "Begin: Pick up data for tindx=", tindx, "t=", t
    grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
    x = []; y = []; ux = []
    for ib in pick_list:
        j = 0 # surface is along the South boundary
        k = 0 # of a 2D grid
        for i in range(flow[ib].ni):
            # Cell closest to surface
            x.append(flow[ib].data['pos.x'][i,j,k])
            ux.append(flow[ib].data['vel.x'][i,j,k])
    # Find the zero-crossing interval,
    # assuming that we start with positive velocity.
    # For no zero-crossing we run to the end.
    i = 0
    while ux[i] >= 0.0 and i < len(ux)-1: i += 1
    # Linearly interpolate the zero-crossing point.
    frac = ux[i-1]/(ux[i-1]-ux[i])
    xzero.append((1.0-frac)*x[i-1] + frac*x[i])
    times.append(t)
    print "t=", t, "xzero=", xzero[-1]

outfile = open("separation-location.data", "w")
outfile.write("# t(s) x(m)\n")
for i in range(len(xzero)):
    outfile.write("%f %f\n" % (times[i], xzero[i]))
outfile.close()

outfile = open("separation-velocity.data", "w")
outfile.write("# t(s) -dx/dt(m/s)\n")
for i in range(1,len(xzero)):
```

```
    outfile.write("%f %f\n" % (times[i], -(xzero[i]-xzero[i-1])/(times[i]-times[i-1])))
outfile.close()

print "Fit an asymptotic function to the location data."
import numpy
x = numpy.array(xzero) * 1000.0 # to get units of mm
t = numpy.array(times) * 1000.0 # to get units of ms
# Drop the first couple of points from the fit.
x = x[2:]; t = t[2:]
def f(t, xf, dx, tau):
    return xf + dx * numpy.exp(-t/tau)
from scipy.optimize import curve_fit
popt, pcov = curve_fit(f, t, x, [60.0, 20.0, 0.4])
print "Fitted parameters:"
print "xf=", popt[0], "mm"
print "dx=", popt[1], "mm"
print "tau=", popt[2], "ms"
print "pcov=", pcov
print "Done"
```

```
#! /usr/bin/env python
# dbl_cone_surface_properties.py
#
# Pick up the simulation data at the last simulated time
# compute an estimate of the shear-stress coefficient and
# output both shear and pressure along the cone surfaces.
#
# PJ, 25-June-2013, adapted from cylinder-flare example

import sys, os
job = "dbl-cone"
print "Determine the latest time."
fp = open(job+".times", "r"); lines = fp.readlines(); fp.close()
tindx = int(lines[-1].strip().split()[0]) # first number of the last line
print "tindx=", tindx

print "Begin: Pick up data for tindx=", tindx
from libprep3 import Vector, cross, dot, vabs
from e3_flow import read_all_blocks
from math import sqrt
#
nb = 28
pick_list = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26] # surface
rho_inf = 6.081e-4 # kg/m**3
p_inf = 18.55 # Pa
u_inf =  2576.0 # m/s
T_inf = 102.2 # K
T_wall = 295.8 # K
from cfpylib.gasdyn import sutherland
mu_inf = sutherland.mu(T_inf, 'N2')
mm = 0.001  # metres
corner1 = Vector(92.08,42.94)*mm
corner2 = Vector(153.69,130.925)*mm
#
grid, flow, dim = read_all_blocks(job, nb, tindx, zipFiles=True)
print "Compute properties for cell-centres along the surface"
outfile = open("surface.data", "w")
outfile.write("# x(m) s(m) tau_w(Pa) Cf Cf_blasius y_plus p(Pa) Cp q(W/m**2) Ch\n")
for ib in pick_list:
    j = 0 # surface is along the South boundary
    k = 0 # of a 2D grid
    print "# start of block"
    for i in range(flow[ib].ni):
        # Cell closest to surface
        x = flow[ib].data['pos.x'][i,j,k]
        y = flow[ib].data['pos.y'][i,j,k]
        ctr = Vector(x, y)
        # Get vertices on surface, for this cell.
        x = grid[ib].x[i,j,k]
```

```
        y = grid[ib].y[i,j,k]
        vtx0 = Vector(x, y)
        x = grid[ib].x[i+1,j,k]
        y = grid[ib].y[i+1,j,k]
        vtx1 = Vector(x, y)
        t1 = (vtx1-vtx0)
        t1.norm() # tangent vector for surface
        midpoint = 0.5*(vtx0+vtx1) # on surface
        normal = cross(Vector(0,0,1),t1)
        normal.norm()
        # Surface to cell-centre distance.
        dy = dot(normal, ctr-midpoint)
        # Distance along surface
        if midpoint.x <= corner1.x:
            # Along the first-cone.
            s = vabs(midpoint)
        elif midpoint.x <= corner2.x:
            # Up the second cone.
            s = vabs(midpoint-corner1) + vabs(corner1)
        else:
            # Along the top surface.
            s = vabs(midpoint-corner2) + vabs(corner1) + vabs(corner2-corner1)
        # Cell-centre flow data.
        rho = flow[ib].data['rho'][i,j,k]
        ux = flow[ib].data['vel.x'][i,j,k]
        uy = flow[ib].data['vel.y'][i,j,k]
        v = Vector(ux, uy)
        vt = dot(v,t1) # velocity component tangent to surface
        mu = flow[ib].data['mu'][i,j,k]
        kgas = flow[ib].data['k[0]'][i,j,k]
        p = flow[ib].data['p'][i,j,k]
        Cp = (p-p_inf)/(0.5*rho_inf*u_inf*u_inf)
        T = flow[ib].data['T[0]'][i,j,k]
        # Shear stress
        dudy = (vt - 0.0) / dy # no-slip wall
        tau_w = mu * dudy    # wall shear stress
        Cf = tau_w / (0.5*rho_inf*u_inf*u_inf)
        u_tau = sqrt(abs(tau_w) / rho) # friction velocity
        y_plus = u_tau * dy * rho / mu
        Rex = rho_inf * u_inf * s / mu_inf
        Cf_blasius = 0.664 / sqrt(Rex)
        # Heat flux
        dTdy = (T - T_wall) / dy # conductive heat flux at the wall
        q = kgas * dTdy
        Ch = q / (0.5*rho_inf*u_inf*u_inf*u_inf)
        #
        outfile.write("%f %f %f %f %f %f %f %f %f %f\n" %
                      (midpoint.x, s, tau_w, Cf, Cf_blasius,
                       y_plus, p, Cp, q, Ch))
        print "s=", s, "tau_w=", tau_w, "Cf=", Cf, "y_plus=", y_plus, \
            "p=", p, "Cp=", Cp, "q=", q, "Ch=", Ch
outfile.close()
print "Done"
```

27.5 Notes

- The experimental data has come from a spreadsheet, kindly provided by Dr Matthew MacLean of CUBRC. Plotting of the pressure was done using dimensional quantities directly with the following GNUPlot script.

```
# surface-pressure.gnuplot
set term postscript eps 20
set output 'surface-pressure.eps'
set title 'Double-cone sharp-nose, pressure along surface'
set ylabel 'p, Pa'
set xlabel 'x, mm'
set key top left
```

```
plot './surface.data' using ($1*1000):($7) with lines \
     lw 3.0 title 'Eilmer3', \
     './notes/indented-cone-pressure.data' \
     using ($2*92.075):($9*6894.8) \
     title 'CUBRC Run 35' with points pt 4
```

- The experimental heat transfer data seemed to have incorrect x-positions for the transducers. x-position data from the spreadsheet was adjusted to correctly locate the transducer just before the separation point and the transducer toward the end of the second-cone surface as seen in the photograph of the physical model (Figure 48).

```
# affine.py
# Scale the x-position of the CUBRC heat-transfer data
# using x_mm = alpha * x/L + beta
# to match a two key points of the computational result.
# 1. separation point x/L=0.338 x=60mm
# 2. second-last transducer x/L=1.610 x=155mm
# Note that this transformation is not necessary for the pressure data.
# PJ, 25-June-2013
from numpy import array, linalg
a = array([[0.338, 1.0],[1.610, 1.0]])
b = array([60.0,155.0])
e = linalg.solve(a,b)
alpha, beta = e
print "alpha=", alpha, "beta=", beta
```

- All other heat-transfer transducer locations were then positioned relative to these points.

```
# surface-heat-transfer.gnuplot
set term postscript eps 20
set output 'surface-heat-transfer.eps'
set title 'Double-cone sharp-nose, heat-flux along surface'
set ylabel 'q, kW/m**2'
set xlabel 'x, mm'
set yrange [0:1500]
set key top right
plot './surface.data' using ($1*1000):($9/1000) with lines \
     lw 3.0 title 'Eilmer3 k*dT/dy', \
     './notes/indented-cone-heat-transfer.data' \
     using ($2*74.69+34.76):($9*11.377) \
     title 'CUBRC Run 35 x-affine' with points pt 4
```

28 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 [18]. 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 $y_e = 1.0$.

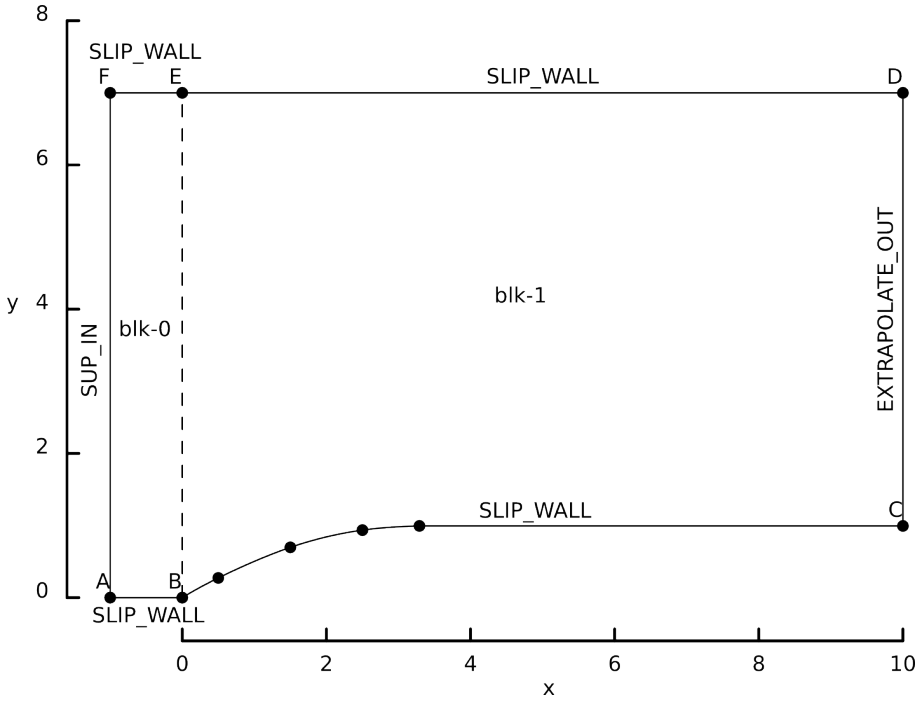


Figure 54: Schematic diagram of the geometry for the sharp body.

The surface pressure (shown in Fig. 56) 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. 57) shows the curved shock clearly.

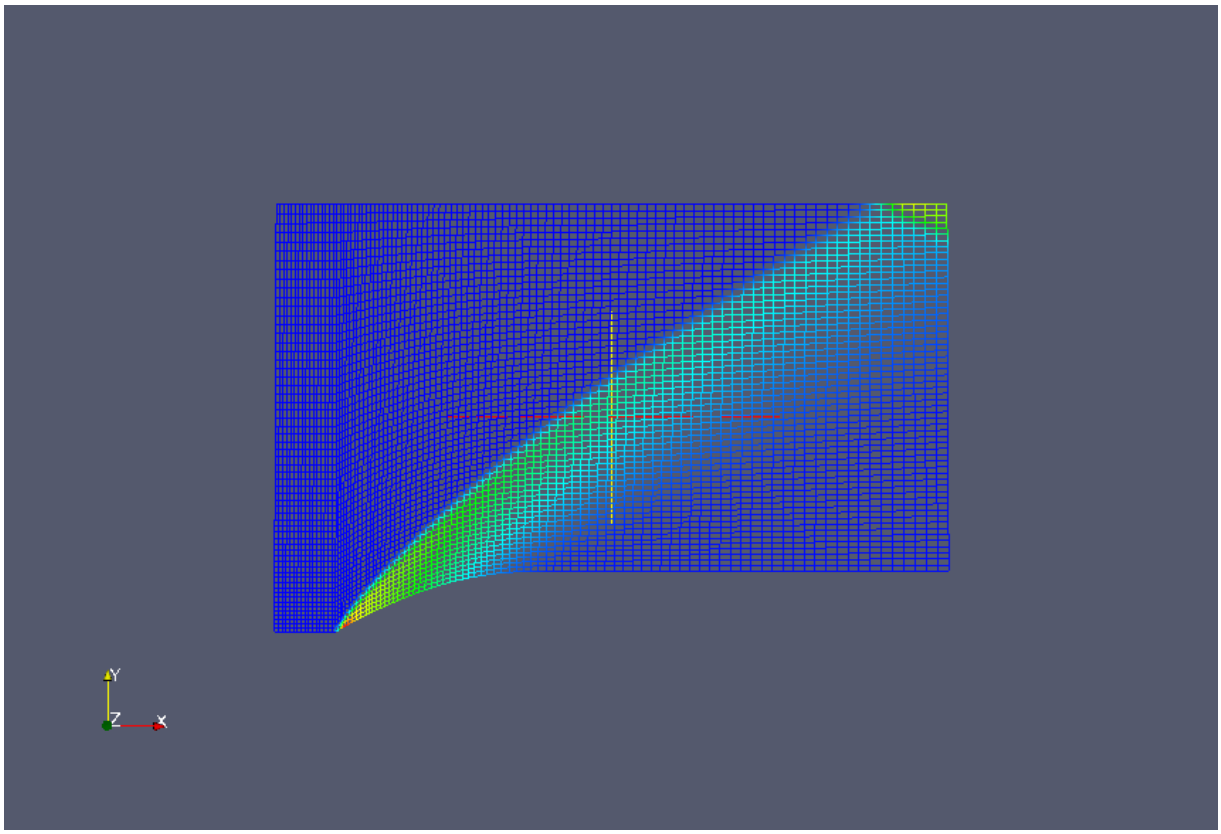


Figure 55: Mesh, coloured by pressure, for the sharp body exercise.

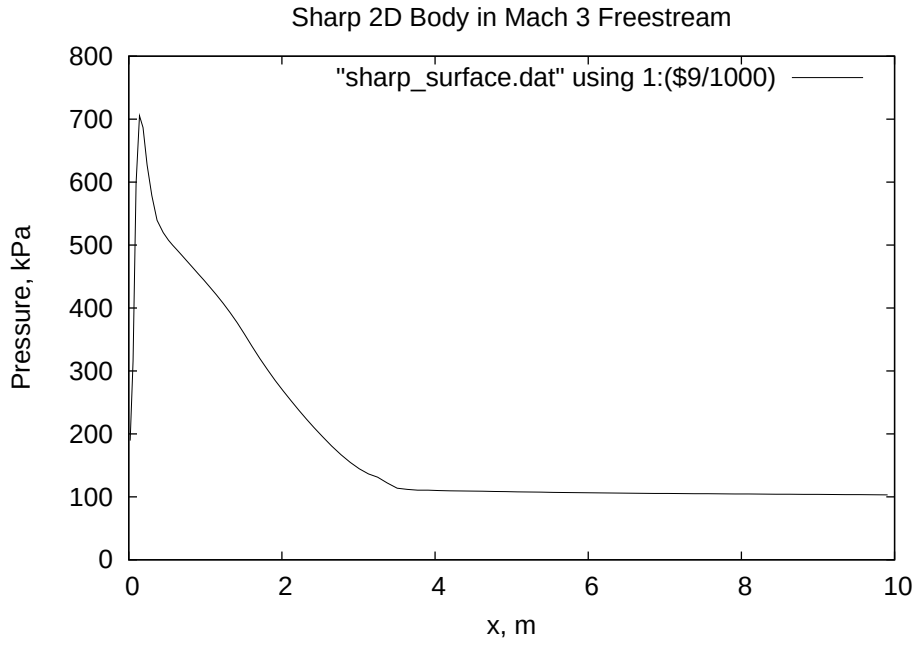


Figure 56: Pressure data along the body surface.

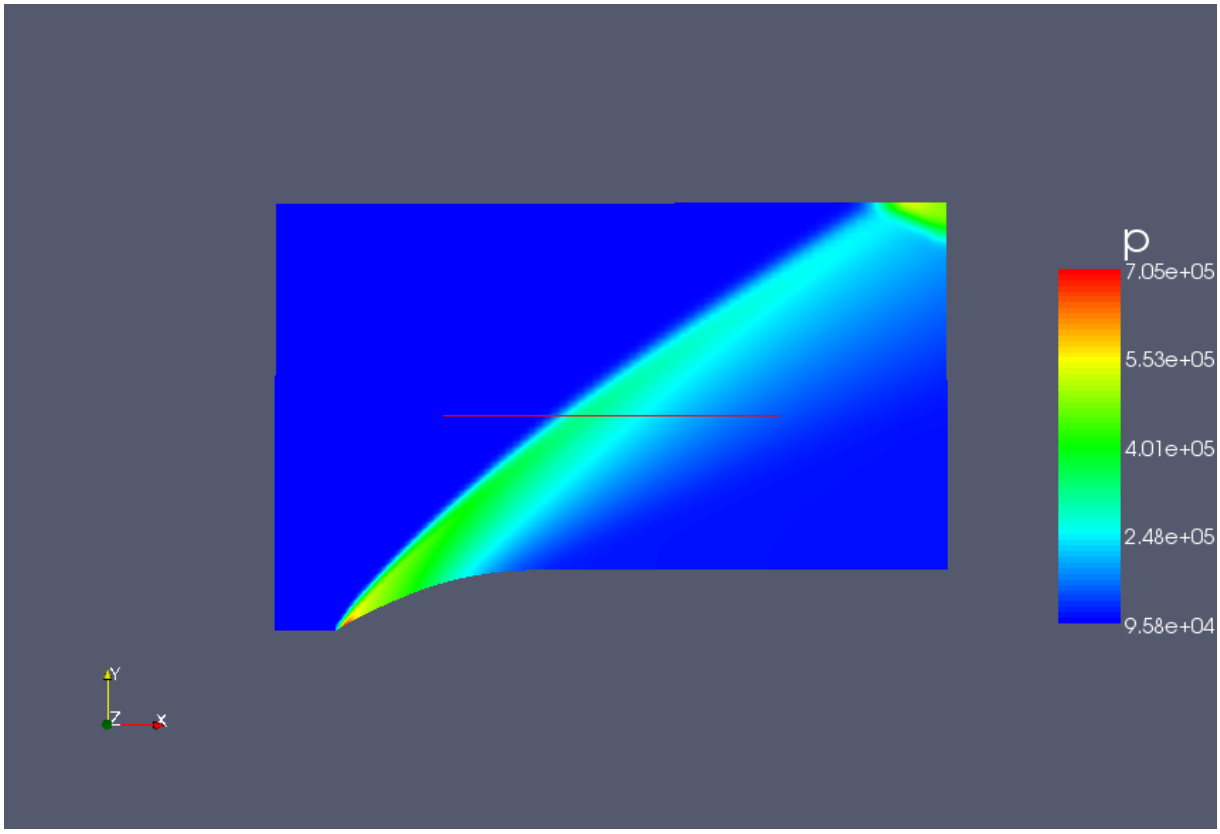


Figure 57: The pressure field for flow over a sharp body. The data has been transformed from cells to points in Paraview. Note that the shock reflects from the upper boundary, which has a SLIP_WALL boundary condition by default.

28.1 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)
```

28.2 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.
```

28.3 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.

29 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. [18].

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

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.

29.1 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
    yval = y(x)
    if yval < 0.0:
        yval = 0.0
    return (x, yval, 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)
```

29.2 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.

30 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 $q_\infty = 1.75\,\text{MPa}$.

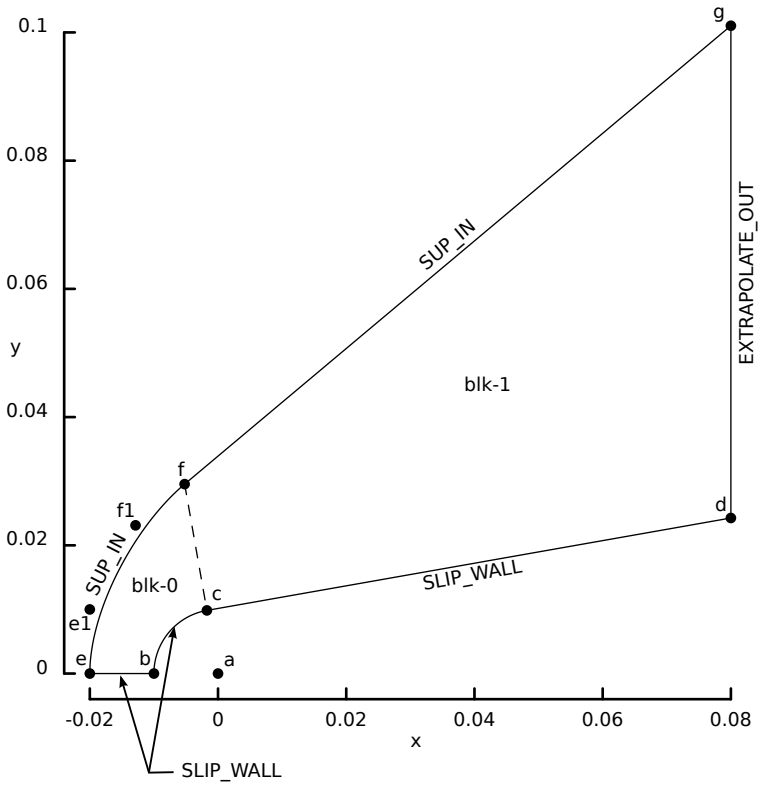


Figure 58: Schematic diagram of the geometry for the blunted 10 degree wedge.

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.60) 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. 61) 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.

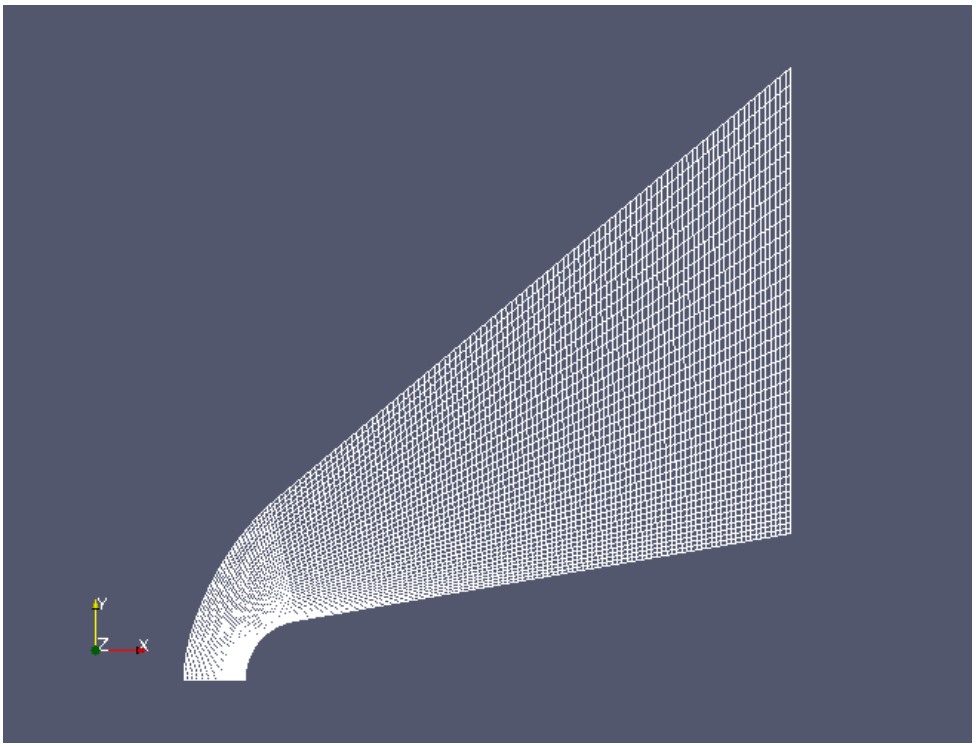


Figure 59: Mesh for the blunt wedge exercise.

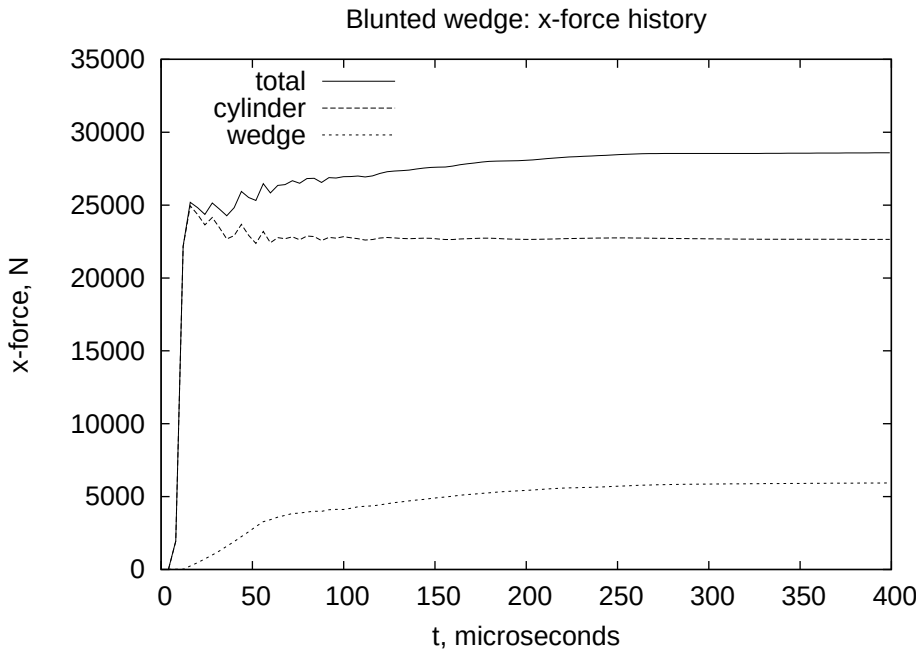


Figure 60: History of the axial forces for the blunt-wedge exercise.

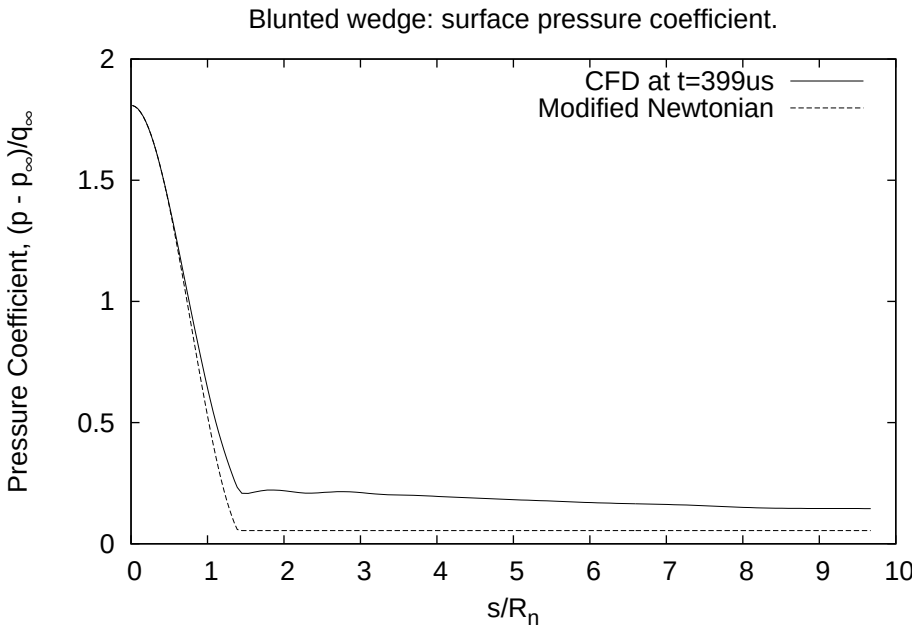


Figure 61: Surface pressure coefficient data for the blunt-wedge exercise.

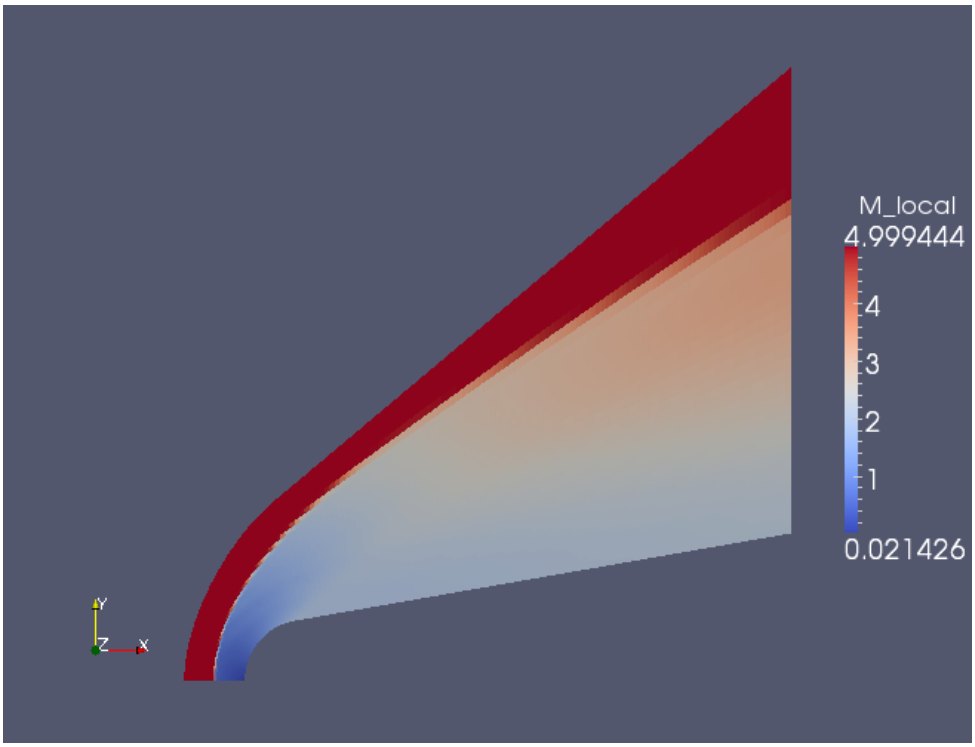


Figure 62: Mach number data for the blunt-wedge exercise.

30.1 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)
```

30.2 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
```

30.3 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;
}
```

31 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 63) 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.[19].

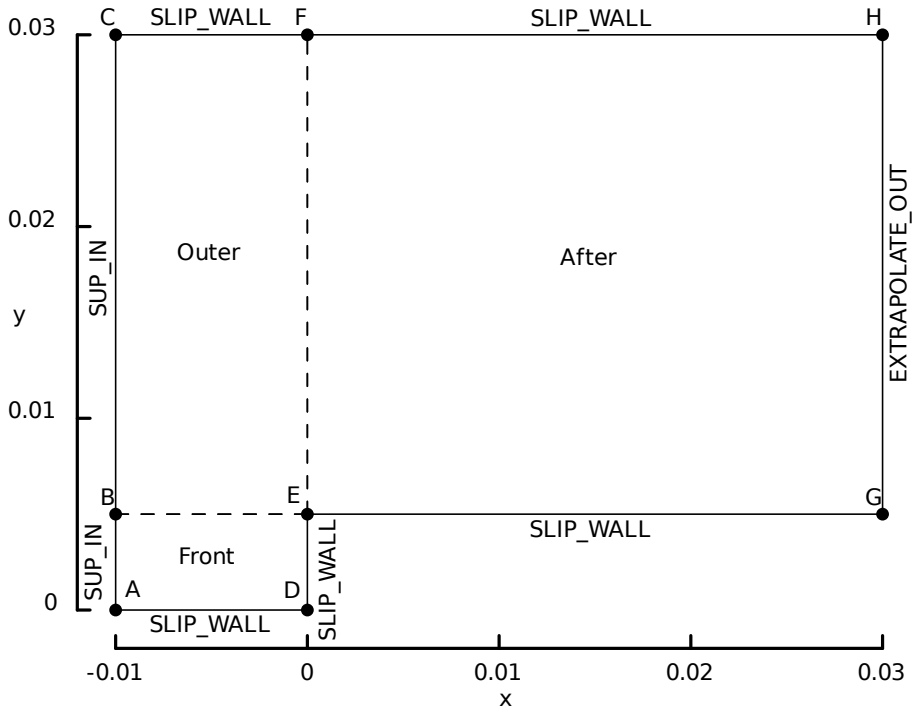


Figure 63: Schematic diagram of the full flow domain around the flat-faced cylinder.

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. 64. 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 65 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.

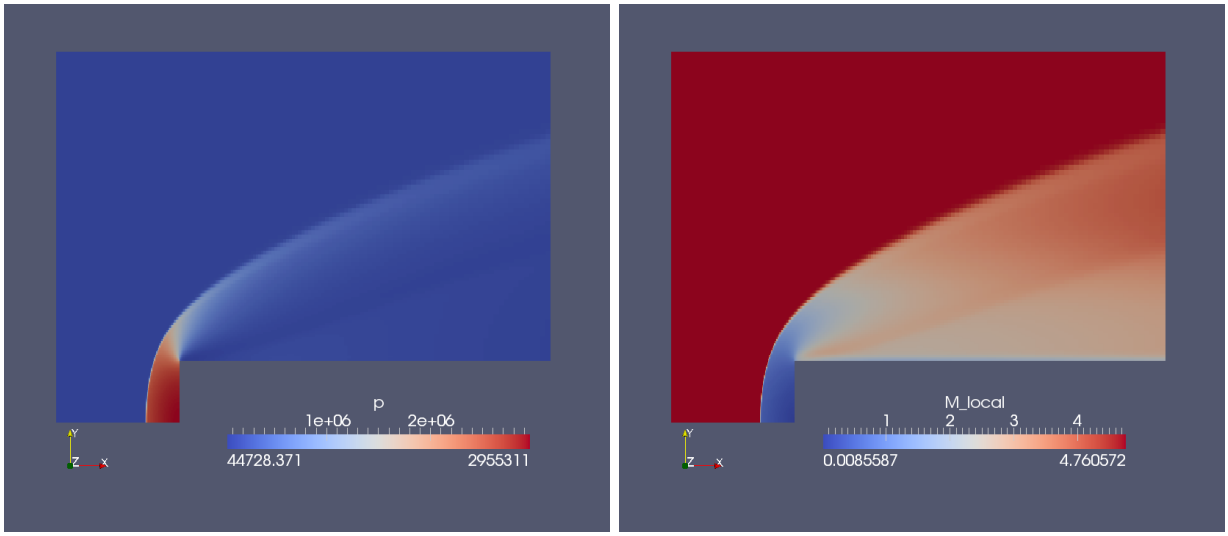


Figure 64: Pressure and Mach number within the flow domain at 50 μs.

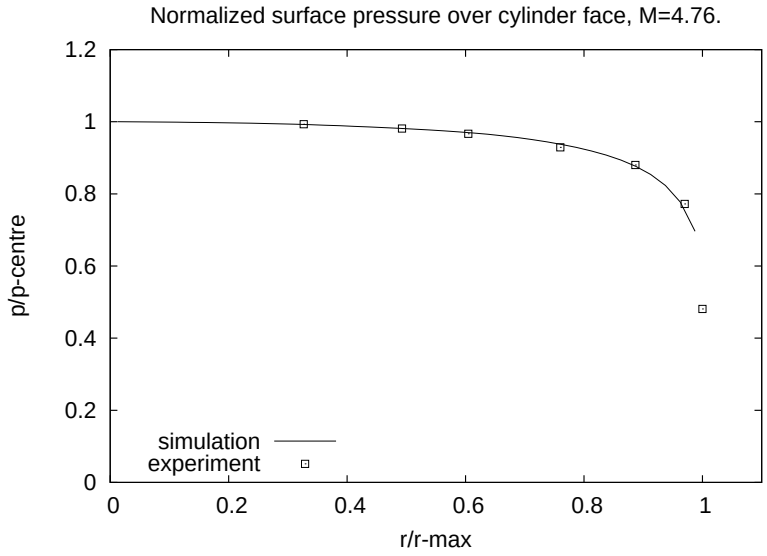


Figure 65: Normalised pressure across the face of the cylinder compared with experimental measurements [19].

31.1 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)
```

31.2 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 pt 4
EOF
```

31.3 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."
}
```

31.4 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...

32 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^o divergent section is available in Ref. [20]. 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.

Figure 66 shows the outline of the simulated flow domain which is set up to approximate the largest subsonic area ratio used in the experiment. A short subsonic section upstream of the throat is included, along with the conical supersonic expansion where the pressure measurements were made. 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. Although the parametric description makes the initial setup of the script a bit more complex than absolutely necessary, it does make the running of the simulation for other radii of curvature very simple.

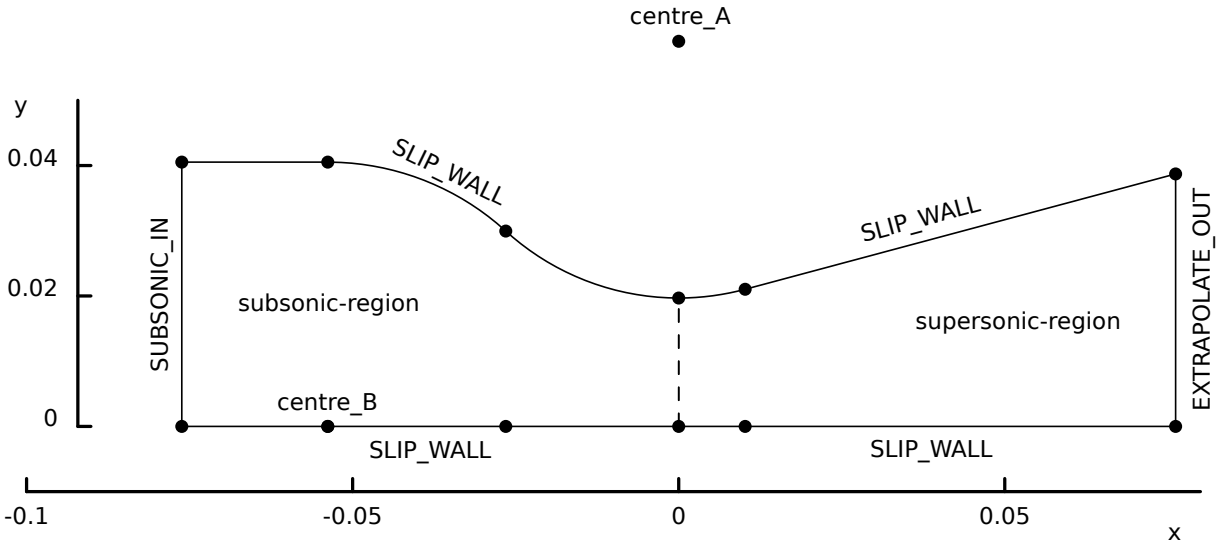


Figure 66: Schematic diagram of the full flow domain for the duct and conical nozzle.

Figure 67 shows the mesh, coloured by Mach number (once the flow has reached steady state). 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 $A_3/A_* = 4.2381$. This is seen to be consistent with the Mach number colouring in the figure and is a good test of the `SubsonicInBC` that is applied at the upstream boundary.

Figure 68 shows the pressure distribution throughout the flow domain at $t = 4.0\,\text{ms}$, once the flow has settled. Note that the inflow boundary has the flow stagnation properties specified as its flow condition but that this condition does not appear in any part of the simulation domain.

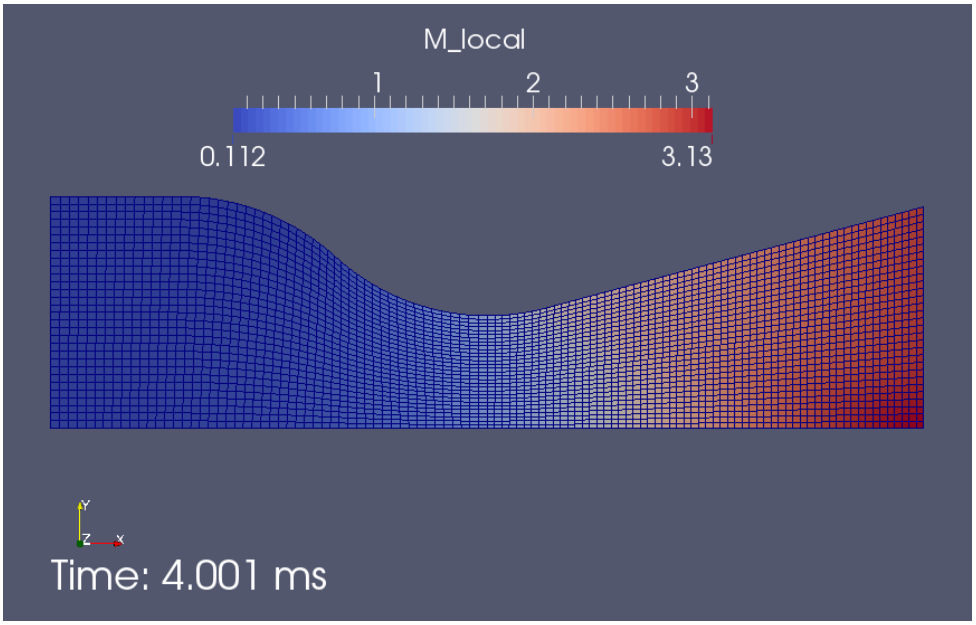


Figure 67: Mesh generated for the axisymmetric nozzle simulation, coloured with Mach number.

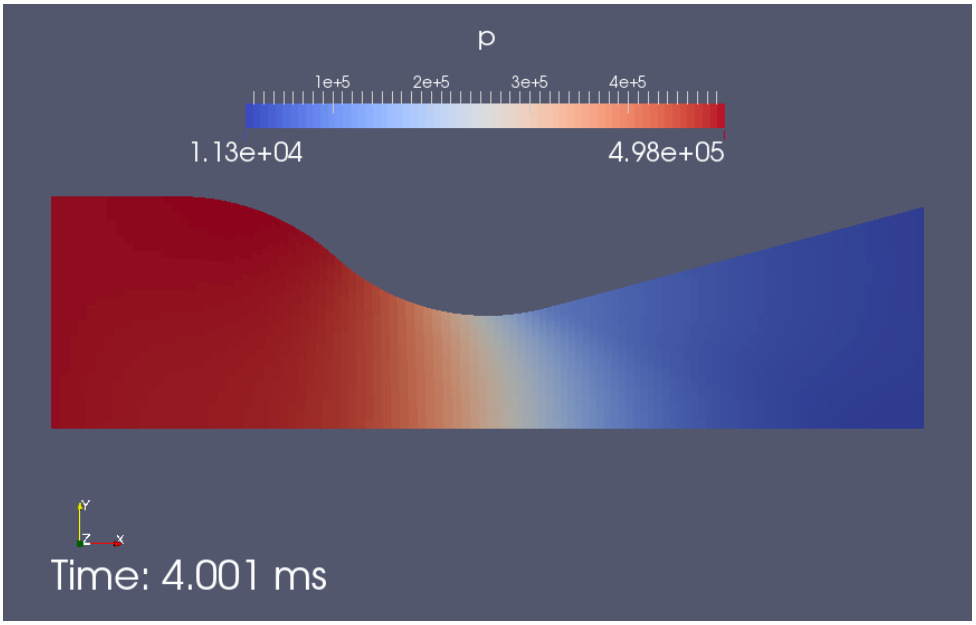


Figure 68: Pressure contours within the flow domain at 4.0 ms.

The flow in the nozzle is largely transient as the stagnation conditions drive gas into the domain but the overall flow becomes steady, as indicated by the histories shown in Fig. 69. Because there is little damping to the gas dynamics, small scale oscillations evident in the pressure history take some to damp out as weak waves bounce around in the subsonic region long after the bulk flow has approached steady state. Figure 70 shows that the simulation matches the experimental data closely.

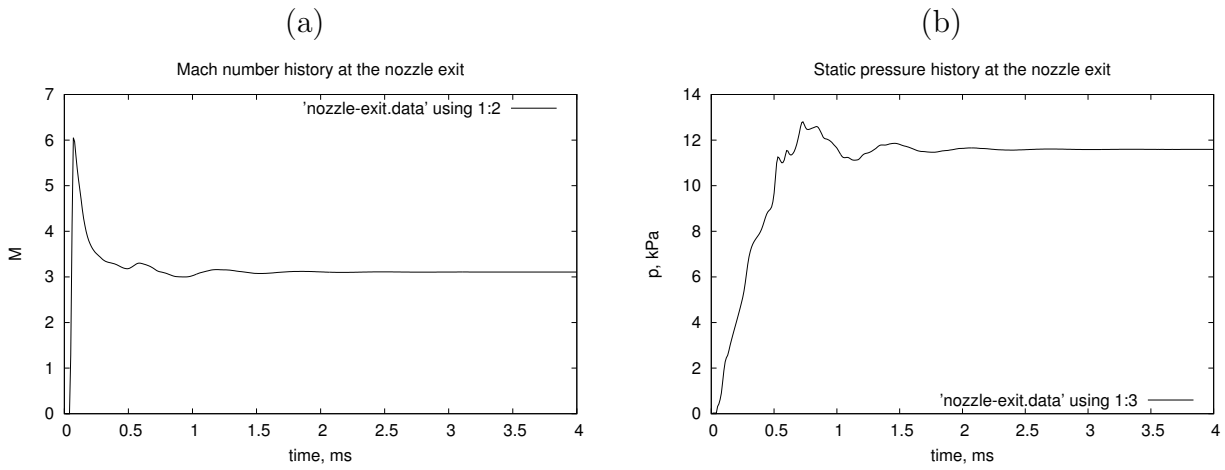


Figure 69: Development of the flow at a "history point" near the centre of the exit plane: (a) Mach number; (b) static pressure.

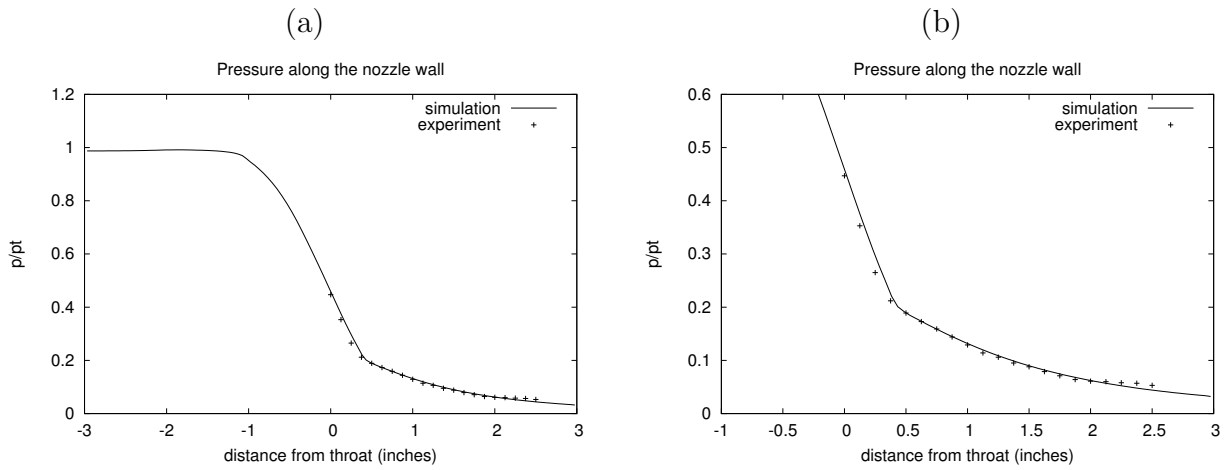


Figure 70: Normalised pressure distribution along the nozzle wall: (a) full length of flow domain; (b) just the supersonic part of the nozzle.

32.1 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'])
# The stagnation gas represents a reservoir condition.
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 = 3.0 * 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 = 50; 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 = 4.0e-3  # seconds
gdata.max_step = 50000
gdata.dt = 1.0e-7
gdata.dt_plot = 0.2e-3
gdata.dt_history = 10.0e-6

sketch.xaxis(-0.10, 0.08, 0.05, -0.01)
sketch.yaxis(  0.0, 0.05, 0.02, -0.015)
sketch.window(-0.10, 0.0, 0.10, 0.05, 0.05, 0.05, 0.25, 0.10)
```

32.2 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
mpirun -np 2 e3mpi.exe --job=back --run
e3post.py --job=back --tindx=all --vtk-xml --add-mach
```

```
# 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 [0.0:4.0]
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 [0.0:4.0]
set xlabel 'time, ms'
set ylabel 'p, kPa'

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

32.3 Notes

- The simulation reaches a final time of 4 ms in 5410 steps and, on an AMD Phenom II X4 840 system, this takes 128 seconds run time.

- 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 = 500.0e3
    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;
}
```

33 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.

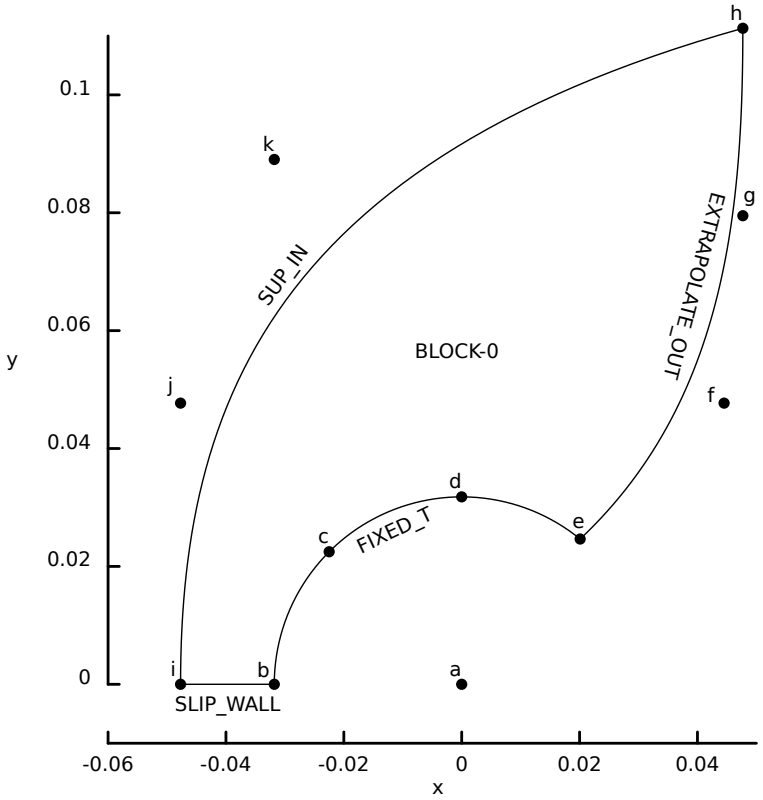


Figure 71: Schematic diagram of the geometry for a sphere wrapped by a single-block grid.

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. [21] with $M_\infty = 13.6$. According to Sawada & Dendou [21], 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. 73 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 [21]).

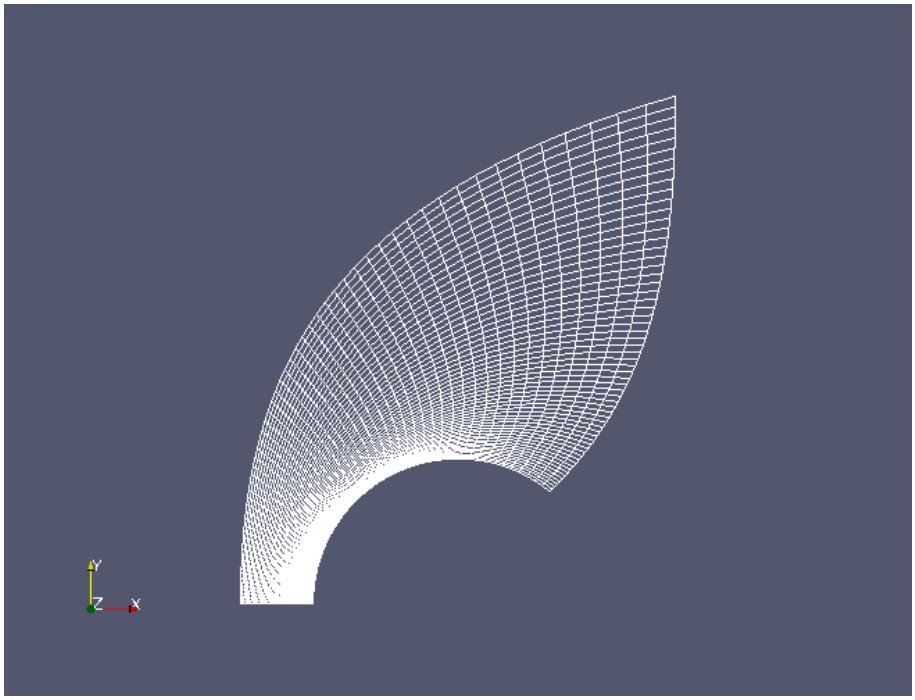


Figure 72: Mesh for flow over a sphere.

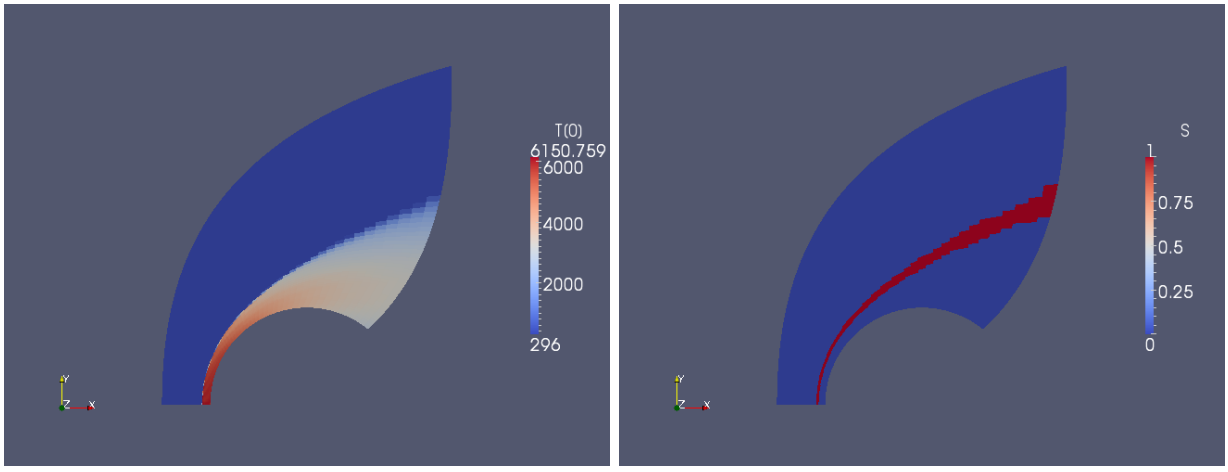


Figure 73: Temperature field and shock-detector (S) for equilibrium-air flow over a sphere.

33.1 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)
```

33.2 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.py --gas=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
```

33.3 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."
}
```

34 Classic shock tube problem

This example is a variation of the "Sod" shock tube problem that is a classic test case for transient flow simulation codes. It models a 1 metre long tube with hot, high-pressure helium in the left half (driver) and low-pressure air in the right half (driven) part of the tube. The conditions are such that high-temperature thermochemical effects are significant in the shock-compressed air that is driven to the right from $t = 0$.

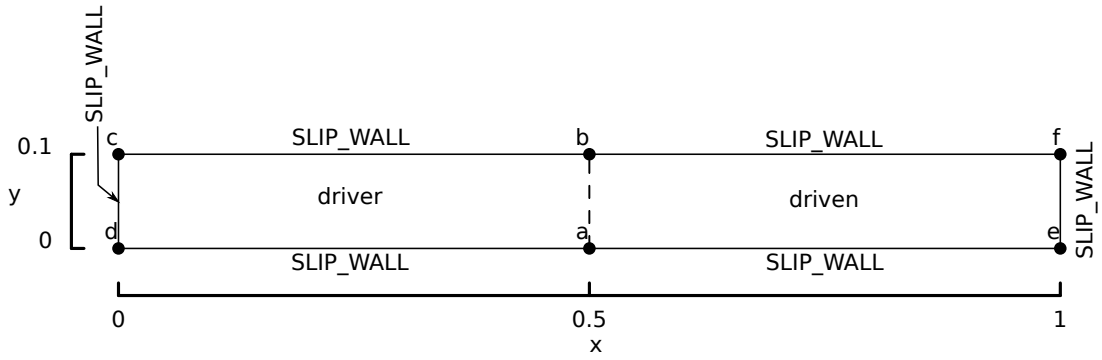


Figure 74: Flow region, as modelled, for the classic shock tube.

Run the case with the following commands:

```
$ cd ~/cfcfd3/examples/eilmer3/2D/classic-shock-tube/
$ ./prep_simulation.sh
$ ./run_simulation.sh
$ ./post_simulation.sh
```

The simulation is run for 100 μs and the data is extracted for plotting against the expected solution, as shown in Figure 75. This reference solution is obtained using finite-wave and shock analysis assuming chemical equilibrium in the driven air. The details of the calculation are found in Python script in Section 34.3.

Convergence of the estimated shock speed (determined by locating the pressure jump with the `locate_shock.py` postprocessing script) is shown in Figure 76. This custom postprocessing script also computes an average of the expended driver gas speed.

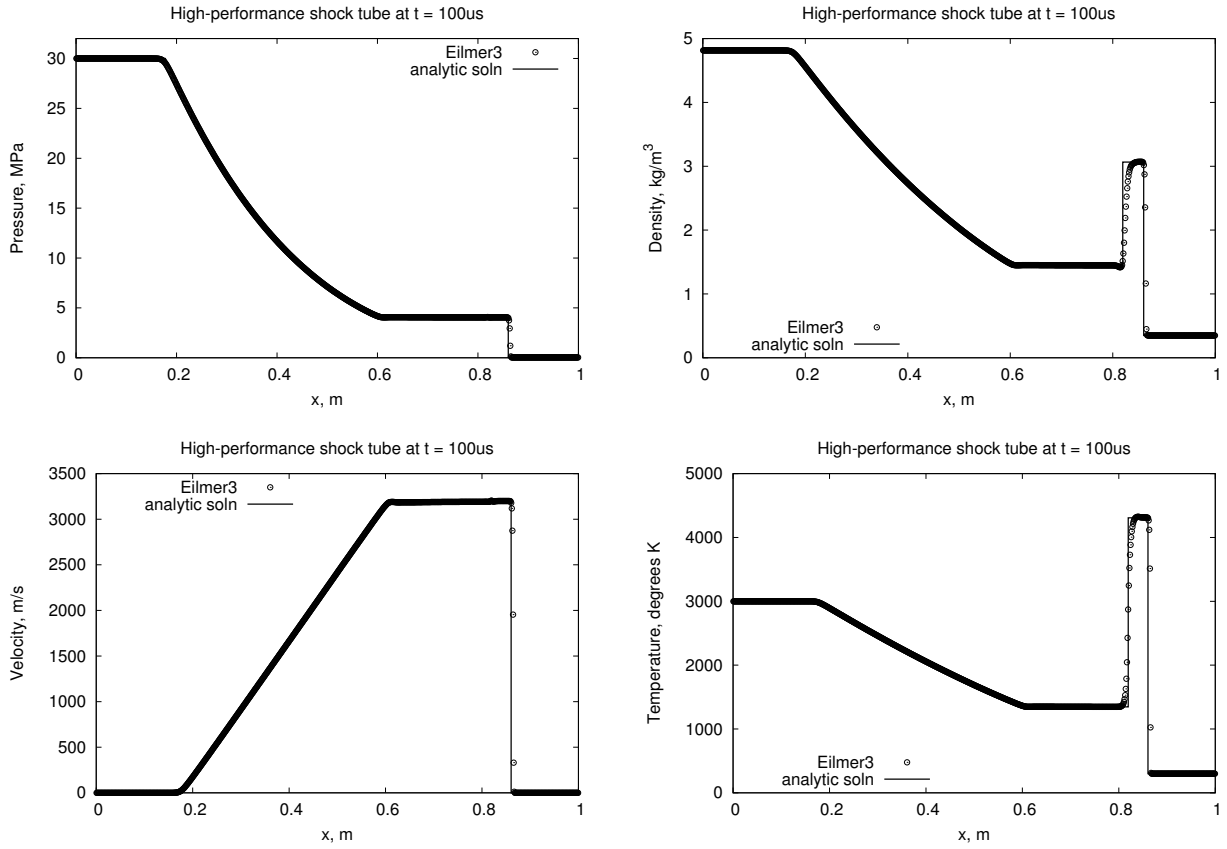


Figure 75: Flow properties along the duct for the Sod shock tube problem for nni=400.

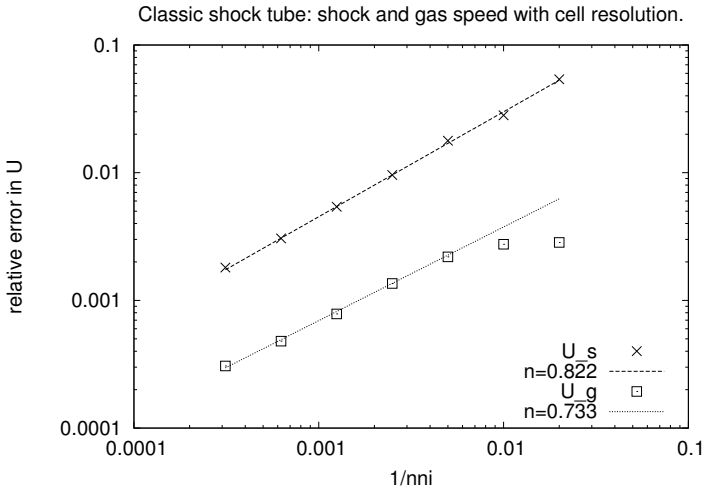


Figure 76: Convergence of estimated shock speed and gas speed.

34.1 Input script (.py)

In the problem setup, below, note the combination of the look-up gas model with the composite gas model. The helium driver gas is the only species in the composite gas and the look-up table gas models all of the chemically-reacting species within the air test gas. The look-up table gas ends up being prepended to the list of species for the simulation.

```
# High-performance shock tube with helium driving air
# in a constant-diameter tube.  The temperatures in the air
# are high enough to induce strong thermochemical effects.
#
# Adapted from examples/mbcns2/sod2/, examples/eilmer3/2D/sod/He-air/.
# Authors: PAJ and RJG
# Date: 24-Mar-2012

gdata.title = "High-performance shock tube with helium driving air."
# Combine a LUT air model with a composite gas of pure helium.
create_gas_file(model="ideal gas", species=['He',],
                fname="LUT-plus-He.lua", lut_file="cea-lut-air.lua.gz")
species_list = select_gas_model(fname="LUT-plus-He.lua")
print "species_list=", species_list

helium = FlowCondition(p=30.0e6, T=3000, massf={'He':1.0})
air = FlowCondition(p=30.0e3, T=300.0, massf={'LUT':1.0})

a = Node(0.5, 0.0, label="a"); b = Node(0.5, 0.1, label="b")
c = Node(0.0, 0.1, label="c"); d = Node(0.0, 0.0,  label="d")
e = Node(1.0, 0.0, label="e"); f = Node(1.0, 0.1, label="f")
south0 = Line(d, a); south1 = Line(a, e) # lower boundary along x-axis
north0 = Line(c, b); north1 = Line(b, f) # upper boundary
# left-end, diaphragm, right-end
west0 = Line(d, c); east0west1 = Line(a, b); east1 = Line(e, f)

# Define the blocks, boundary conditions and set the discretisation.
blk_0 = Block2D(make_patch(north0, east0west1, south0, west0),
                nni=400, nnj=2,
                fill_condition=helium, label="driver")
blk_1 = Block2D(make_patch(north1, east1, south1, east0west1),
                nni=400, nnj=2,
                fill_condition=air, label="driven")
identify_block_connections()

# Some simulation parameters
gdata.flux_calc = ADAPTIVE
gdata.max_time = 100.0e-6
gdata.max_step = 8000
gdata.dt = 1.0e-9

sketch.xaxis(0.0, 1.0, 0.5, -0.05)
sketch.yaxis(0.0, 0.1, 0.1, -0.05)
sketch.window(0.0, 0.0, 1.0, 0.1, 0.02, 0.02, 0.17, 0.035)
```

34.2 Shell scripts

```
#/bin/bash

if [ -f ./cea-lut-air.lua.gz ]
then
    echo "Found LUT file already in place."
else
    echo "Generate LUT file for air."
    build-cea-lut.py --gas=air
fi
e3prep.py --job=cst --do-svg
```

```
#/bin/bash

mpirun -np 2 e3mpi.exe --job=cst --run
```

```
#!/bin/bash

# Extract the profile along the shock tube.
# slice-list=block-range,i-range,j-range,k-range
#             all blocks - :
#             all i's - :
#             constant j - 0
#             constant k - 0 (not relevant in 2D anyway)
e3post.py --job=cst --slice-list=":,:,0,0" --output-file=profile.data

# Plot the data along the x-axis.
gnuplot <<EOF
set term postscript eps enhanced 20
set output "cst-p.eps"
set title "High-performance shock tube at t = 100us"
set xlabel "x, m"
set ylabel "Pressure, MPa"
set xrange [0:1]
set yrange [0:32.0]
plot "profile.data" using 1:(\$9/1.0e6) t 'Eilmer3' with points pt 6, \
     "exact.data" using 1:(\$3/1.0e6) t 'analytic soln' with lines lt 1 lw 3
EOF

gnuplot <<EOF
set term postscript eps enhanced 20
set output "cst-rho.eps"
set title "High-performance shock tube at t = 100us"
set xlabel "x, m"
set ylabel "Density, kg/m^3"
set xrange [0:1]
set yrange [0:5]
set key left bottom
plot "profile.data" using 1:5 t 'Eilmer3' with points pt 6, \
     "exact.data" using 1:2 t 'analytic soln' with lines lt 1 lw 3
EOF

gnuplot <<EOF
set term postscript eps enhanced 20
set output "cst-u.eps"
set title "High-performance shock tube at t = 100us"
set xlabel "x, m"
set ylabel "Velocity, m/s"
set xrange [0:1]
set yrange [0:3500]
set key left top
plot "profile.data" using 1:6 t 'Eilmer3' with points pt 6, \
```

```
     "exact.data" using 1:5 t 'analytic soln' with lines lt 1 lw 3
EOF

gnuplot <<EOF
set term postscript eps enhanced 20
set output "cst-T.eps"
set title "High-performance shock tube at t = 100us"
set xlabel "x, m"
set ylabel "Temperature, degrees K"
set xrange [0:1]
set yrange [0:5000]
set key left bottom
plot "profile.data" using 1:22 t 'Eilmer3' with points pt 6, \
     "exact.data" using 1:4 t 'analytic soln' with lines lt 1 lw 3
EOF
```

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

gnuplot <<EOF
set term postscript eps 20
set output "cst-errors.eps"
set title "Classic shock tube: shock and gas speed with cell resolution."
set xlabel "1/nni"
set ylabel "relative error in U"
set logscale xy
set key right bottom
e(x) = e0 * (nni*x)**n
plot "speeds.data" using (1.0/\$1):(\$2/3603.687-1.0) title "U_s" with points pt 2 ps 1.5, \
     nni = 400, e0 = 0.00960, n = 0.822, e(x) title "n=0.822" lw 2, \
     "speeds.data" using (1.0/\$1):(1.0-\$3/3194.170) title "U_g" with points pt 4 ps 1.5, \
     nni = 400, e0 = 0.00136, n = 0.733, e(x) title "n=0.733" lw 2
EOF
```

34.3 Solution using finite wave and shock analysis

The NASA CEA program can be used by a library module to provide convenient estimates of the thermochemical state of gas mixtures at equilibrium. The following script shows how to use that library to compute the flow for the classic shock tube where the temperatures in the driven air test gas are large enough to allow significant thermochemical effects. Beyond gas state estimation, the library provides analysis functions for simple flow processes such as shock and finite, isentropic waves.

```
#!/usr/bin/env python
"""
classic_shock_tube.py

Moderately high-performance shock tube with helium driving air.
Done as an example of using gas_flow functions but can be
compared the Eilmer3 sod shock tube example.

PJ, 22-Mar-2012
"""

import sys, os
sys.path.append(os.path.expandvars("$HOME/e3bin"))

from cfpylib.gasdyn.cea2_gas import Gas
from cfpylib.gasdyn.gas_flow import normal_shock, finite_wave_dp, normal_shock_p2p1
from cfpylib.nm.zero_solvers import secant

def main():
    print "Helium driver gas"
    state4 = Gas({'He':1.0})
    state4.set_pT(30.0e6, 3000.0)
    print "state4:"
    state4.write_state(sys.stdout)
    #
    print "Air driven gas"
    state1 = Gas({'Air':1.0})
    state1.set_pT(30.0e3, 300.0)
    print "state1:"
    state1.write_state(sys.stdout)
    #
    print "\nNow do the classic shock tube solution..."
    # For the unsteady expansion of the driver gas, regulation of the amount
    # of expansion is determined by the shock-processed test gas.
    # Across the contact surface between these gases, the pressure and velocity
    # have to match so we set up some trials of various pressures and check
    # that velocities match.
    def error_in_velocity(p3p4, state4=state4, state1=state1):
        "Compute the velocity mismatch for a given pressure ratio across the expansion."
        # Across the expansion, we get a test-gas velocity, V3g.
        p3 = p3p4*state4.p
        V3g, state3 = finite_wave_dp('cplus', 0.0, state4, p3)
        # Across the contact surface.
        p2 = p3
        print "current guess for p3 and p2=", p2
        V1s, V2, V2g, state2 = normal_shock_p2p1(state1, p2/state1.p)
        return (V3g - V2g)/V3g
    p3p4 = secant(error_in_velocity, 0.1, 0.11, tol=1.0e-3)
    print "From secant solve: p3/p4=", p3p4
    print "Expanded driver gas:"
    p3 = p3p4*state4.p
    V3g, state3 = finite_wave_dp('cplus', 0.0, state4, p3)
    print "V3g=", V3g
    print "state3:"
    state3.write_state(sys.stdout)
    print "Shock-processed test gas:"
    V1s, V2, V2g, state2 = normal_shock_p2p1(state1, p3/state1.p)
```

```
    print "V1s=", V1s, "V2g=", V2g
    print "state2:"
    state2.write_state(sys.stdout)
    assert abs(V2g - V3g)/V3g < 1.0e-3
    #
    # Make a record for plotting against the Eilmer3 simulation data.
    # We reconstruct the expected data along a tube 0.0 <= x <= 1.0
    # at t=100us, where the diaphragm is at x=0.5.
    x_centre = 0.5 # metres
    t = 100.0e-6 # seconds
    fp = open('exact.data', 'w')
    fp.write('# 1:x(m)  2:rho(kg/m**3) 3:p(Pa) 4:T(K) 5:V(m/s)\n')
    print 'Left end'
    x = 0.0
    fp.write('%g %g %g %g %g\n' % (x, state4.rho, state4.p, state4.T, 0.0))
    print 'Upstream head of the unsteady expansion.'
    x = x_centre - state4.a * t
    fp.write('%g %g %g %g %g\n' % (x, state4.rho, state4.p, state4.T, 0.0))
    print 'The unsteady expansion in n steps.'
    n = 100
    dp = (state3.p - state4.p) / n
    state = state4.clone()
    V = 0.0
    p = state4.p
    for i in range(n):
        rhoa = state.rho * state.a
        dV = -dp / rhoa
        V += dV
        p += dp
        state.set_ps(p, state4.s)
        x = x_centre + t * (V - state.a)
        fp.write('%g %g %g %g %g\n' % (x, state.rho, state.p, state.T, V))
    print 'Downstream tail of expansion.'
    x = x_centre + t * (V3g - state3.a)
    fp.write('%g %g %g %g %g\n' % (x, state3.rho, state3.p, state3.T, V3g))
    print 'Contact surface.'
    x = x_centre + t * V3g
    fp.write('%g %g %g %g %g\n' % (x, state3.rho, state3.p, state3.T, V3g))
    x = x_centre + t * V2g  # should not have moved
    fp.write('%g %g %g %g %g\n' % (x, state2.rho, state2.p, state2.T, V2g))
    print 'Shock front'
    x = x_centre + t * V1s  # should not have moved
    fp.write('%g %g %g %g %g\n' % (x, state2.rho, state2.p, state2.T, V2g))
    fp.write('%g %g %g %g %g\n' % (x, state1.rho, state1.p, state1.T, 0.0))
    print 'Right end'
    x = 1.0
    fp.write('%g %g %g %g %g\n' % (x, state1.rho, state1.p, state1.T, 0.0))
    fp.close()
    return

if __name__ == '__main__':
    main()
    print "Done."
```

34.4 Extracting shock location and getting average gas speed

The following script is and example of how to pick up a full block of data with the post-processing library functions and then look within that flow data for particular features.

```
#! /usr/bin/env python
"""
locate_shock.py -- Locate the shock by its pressure jump.

PJ, 12-Apr-2012
"""
print "Begin..."
import sys, os, gzip
sys.path.append(os.path.expandvars("$HOME/e3bin"))
from e3_flow import StructuredGridFlow

# Block 1 contains the shock and the fully-expanded driver gas.
fileName = 'flow/t9999/cst.flow.b0001.t9999.gz'
fp = gzip.open(fileName, "r")
blockData = StructuredGridFlow()
blockData.read(fp)
fp.close()

# We expect the shock to have progressed some way along the i-index.
# Start the search from the right and move left.
k = 0; j = 0; i = blockData.ni -1
p_trigger = 2.0e6  # Pa
x_old = blockData.data['pos.x'][i,j,k]
p_old = blockData.data['p'][i,j,k]
while i >= 0:
    i -= 1
    x = blockData.data['pos.x'][i,j,k]
    p = blockData.data['p'][i,j,k]
    if p > p_trigger: break
    x_old, p_old = x, p
frac = (p_trigger - p_old) / (p - p_old)
x_loc = x_old * (1.0 - frac) + x * frac
t_final = 100.0e-6 # seconds
print "shock at x=", x_loc, "m, speed=", (x_loc - 0.5)/t_final, "m/s"

# Also compute average gas speed of the expanded driver gas
# over a representative region.
u_sum = 0; n = 0;
for i in range(blockData.ni):
    x = blockData.data['pos.x'][i,j,k]
    u = blockData.data['vel.x'][i,j,k]
    if x >= 0.7 and x <= 0.8:
        u_sum += u; n += 1
u_sum /= n
print "average u_g=", u_sum, "m/s"
print "Done."
```

34.5 Notes

- The simulation with `nni = 400` takes about 13 seconds on a recent (2011) machine with an AMD Phenom 9650 quad-core processor.

35 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 [22] 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.

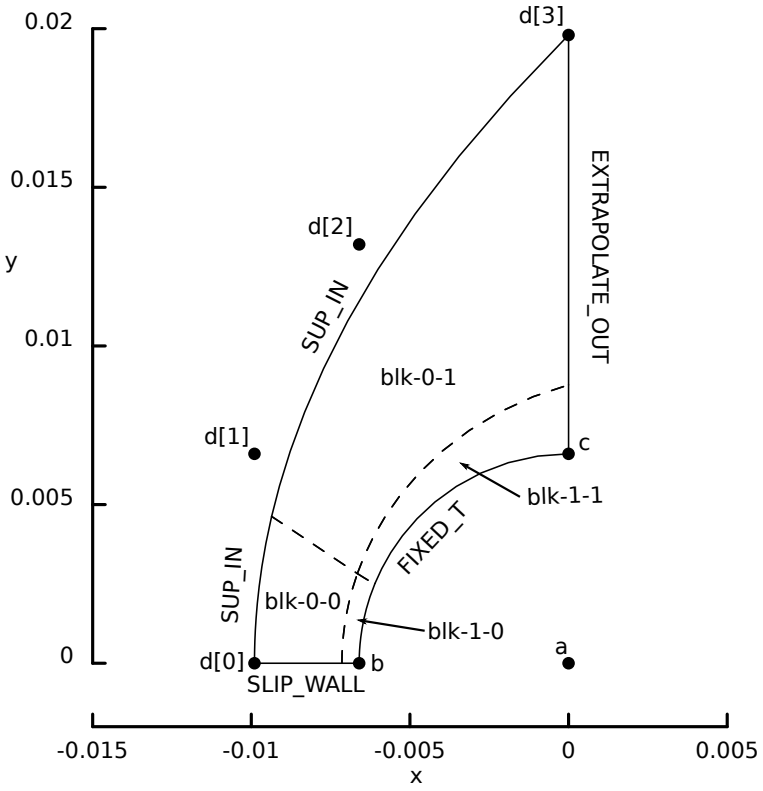


Figure 77: Schematic diagram of the geometry for a sphere wrapped by a SuperBlock2D grid.

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 [22], 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 78 through 82 which show the temperature field at the end of each stage and the mesh used for that stage.

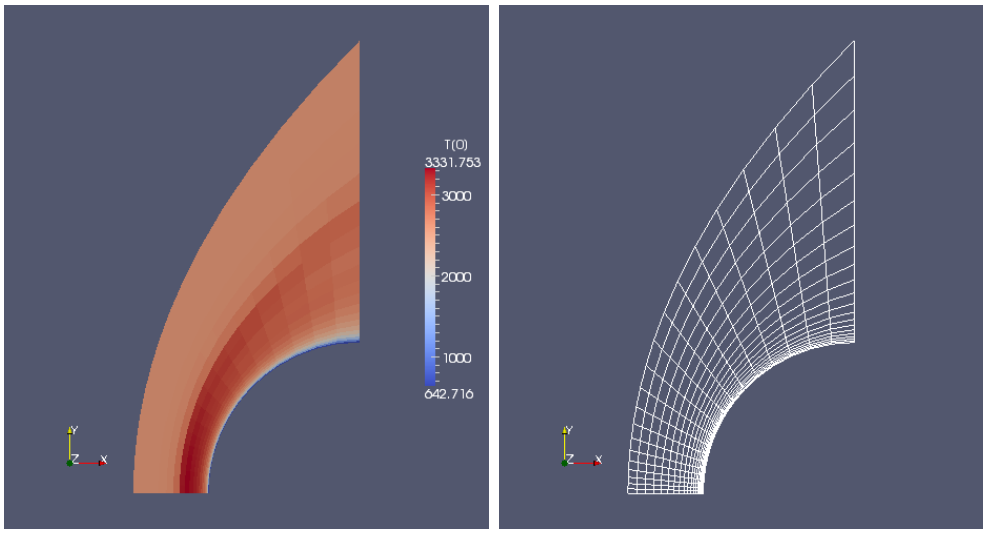


Figure 78: Temperature field and mesh for stage 0. The control points for the Bezier curve have been set so as to accommodate a shock in ideal (nonreacting) air and the clustering is fairly strong so that the boundary layer on the sphere surface may be resolved. The wall-clock time required to run this 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 83 shows the distribution of heat transfer around the nose compared with the experimental data reported Kemp, Rose and Detra [23]. In the simulation data, there are small disturbances at the corners of blocks (at approximately 20 degrees and then again approaching 90 degrees) but they are quite small.

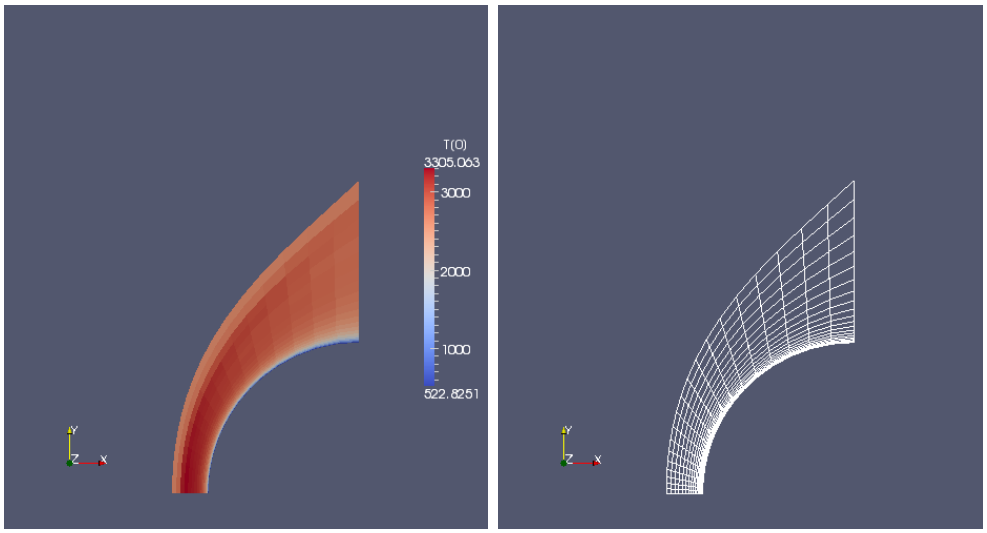


Figure 79: Temperature field and mesh for stage 1. The Bezier points have been adapted to the shock from stage 0 but the number of cells in each direction remains at 20×20, as for stage 0. With the finer cells, the size of the time step decreased and this stage required 55 seconds of wall-clock time to extend the simulation a further 10 body lengths in 23125 time steps. $\dot{q}_s = 2.260\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 2.94$

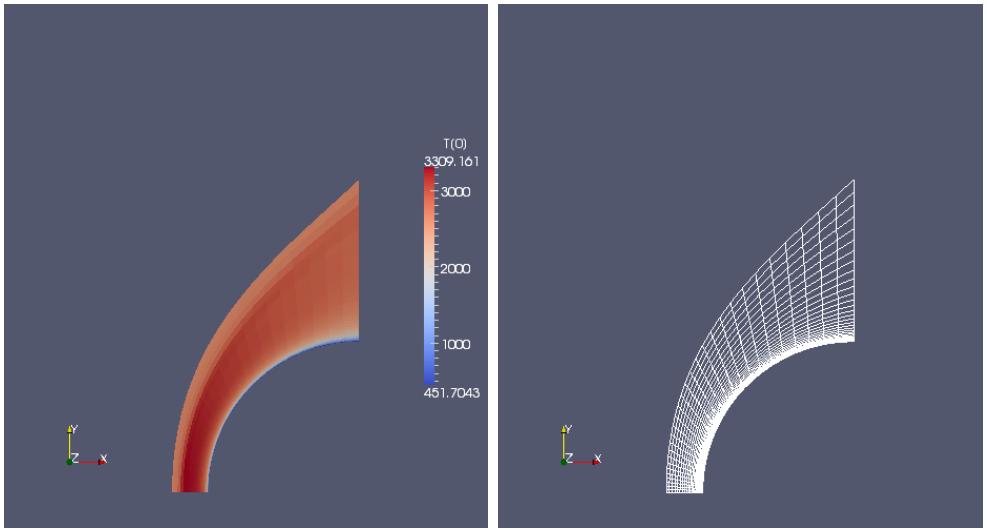


Figure 80: Temperature field and mesh for stage 2. The Bezier points have not been adapted further for this stage but the number of cells has been increased to 30×30. The size of the time step decreased further and this stage required 130 seconds of wall-clock time to extend the simulation only 5 body lengths (13.5 μs) in 24293 time steps. $\dot{q}_s = 2.257\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 2.39$

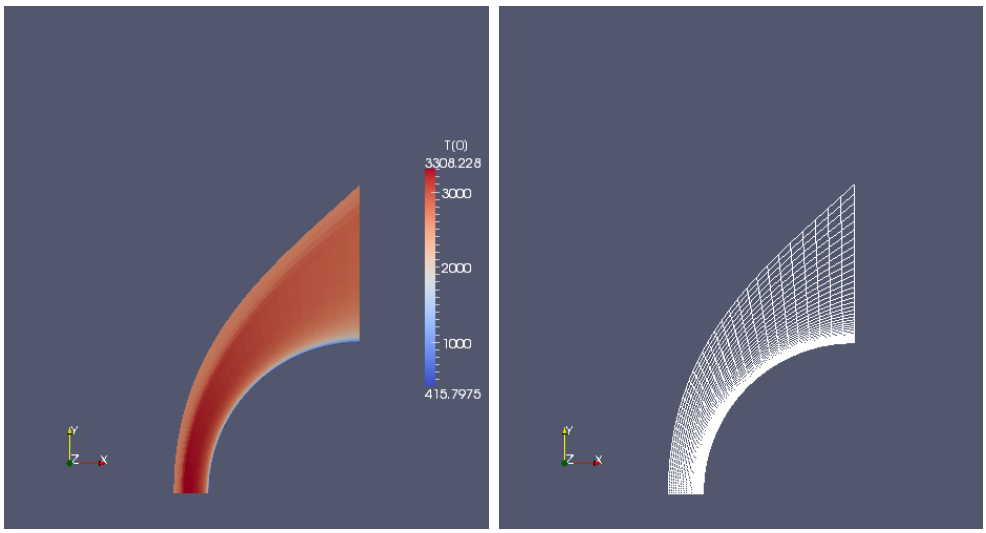


Figure 81: Temperature field and mesh for stage 3. The Bezier points have been adapted to the shock from stage 2 and the cells have been increased to 40×40. The size of the time step is now 0.319 ns and this stage required 469 seconds of wall-clock time to extend the simulation only 5 body lengths (13.5 μs) in 42660 time steps. $\dot{q}_s = 2.260\,\mathrm{MW/m^2}$ and $\mathrm{Re}_{wall} = 1.99$

.

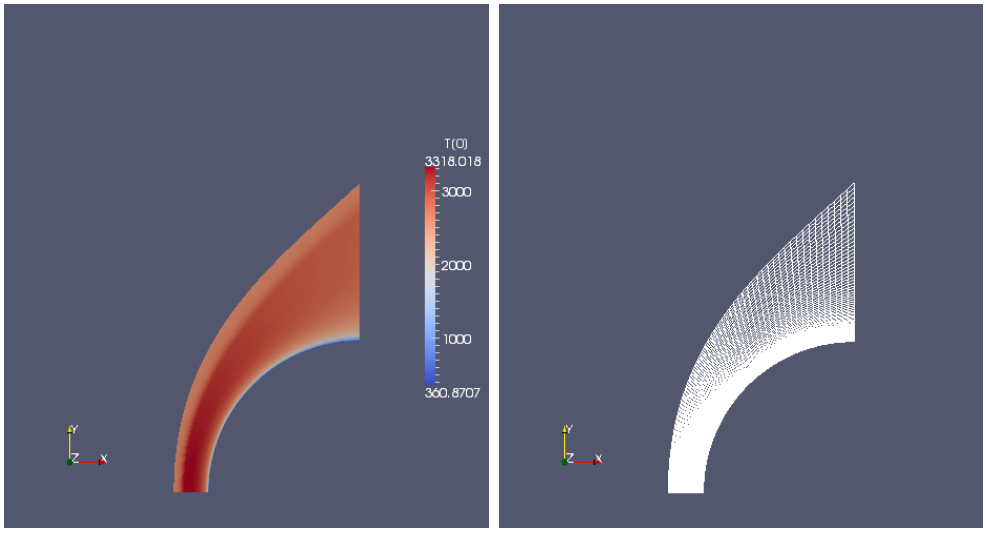


Figure 82: Temperature field and mesh for stage 4. The Bezier points have not been further adapted but the number of cells has been increased to 80×80 to test the sensitivity of the heat transfer estimate. The size of the time step is now 0.086 ns and this stage required 8950 seconds of wall-clock time 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 $\mathrm{Re}_{wall} = 1.25$

.

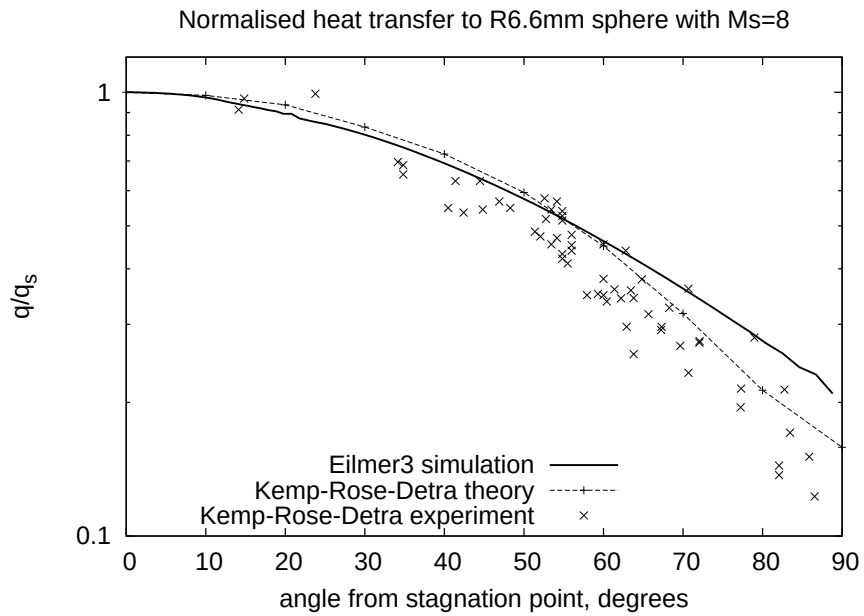


Figure 83: Temperature around the sphere for stage 4. The experimental data is from Kemp, Rose and Detra [23].

.

35.1 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-ions.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, 5)
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)
```

35.2 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 -q --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/t0005/%s.flow.b%04d.t0005.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."
```

35.3 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=5 --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
```

35.4 Notes

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

36 Dissociating nitrogen 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.

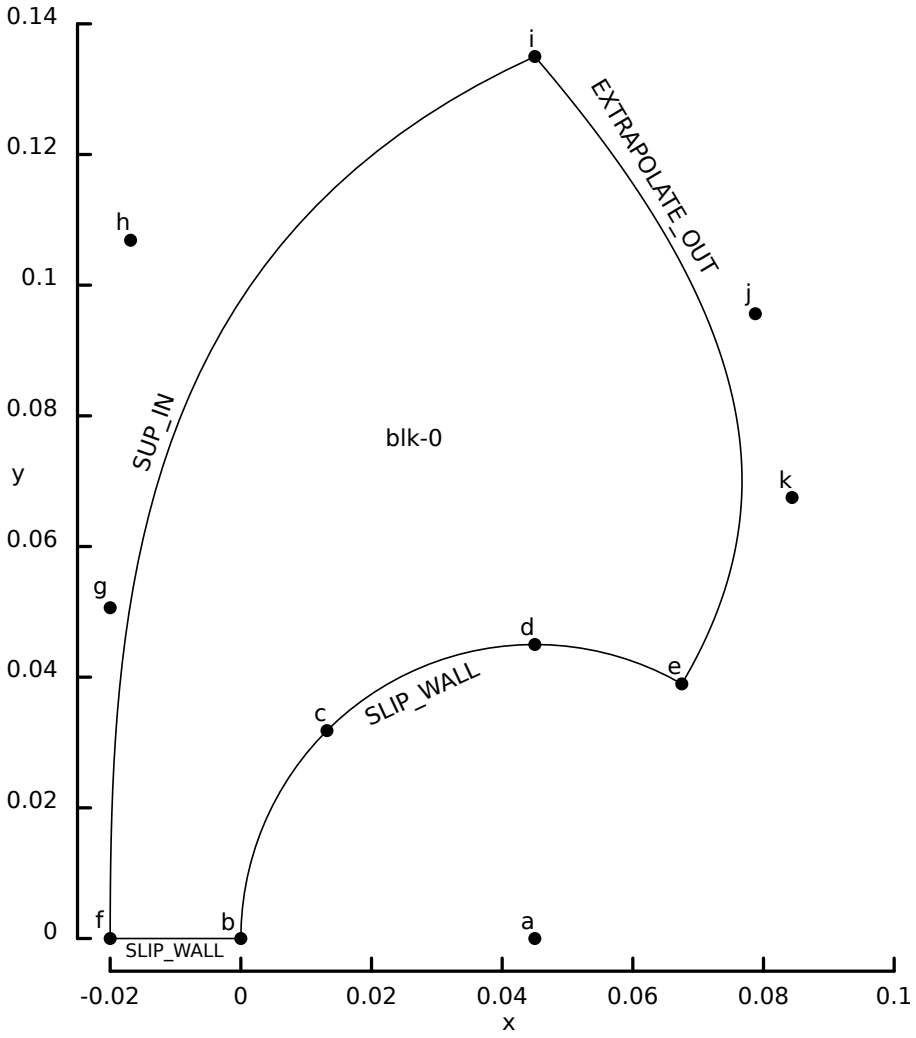


Figure 84: Schematic diagram of the geometry for the bluff body.

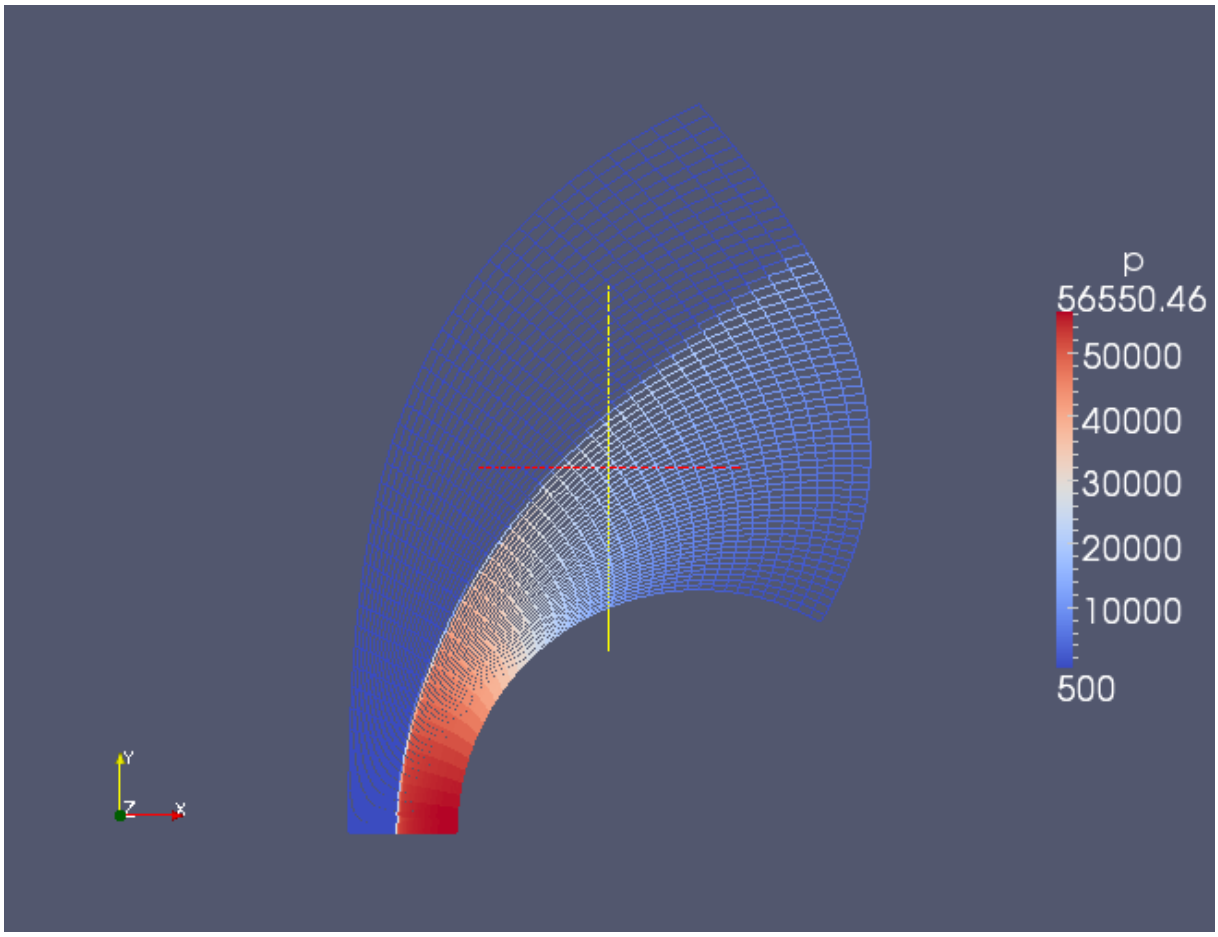


Figure 85: Mesh, coloured by pressure, for the n90 bluff body exercise.

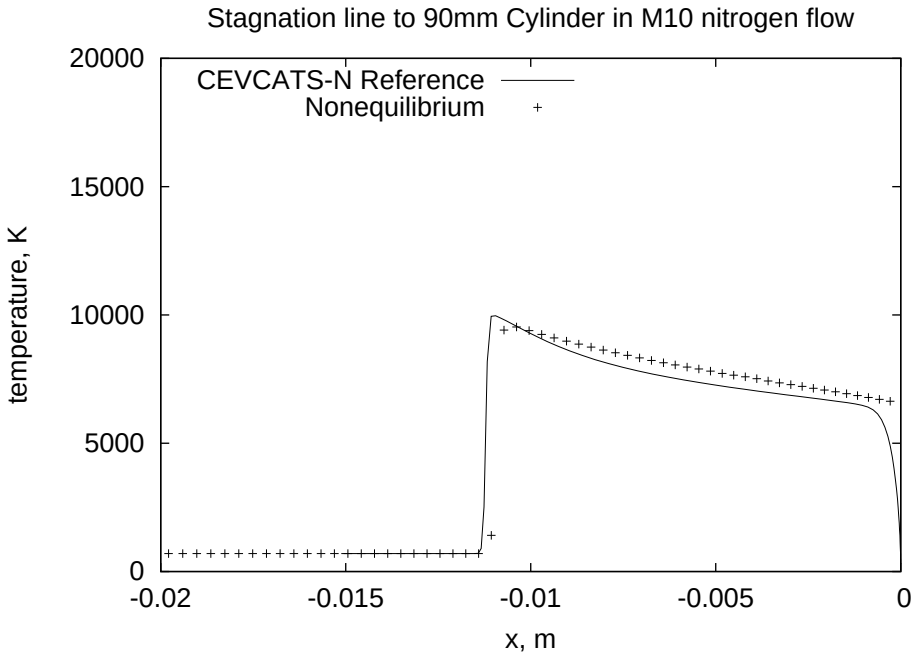


Figure 86: Temperature data along the stagnation streamline.

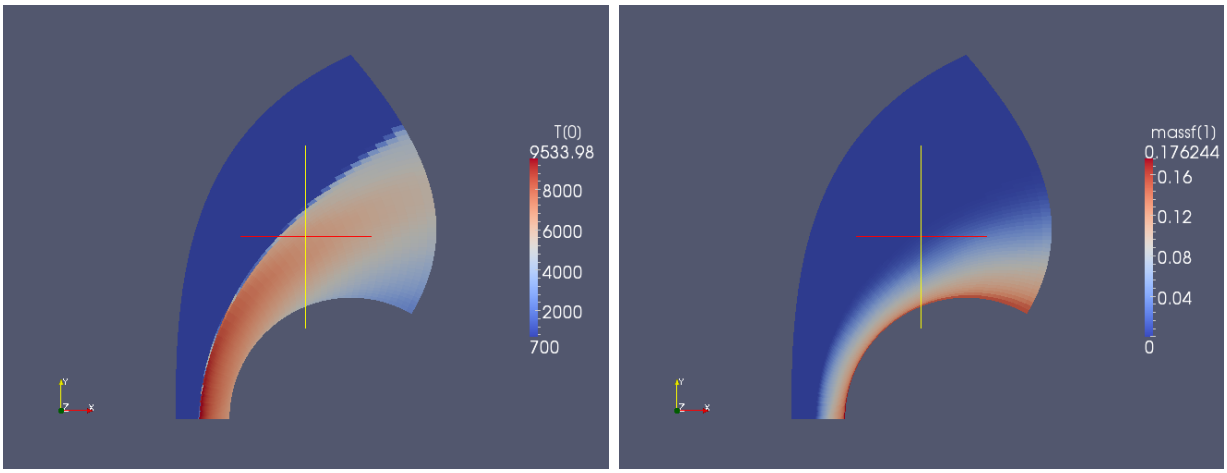


Figure 87: Temperature and mass fraction of nitrogen atoms for the n90 bluff body exercise.

36.1 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)
```

36.2 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}
}
```

36.3 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"
```

36.4 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.

37 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 [24]. 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 88 shows the periodic structure caused by the unsteady combustion of the gas mixture over the projectile.

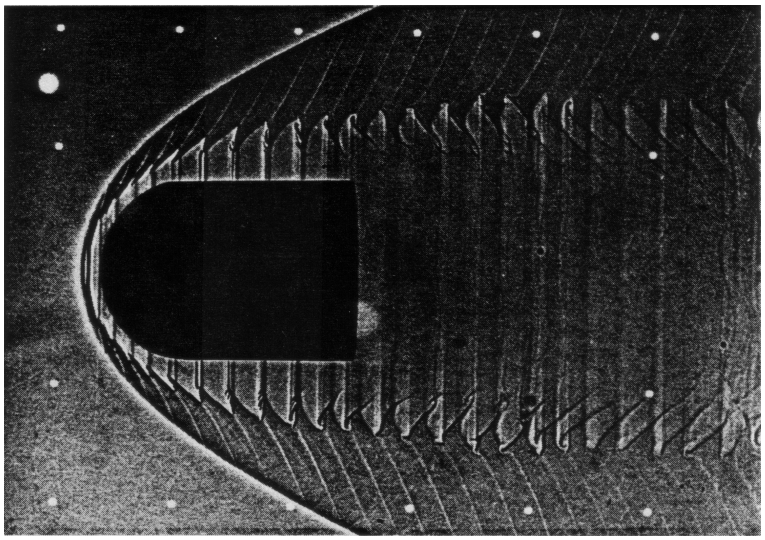


Figure 88: Shadowgraph of the unsteady flow of reacting hydrogen and air over a ballistic-range projectile at a Mach number of 4.79. This particular image has been scanned from Greg Wilson's PhD thesis [25].

The free-stream condition is $p_\infty = 320\,\text{mm Hg}$, $T_\infty = 292\,\text{K}$, $u_\infty = 1.931\,\text{km/s}$, 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 37.2 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 90. The solution is all very steady, well behaved, and quite wrong. The reaction front has merged into the shock and

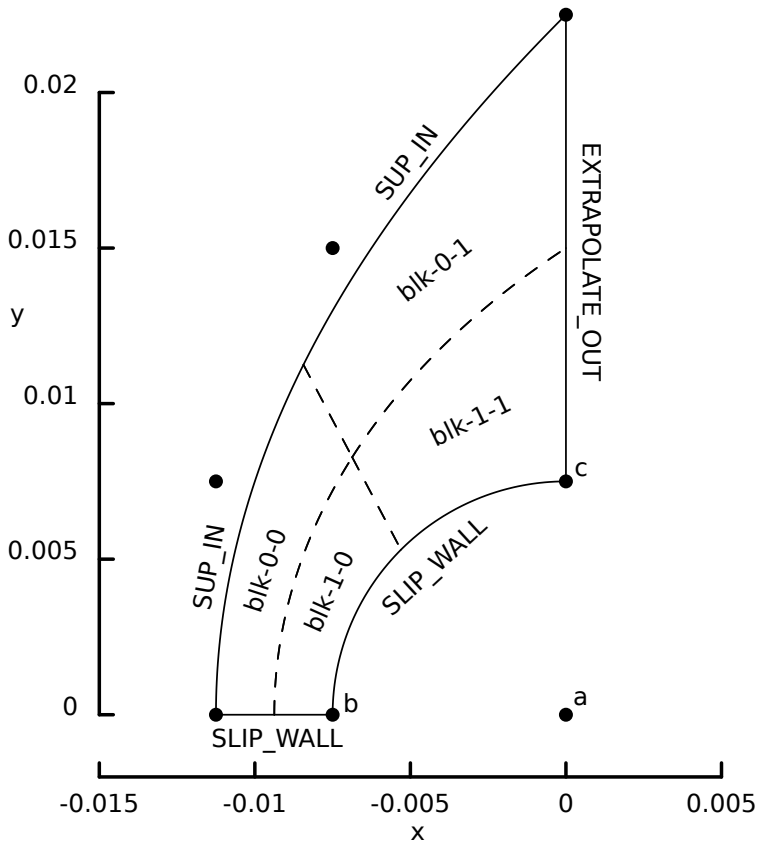


Figure 89: Schematic diagram of the geometry for a sphere wrapped by a SuperBlock2D grid. Although 2×2 blocks are shown here, we are typically impatient and use 4×6 blocks as shown in the input scripts.

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 90. Here, the reaction front is clearly separated from the incident 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.

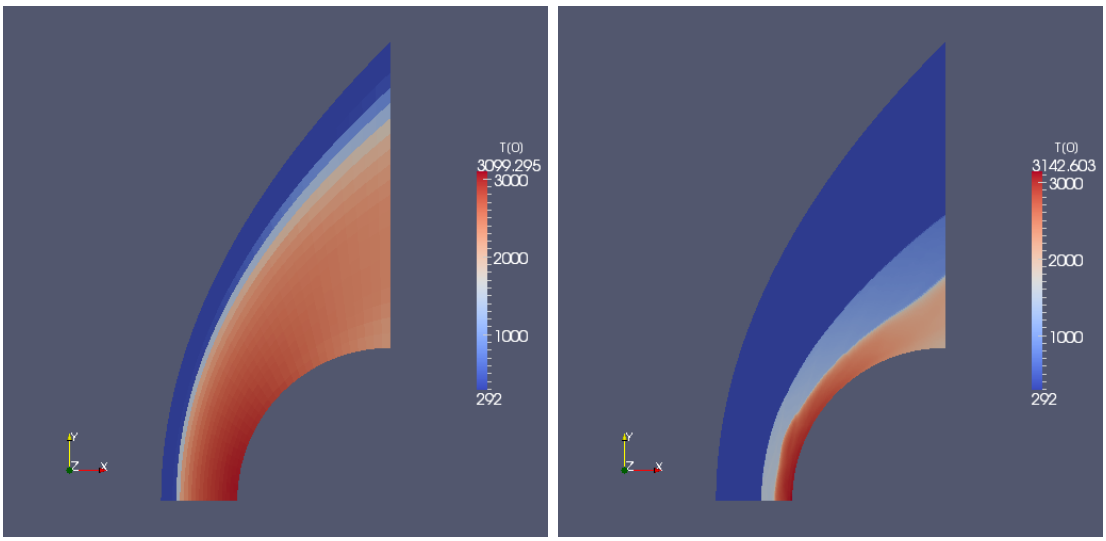


Figure 90: Temperature field at t=190 μs for 20×30 cells (left) and 200×300 cells (right).

Figure 91 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.

37.1 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
p_init = p_inf / 5
```

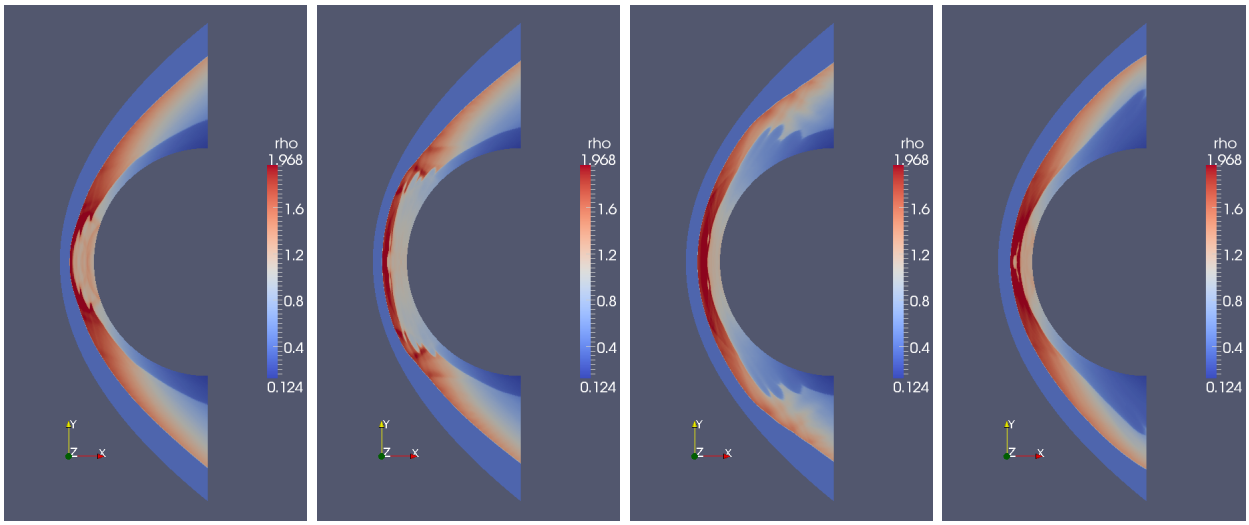


Figure 91: Density field at t=11, 14, 17, 20 μs in the `lehr1` second-stage simulation.

```
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)
```

37.2 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
```

37.3 Notes

- None

38 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 ~/cfcfd3/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 92 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 input script in Section 38.1

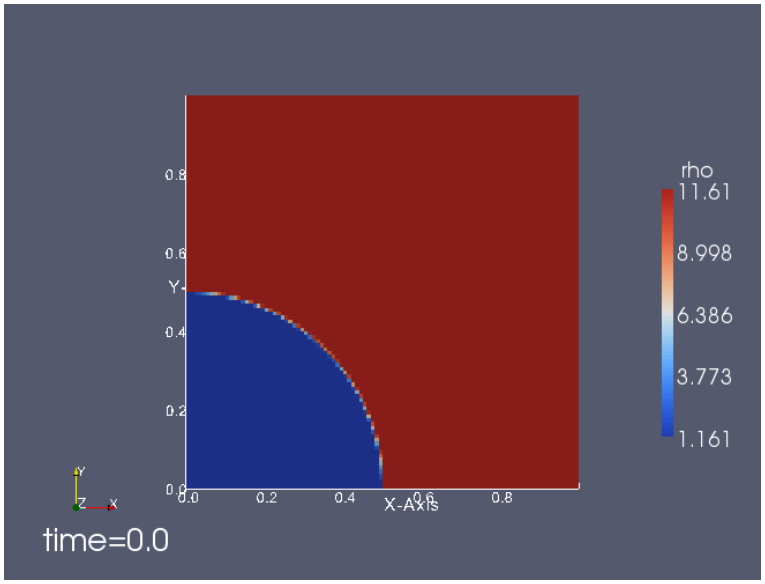


Figure 92: Initial density field for the implosion problem.

Figure 93 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 94 shows the density profiles for a number of radial slices.

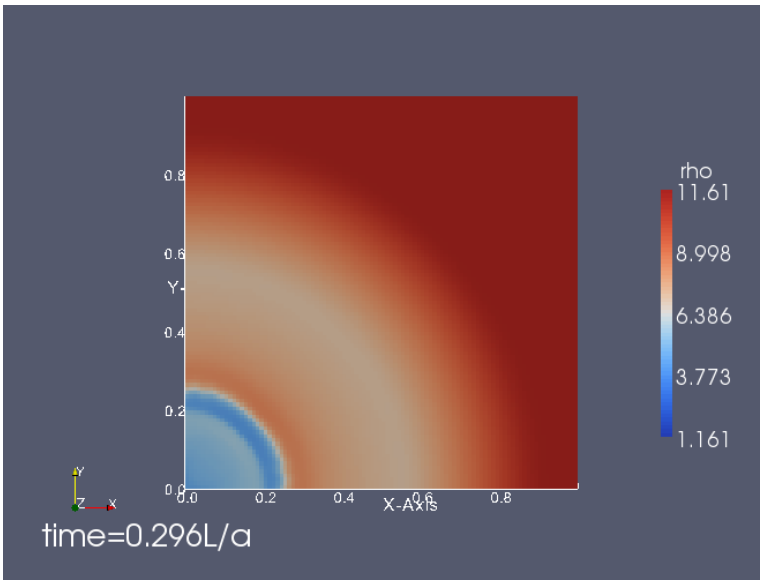


Figure 93: Final density field for the implosion problem.

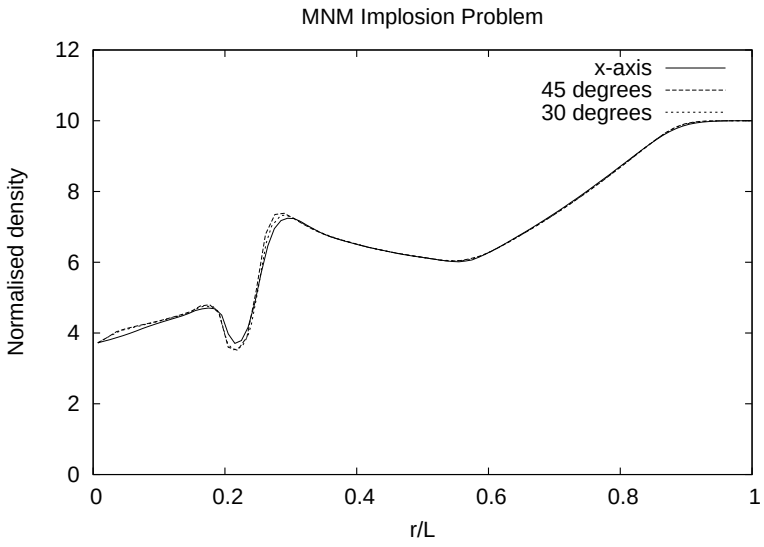


Figure 94: Density profiles at the final time for the implosion problem.

38.1 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"
```

38.2 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
```

38.3 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`.

39 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. 95.

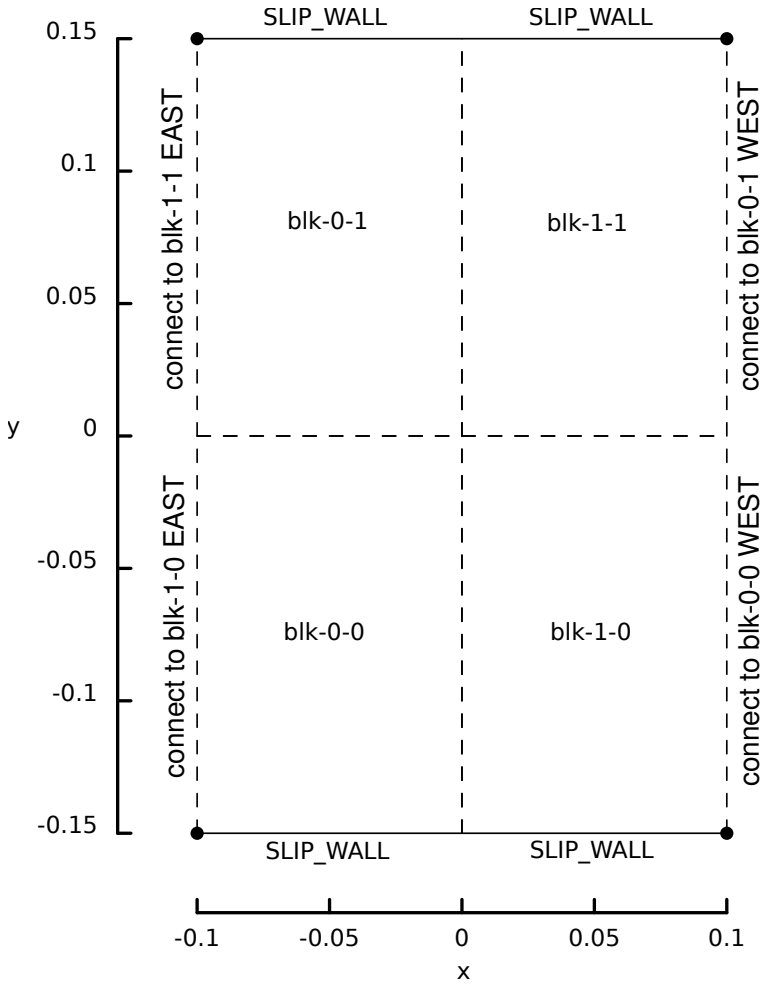


Figure 95: Computational domain for the periodic shear layer.

Figure 96 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. 39.1).

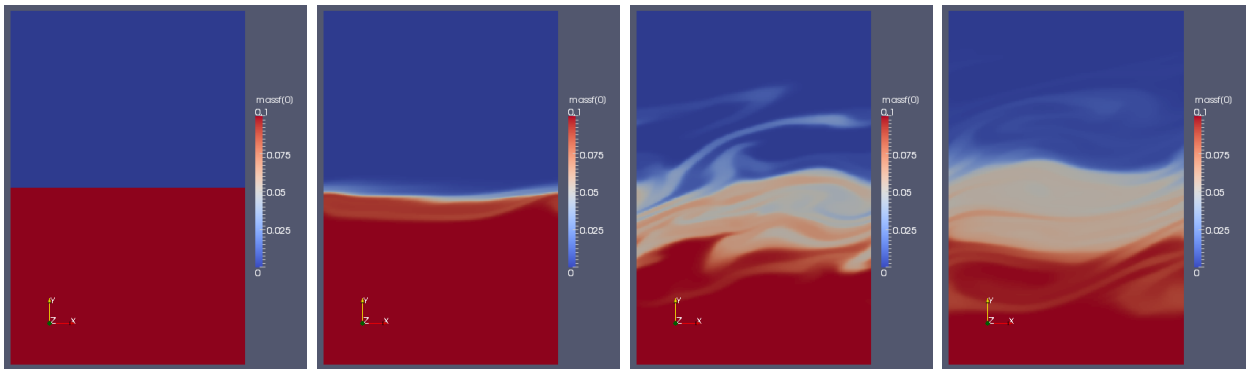


Figure 96: Mass fraction for helium across the periodic shear layer at times 0, 5.1 ms, 10.2 ms and 15 ms.

Figure 97 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.

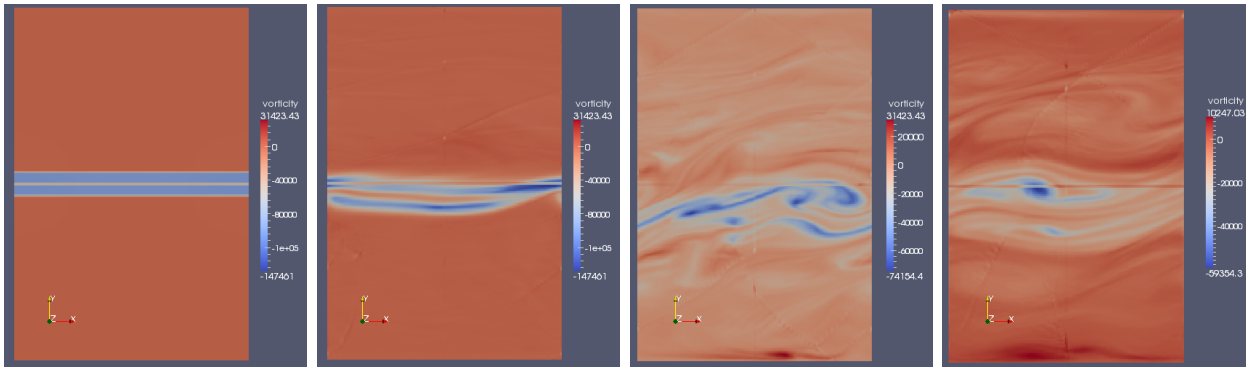


Figure 97: Vorticity field for the periodic shear layer at times 0, 5.1 ms, 10.2 ms and 15 ms.

39.1 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,
                      check_corner_locations=False)

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)
```

39.2 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
```

39.3 Notes

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

40 Mach 1.5 flow over a 20-degree 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 12. Use the following commands:

```
$ cd ~/cfcfd3/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 40.3.

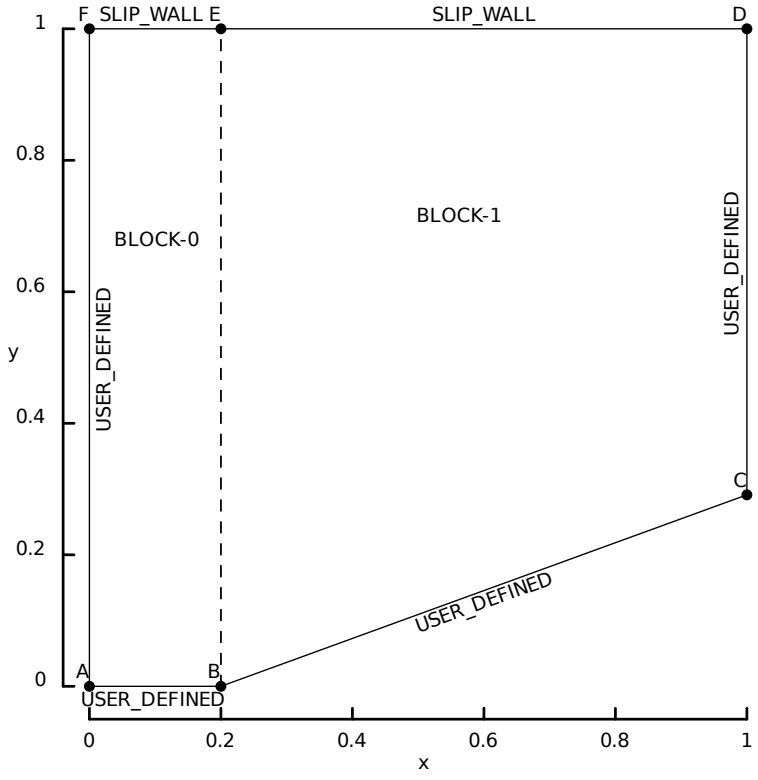


Figure 98: Schematic diagram of the geometry for a cone with 20 degree half-angle and user-defined boundaries.

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 [10] and are set to give a Mach number of 1.5. From Chart 5 in Ref. [11], the expected steady-state

shock wave angle is 49^o 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 101 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.

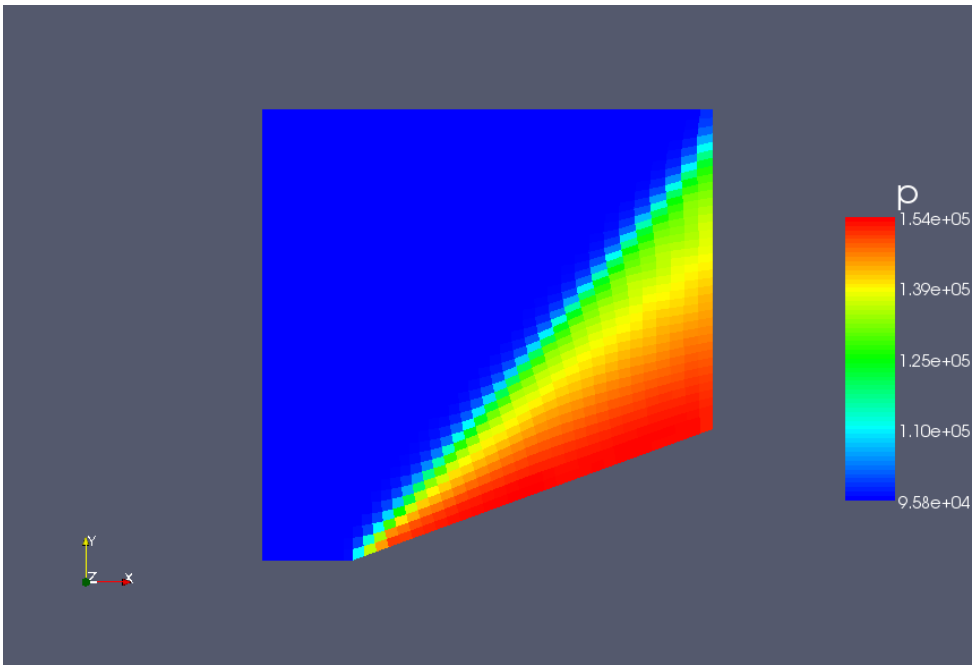


Figure 99: Pressure data for flow over a cone with 20 degree half-angle. The shock profile is not yet straight and the pressure field near the cone surface is not conically symmetric, although it would become more so if we continued the simulation.

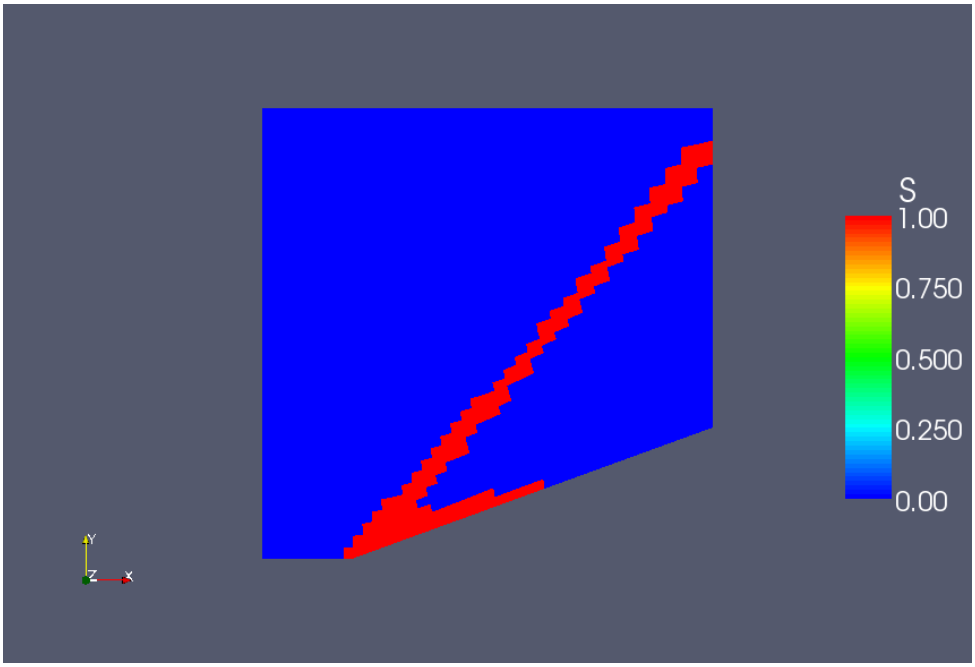


Figure 100: Shock-sensor data for flow over a cone with 20 degree half-angle. For the `adaptive` flux calculator, this sensor indicates the regions of the flow where the more dissipative scheme should be used.

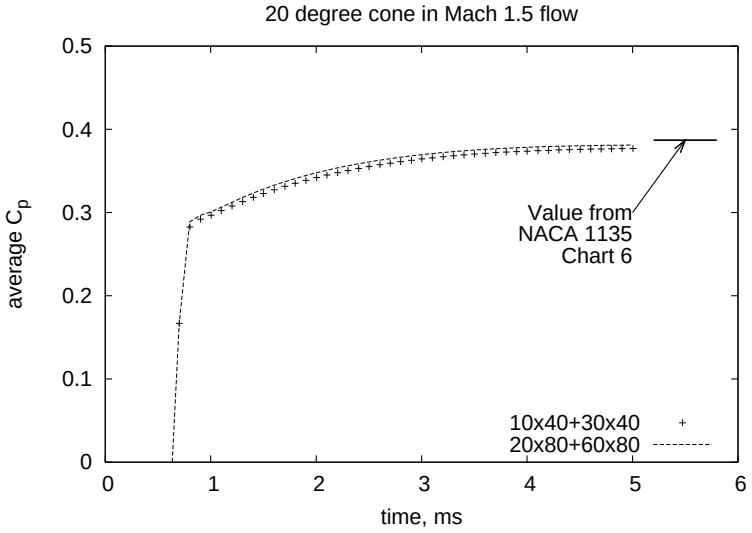


Figure 101: Evolution of the axial (drag) force for flow over a cone with 20 degree half-angle for two mesh resolutions.

40.1 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
## 24-Apr-2012 -- update source terms and supersonic-in Lua files

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, "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.set_BC(WEST, USER_DEFINED, filename="udf-supersonic-in.lua", sets_conv_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 # 0=standard case; 1=energy-addition test
gdata.compression_tolerance = -0.05 # the old default value
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)
```

40.2 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.
   -- print("Hello from function ghost_cell.")
   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.
   -- print("Hello from function interface.")
   face = {}
   face.u = 1000.0
   face.v = 0.0
   face.w = 0.0
   face.T = {[0]=1103.0,}
   face.massf = {[0]=1.0,}
   return face
end


function convective_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.
   -- print("Hello from function flux.")
   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("nblks=", nblks)

function at_timestep_start(args)
   if (args.step ~= 0) then
      -- do nothing, just leave
      return
   end
   -- For the 0th step only
   mass = 0.0
   for ib=0,(nblks-1) do
      imin = blks[ib].imin; imax = blks[ib].imax
      jmin = blks[ib].jmin; jmax = blks[ib].jmax
      blk_id = blks[ib].id
      for j=jmin,jmax do
         for i=imin,imax do
            cell = sample_flow(blk_id, i, j, k)
            -- We are only given p and T
            -- so need to compute density
            -- using gas model
            Q = create_empty_gas_table()
            Q.p = cell.p
            Q.T = cell.T
            for isp=0,(nsp-1) do Q.massf[isp] = cell.massf[isp] end
            eval_thermo_state_pT(Q)
            rho = Q.rho
            -- Now we can compute mass in cell using volume of cell
            mass = mass + rho*cell.vol
         end
      end
   end
   print("Mass (kg) of gas in domain: ", mass)
   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
   -- cell_data table contains most else
   src = {}
   src.mass = 0.0
   src.momemtum_x = 0.0
   src.momentum_y = 0.0
   src.momentum_z = 0.0
   if cell_data.x > 0.2 and cell_data.x < 0.4 and
      cell_data.y > 0.2 and cell_data.y < 0.3 then
      src.total_energy = 100.0e+6  -- J/m**3
      src.energies = {[0]=100.0e6}
   else
      src.total_energy = 0.0
      src.energies = {[0]=0.0}
   end
   src.species = {[0]=0.0,}
   return src
end
```

40.3 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."
```

40.4 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.

41 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 [26]. 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. 102) 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 $p_i = 100\,\text{kPa}$.

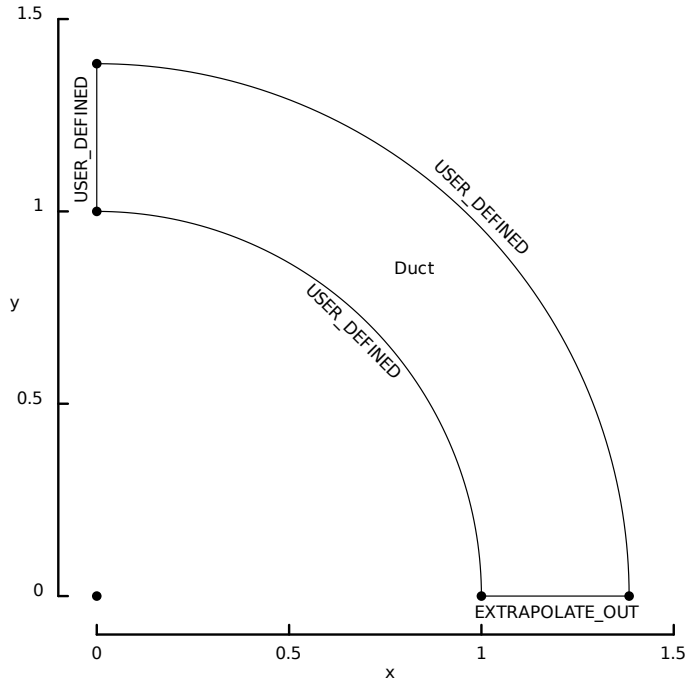


Figure 102: Schematic diagram of the first quadrant domain for the compressible-flow vortex.

Figure 103 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.

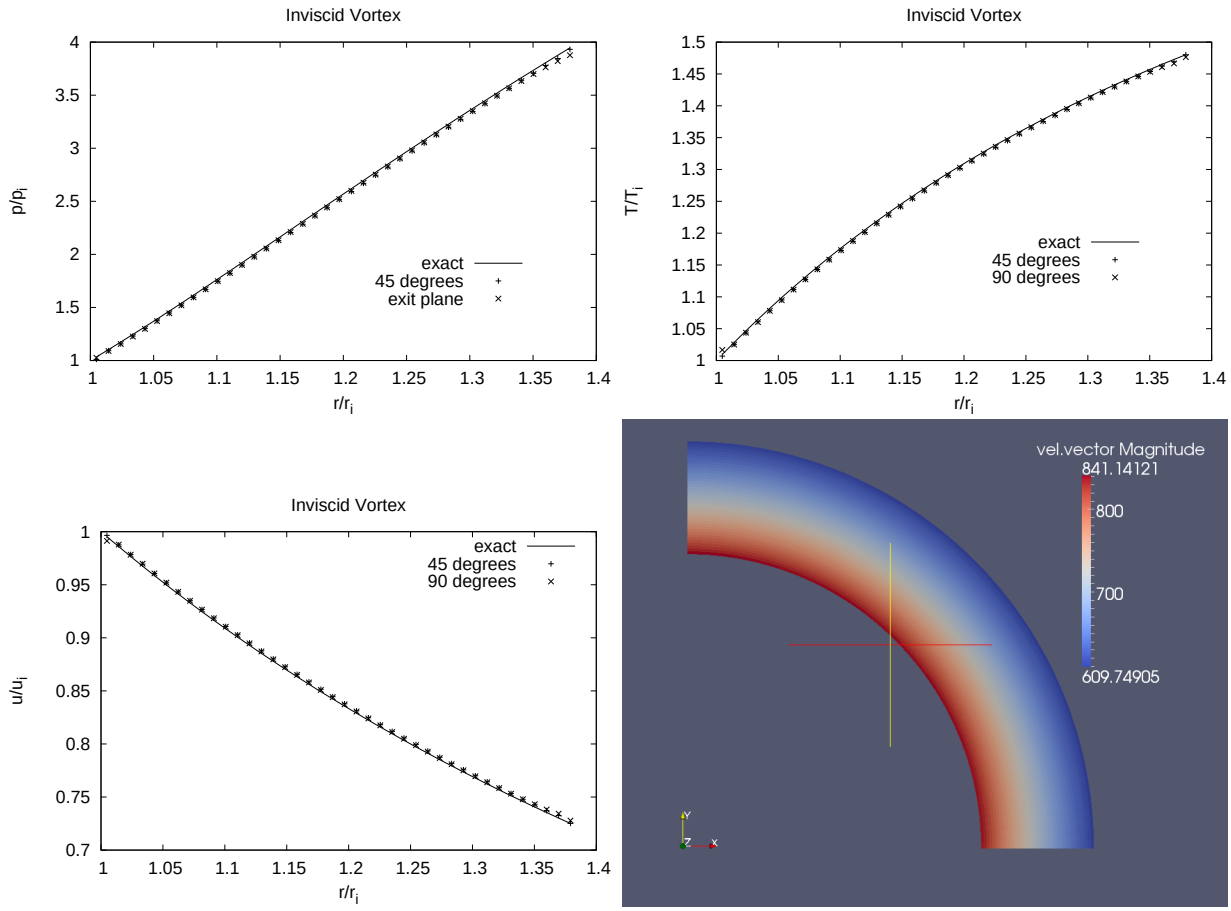


Figure 103: Radial distributions of normalized pressure, temperature and velocity. Also, the bottom right image shows the flow speed over the simulated domain

41.1 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)
```

41.2 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  -- indexed from 0 to nmodes-1
   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 = {[0]=T,}
   ghost2.u = math.sin(theta) * speed
   ghost2.v = -math.cos(theta) * speed
   ghost2.w = 0.0
   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)

   face = {}
   face.u = math.sin(theta) * speed
   face.v = -math.cos(theta) * speed
   face.w = 0.0
   face.T = {[0]=T,}
   face.massf = {[0]=1.0,}
   return face
end
```

41.3 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
```

41.4 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 bottom right
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 bottom right
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
}
```

42 Method of manufactured solutions – Euler flow

The method of manufactured solutions as a code verification exercise for inviscid flow. This shows a sophisticated use of the user-defined source terms to add the extra pieces required to model a known (manufactured) flow solution.

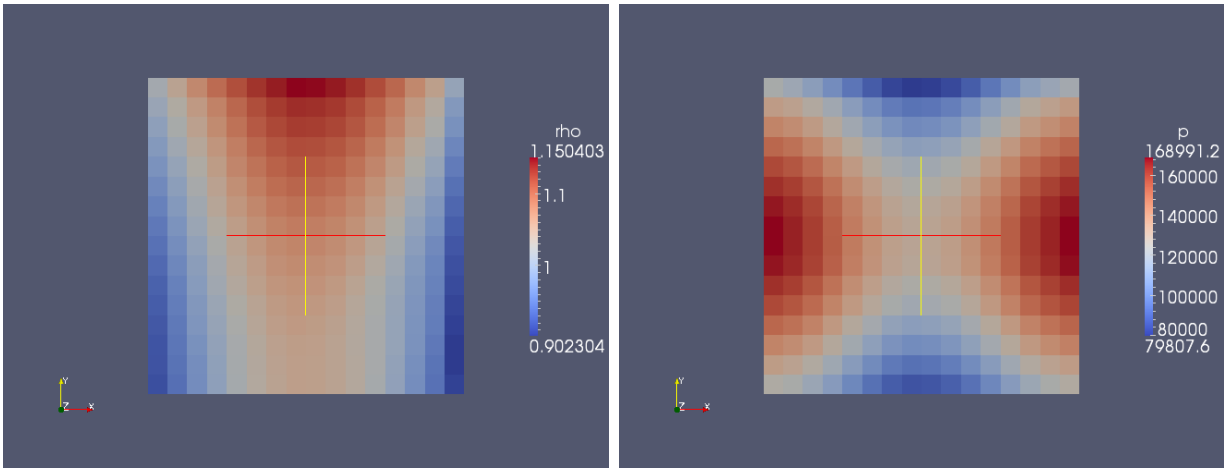


Figure 104: Density and pressure fields for the steady-state solution for the Method of Manufactured Solutions.

42.1 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
```

42.2 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
```

42.3 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
```

42.4 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
```

42.5 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}
```

42.6 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.

43 Method of manufactured solutions – Viscous flow

This extends the method of manufactured solutions as a code verification exercise to viscous flow. If you thought that the user-defined source terms for the Euler case (Section 42) were ugly, the viscous terms are so bad we no longer look at them. Here all of the source-term code is machine generated, principally by the SymPy computer algebra system (http://sympy.org).

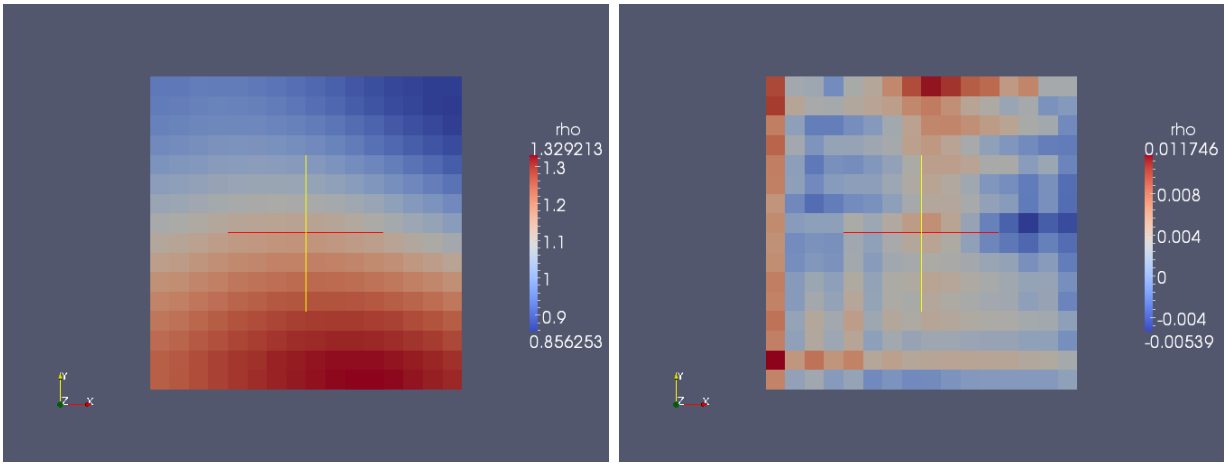


Figure 105: Density and error-in-density fields for the steady-state solution for the viscous (case=2) Method of Manufactured Solutions.

Note the smooth solution for the density field but the pattern of errors hinting at the 4 blocks used in this simulation. With respect to interior points in a block, the viscous terms are estimated with fewer data along the edges and at the corners.

Rowan might like to add his convergence plots here...

43.1 Input script (.py)

```
# mms.py
# This file can be used to simulate the
# Method of Manufactured Solutions test case.
#
# Author: Rowan J. Gollan
# Updated: 05-Feb-2008
# Generalized to the viscous case by PJ, June 2011.
#

# Read some case parameters from a fixed file format.
fp = open('case.txt', 'r');
case_str = fp.readline().strip()
case = int(case_str)
flux_calc_str = fp.readline().strip()
flux_calc = fluxcalcIndexFromName[flux_calc_str]
x_order_str = fp.readline().strip()
x_order = int(x_order_str)
blocking = fp.readline().strip()
nn_str = fp.readline()
nn = int(nn_str)
fp.close()

gdata.title = "Method of Manufactured Solutions, Case=%d." % case

select_gas_model(fname='very-viscous-air.lua')
p0 = 1.0e5
T0 = p0 / 287.0  # rho0 = 1.0
if case == 1 or case == 3:
    # Supersonic inviscid flow
    u0 = 800.0; v0 = 800.0
    gdata.viscous_flag = 0
elif case == 2 or case == 4:
    # Subsonic viscous flow
    u0 = 70.0; v0 = 90.0
    gdata.viscous_flag = 1
else:
    print "UNKNOWN CASE"
    sys.exit()

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")

if case == 1 or case == 3:
    bc_list = [ExtrapolateOutBC(x_order=1), ExtrapolateOutBC(x_order=1),
               UserDefinedBC("udf-bc.lua"), UserDefinedBC("udf-bc.lua")]
elif case == 2 or case == 4:
    bc_list = [UserDefinedBC("udf-bc.lua"),]*4

if blocking == 'single':
    blk = Block2D(make_patch(Line(c,d), Line(b,d), Line(a,b), Line(a,c)),
                  nni=nn, nnj=nn,
                  bc_list=bc_list,
                  fill_condition=initial, label="blk")
elif blocking == 'multi':
    blk = SuperBlock2D(make_patch(Line(c,d), Line(b,d), Line(a,b), Line(a,c)),
                       nni=nn, nnj=nn, nbi=4, nbj=4,
                       bc_list=bc_list,
                       fill_condition=initial, label="blk")
else:
    print "UNKOWN BLOCKING SELECTION:", blocking
    sys.exit()

gdata.udf_file = "udf-source.lua"
gdata.udf_source_vector_flag = 1
```

```
gdata.flux_calc = flux_calc
gdata.x_order = x_order
if case == 1 or case == 3:
    gdata.max_time = 60.0e-3
    gdata.max_step = 1000000
    gdata.dt = 1.0e-6
    gdata.cfl = 0.5
elif case == 2 or case == 4:
    gdata.max_time = 150.0e-3
    gdata.max_step = 3000000
    gdata.dt = 1.0e-7
    gdata.cfl = 0.5
# For the verification tests,
# do NOT use the limiters
gdata.apply_limiter_flag = 0
gdata.extrema_clipping_flag = 0
gdata.stringent_cfl = 1
gdata.dt_plot = gdata.max_time/20.0
```

43.2 Boundary condition file (.lua)

```
-- Lua script for the boundaries of a Manufactured Solution
--
-- Author: Rowan J. Gollan
-- Date: 04-Feb-2008
-- Generalised by PJ, May-June-2011

pi = math.pi
cos = math.cos
sin = math.sin
exp = math.exp

L = 1.0
R = 287.0
gam = 1.4

file = io.open("case.txt", "r")
case = file:read("*n")
file:close()

if case == 1 or case == 3 then
   -- Supersonic flow
   rho0=1.0; rhox=0.15; rhoy=-0.1; rhoxy=0.0; arhox=1.0; arhoy=0.5; arhoxy=0.0;
   u0=800.0; ux=50.0; uy=-30.0; uxy=0.0; aux=1.5; auy=0.6; auxy=0.0;
   v0=800.0; vx=-75.0; vy=40.0; vxy=0.0; avx=0.5; avy=2.0/3; avxy=0.0;
   p0=1.0e5; px=0.2e5; py=0.5e5; pxy=0.0; apx=2.0; apy=1.0; apxy=0.0
end

if case == 2 or case == 4 then
   -- Subsonic flow
   rho0=1.0; rhox=0.1; rhoy=0.15; rhoxy=0.08; arhox=0.75; arhoy=1.0; arhoxy=1.25;
   u0=70.0; ux=4.0; uy=-12.0; uxy=7.0; aux=5.0/3; auy=1.5; auxy=0.6;
   v0=90.0; vx=-20.0; vy=4.0; vxy=-11.0; avx=1.5; avy=1.0; avxy=0.9;
   p0=1.0e5; px=-0.3e5; py=0.2e5; pxy=-0.25e5; apx=1.0; apy=1.25; apxy=0.75
end

w0=0.0

if case == 1 or case == 2 then
   function S(x, y) return 1.0 end
else
   function S(x, y)
      rsq = (x - L/2)^2 + (y - L/2)^2
      return exp(-16.0*rsq/(L*L))
   end
end

function rho(x, y)
   return rho0 + S(x,y)*rhox*sin(arhox*pi*x/L) + S(x,y)*rhoy*cos(arhoy*pi*y/L)
          + S(x,y)*rhoxy*cos(arhoxy*pi*x*y/(L*L))
end

function u(x, y)
   return u0 + S(x,y)*ux*sin(aux*pi*x/L) + S(x,y)*uy*cos(auy*pi*y/L)
          + S(x,y)*uxy*cos(auxy*pi*x*y/(L*L))
end

function v(x, y)
   return v0 + S(x,y)*vx*cos(avx*pi*x/L) + S(x,y)*vy*sin((avy*pi*y)/L)
          + S(x,y)*vxy*cos(avxy*pi*x*y/(L*L))
end

function p(x, y)
   return p0 + S(x,y)*px*cos((apx*pi*x)/L) + S(x,y)*py*sin(apy*pi*y/L)
          + S(x,y)*pxy*sin(apxy*pi*x*y/(L*L))
end

function fill_table(t, x, y)
   t.p = p(x, y)
```

```
   t_rho = rho(x, y)
   t.u = u(x, y)
   t.v = v(x, y)
   t.w = 0.0
   t.T = {}
   t.T[0] = t.p/(t_rho*R)      -- temperature, K
   t.massf = {}     -- mass fractions to be provided as a table
   t.massf[0] = 1.0 -- mass fractions are indexed from 0 to nsp-1
   t.Tvib = {}   -- vibrational temperatures also indexed from 0
   return 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}
   -- Set constant conditions across the whole boundary.
   x = args.x; y = args.y
   i = args.i; j = args.j; k = args.k
   ghost1 = {}
   ghost2 = {}
   if args.which_boundary == NORTH then
      cell = sample_flow(block_id, i, j+1, k)
      ghost1 = fill_table(ghost1, cell.x, cell.y)
      cell = sample_flow(block_id, i, j+2, k)
      ghost2 = fill_table(ghost2, cell.x, cell.y)
   elseif args.which_boundary == SOUTH then
      cell = sample_flow(block_id, i, j-1, k)
      ghost1 = fill_table(ghost1, cell.x, cell.y)
      cell = sample_flow(block_id, i, j-2, k)
      ghost2 = fill_table(ghost2, cell.x, cell.y)
   elseif args.which_boundary == EAST then
      cell = sample_flow(block_id, i+1, j, k)
      ghost1 = fill_table(ghost1, cell.x, cell.y)
      cell = sample_flow(block_id, i+2, j, k)
      ghost2 = fill_table(ghost2, cell.x, cell.y)
   else -- WEST
      cell = sample_flow(block_id, i-1, j, k)
      ghost1 = fill_table(ghost1, cell.x, cell.y)
      cell = sample_flow(block_id, i-2, j, k)
      ghost2 = fill_table(ghost2, cell.x, cell.y)
   end
   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
   face = {}
   face.u = u(x, y)
   face.v = v(x, y)
   face_p = p(x, y)
   face_rho = rho(x, y)
   face.w = 0.0
   face.T = {}
   face.T[0] = face_p/(face_rho*R)
   face.massf = {}
   face.massf[0] = 1.0
   return face
end
```

43.3 Source term file (.lua)

The source terms are generated with the aid of the SymPy computer algebra system and inserted into the following template. The expressions for `fmass`, `fxmom`, `fymom`, and `fe` turn out to be 10, 23, 23 and 124 lines of 80-column text.[21]

```
-- udf-source-template.lua
-- Lua template for the source terms of a Manufactured Solution.
--
-- PJ, 29-May-2011
-- RJG, 06-Jun-2014
--   Declared maths functions as local

-- dummy functions to keep eilmer3 happy
function at_timestep_start(args) return nil end
function at_timestep_end(args) return nil end

local sin = math.sin
local cos = math.cos
local exp = math.exp
local pi = math.pi

function source_vector(args, cell)
   src = {}
   x = cell.x
   y = cell.y
<insert-source-terms-here>
   src.mass = fmass
   src.momentum_x = fxmom
   src.momentum_y = fymom
   src.momentum_z = 0.0
   src.total_energy = fe
   src.species = {}
   src.species[0] = src.mass
   return src
end
```

The Python script to do the real work is:

```
# Author: Rowan J. Gollan
# Place: The University of Queensland, Brisbane, Australia
# Date: 06-Jun-2014
#
# This script is used to generate the analytical source
# terms required to run the Method of Manufactured Solutions
# test case. The generated code is in Fortran95 and it can
# be converted to Lua with a separate script.
#
# This is an exercise in using sympy to generate the source
# terms. It is a transliteration of PJ's original work
# done using Maxima.

from sympy import *
from analytic_solution import *

Rgas, g, Prandtl, Cv, Cp = symbols('Rgas g Prandtl Cv Cp')
Rgas = 287.0
g = 1.4
Prandtl = 1.0
Cv = Rgas/(g-1)
```

[21] For the Maxima generated version. The SymPy version is similar but the source text is no longer wrapped at 80 characters.

```
Cp = g*Cv

mu, k = symbols('mu k')
mu = 10.0
k = Cp*mu/Prandtl
if case == 1 or case == 3:
    mu = 0.0
    k = 0.0

# Thermodynamic behvaiour, equation of state and energy equation
e, T, et = symbols('e T et')
e = p/rho/(g-1)
T = e/Cv
et = e + u*u/2 + v*v/2

# Heat flux
qx, qy = symbols('qx qy')
qx = -k*diff(T, x)
qy = -k*diff(T, y)

# Shear stress
tauxx, tauyy, tauxy = symbols('tauxx tauyy tauxy')
tauxx = 2./3*mu*(2*diff(u, x) - diff(v, y))
tauyy = 2./3*mu*(2*diff(v, y) - diff(u, x))
tauxy = mu*(diff(u, y) + diff(v, x))

# Navier-Stokes equations in conservative form
t, fmass, fxmom, fymom, fe = symbols('t fmass fxmom fymom fe')
fmass = diff(rho, t) + diff(rho*u, x) + diff(rho*v, y)
fxmom = diff(rho*u, t) + diff(rho*u*u+p-tauxx, x) + diff(rho*u*v-tauxy, y)
fymom = diff(rho*v, t) + diff(rho*v*u-tauxy, x) + diff(rho*v*v+p-tauyy, y)
fe = diff(rho*et, t) + diff(rho*u*et+p*u-u*tauxx-v*tauxy+qx, x) + diff(rho*v*et+p*v-u*tauxy-v*tauyy+qy, y)

if __name__ == '__main__':
    import re
    from sympy.utilities.codegen import codegen
    print 'Generating manufactured source terms.'
    [(f_name, f_code), (h_name, f_header)] = codegen(
        [("fmass", fmass), ("fxmom", fxmom), ("fymom", fymom), ("fe", fe)],
        "F95", "test", header=False)
    # Convert F95 to Lua code
    # This is heavily borrowed PJ's script: f90_to_lua.py
    # First we'll do some replacements
    f_code = f_code.replace('**', '^')
    f_code = f_code.replace('d0', '')
    # Now we'll break into lines so that we can completely remove
    # some lines and tidy others
    lines = f_code.split('\n')
    lines[:] = [l.lstrip() for l in lines if ( not l.startswith('REAL*8') and
                                               not l.startswith('implicit') and
                                               not l.startswith('end')) ]
    # Now reassemble but collect the split lines into a large line
    buf = ""
    f_code = ""
    for i,l in enumerate(lines):
        if l.endswith('&'):
            buf = buf + l[:-2]
        else:
            if buf == "":
                f_code = f_code + l + '\n'
            else:
                f_code = f_code + buf + l + '\n'
                buf = ""

    # Keep a tally of lines that end with an open function call
    # so that we can fix these
#    fn_list = ['cos', 'sin', 'exp']
#    open_call_lines = []
#    for i,l in enumerate(lines):
#        for fn in fn_list:
#            if l.endswith(fn + ' '):
#                open_call_lines.append(i)
#    follow_on_lines = [i+1 for i in open_call_lines]
```

```
    # Now rebuild as a single string
    # For the special lines with open function
    # calls, we'll append the following.
    # For all other lines, we add them just
    # as they are.
#    f_code = ""
#    for i,l in enumerate(lines):
#        if i in open_call_lines:
#            f_code += l[:-1] + lines[i+1].lstrip() + '\n'
#        elif i in follow_on_lines:
#            continue
#        else:
#            f_code += l + '\n'

    fin = open('udf-source-template.lua', 'r')
    template_text = fin.read()
    fin.close()
    lua_text = template_text.replace('<insert-source-terms-here>',
                                     f_code)

    fout = open('udf-source.lua', 'w')
    fout.write(lua_text)
    fout.close()
    print 'Done converting to Lua.'
```

Also, a user-defined gas model is needed so that a very large value for viscosity can be specified:

```
-- very-viscous-air.lua
--
-- User-defined gas model adapted from Rowan's example
-- PJ, 08-Jun-2011

-- Mandatory, set nsp and nmodes
model = 'user-defined'
nsp = 1
nmodes = 1

-- Local parameters for model
local R0 = 8.31451
local R = 287.0
local gamma = 1.4
local C_v = R / (gamma - 1)
local C_p = R + C_v

local mu0 = 1.0e1
local Pr = 1.0
local k0 = mu0 * C_p / Pr

-- Local helper functions
local sqrt, pow = math.sqrt, math.pow
local function sound_speed(gamma, R, T)
   return sqrt(gamma*R*T)
end

-- Mandatory function:
function eval_thermo_state_rhoe(Q)
   -- Assume rho and e[0] are given, compute the
   -- remaining thermodynamic variables.
   -- Remember: we need to access the temperature
   -- and energy as the 0th entry in an array
   -- of possible energies/temperatures.
   Q.T[0] = Q.e[0]/C_v
   Q.p = Q.rho*R*Q.T[0]
   Q.a = sound_speed(gamma, R, Q.T[0])
   -- 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 constants.
   Q.mu = mu0
   Q.k[0] = k0
   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[0][0] = 0.0
   return Q
end
```

43.4 Shell scripts

The coordination of the scripts to generate the simupation input files is handled at preparation stage.

```
#!/bin/bash
python make_source_terms.py
cp mms-regular.py mms.py
e3prep.py --job=mms
```

And, since we're is a hurry and have a nice new quad-core machine at home, we use the MPI version of the code to run the simulation.

```
#!/bin/bash

time mpirun -np 16 e3mpi.exe --job=mms --run
```

As for the simpler Euler case (Section 42), 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=mms --tindx=6 --gmodel-file="very-viscous-air.lua" \
    --compare-job=mms --compare-tindx=20
e3post.py --job=mms --tindx=7 --gmodel-file="very-viscous-air.lua" \
    --compare-job=mms --compare-tindx=20

echo "------------------------------------------------------------------------"
echo "Check simulation against analytical data:"
e3post.py --job=mms --tindx=20 --gmodel-file="very-viscous-air.lua" \
    --ref-function=analytic_solution.py \
    --global-norm-list="rho,L2"

echo "------------------------------------------------------------------------"
echo "Generate VTK files for plotting:"
e3post.py --job=mms --tindx=20 --gmodel-file="very-viscous-air.lua" \
    --vtk-xml
```

43.5 Python reference-function files

```
# analytic_solution.py
# Python version of the analytic solution described in Appendix A of
# C.J. Roy, C.C. Nelson, T.M. Smith and C.C. Ober
# Verification of Euler/Navier-Stokes codes using the method
# of manufactured solutions.
# Int J for Numerical Methods in Fluids 2004; 44:599-620
#
# PJ, 28-May-2011
# It essentially Rowan's code with more and renamed variables
# to bring it closer to the original paper.
# PJ, 30-June-2012
# Scale the disturbance to reduce its magnitude away from the centre.
# RJG, 06-June-2014
# Re-worked completely to use sympy

from sympy import *
R_air = 287.0
# Read case no.
fp = open('case.txt', 'r');
case_str = fp.readline().strip()
case = int(case_str)
fp.close()
# constants
L = 1.0
if case == 1 or case == 3:
    # Supersonic flow
    rho0=1.0; rhox=0.15; rhoy=-0.1; rhoxy=0.0; arhox=1.0; arhoy=0.5; arhoxy=0.0;
    u0=800.0; ux=50.0; uy=-30.0; uxy=0.0; aux=1.5; auy=0.6; auxy=0.0;
    v0=800.0; vx=-75.0; vy=40.0; vxy=0.0; avx=0.5; avy=2.0/3; avxy=0.0;
    p0=1.0e5; px=0.2e5; py=0.5e5; pxy=0.0; apx=2.0; apy=1.0; apxy=0.0

if case == 2 or case == 4:
    # Subsonic flow
    rho0=1.0; rhox=0.1; rhoy=0.15; rhoxy=0.08; arhox=0.75; arhoy=1.0; arhoxy=1.25;
    u0=70.0; ux=4.0; uy=-12.0; uxy=7.0; aux=5.0/3; auy=1.5; auxy=0.6;
    v0=90.0; vx=-20.0; vy=4.0; vxy=-11.0; avx=1.5; avy=1.0; avxy=0.9;
    p0=1.0e5; px=-0.3e5; py=0.2e5; pxy=-0.25e5; apx=1.0; apy=1.25; apxy=0.75

x, y, rho, u, v, p, S = symbols('x y rho u v p S')

if case == 1 or case == 2:
    S = 1.0
else:
    S = exp(-16.0*((x-L/2)*(x-L/2) + (y-L/2)*(y-L/2))/(L*L))

rho = rho0 + S*rhox*sin(arhox*pi*x/L) + S*rhoy*cos(arhoy*pi*y/L) + \
   S*rhoxy*cos(arhoxy*pi*x*y/(L*L));
u =  u0 + S*ux*sin(aux*pi*x/L) + S*uy*cos(auy*pi*y/L) + S*uxy*cos(auxy*pi*x*y/(L*L));
v =  v0 + S*vx*cos(avx*pi*x/L) + S*vy*sin(avy*pi*y/L) + S*vxy*cos(avxy*pi*x*y/(L*L));
p =  p0 + S*px*cos(apx*pi*x/L) + S*py*sin(apy*pi*y/L) + S*pxy*sin(apxy*pi*x*y/(L*L));

def ref_function(x1, y1, z1, t):
    inp = {x:x1, y:y1}
    rho1 = rho.subs(inp).evalf()
    p1 = p.subs(inp).evalf()
    T1 = p1/(rho1*R_air)
    u1 = u.subs(inp).evalf()
    v1 = v.subs(inp).evalf()
    return {"rho":rho1, "p":p1, "T":T1, "vel.x":u1, "vel.y":v1}

if __name__ == "__main__":
    pt = {x:0.5, y:0.5}
    print 'rho=', rho.subs(pt).evalf(), \
        'u=', u.subs(pt).evalf(), \
        'v=', v.subs(pt).evalf(), \
        'p=', p.subs(pt).evalf()
```

43.6 Notes

- This simulation requires more than 14 minutes on a single core of an AMD Phenom II processor to reach a final time of 80 ms in 6803 steps. With 4 cores running MPI, the wall-clock time is about 4 minutes.

44 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.

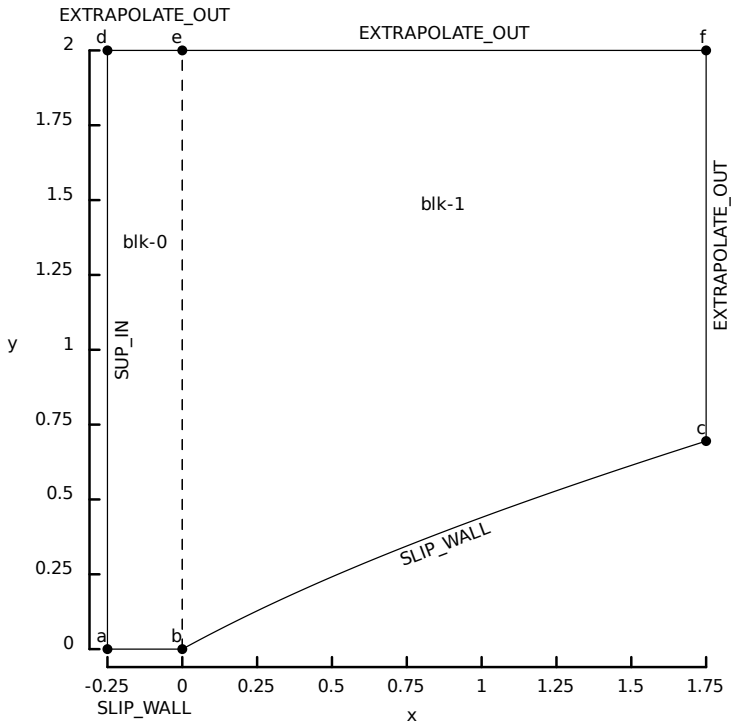


Figure 106: Layout for the oblique detonation wave simulation.

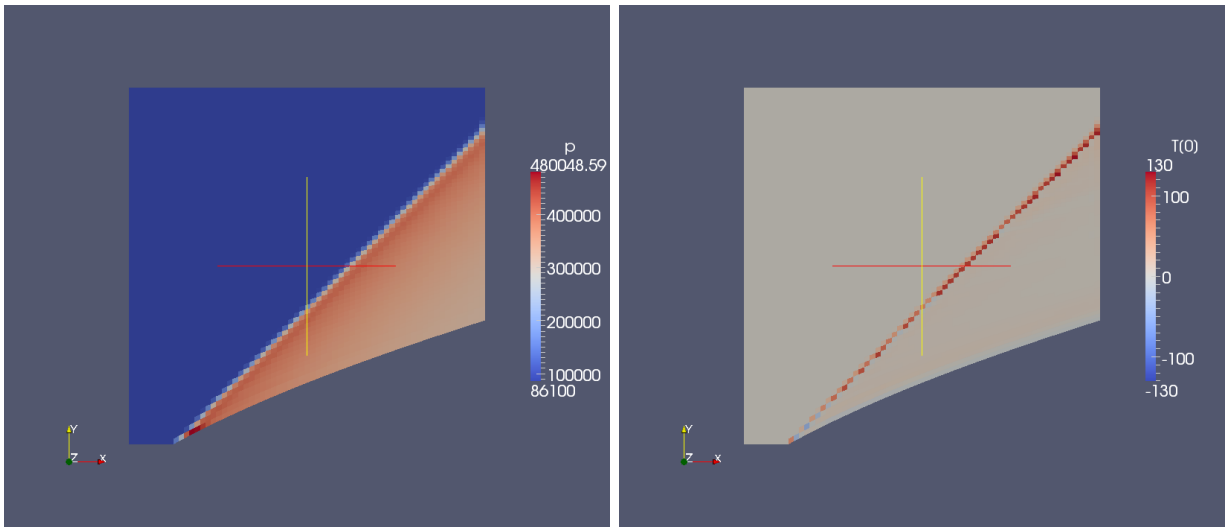


Figure 107: Pressure and temperature-difference fields for the steady-state solution. The temperature difference is the computed flow temperature minus the analytic solution temperature.

44.1 Input script (.py)

```
# odw.py
#
# A Python input file to describe the oblique
# detonation wave used as a verification case.
#
# This 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

# Read discretisation from a fixed file format.
fp = open('case.txt', 'r')
nn_str = fp.readline().strip()
nn = int(nn_str)

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 = nn
nny = nn

from oblique_detonation import *
from math import pi
od =  ObliqueDetonation(pi/4.0, 300.0, 3.0, 1.0)
wall = PyFunctionPath(od.create_wall_function(0.0, xmax))

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
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 = SuperBlock2D(
    psurf=make_patch(north0, east0west1, south0, west0),
    nni=nnx0, nnj=nny,
    nbi=1, nbj=8,
    bc_list=[ExtrapolateOutBC(x_order=1), AdjacentBC(), SlipWallBC(), SupInBC(inflow)],
    fill_condition=inflow,
    label="blk-0"
    )
blk_1 = SuperBlock2D(
    psurf=make_patch(north1, east1, south1, east0west1),
    nni=nnx1, nnj=nny,
    nbi=7, nbj=8,
    bc_list=[ExtrapolateOutBC(x_order=1), ExtrapolateOutBC(x_order=1), 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 = 2.0e-2  # seconds
gdata.max_step = 300000
gdata.dt = 1.0e-6
gdata.dt_plot = gdata.max_time/40.0
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)
```

44.2 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.atomic_constituents = {}
A.charge = 0
A.M = {
  value = 8.31451/287.0,
  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.atomic_constituents = {}
B.charge = 0
B.M = {
  value = 8.31451/287.0,
  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",
}
```

44.3 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
```

44.4 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-ref-function.py --gmodel-file="binary-gas.lua" --tindx=9999
```

44.5 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 cfpylib.nm.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 create_wall_function(self, xmin, xmax):

        def wall(t):
            # Map t --> x
            x = t*(xmax - xmin)
            (X, Yw) = self.find_XYw_from_x(x)
            (x, y) = self.transform_XY_2_xy(X, Yw)
            return (x, y, 0.0)

        return wall

    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."
```

44.6 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.

45 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.

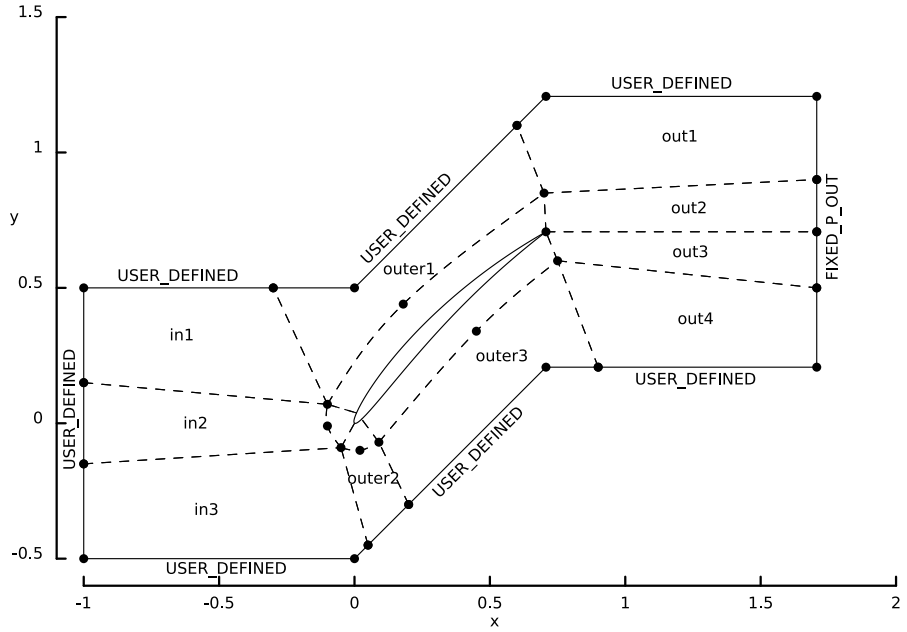


Figure 108: Overview of the flow geometry, showing the interior block boundaries and a number of anchor points. A number of the automatically-generated labels have been removed and others have been moved to make the diagram clearer.

45.1 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
# Accept defaults for air giving R=287.1, gamma=1.4
```

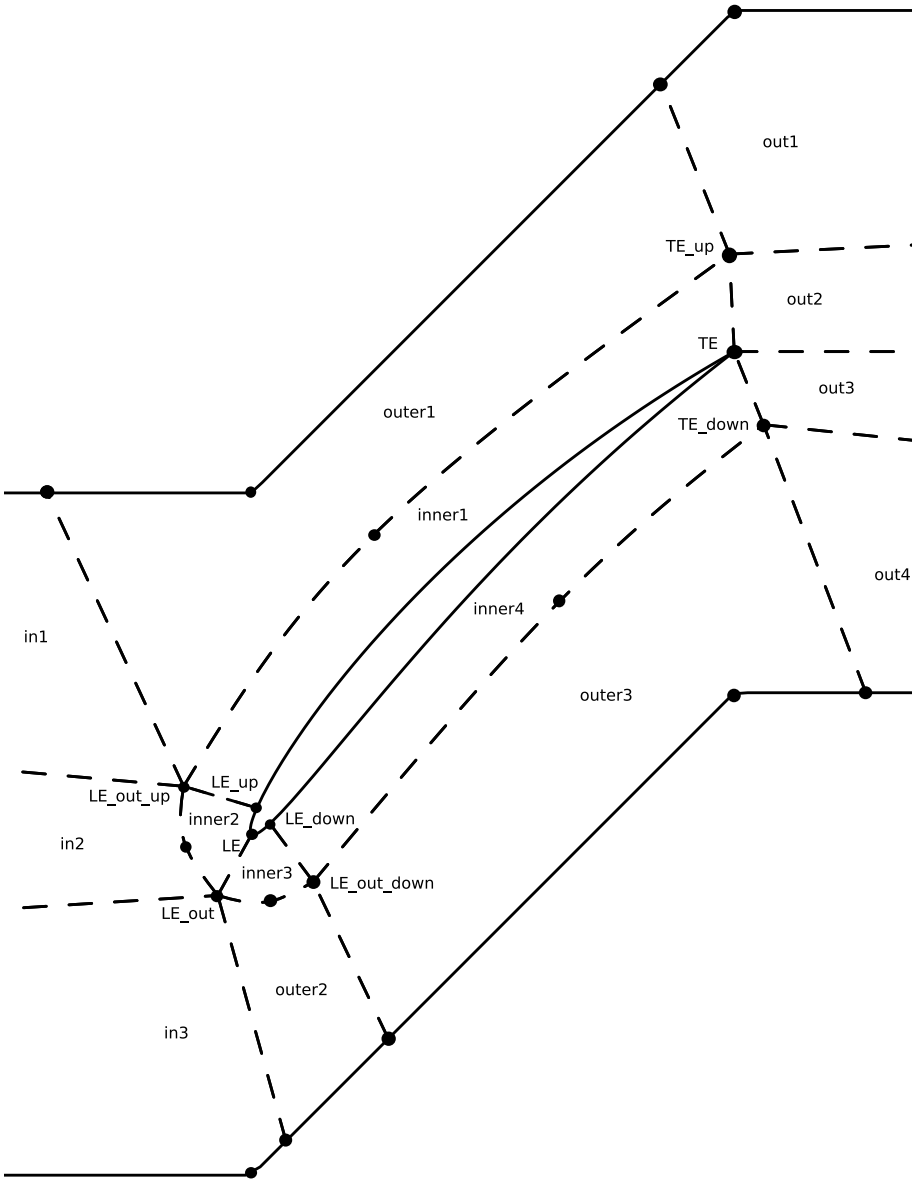


Figure 109: A further-edited diagram showing the blade surface and the arrangement of the inner blocks. More of the anchor-points are labelled.

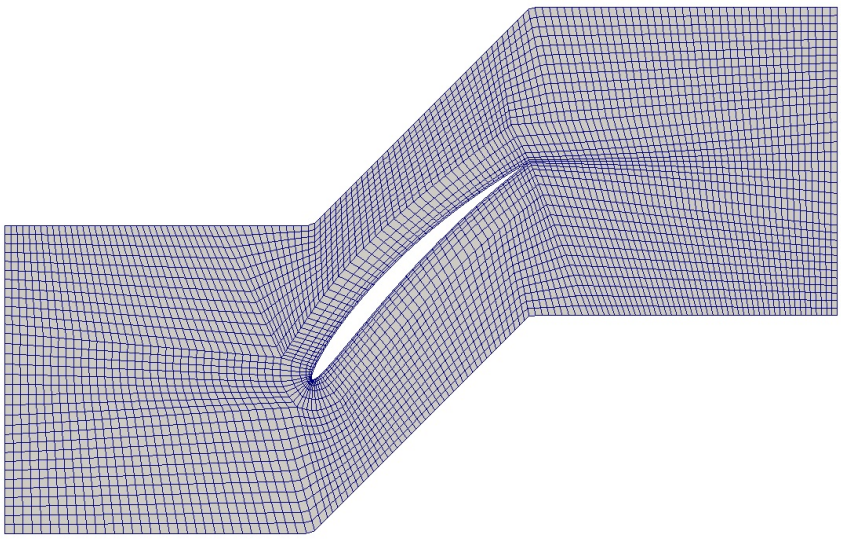


Figure 110: Mesh around the subsonic compressor blade.

Figure 111: Mach number field for flow over a subsonic compressor blade.

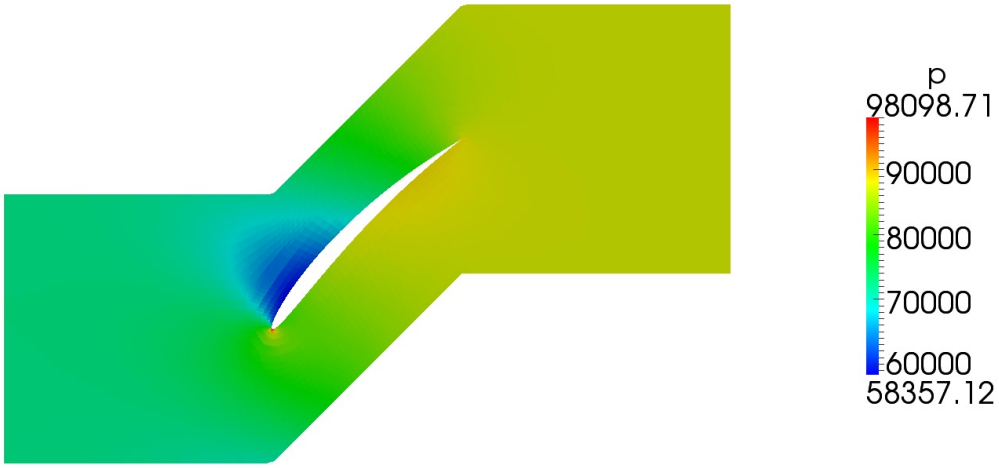


Figure 112: Pressure field for flow over a subsonic compressor blade.

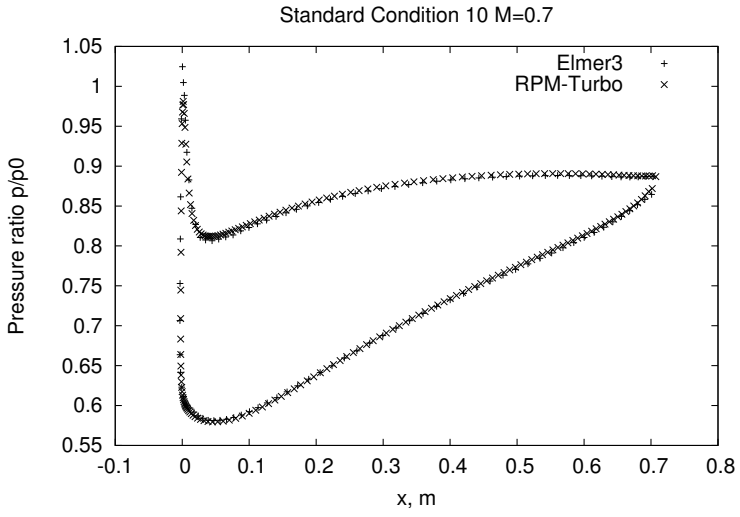


Figure 113: Pressure around the blade surface; comparison with RPM-Turbo reference data.

```
select_gas_model(model='ideal gas', species=['air'])
gdata.viscous_flag = 0 # inviscid simulation
gdata.gasdynamic_update_scheme = "euler"
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)
```

45.2 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
```

45.3 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 --tindx=all

# 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 "Eilmer3" 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.
```

45.4 Notes

- Run time is approximately 11600 seconds for 276720 steps on a computer with an AMD Phenom 9650 2.7 GHz processor.

46 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.

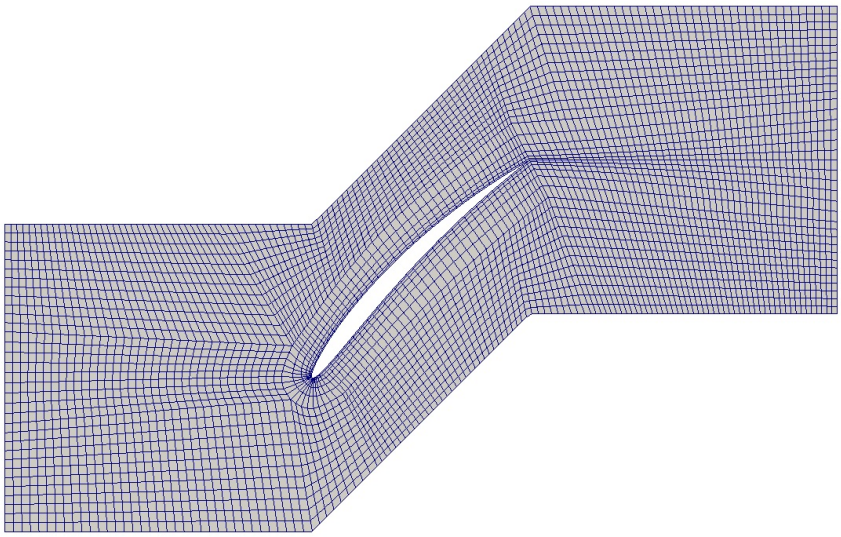


Figure 114: Mesh around the subsonic compressor blade.

46.1 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 115: Mach number field for flow over a subsonic compressor blade.

Figure 116: Pressure field for flow over a subsonic compressor blade.

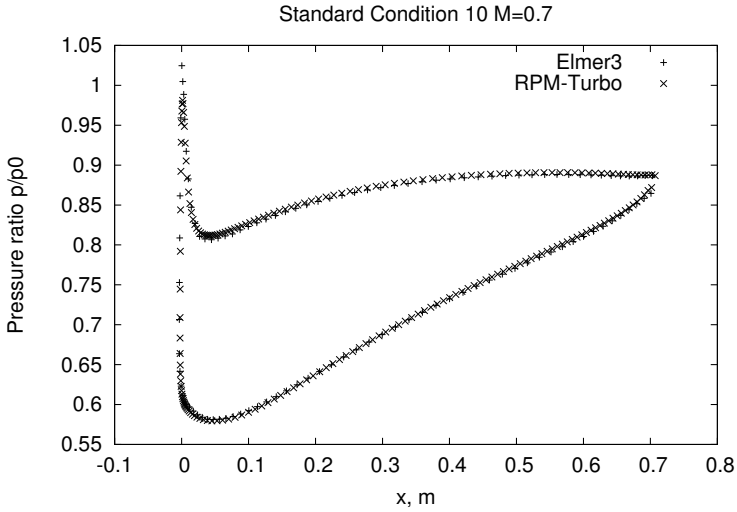


Figure 117: Pressure around the blade surface; comparison with RPM-Turbo reference data.

```
from cfpylib.geom.transform_pyfunc import rotate_pyfunc

#-----------------------------------------------------
# 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.

    """
    if abs(s) < 1.0e-12: s = 0
    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)
```

46.2 Notes

- Run time is approximately 11700 seconds for 280460 steps on a computer with an AMD Phenom 9650 2.7 GHz processor.

47 Couette Flow

This case is contributed by Jason Qin and computes the Couette flow between two parallel plates, one is a moving wall with a translational velocity while the other stationary wall. The flow is driven by the virtue of viscous drag force acting on the fluid and the applied pressure gradient parallel to the plates. This test case exercises the *moving-wall* boundary condition.

The boundary conditions are shown in Figure 118, with the NORTH and SOUTH faces set as the moving-wall and adiabatic boundary conditions, respectively. The velocity of the NORTH face is set as 100 m/s. The function `connect_blocks_2D` is used to connect the WEST and EAST faces, which can be regarded as periodic boundary conditions.

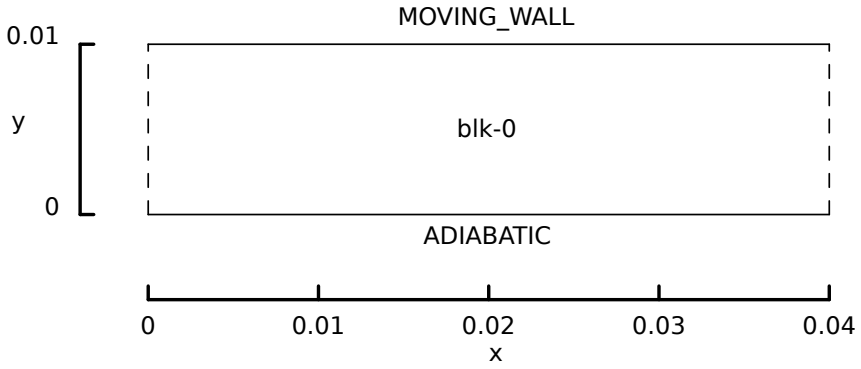
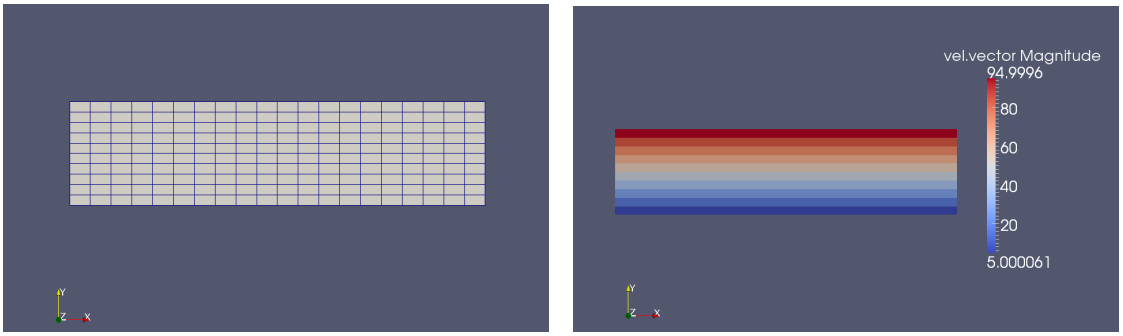


Figure 118: Flow domain for viscous flow between two parallel plates.

The mesh of 20×10 for this simple domain is plotted in Figure 119a and the velocity coutour is shown in Figure 119b. The maximum velocity at is approximately 95 m/s, slightly less than the translational velocity of moving wall, as expected for a cell-centre value.



(a) Mesh. (b) Velocity magnitude.

Figure 119: Uniform mesh and resulting velocity field for the two-dimensional Couette flow example.

Since the initial velocity profile along the height is set as linear, the solution achieves

steady state condition quickly. The final velocity profile is the same as the initial profile, as shown in Figure 120.

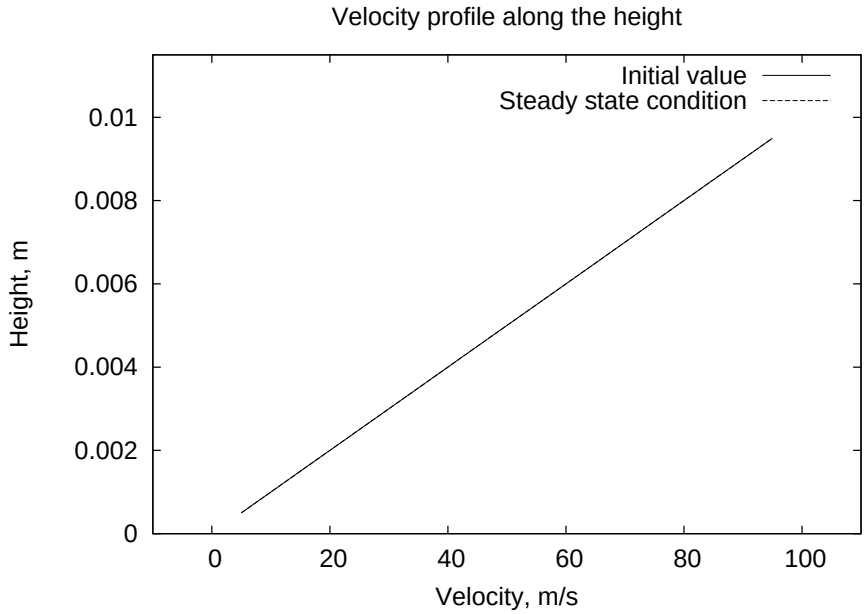


Figure 120: Velocity profile across the channel.

47.1 Input script (.py)

```
# couette.py
# Jason (Kan) Qin, November 2013

gdata.dimensions = 2
select_gas_model(model='ideal gas', species=['air'])

x_max = 0.040
y_max = 0.010
nx = 20
ny = 10

def simple_rectangle(r, s, t=0.0):
    global x_max, y_max
    return (x_max*r, y_max*s, 0.0)

p_inf = 100.0e3  # Pa
u_max = 100.0    # m/s

initial = FlowCondition(p=p_inf, u=u_max, v=0.0)
def initial_flow(x, y, z):
    global y_max, T_inf, p_inf, u_max
    u = u_max * y / y_max
    return FlowCondition(p=p_inf, u=u, v=0.0, add_to_list=0).to_dict()

blk = Block2D(PyFunctionSurface(simple_rectangle),
              nni=nx, nnj=ny,
              fill_condition=initial_flow,
              cf_list=4*[None,])

blk.set_BC("NORTH", "MOVING_WALL", r_omega=[0.0,0.0,0.0],v_trans=[u_max,0.0,0.0])
```

```
blk.bc_list[SOUTH] = AdiabaticBC()

# the WEST face is connected with the EAST face
connect_blocks_2D(blk,WEST,blk,EAST,check_corner_locations=False)

identify_block_connections()

gdata.title = "Couette flow (Just at start-up)"
gdata.viscous_flag = 1
gdata.flux_calc = ADAPTIVE
gdata.max_time = 50e-3
gdata.max_step = 20000
gdata.dt = 1.0e-9
gdata.dt_plot = 1e-3
gdata.dt_history = 1e-3
# The following scales provide a reasonable picture.
sketch.xaxis(0.0, 0.040, 0.01, -0.005)
sketch.yaxis(0.0, 0.010, 0.01, -0.004)
sketch.window(0.0, 0.0, 0.040, 0.010, 0.05, 0.05, 0.15, 0.075)
```

47.2 Shell scripts

```
#!/bin/sh
# couette.sh

e3prep.py --job=couette --do-svg
e3post.py --job=couette --vtk-xml --tindx=0

time e3shared.exe --job=couette --run

e3post.py --job=couette --vtk-xml --tindx=last

e3post.py --job=couette --output-file=dudy0.dat --tindx=0 \
          --slice-list="0,1,:,0"
e3post.py --job=couette --output-file=dudy1.dat --tindx=last \
          --slice-list="0,1,:,0"

gnuplot <<EOF
set term postscript eps 20
set output "velocity.ps"
set title "Velocity profile along the height"
set ylabel "Height, m"
set xlabel "Velocity, m/s"
set yrange [0.0:0.0115]
set xrange [-10.0:110.0]
plot "dudy0.dat" using 6:2 with lines title "Initial value", \
     "dudy1.dat" using 6:2 with lines title "Steady state condition"
EOF
```

47.3 Notes

- None

48 Radiating argon shock layer with thermochemical nonequilibrium

48.1 Experiment description

Rutowski *et al.* [27] measured the total and radiative heat fluxes at the stagnation point of a 1 inch diameter hemisphere immersed in a freestream flow of shock heated argon. A schematic of the experimental setup is depicted in Figure 121. The hemispherical model was placed in the test section of a 3 inch diameter stainless steel shock tube at the Lockheed Research Laboratories [28]. Incident shock waves with velocities up to 4.3 km/s ($M_s = 13.2$) were driven through the argon test gas at an initial pressure of 10 Torr. Total heat transfer at the stagnation point was measured with a surface mounted calorimetric gauge, in which a thin strip of polished platinum is exposed to the flow and the heat transfer determined from change in resistivity. Radiative heat transfer was measured with a similar gauge mounted behind a sapphire window that allowed transmission in the wavelength range $180 \leq \lambda \leq 6000$ nm. The platinum strip was determined to have a weighted average absorptivity of 0.4, however some experiments were also performed with a thin layer of camphor lampblack to give an absorptivity of 1.0. The error in the total heat transfer and radiative measurements were estimated to be $\pm 5\,\%$ and $\pm 15\,\%$, respectively.

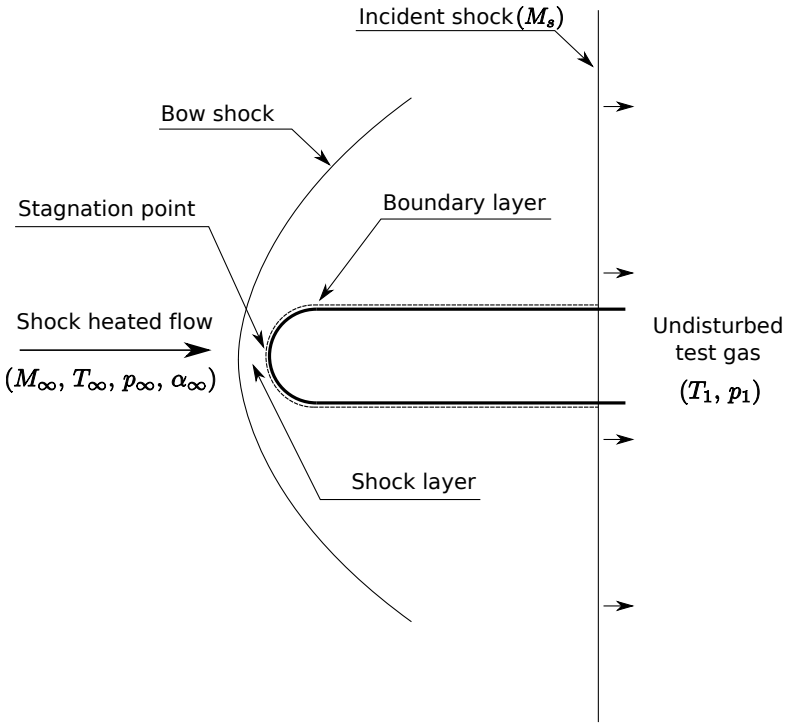


Figure 121: Schematic diagram of hemispherical model immersed in shock heat flow (adapted from Rutowski *et al.* [27]).

48.2 Simulation description

In the present work the experiment with a shock Mach number of 12.7 is considered. This condition has three experiment datapoints available for comparison. The simulation is run in three parts:

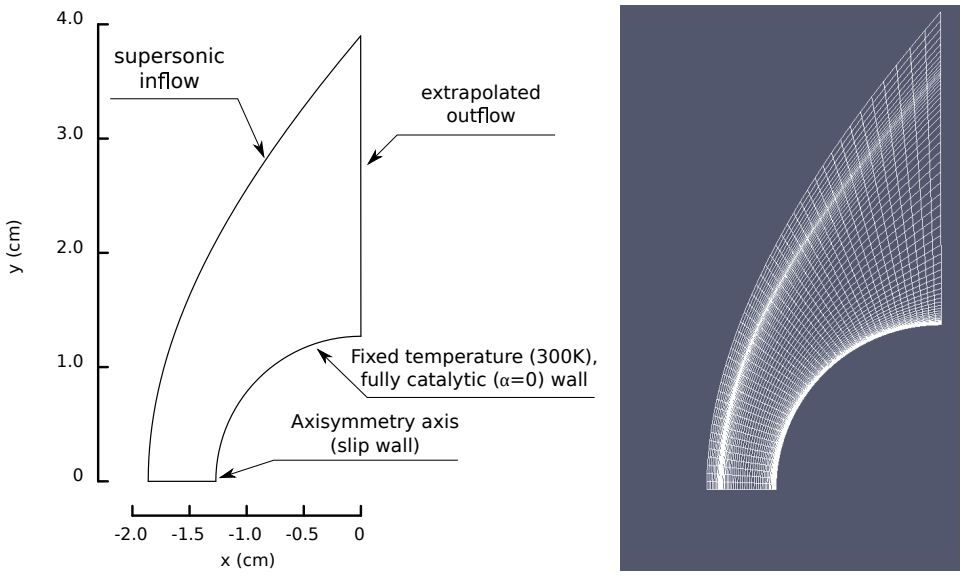
1. Inviscid (Euler equations, 10 body lengths of flow)

2. Viscous (Navier–Stokes equations, 5 body length of flow)

3. Viscous with radiation-flowfield coupling (Navier–Stokes equations, 2 body lengths of flow with 2 radiation transport calculations)

As the radiation–flowfield coupling is not very strong for this case, just two iterations between the radiation-transport solver and the flowfield solver were required to achieve a converged solution.

The computational domain and boundary conditions applied in the viscous stages of the simulation are illustrated in Figure 122a. The stainless steel surface is modelled as a fixed temperature, fully catalytic wall at 300 K. The computation grid for the viscous stages of the simulation is shown in Figure 122b. Clustering is applied in the vicinity of the shock front and boundary layer to enable the strong density gradients in these regions to be adequately captured.



(a) Computation domain (b) Computation grid

Figure 122: Computational domain and grid for Navier–Stokes simulations of the Rutowski and Bershader [27] experiments.

48.2.1 Thermodynamics

The argon plasma is modelled via the consideration of three species (Ar, Ar^+ and e^-) and two temperatures (a heavy particle translation temperature, T, and a combined free electron and bound electronic temperature, T_e). This allows the nonequilibrium between heavy-particle and free-electron translation to be captured whilst acknowledging the efficiency of bound electronic excitation via free electron impact. The electronic energy of the heavy particle species are calculated assuming Boltzmann distribution of the electronic state populations. The electronic level structure of Ar and Ar^+ are represented with 8 and 13 grouped levels, respectively, using the energy level from NIST ASD [29].

48.2.2 Viscous transport

Viscosity and thermal conductivity are calculated via the Gupta-Yos model [30]. The collision integrals are compiled from Wright *et al.* [31], Levin *et al.* [32] and Mason *et al.* [33]. Species mass diffusion is not considered.

48.2.3 Chemical reactions

Three chemical reactions are considered:

$$\begin{aligned} \mathrm{Ar} + \mathrm{Ar} &\leftrightharpoons \mathrm{Ar}^+ + \mathrm{e}^- + \mathrm{Ar} && (1)\\ \mathrm{Ar} + \mathrm{Ar}^+ &\leftrightharpoons \mathrm{Ar}^+ + \mathrm{e}^- + \mathrm{Ar}^+ && (2)\\ \mathrm{Ar} + \mathrm{e}^- &\leftrightharpoons \mathrm{Ar}^+ + \mathrm{e}^- + \mathrm{e}^- && (3)\\ & && (4) \end{aligned}$$

where photoionization has been omitted based on its relatively minor contribution for other high enthalpy conditions [34] . The reaction rates are determined via fitting an Arrhenius equaion to the two-stage model proposed by Petschek *et al.* [35]:

$$k_{f,M}(T_M) = S^*_{Ar-M}\sqrt{\frac{32}{\pi}\left(\frac{m_{Ar}+m_M}{m_{Ar}m_M}\right)}\cdot(k_B T_M)^{\frac{3}{2}}\left(\frac{\Theta_1}{2T_M}+1\right)\exp\left(-\frac{\Theta_1}{T_M}\right) \qquad (5)$$

where S^*_{Ar-M} is the first excitation collision cross-section for colliding particle M, T_M is the translational temperature of the colliding particle and Θ_1 is the characteristic temperature of the first excited state of argon (134,800 K). Glass *et al.* [36] found good agreement with shock tube electron density profiles using $S^*_{Ar-Ar} = 1.0 \times 10^{-19}$ and $S^*_{Ar-e^-} = 4.9 \times 10^{-18}\,\mathrm{cm}^2/\mathrm{eV}$; these cross-section are therefore used in the present work.

48.2.4 Thermal energy exchange

Translational energy exchange due to elastic collisions between free electrons and heavy particles is calculated via the model proposed by Appleton *et al.* [37]. The e^-–Ar effective elastic collision cross-section are taken from Jaffrin [38].

48.2.5 Radiation transport

A photon Monte–Carlo model is implemented to numerically solve for the both the radiative divergence throughout the flowfield ($\nabla \cdot \vec{q}_{\text{rad}}$), and the radiative heating incident on solid surfaces (q_{rad}). The basis of photon Monte–Carlo models [39, 40, 41] is the modelling of radiation transport by a collection of photon bundles with statistically determined properties. For the present simulations, a maximum of 32768 photons-per-cell are emitted, and their absorption throughout the flowfield is modelled via the partitioned energy model [42]. See § 4.5 of the Eilmer3 theory book (http://cfcfd.mechmining.uq.edu.au/pdf/eilmer3-theory-book.pdf) for a detailed description of the model.

48.2.6 Radiation spectra

The radiation spectra of the argon plasma are calculated via the Photaura model (see http://cfcfd.mechmining.uq.edu.au/pdf/photaura-users-guide.pdf). Three radiation mechanisms are considered:

1. Bound-bound line radiation
2. Photoionization continuum radiation
3. Bremsstrahlung continuum radiation

425 individual Ar lines and 307 individual Ar^+ lines from the NIST ASD [29] are considered, and photoionization cross-sections are obtained from TOPBase [43]. The experimentally measured Stark widths for 48 Ar lines collated by Griem [44] are implemented, with the remaining lines using the empirical fit suggested by Park [45]. The upper and lower state populations for the Ar atom are determined via application of a collisional-radiative model in the QSS limit [46]. The spectral grid is uniformally distributed in wavenumber space in the range $1000 \leq \eta \leq 150000$ $^{1/\text{cm}}$, with a resolution of 1 point / 10 $^{1/\text{cm}}$.

48.3 Results

Temperature solutions from Eilmer3 simulation of the Rutowski hemisphere with radiation-flowfield coupling are presented in Figure 123. Immediately behind the shock there is a region of thermal nonequilibrium, with the electron temperature being significantly lower

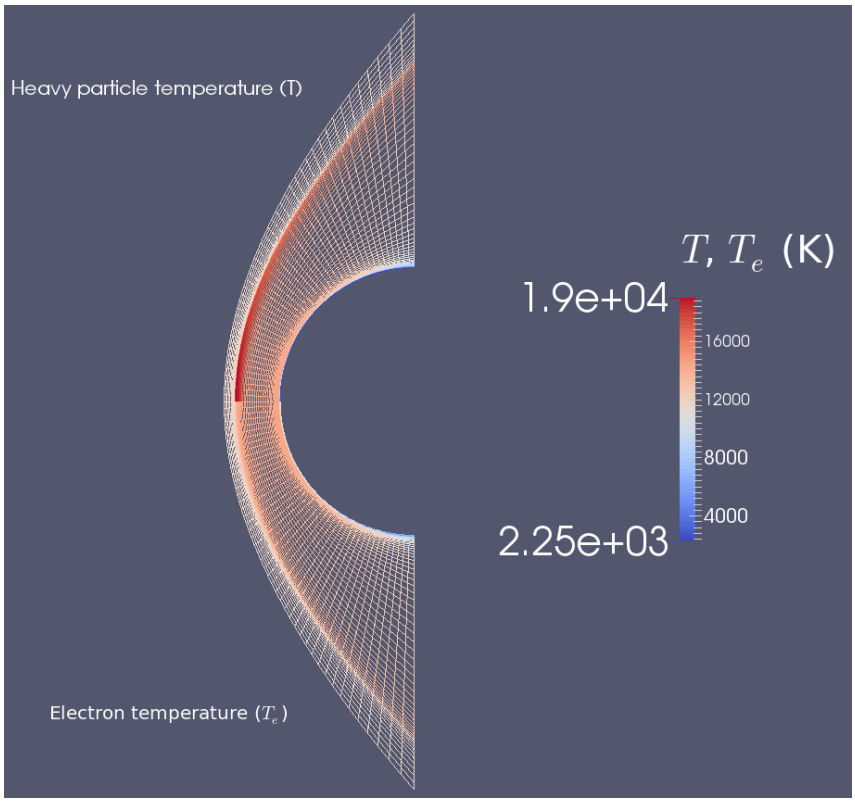


Figure 123: Temperature solutions from Eilmer3 simulation of the Rutowski hemisphere with radiation-flowfield coupling.

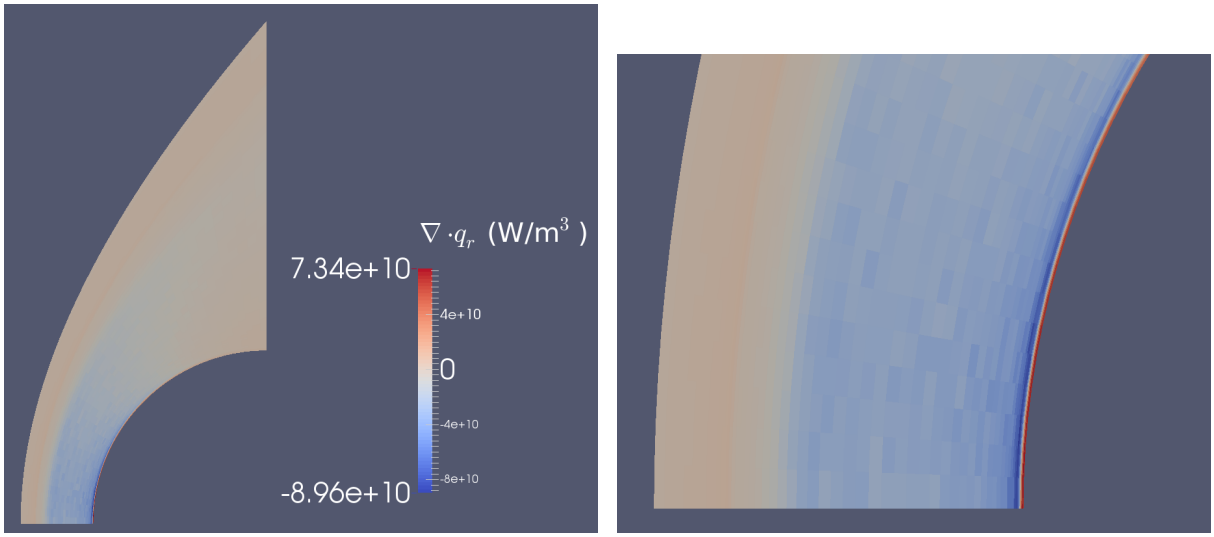
than the heavy particle temperature. Both temperatures radidly drop towards the 300K wall temperature near the model surface.

The radiation solution is presented in Figure 124. The hot shock layer is a net emitter of radiation (blue), whilst the cool and dense boundary layer is a net absorbed of radiation (red).

The computed surface radiative heating profiles are compared with the experiment measurements in Figure 125. The computed result at the stagnation point in the spectral range $67 \leq \lambda \leq 10000\,\text{nm}$ of approximately 5.1 kW/cm^2 is in agreement with the blackened guage data to within the measurement uncertainty bounds. The computed result for the supposed range of sensitivity for the radiometer ($180 \leq \lambda \leq 6000\,\text{nm}$), however, is slightly lower than the measured data. A finer spectral grid may improve the agreement with experiment by allowing the peaks of the atomic lines to be better resolved.

48.4 Run script (.sh)

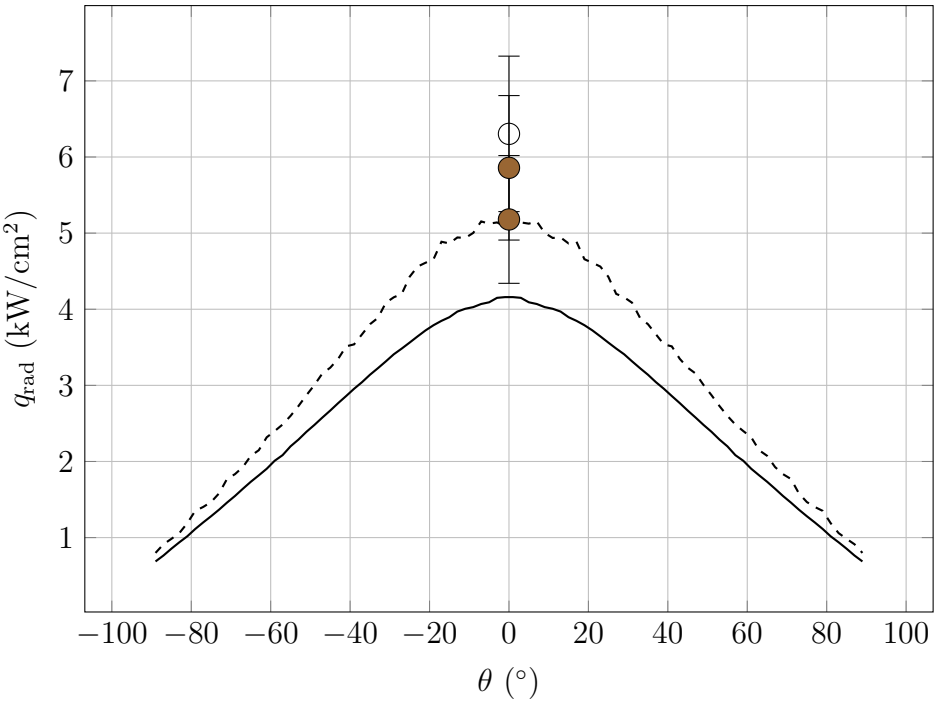
```
#!/bin/bash
#
# Radiation argon shock layer test case.
#
# DFP, 2-June-2014
```



(a) Full computational domain (b) Stagnation point detail

Figure 124: Radiation solution from Eilmer3 simulation of the Rutowski hemisphere with radiation-flowfield coupling.

Figure 125: Radiative heating on the hemisphere surface: Eilmer3 for $67 \leq \lambda \leq 10000\,\text{nm}$ (- - -), Eilmer3 for $180 \leq \lambda \leq 6000\,\text{nm}$ (——), experiment with blackened gauge (●) and experiment with adjust gauge (○)



```
echo "Run the inviscid stage"
cd part1-inviscid/
e3prep.py --job=hemisphere > LOGFILE_PREP
mpirun -np 4 e3mpi.exe -f hemisphere -r > LOGFILE_RUN
cd ..

echo "Run the viscous stage"
cd part2-viscous/
e3prep.py --job=hemisphere > LOGFILE_PREP
echo "Adding viscous effects"
mpirun -np 4 e3mpi.exe -f hemisphere -q -r > LOGFILE_RUN
echo "Increasing CFL number to 0.5"
set_control_parameter.py hemisphere.control cfl 0.5
set_control_parameter.py hemisphere.control max_time 1.88964e-05
mpirun -np 4 e3mpi.exe -f hemisphere -t 1 -q -r >> LOGFILE_RUN

echo "Run the viscous with radiation stage"
cd part3-viscous-with-radiation/
radmodel.py -i Ar-nonequilibrium-radiation.py -L rad-model.lua > LOGFILE_PREP
e3prep.py --job=hemisphere >> LOGFILE_PREP

echo "Run e3mpi for one body length on new grid"
mpirun -np 4 e3mpi.exe -f hemisphere -q -r > LOGFILE_RUN
get_residuals.py 0 residuals-0.txt
mv e3mpi*.log e3mpi.log.part1/
echo "First radiation transport calculation"
e3rad.exe -f hemisphere -q -t 1 -r > LOGFILE_RUN

echo "Run e3mpi for another body length with radiation coupling"
set_control_parameter.py hemisphere.control max_time 7.55855e-06
mpirun -np 4 e3mpi.exe -f hemisphere -q -t 2 -r >> LOGFILE_RUN
get_residuals.py 0 residuals-1.txt
echo "Final radiation transport calculation"
radmodel.py -i Ar-nonequilibrium-radiation-180to6000nm.py -L rad-model.lua >> LOGFILE_PREP
e3rad.exe -f hemisphere -q -t 3 -r >> LOGFILE_RUN

echo "Extract final surface heat flux profile"
e3post.py --job=hemisphere --tindx=4 --heat-flux-list="2:3,1,:,:,:" > LOGFILE_POST
cd ..

# Compute the radiative heat flux error with respect to the experiment
# measurement using average of the two experimental datapoints with
# blackened gauges at Ms=12.7 from Figure 9
./compute_qrad_error.py part3-viscous-with-radiation/hf_profile.data 5.5188e7 > LOGFILE_COMPARE
```

48.5 Eilmer3 input scripts (.py)

48.5.1 Part 1 – inviscid flow

```
## \file hemisphere.py
## \brief Mach 12.7 condition from Rutowski and Bershader (1964)
## \author DFP, 28-May-2014
##

from cfpylib.grid.shock_layer_surface import *

gdata.title  = "Shock heated argon flow over a 1/2 inch hemisphere"
gdata.title += "- part 1: inviscid"
print gdata.title

# axisymmetry
gdata.axisymmetric_flag = 1

# gas model
species = select_gas_model(  model = "two temperature gas", \
                           species = [ "Ar", "Ar_plus", "e_minus" ] )
```

```
gm = get_gas_model_ptr()
nsp = gm.get_number_of_species()
ntm = gm.get_number_of_modes()

# kinetics
set_reaction_update("../../kinetic-models/Ar-2T-chemical-reactions.lua")
set_energy_exchange_update("../../kinetic-models/Ar-2T-energy-exchange.lua")

# flow conditions - shock heated argon, initially at 10 Torr and 300K
T_wall = 300.0
Ms = 12.7
from cfpylib.gasdyn.cea2_gas import *
reactants = { "Ar" : 1.0, "Ar+" : 0.0, "e-" : 0.0 }
cea = Gas( reactants, onlyList=reactants.keys(), with_ions=True, \
           trace=1.0e-20 )
cea.set_pT(p=1333.3,T=300.0)
Us = cea.a * Ms
print "Us = ", Us
cea.shock_process(Us)
rho_inf = cea.rho
T_inf = [ cea.T ]*ntm
massf_inf = []
for isp,sp in enumerate(species):
    cea_sp = sp.replace("_plus","+").replace("_minus","-")
    massf_inf.append( cea.species[cea_sp] )
u_inf = Us - cea.u2

# do some calculations to get pressure and Mach number
Q = Gas_data(gm)
Q.rho = rho_inf
for itm in range(ntm):
    Q.T[itm] = T_inf[itm]
mf_sum = 0.0
for isp in range(nsp):
    Q.massf[isp] = massf_inf[isp]
    mf_sum += Q.massf[isp]
massf_inf = []
for isp in range(nsp):
    Q.massf[isp] /= mf_sum
    massf_inf.append( Q.massf[isp] )
gm.eval_thermo_state_rhoT(Q)
Q.print_values(False)
M_inf = u_inf / Q.a
p_inf = Q.p
print "M_inf = %0.2f" % ( M_inf )

# inflow and initial conditions
inflow  = FlowCondition(p=p_inf,      u=u_inf, v=0.0, T=T_inf, \
                        massf=massf_inf)
initial = FlowCondition(p=p_inf/10.0, u=0.0,   v=0.0, T=T_inf, \
                        massf=massf_inf)

# geometry
Rn = 1.27e-2
psurf, west = make_parametric_surface( 1.0, 1.0, M_inf, Rn, \
                                       axi=gdata.axisymmetric_flag )

# mesh clustering
cf_list=[ None ] * 4

# boundary conditions
bc_list=[ExtrapolateOutBC(),                    # outflow
         FixedTBC(T_wall),                      # surface
         SlipWallBC(),                          # symmetry
         SupInBC(inflow)]                       # inflow

# catalytic boundary conditions
wc_bc_list=[NonCatalyticWBC()]*4

blk_0 = SuperBlock2D(psurf=psurf,
             fill_condition=initial,
             nni=40, nnj=30,
             nbi=2, nbj=2,
```

```
            cf_list=cf_list,
            bc_list=bc_list,
            wc_bc_list=wc_bc_list,
            label="BLOCK-0")

identify_block_connections()

# global simulation parameters
gdata.viscous_flag = 0
gdata.viscous_delay = 0.1 * Rn / u_inf
gdata.viscous_factor_increment = 1.0e-3
gdata.diffusion_flag = 0
gdata.diffusion_model = "ConstantLewisNumber"
gdata.electric_field_work_flag = 0
gdata.reaction_time_start = 0 * Rn / u_inf
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.max_time = Rn * 10 / u_inf    # 10 body lengths
gdata.reaction_time_start = Rn * 1 / u_inf
gdata.max_step = 230000
gdata.dt = 1.0e-10
gdata.stringent_cfl = 1
gdata.dt_plot = Rn * 1 / u_inf    # 10 solutions
gdata.cfl = 1.0
gdata.cfl_count = 10
gdata.print_count = 20

sketch.scales(0.03/Rn, 0.03/Rn)
sketch.origin(0.0, 0.0)
sketch.xaxis(-2.0e-2, 0.0, 0.5e-2, -0.3e-2)
sketch.yaxis(0.0, 4.0e-2, 1.0e-2, -0.3e-2)
```

48.5.2 Part 2 – viscous flow

```
## \file hemisphere.py
## \brief Mach 12.7 condition from Rutowski and Bershader (1964)
## \author DFP, 28-May-2014
##

from cfpylib.grid.shock_layer_surface import *

gdata.title  = "Shock heated argon flow over a 1/2 inch hemisphere"
gdata.title += "- part 2: viscous"
print gdata.title

# axisymmetry
gdata.axisymmetric_flag = 1

# gas model
species = select_gas_model(  model = "two temperature gas", \
                           species = [ "Ar", "Ar_plus", "e_minus" ] )
gm = get_gas_model_ptr()
nsp = gm.get_number_of_species()
ntm = gm.get_number_of_modes()

# kinetics
set_reaction_update("../../kinetic-models/Ar-2T-chemical-reactions.lua")
set_energy_exchange_update("../../kinetic-models/Ar-2T-energy-exchange.lua")

# flow conditions - shock heated argon, initially at 10 Torr and 300K
T_wall = 300.0
Ms = 12.7
from cfpylib.gasdyn.cea2_gas import *
reactants = { "Ar" : 1.0, "Ar+" : 0.0, "e-" : 0.0 }
cea = Gas( reactants, onlyList=reactants.keys(), with_ions=True, \
           trace=1.0e-20 )
cea.set_pT(p=1333.3,T=300.0)
```

```
Us = cea.a * Ms
print "Us = ", Us
cea.shock_process(Us)
rho_inf = cea.rho
T_inf = [ cea.T ]*ntm
massf_inf = []
for isp,sp in enumerate(species):
    cea_sp = sp.replace("_plus","+").replace("_minus","-")
    massf_inf.append( cea.species[cea_sp] )
u_inf = Us - cea.u2

# do some calculations to get pressure and Mach number
Q = Gas_data(gm)
Q.rho = rho_inf
for itm in range(ntm):
    Q.T[itm] = T_inf[itm]
mf_sum = 0.0
for isp in range(nsp):
    Q.massf[isp] = massf_inf[isp]
    mf_sum += Q.massf[isp]
massf_inf = []
for isp in range(nsp):
    Q.massf[isp] /= mf_sum
    massf_inf.append( Q.massf[isp] )
gm.eval_thermo_state_rhoT(Q)
Q.print_values(False)
M_inf = u_inf / Q.a
p_inf = Q.p
print "M_inf = %0.2f" % ( M_inf )

# inflow and initial conditions (continuation from part 1)
inflow  = FlowCondition(p=p_inf,      u=u_inf, v=0.0, T=T_inf, \
                        massf=massf_inf)
initial = ExistingSolution(rootName="hemisphere", \
                           solutionWorkDir="../part1-inviscid/", \
                           nblock=4, tindx=10)

# geometry
Rn = 1.27e-2
gamma = 0.2
print "WARNING: the shock fitting procedure takes a long time as the"
print "         Billig function is difficult to solve at this Mach"
print "         number."
shock, nodes = fit_billig2shock( initial, gdata.axisymmetric_flag, \
                                 M_inf, Rn, None, show_plot=False )
psurf, west = make_parametric_surface( M_inf=M_inf, R=Rn, \
                                       axi=gdata.axisymmetric_flag, \
                                       east=None, shock=shock, \
                                       f_s=1.0/(1.0-gamma) )

# boundary conditions
bc_list=[ExtrapolateOutBC(),              # outflow
         FixedTBC(T_wall),                # surface
         SlipWallBC(),                    # symmetry
         SupInBC(inflow)]                 # inflow

# catalycity boundary conditions
wc_bc_list=[NonCatalyticWBC(),                # outflow
           SuperCatalyticWBC([1.0,0.0,0.0]),  # surface
           NonCatalyticWBC(),                 # symmetry
           NonCatalyticWBC()]                 # inflow

# mesh clustering
beta0 = 1.1; dx0 = 5.0e-1; dx1 = 5.0e-2
beta1 = 1.0
cf_list = [BHRCF(beta0,dx0,dx1,gamma),    # outflow
           RCF(0,1,beta1),                # surface
           BHRCF(beta0,dx0,dx1,gamma),    # symmetry
           RCF(0,1,beta1)]                # inflow

# computation domain
blk_0 = SuperBlock2D(psurf=psurf,
             fill_condition=initial,
```

```
            nni=60, nnj=45,
            nbi=2, nbj=2,
            cf_list=cf_list,
            bc_list=bc_list,
            wc_bc_list=wc_bc_list,
            label="BLOCK-0")

identify_block_connections()

# global simulation parameters
gdata.viscous_flag = 1
gdata.viscous_delay = 0.001 * Rn / u_inf
gdata.viscous_factor_increment = 1.0e-4
# NOTE: diffusion is currently turned off
gdata.diffusion_flag = 0
gdata.diffusion_delay = 0.001 * Rn / u_inf
gdata.diffusion_factor_increment = 1.0e-4
gdata.diffusion_model = "Ramshaw-Chang"
# NOTE: if an ambipolar diffusion model is being used, the electric
#       field work term should be included
gdata.electric_field_work_flag = gdata.diffusion_flag
gdata.reaction_time_start = 0 * Rn / u_inf
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.max_time = Rn * 1 / u_inf    # 1 body length
gdata.max_step = 2300000
gdata.dt = 1.0e-10
gdata.stringent_cfl = 1
gdata.dt_plot = Rn * 1 / u_inf    # 1 solution
# NOTE: the CFL number can be increased to 0.5 after the viscous terms
#       have been added
gdata.cfl = 1.0e-1
gdata.cfl_count = 1
gdata.print_count = 10

sketch.scales(0.03/Rn, 0.03/Rn)
sketch.origin(0.0, 0.0)
sketch.xaxis(-2.0e-2, 0.0, 0.5e-2, -0.3e-2)
sketch.yaxis(0.0, 4.0e-2, 1.0e-2, -0.3e-2)
```

48.5.3 Part 3 – viscous flow with radiation coupling

```
## \file hemisphere.py
## \brief Mach 12.7 condition from Rutowski and Bershader (1964)
## \author DFP, 28-May-2014
##

from cfpylib.grid.shock_layer_surface import *

gdata.title  = "Shock heated argon flow over a 1/2 inch hemisphere"
gdata.title += "- part 3: viscous with radiation coupling"
print gdata.title

# axisymmetry
gdata.axisymmetric_flag = 1

# gas model
species = select_gas_model(  model = "two temperature gas", \
                           species = [ "Ar", "Ar_plus", "e_minus" ] )
gm = get_gas_model_ptr()
nsp = gm.get_number_of_species()
ntm = gm.get_number_of_modes()

# kinetics
set_reaction_update("../../kinetic-models/Ar-2T-chemical-reactions.lua")
set_energy_exchange_update("../../kinetic-models/Ar-2T-energy-exchange.lua")
```

```
# radiation model
# NOTE: update frequency of 0 means e3mpi.exe and e3shared.exe will not
#       try and compute the radiation source term - we leave this to
#       the dedicated, parallelised radiation solver, e3rad.exe
select_radiation_model(input_file="rad-model.lua", update_frequency=0, \
                       scaling=True)

# flow conditions - shock heated argon, initially at 10 Torr and 300K
T_wall = 300.0
Ms = 12.7
from cfpylib.gasdyn.cea2_gas import *
reactants = { "Ar" : 1.0, "Ar+" : 0.0, "e-" : 0.0 }
cea = Gas( reactants, onlyList=reactants.keys(), with_ions=True, \
           trace=1.0e-20 )
cea.set_pT(p=1333.3,T=300.0)
Us = cea.a * Ms
print "Us = ", Us
cea.shock_process(Us)
rho_inf = cea.rho
T_inf = [ cea.T ]*ntm
massf_inf = []
for isp,sp in enumerate(species):
    cea_sp = sp.replace("_plus","+").replace("_minus","-")
    massf_inf.append( cea.species[cea_sp] )
u_inf = Us - cea.u2

# do some calculations to get pressure and Mach number
Q = Gas_data(gm)
Q.rho = rho_inf
for itm in range(ntm):
    Q.T[itm] = T_inf[itm]
mf_sum = 0.0
for isp in range(nsp):
    Q.massf[isp] = massf_inf[isp]
    mf_sum += Q.massf[isp]
massf_inf = []
for isp in range(nsp):
    Q.massf[isp] /= mf_sum
    massf_inf.append( Q.massf[isp] )
gm.eval_thermo_state_rhoT(Q)
Q.print_values(False)
M_inf = u_inf / Q.a
p_inf = Q.p
print "M_inf = %0.2f" % ( M_inf )

# inflow and initial conditions (continuation from part 2)
inflow  = FlowCondition(p=p_inf,      u=u_inf, v=0.0, T=T_inf, \
                        massf=massf_inf)
initial = ExistingSolution(rootName="hemisphere", \
                           solutionWorkDir="../part2-viscous/", \
                           nblock=4, tindx=5)

# geometry
Rn = 1.27e-2
gamma = 0.2
print "WARNING: the shock fitting procedure takes a long time as the"
print "         Billig function is difficult to solve at this Mach"
print "         number."
shock, nodes = fit_billig2shock( initial, gdata.axisymmetric_flag, \
                                 M_inf, Rn, None, show_plot=False )
psurf, west = make_parametric_surface( M_inf=M_inf, R=Rn, \
                                       axi=gdata.axisymmetric_flag, \
                                       east=None, shock=shock, \
                                       f_s=1.0/(1.0-gamma) )

# boundary conditions
bc_list=[ExtrapolateOutBC(),              # outflow
         FixedTBC(T_wall),                # surface
         SlipWallBC(),                    # symmetry
         SupInBC(inflow)]                 # inflow

# catalycity boundary conditions
wc_bc_list=[NonCatalyticWBC(),                # outflow
```

```
           SuperCatalyticWBC([1.0,0.0,0.0]),  # surface
           NonCatalyticWBC(),                 # symmetry
           NonCatalyticWBC()]                 # inflow

# mesh clustering
beta0 = 1.1; dx0 = 5.0e-1; dx1 = 5.0e-2
beta1 = 1.0
cf_list = [BHRCF(beta0,dx0,dx1,gamma),    # outflow
           RCF(0,1,beta1),                # surface
           BHRCF(beta0,dx0,dx1,gamma),    # symmetry
           RCF(0,1,beta1)]                # inflow

# computation domain
blk_0 = SuperBlock2D(psurf=psurf,
             fill_condition=initial,
             nni=60, nnj=45,
             nbi=2, nbj=2,
             cf_list=cf_list,
             bc_list=bc_list,
             wc_bc_list=wc_bc_list,
             label="BLOCK-0")

identify_block_connections()

# global simulation parameters
gdata.viscous_flag = 1
# NOTE: diffusion is currently turned off
gdata.diffusion_flag = 0
gdata.diffusion_model = "Ramshaw-Chang"
# NOTE: if an ambipolar diffusion model is being used, the electric
#       field work term should be included
gdata.electric_field_work_flag = gdata.diffusion_flag
gdata.reaction_time_start = 0 * Rn / u_inf
gdata.flux_calc = ADAPTIVE
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.max_time = Rn * 1 / u_inf    # 1 body length
gdata.max_step = 2300000
gdata.dt = 1.0e-10
gdata.stringent_cfl = 1
gdata.dt_plot = Rn * 1 / u_inf    # 1 solution
# NOTE: the CFL number can be increased to 0.5 after the viscous terms
#       have been added
gdata.cfl = 5.0e-1
gdata.cfl_count = 1
gdata.print_count = 10

sketch.scales(0.03/Rn, 0.03/Rn)
sketch.origin(0.0, 0.0)
sketch.xaxis(-2.0e-2, 0.0, 0.5e-2, -0.3e-2)
sketch.yaxis(0.0, 4.0e-2, 1.0e-2, -0.3e-2)
```

48.6 Chemical reaction script (.lua)

```
-- Ar-2T-chemical-reactions.lua
--
-- Original reaction rates from:
--
-- Hoffert, M.I. and Lien, H. (1967)
-- Quasi-one-dimensional, nonequilibrium gas dynamics of partially
-- ionized two-temperature Argon
-- Physics of Fluids, Volume 10 Number 8 pp 1769-1777 Aug. 1967
--
-- New cross-sections from:
--
-- Glass, I.I and Liu, W.S. (1978)
-- Effects of hydrogen impurities on shock structure and stability in
-- ionizing monatomic gases. Part 1. Argon
-- Journal of Fluid Mechanics, Vol. 84 Part 1 pp 55-77 1978
--
-- The presented rates are in the form:
--     k = A ( Ta / T + 2 ) exp( - Ta / T )
-- and have therefore been curve-fitted to the Generalized Arrhenius
-- form for numerical implementation.  The Argon impact reaction was
-- curve fitted in the temperature range of [10500,35000] and the
-- electron impact reaction was curve fitted in the temperature range of
-- [500,20000].  The respective maximum errors were 0.0259% and 0.0078%.
--
-- Author: Daniel F. Potter
-- Date: 18-Apr-2012
-- Place: DLR, Goettingen, Germany
--
-- History:
--   18-Apr-2012: - Initial implementation
--   01-Oct-2013: - Updated with better cross-sections from Glass and Liu
--   28-May-2014: - Aesthetic improvements

scheme_t = {
    update = "chemical kinetic ODE MC",
    temperature_limits = {
        lower = 20.0,
        upper = 100000.0
    },
    error_tolerance = 0.000001
}

-- Argon-impact ionization

Q_hoffert = 1.2e-19     -- cm2/eV
Q_glass   = 1.0e-19     -- cm2/eV
f         = Q_glass / Q_hoffert

reaction{
   'Ar + Ar <=> Ar+ + Ar + e-',
   fr={'Arrhenius', A=f*8.996906e06, n=1.004, T_a=129441.6 },
   ec={model='from thermo',iT=-1,species="Ar", mode="translation"}
}

-- Electron-impact ionization

Q_hoffert = 7.0e-18     -- cm2/eV
Q_glass   = 4.9e-18     -- cm2/eV
f         = Q_glass / Q_hoffert

reaction{
   'Ar + e- <=> Ar+ + e- + e-',
   fr={'Park', A=f*9.039202e11, n=0.867, T_a=132482.8,
       p_name='e_minus', p_mode='translation', s_p=1.0,
       q_name='NA', q_mode='NA'
   },
   chemistry_energy_coupling={
      {species='e_minus', mode='translation',
```

```
        model='electron impact ionization', T_I=181700.0}
    },
    ec={model='from thermo',iT=-1,species="e_minus", mode="translation"}
}
```

48.7 Thermal energy exchange script (.lua)

```
-- Ar-2T-energy-exchange.lua
--
-- Electron-translation thermal energy exchange for the Ar,Ar+,e- system
-- via the Appleton and Bray (1967) model. Heavy-particle excitation
-- cross sections have been curve fitted from the data presented in:
--
-- Hoffert, M.I. and Lien, H. (1967)
-- Quasi-one-dimensional, nonequilibrium gas dynamics of partially
-- ionized two-temperature Argon
-- Physics of Fluids, Volume 10 Number 8 pp 1769-1777 Aug. 1967
--
-- Author: Daniel F. Potter
-- Date: 18-Apr-2012
-- Place: DLR, Goettingen, Germany
--
-- History:
--   18-Apr-2012: - Initial implementation
--   28-May-2014: - Aesthetic improvements

mechanism{
    'e- ~~ Ar : E-T',
    rt={'Appleton-Bray:TwoRangeNeutral',
        T_switch=10000.0,
        sigma_low_T={  3.9e-21, -5.51e-25, 5.95e-29},
        sigma_high_T={-3.5e-21,  7.75e-25, 0.0}
    }
}

mechanism{
    'e- ~~ Ar+ : E-T',
    rt={'Appleton-Bray:Ion'}
}
```

48.8 Radiation model (for flowfield coupling) script (.py)

```
# 1. transport model
gdata.transport_model = "monte carlo"
gdata.nrays = 32768
gdata.clustering = "by area"
gdata.absorption = "partitioned energy"

# 2. spectral model
gdata.spectral_model = "photaura"
gdata.lambda_min = 1.0e7 / 150000.0
gdata.lambda_max = 1.0e7 / 1000.0
gdata.spectral_points = int ( ( 1.0e7 / gdata.lambda_min - 1.0e7 / gdata.lambda_max ) * 0.1 )
gdata.adaptive_spectral_grid = False

params = {
"species"               : [ 'Ar', 'Ar_plus', 'e_minus' ],
"radiators"             : [ 'Ar', 'Ar_plus', 'e_minus' ],
"QSS_radiators"         : [ 'Ar' ],
"no_emission_radiators" : [],
```

```
"iTe"                   : 1,
"atomic_level_source"   : "NIST_ASD",
"atomic_line_source"    : "NIST_ASD",
"atomic_PICS_source"    : "TOPBase",
"allow_inexact_Stark_matches" : True,
"require_PICS_term_match" : False
}

declare_radiators( params, gdata )
```

48.9 Radiation model (for experiment comparison) script (.py)

```
# 1. transport model
gdata.transport_model = "monte carlo"
gdata.nrays = 32768
gdata.clustering = "by area"
gdata.absorption = "partitioned energy"

# 2. spectral model
gdata.spectral_model = "photaura"
gdata.lambda_min = 180.
gdata.lambda_max = 6000.
gdata.spectral_points = int ( ( 1.0e7 / gdata.lambda_min - 1.0e7 / gdata.lambda_max ) * 0.1 )
gdata.adaptive_spectral_grid = False

params = {
"species"               : [ 'Ar', 'Ar_plus', 'e_minus' ],
"radiators"             : [ 'Ar', 'Ar_plus', 'e_minus' ],
"QSS_radiators"         : [ 'Ar' ],
"no_emission_radiators" : [],
"iTe"                   : 1,
"atomic_level_source"   : "NIST_ASD",
"atomic_line_source"    : "NIST_ASD",
"atomic_PICS_source"    : "TOPBase",
"allow_inexact_Stark_matches" : True,
"require_PICS_term_match" : False
}

declare_radiators( params, gdata )
```

48.10 Radiation error checking script (.py)

```
#!/usr/bin/env python

import sys

qrad = {}
qrad["measured"] = float(sys.argv[2])

ifile = open(sys.argv[1],"r")
lines = ifile.readlines()
ifile.close()

for line in lines:
    tks = line.split()
    if len(tks)==0: continue
    if tks[0]=="#": continue
    elif float(tks[0])==0.0:
        qrad["calculated"] = float(tks[3])
        break
```

```
error = abs(qrad["measured"] - qrad["calculated"])/qrad["measured"] * 100.0

print "qrad error = %0.1f percent" % ( error )
```

48.11 Notes

- The radiation-flowfield coupling is relatively weak for this case, and therefore the flowfield solution with and without radiation are relatively similar. For cases where radiation-flowfield coupling is stronger, difficulties may arise when the scaling of the radiation source term is turned on.

- The radiation portion of this simulation can be run in parallel on a shared memory computer. See http://cfcfd.mechmining.uq.edu.au/eilmer3.html for intructions on how to compile `e3rad.exe` for parallel computations.

49 Microscale combustion

Here is an example of a reacting flow at submillimeter scale. The gas mixture of stoichiometric methane/air is fed to a 2-D micro-channel at the dimension of 5 mm (in length) × 0.6 mm (in height). However, using a symmetry assumption the computational domain is only 0.3 mm in height. The reaction mechanism of 19-species and 84-reaction methane/air chemistry (DRM19) [47] is used in the simulation. The method of "IgnitionZone" is switched on for the initial 1 ms in order to trigger the combustion and then switched off subsequently. Figure 126 shows the computational domain.

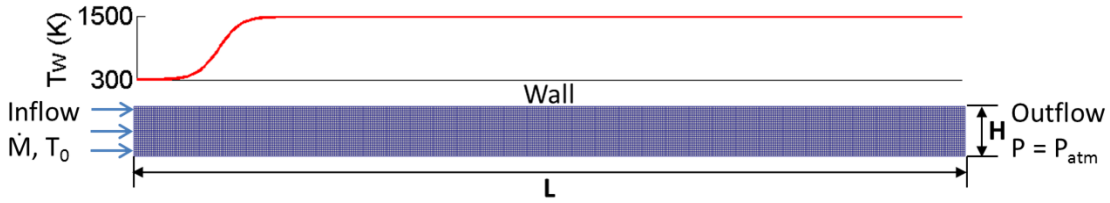


Figure 126: Computational domain of the planar micro-channel and boundary conditions used.

49.1 Input script (.py)

```
# Micro-combustion
# 5 mm x 0.3 mm channel
# methane/air, V = 40 cm/s, Phi = 1.0
# Xin Kang

gdata.title = "full channel simulation"
gdata.dimensions  = 2
gdata.axisymmetric_flag = 0
gdata.viscous_flag = 1


# Gas Model Set-up

select_gas_model(fname='thermally-perfect-drm19.lua')
set_reaction_scheme("drm19.lua",reacting_flag =1)
gdata.diffusion_model = 'ConstantLewisNumber'
gdata.diffusion_lewis_number = 1.0
gdata.diffusion_flag = 1


# Flow Conditions

molef = {'N2':7.52, 'O2':2.0, 'CH4':1.0}
gmodel = get_gas_model_ptr()
massf=gmodel.to_massf(molef)
P_exit = 1.01325e5 #Pa, 1 atm
T0 = 300.0 #K, total inlet stagnation temperature
u0 = 0.4 #m/s, expected combustor inlet velocity
inflow = FlowCondition(p=P_exit,T=T0,u=u0,v=0,massf=massf)


# Geometry
```

```
L = 5.0e-3 #m, length of the channel
h = 0.3e-3 #m, full channel simulation

a = Node(0.0,0.0)
b = Node(0.0,h)
c = Node(L,0.0)
d = Node(L,h)

ab = Line(a,b)
ac = Line(a,c)
cd = Line(c,d)
bd = Line(b,d)


# Block Configuration
# Ensure only one blocks along y axis to facilitate udf lua function

nxcells0 = 390
nycells0 = 23
nbi0 = 64
nbj0 = 1


blk0 = SuperBlock2D(make_patch(bd, cd, ac, ab),
                    nni = nxcells0, nnj = nycells0, nbi=nbi0, nbj=nbj0,
                    bc_list = [UserDefinedBC(filename="udf-wall.lua", is_wall=1),\
                               FixedPOutBC(P_exit),\
                               UserDefinedBC(filename="udf-wall.lua", is_wall=1),\
                               UserDefinedBC(filename="udf-massflux-in.lua")],
                    fill_condition = inflow,
                    cf_list = [None, None, None, None])

# Make Block connections
identify_block_connections()


# IgnitionZone

point0 = Vector3(0.5*L, 0.0)
point1 = Vector3(point0.x + 0.05*L, h)
IgnitionZone(2000.0, point0, point1)
gdata.ignition_time_stop = 1.0e-3 # s


# History locations

HistoryLocation(0.0, 0.0)
HistoryLocation(0.0, h/2.0)
HistoryLocation(0.0, h)
HistoryLocation(L/2.0, 0.0)
HistoryLocation(L/2.0, h/2.0)
HistoryLocation(L/2.0, h)
HistoryLocation(L, 0.0)
HistoryLocation(L, h/2.0)
HistoryLocation(L, h)


# Simulation Parameters

gdata.flux_calc = AUSM_PLUS_UP
gdata.gasdynamic_update_scheme = "classic-rk3"
gdata.cfl = 0.3
gdata.max_time = 60.0e-3  # seconds
gdata.max_step = 20000000
gdata.dt = 3.0e-11
gdata.dt_plot = gdata.max_time/600.0
gdata.dt_history = 1.0e-7
```

49.2 UDF Boundary conditions

At the inlet of the channel, an inflow boundary condition based on the characteristic wave relations [48] is employed. It is capable of absorbing acoustic waves due to the chemical heat release and heat exchange between the flow and the wall. The gas total temperature (T_0), mass flow rate ($\dot{M}$) and incoming species mass fractions are specified.

```
-- udf-massflux-in.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedGhostCell BC.


function ghost_cell(args)
   -- Function that returns the flow states for a ghost cells.
   -- For use in the inviscid flux calculations.

--------------------------!!!Input parameters-----------------------------------
   dt_plot = 1e-4 -- timestep for saving the ghost cell information, s
   mass = 1.122497365547*0.4 -- mass flow rate, kg/s/m2
   T0 = 300 --total temperature, K
   massf = {}
   for isp=0,(nsp-1) do
      massf[isp] = 0.000000e+00
   end
   massf[3] = 2.201527e-01  --O2
   massf[10] = 5.518596e-02 --CH4
   massf[19] = 7.246613e-01 --N2
   -- mass fractions are indexed from 0 to nsp-1
-------------------------------------------------------------------------------

------------------for the very first timestep, set values in the ghost cell------
   if (args.t == 0) then
      cell0 = sample_flow(block_id, args.i, args.j, args.k)
      filename = "update-"..string.format("%04d", args.j)..".data"
      file = io.open(filename, "w")
      file:write(dt_plot,"\t",cell0.u,"\t",cell0.p,"\t",cell0.T[0],"\n")
      file:close()
   end

   if (args.t_step == 0 and args.j == 2) then
----- initialize the data records in each ghost cell along j axis
      step_previous = {}
      dt_save = {}
      update_u = {}
      update_p = {}
      update_T = {}
   end

   if (args.t_step == 0) then
      step_previous[args.j] = args.t_step
      filename = "update-"..string.format("%04d", args.j)..".data"
      file = io.open(filename, "r")
      dt_save[args.j], update_u[args.j], update_p[args.j], update_T[args.j] \
      = file:read("*number","*number","*number","*number")
      file:close()
   end
-------------------------------------------------------------------------------

   Q = create_empty_gas_table()
   Q.p = update_p[args.j]
   Q.T[0] = update_T[args.j]
   for isp=0,(nsp-1) do
      Q.massf[isp] = massf[isp]
   end
   eval_thermo_state_pT(Q)
   eval_sound_speed(Q)
   a = Q.a
   Cp = eval_Cp(Q)
```

```
   gamma = eval_gamma(Q)
   R = eval_R(Q)
   rho = Q.rho
   u = update_u[args.j]
   p = update_p[args.j]
   M = math.abs(u/a)
   -- Sample the flow field from the inner cells near the boundary.
   cell1 = sample_flow(block_id, args.i, args.j, args.k)
   x1 = cell1.x
   u1 = cell1.u
   p1 = cell1.p
   cell2 = sample_flow(block_id, args.i+1, args.j, args.k)
   x2 = cell2.x
   u2 = cell2.u
   p2 = cell2.p
   cell3 = sample_flow(block_id, args.i+2, args.j, args.k)
   x3 = cell3.x
   u3 = cell3.u
   p3 = cell3.p
   cell4 = sample_flow(block_id, args.i+3, args.j, args.k)
   x4 = cell4.x
   u4 = cell4.u
   p4 = cell4.p


   if (args.t_step ~= step_previous[args.j]) then
      -- NSCBC Wave amplitude and LODI relations by T.J.Poinsot:
      dpdx = (-25/12*p+4*p1-3*p2+4/3*p3-1/4*p4)/(x2-x1)
      dudx = (-25/12*u+4*u1-3*u2+4/3*u3-1/4*u4)/(x2-x1)
      L1 = (u-a)*(dpdx-rho*a*dudx) -- sound wave at speed u-c
      L2 = (1-M)/(M+1/(gamma-1))*L1 -- entropy wave at speed u
      L5 = (M-1)*(M*(gamma-1)-1)/(M+1)/(M*(gamma-1)+1)*L1
      -- sound wave at speed u+c

      -- As total temperature and mass flow rate are specified,
      -- only continuity equation needs to be solved on the boundary:
      d1 = 1/a/a*(L2+0.5*(L1+L5))
      rho = rho-d1*args.dt
      step_previous[args.j] = args.t_step
   end

   update_u[args.j] = mass/rho
   update_T[args.j] = T0-0.5/Cp*update_u[args.j]*update_u[args.j]
   update_p[args.j] = rho*R*update_T[args.j]


   -- update ghost cells
   ghost = {}
   ghost.T = {}  -- temperatures, K (as a table)
   ghost.T[0] = update_T[args.j]
   ghost.u = update_u[args.j] -- x-velocity, m/s
   ghost.v = 0.0 -- y-velocity, m/s
   ghost.w = 0.0 -- z-velocity, m/s
   ghost.p = update_p[args.j] -- pressure, Pa
   ghost.massf = massf -- mass fractions


------------------save ghost cell information every dt_plot----------------------
   if (args.t >= dt_save[args.j]) then
      dt_save[args.j] = dt_save[args.j] + dt_plot
      filename = "update-"..string.format("%04d", args.j)..".data"
      file = io.open(filename, "w")
      file:write(dt_save[args.j],"\t",update_u[args.j],"\t",update_p[args.j],\
      "\t",update_T[args.j],"\n")
      file:close()

      filename1 = "data-records.data"
      file = io.open(filename1, "a")
      file:write(args.t_step,"\t",args.t_level,"\t",args.dt,"\t",d1,"\t",rho,\
      "\t",ghost.u,"\t",ghost.p,"\t",ghost.T[0],"\n")
      file:close()
   end
--------------------------------------------------------------------------------
```

```
   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
```

At the wall of the channel, a hyperbolic tangent temperature profile is prescribed. The temperature ramps from 300 K to 1500 K over the initial 1 mm of the channel length and maintains at 1500K for the rest length of the combustor.

```
-- udf-wall.lua
-- Lua script for the user-defined functions
-- called by the UserDefinedBC boundary condition.


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 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
   Tleft = 300
   Tright = 1500
   cell = sample_flow(block_id, args.i, args.j, args.k)
   cell.u, cell.v = 0
   cell.T = {}  -- temperatures, K (as a table)
   x = args.x
   if (x >= 0.0 and x <= 1e-3) then
   cell.T[0] = ((Tright-Tleft)*(1-math.exp(-1e4*(x-0.5e-3)))\
```

```
   /(1+math.exp(-1e4*(x-0.5e-3)))+(Tright+Tleft))/2 --hypertangent profile
   else
   cell.T[0] = Tright
   end
   return cell
end
```

49.3 Running the simulation

This simulation is running on Barrine cluster (the High-Performance Computing Unit at UQ) using 64 processors, with a shell script:

```
#!/bin/bash -l
#PBS -S /bin/bash
#PBS -N micro-combustion
#PBS -q workq
#PBS -l select=16:ncpus=4:NodeType=medium:mpiprocs=4
#PBS -A uq-MechMinEng
#PBS -l walltime=48:00:00
echo "-----------------------------------------"
echo "Begin MPI job..."
date
cd $PBS_O_WORKDIR
mpirun -np 64 e3mpi.exe --job=microchannel --run > LOGFILE
echo "End MPI job."
date
```

This heavy task will take approximately 10 days wall clock time to reach a stable solution.

49.4 Results

Figure 127 plots the temporal evolution of CH_3 radicals which are responsible for the establishment of the flame front. In the initial 0.1 ms, CH_3 radicals are generated and accumulated near the walls. Then the flame is established and bifurcated into two branches: The main flame propagates upstream and finally gets stablized, while the bifurcated flame propagtes downstream and then flows out of the domain. After around 4 ms, a stable solution is obtained.

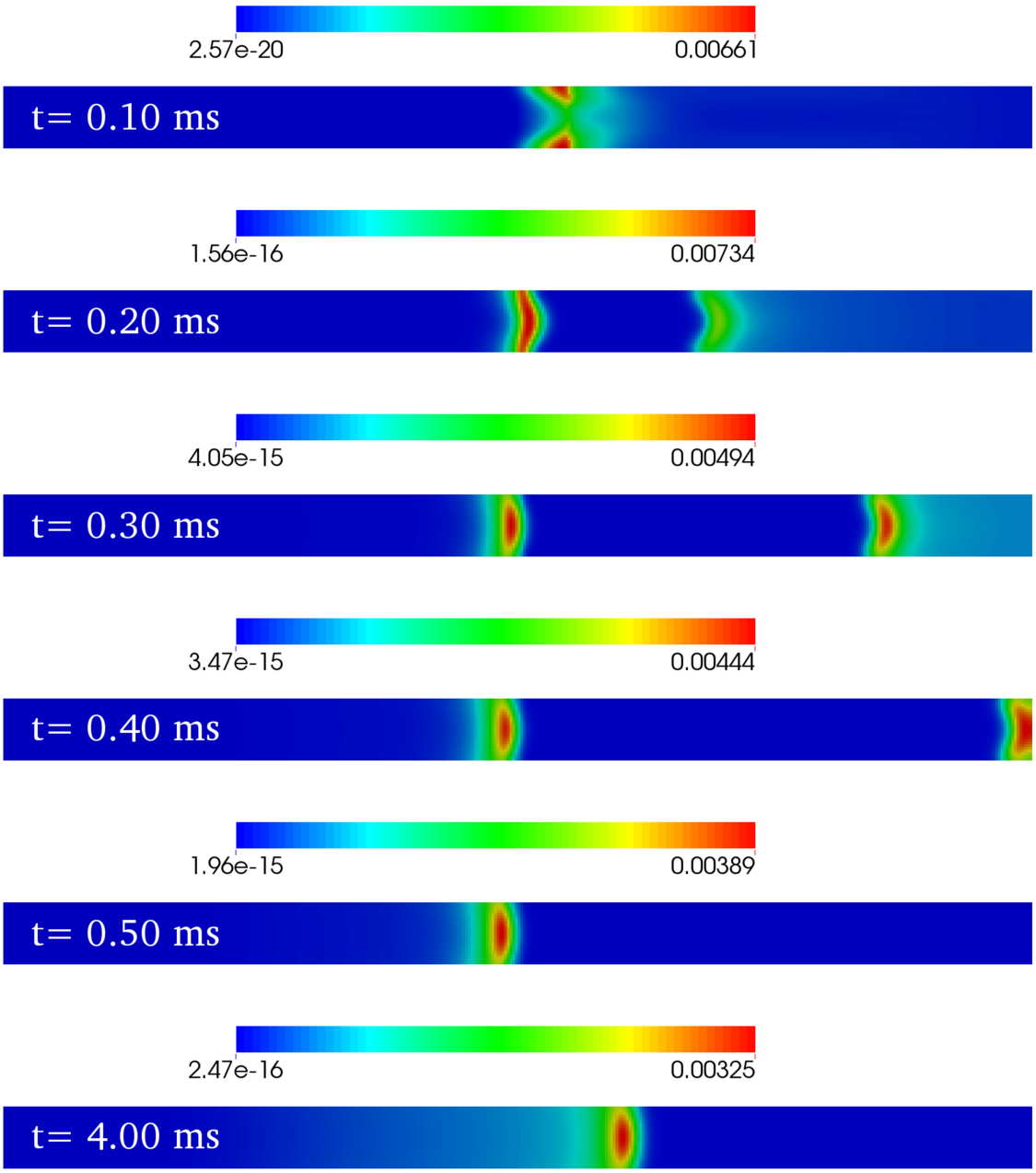


Figure 127: Temporal evolution of CH_3 radical concentrations.

Part V
Examples for 3D flow

50 Mach 1.5 flow over a 10-degree 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 ~/cfcfd3/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 50.2 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 128 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 [10] 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. [11], the expected steady-state shock wave angle is 57^o

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[22], one may view the transient development of the planar shock travelling over the ramp and establishing the steady-flow oblique shock seen in Fig. 128. Starting with VTK parallel file `simple_ramp.t0000.pvtu`, ParaView understands the time-stamp sequence numbering of the VTK output 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 50.3, 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

[22]The Parallel Visualization Application (http://www.paraview.org) developed by Kitware (http://www.kitware.com) is freely available for download.

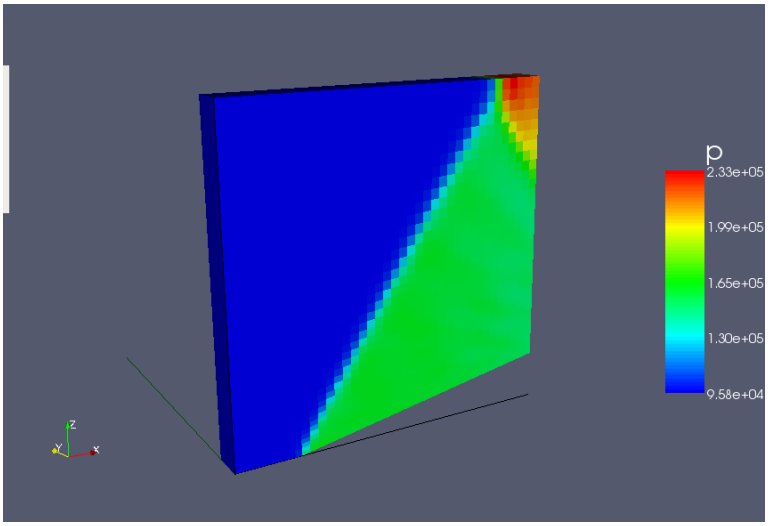


Figure 128: Filled surface representation of the cells Colours representing pressure at $t = 5.0\,\text{ms}$. The 10^o slope on the ramp is seen running up to the right. Note that the shock propagating from the start of the ramp is nearly straight until it approaches the top surface of the simulation domain where it is reflected. This PDF figure was generated with Paraview from the final solution file.

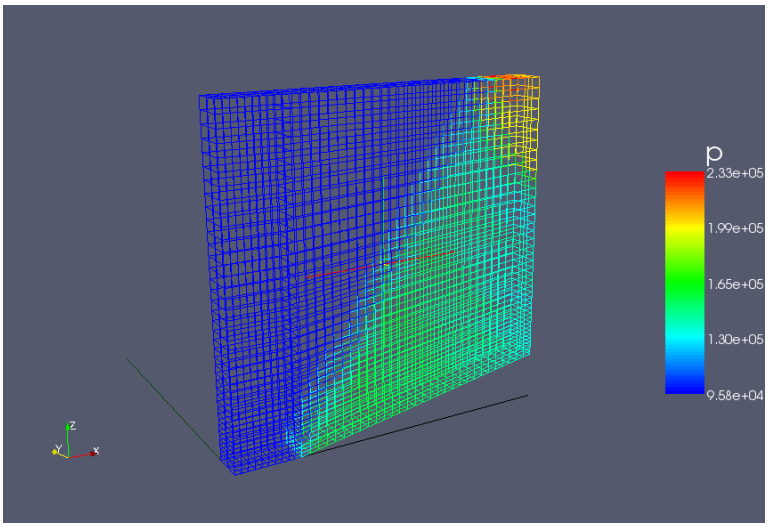


Figure 129: Wireframe representation of the cells on the outer surfaces of the blocks. The grid is coloured, representing pressure at $t = 5.0\,\text{ms}$.

for any particular block and makes the data available as a multidimensional array. The `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.

50.1 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
#     Nov 2013 Test manual block connection with flow vector reorientation
# ----------------------------------------------------------------------

# ---------------- 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 10 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()
# Let's manually connect and exercise the flow reorientation code.
connect_blocks_3D(blk0, blk1, [(1,0),(5,4),(6,7),(2,3)],
                  reorient_vector_quantities=True,
                  nA=[1.0,0.0,0.0], t1A=[0.0,1.0,0.0],
                  nB=[1.0,0.0,0.0], t1B=[0.0,1.0,0.0])
```

50.2 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
```

50.3 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."
```

50.4 Notes

- None.

51 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 ~/cfcfd3/examples/eilmer3/3D/sod/
$ ./sod_run_and_plot.sh
```

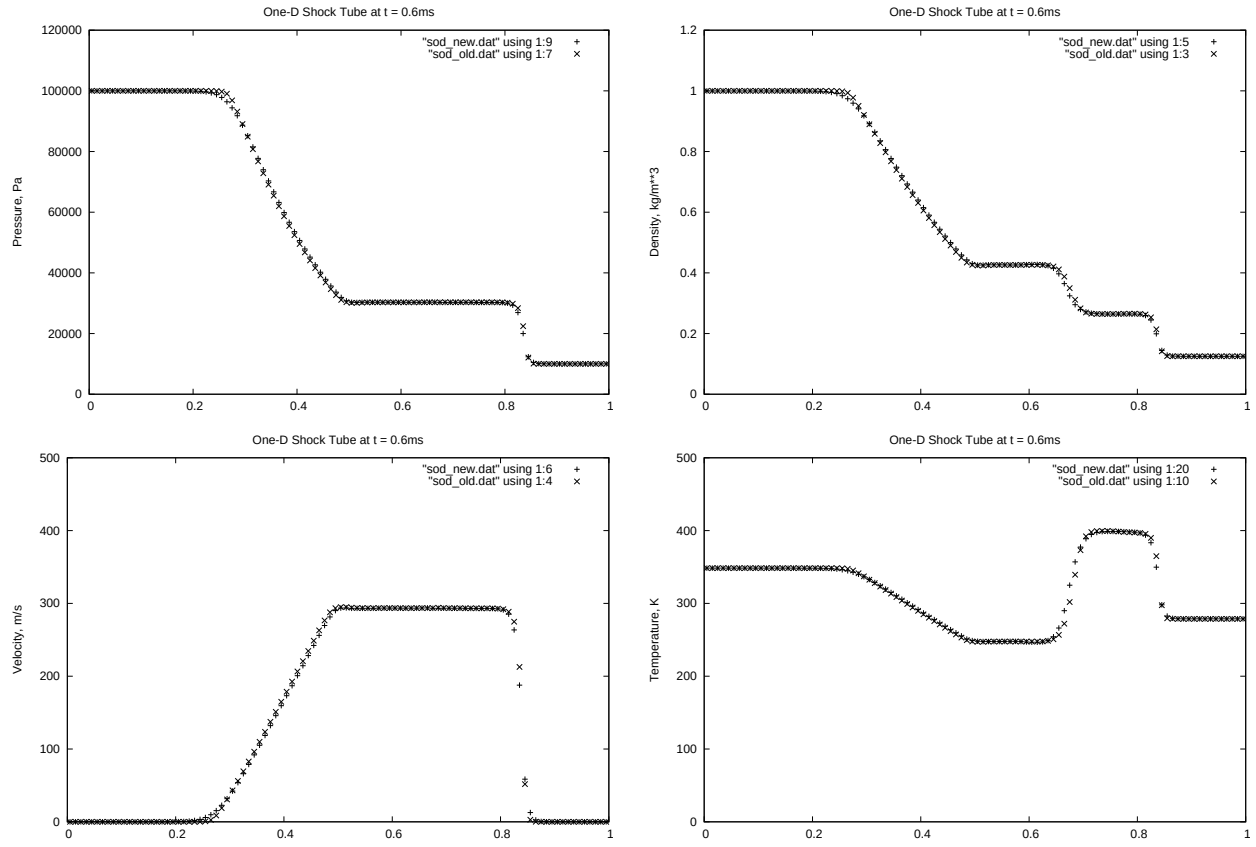


Figure 130: Flow properties along the duct for the Sod shock tube problem.

51.1 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
```

51.2 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 ps 1 pt 1, \
     "sod_old.dat" using 1:7 with points ps 1 pt 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 ps 1 pt 1, \
     "sod_old.dat" using 1:3 with points ps 1 pt 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 ps 1 pt 1, \
     "sod_old.dat" using 1:4 with points ps 1 pt 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 ps 1 pt 1, \
     "sod_old.dat" using 1:10 with points ps 1 pt 2
EOF
```

51.3 Notes

- None

52 Injection of hydrogen into a nitrogen stream

Figure 131 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.

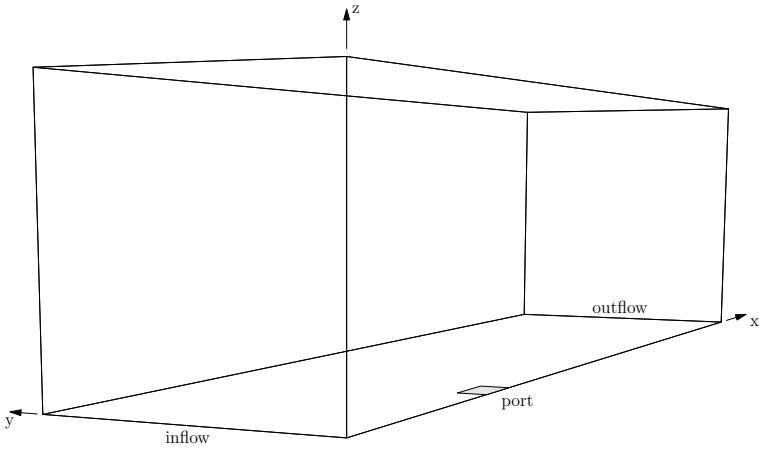


Figure 131: Wireframe representation of the duct with the hydrogen injection port shaded on the bottom surface. The main stream flow is from the closest (West) boundary to the furtherest (East) boundary.

The flow domain consists of 6 blocks filling the whole flow domain as shown in Figure 132. 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 133 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.

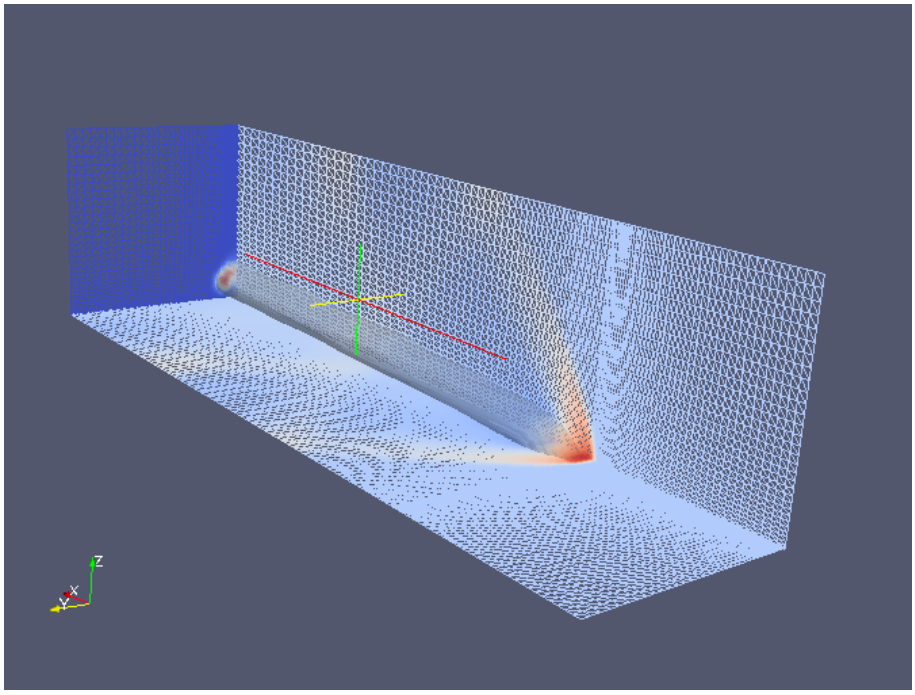


Figure 132: Wireframe representation of the surface grid on three surfaces, coloured by pressure on the bottom and symmetry surfaces and coloured by mass fraction on the outflow surface. Also plotted is a contour surface for a mass-fraction of hydrogen with a value 0.1, coloured by pressure and made partially transparent so that the planar surfaces show through.

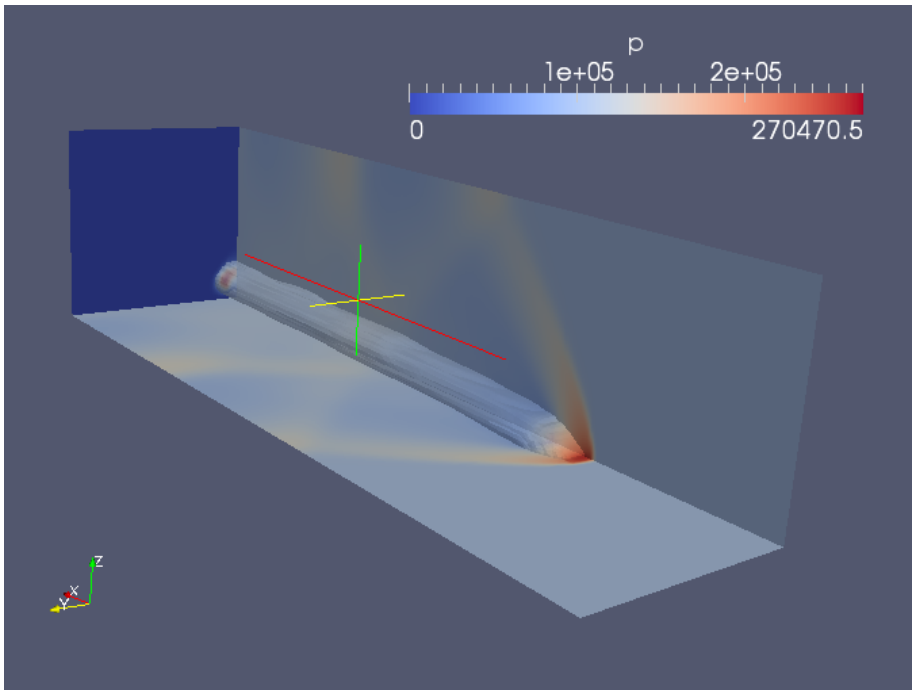


Figure 133: Filled surface representation of pressure and mass-fraction for hydrogen at the final time.

52.1 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")
```

52.2 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
```

52.3 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.

53 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 thermochemistry regions in the flow field and temperatures that rise above 20 000 K. Two simulations of the cylinder flow are presented: the first with chemical nonequilibrium and thermal equilibrium, and the second with both chemical and thermal nonequilibrium.

The flow domain shown is made up of 4 block-structured grids as shown in Figure 134 and number of the surface grids are are indicated in Figure 135 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.

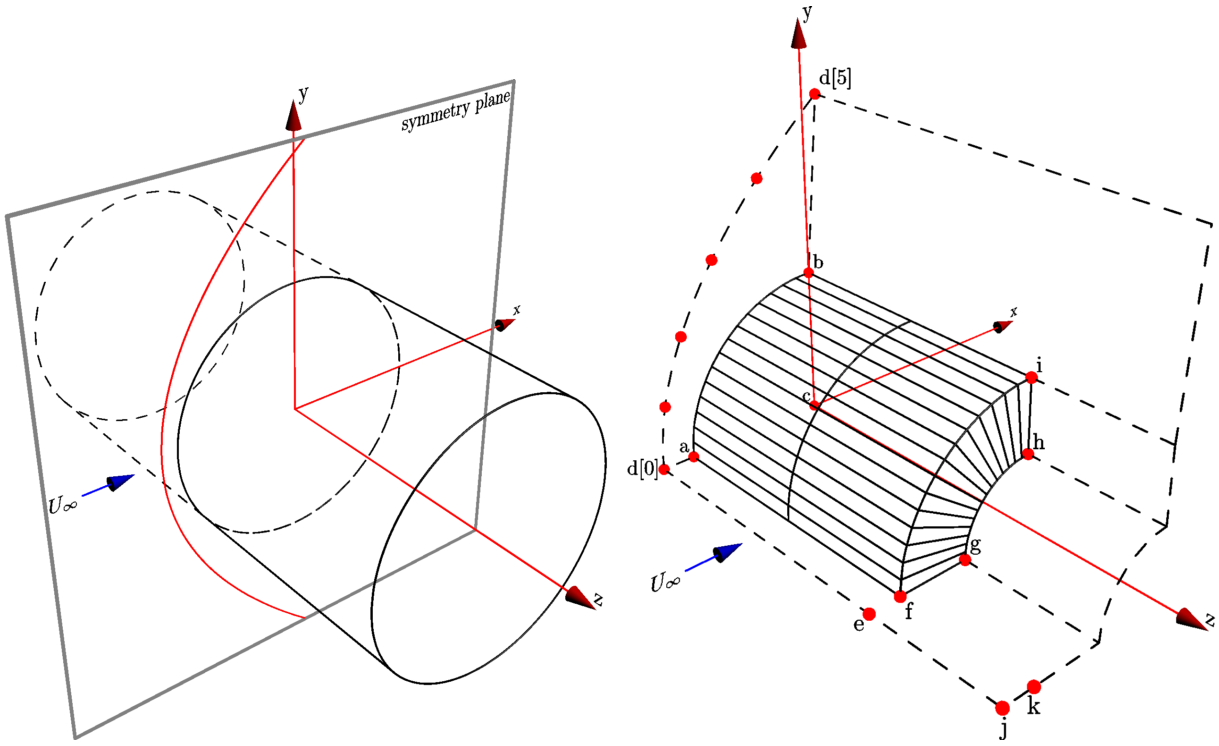


Figure 134: Left: full cylinder with the expected shock location scribed on the symmetry plane. Right: layout of finite-cylinder simulation with one-quarter of forward-facing half of the cylinder surface shown as wire-frame. Some of the edges of the flow domain are shown dashed and the labelled nodes 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.

53.1 Chemical nonequilibrium and thermal equilibrium

Here we describe the finite-cylinder simulations with chemical nonequilibrium and thermal equilibrium. This means chemical reactions are permitted to occur at a finite-rate (chemical nonequilibrium), but all thermal modes are assumed to be governed by a single temperature (thermal equilibrium).

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 136). 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.

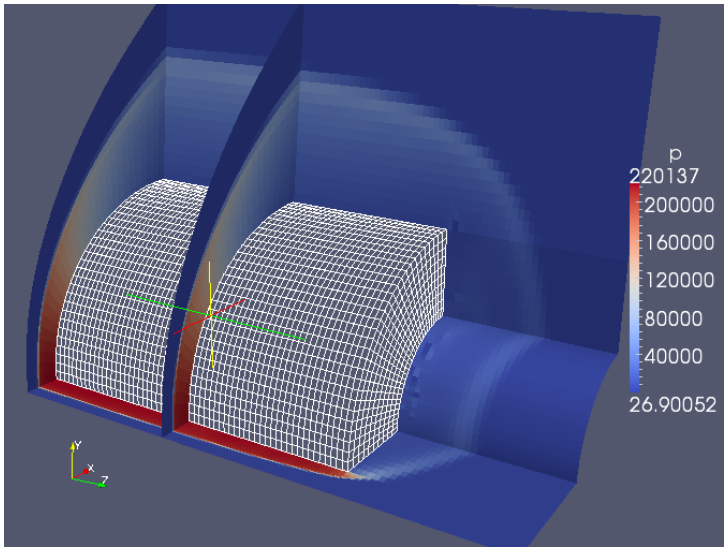


Figure 135: A selection of surface grids from the finite-cylinder simulation with chemical nonequilibrium, shown as wire-frame on the cylinder surface and coloured by pressure in the flow field. This PNG figure was generated with Paraview using block surfaces extracted from final solution file.

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.

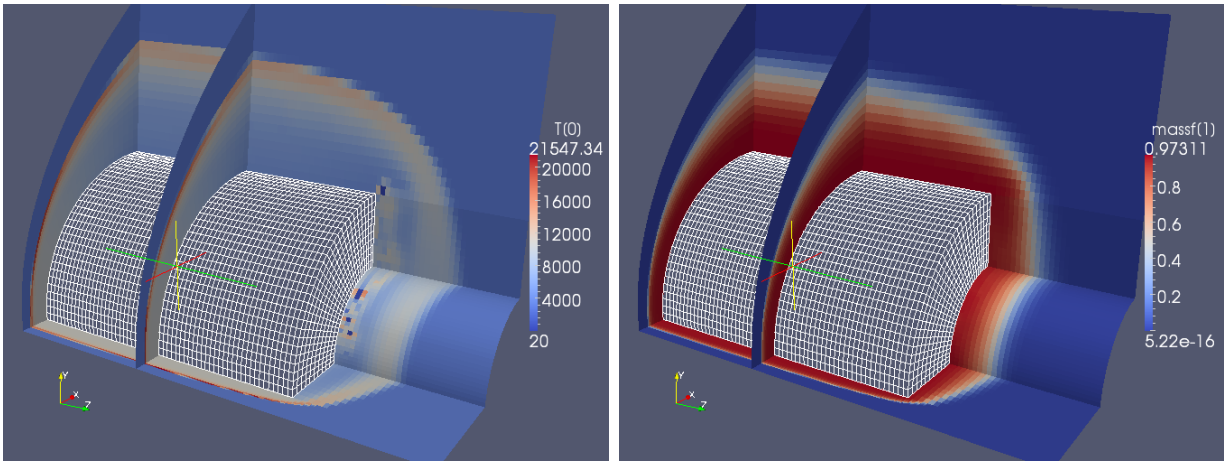


Figure 136: Static temperature and mass fraction of nitrogen atoms in the flow field from the chemical nonequilibrium simulation.

53.1.1 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
```

53.1.2 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}
}
```

53.1.3 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
```

53.1.4 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."
```

53.1.5 Notes

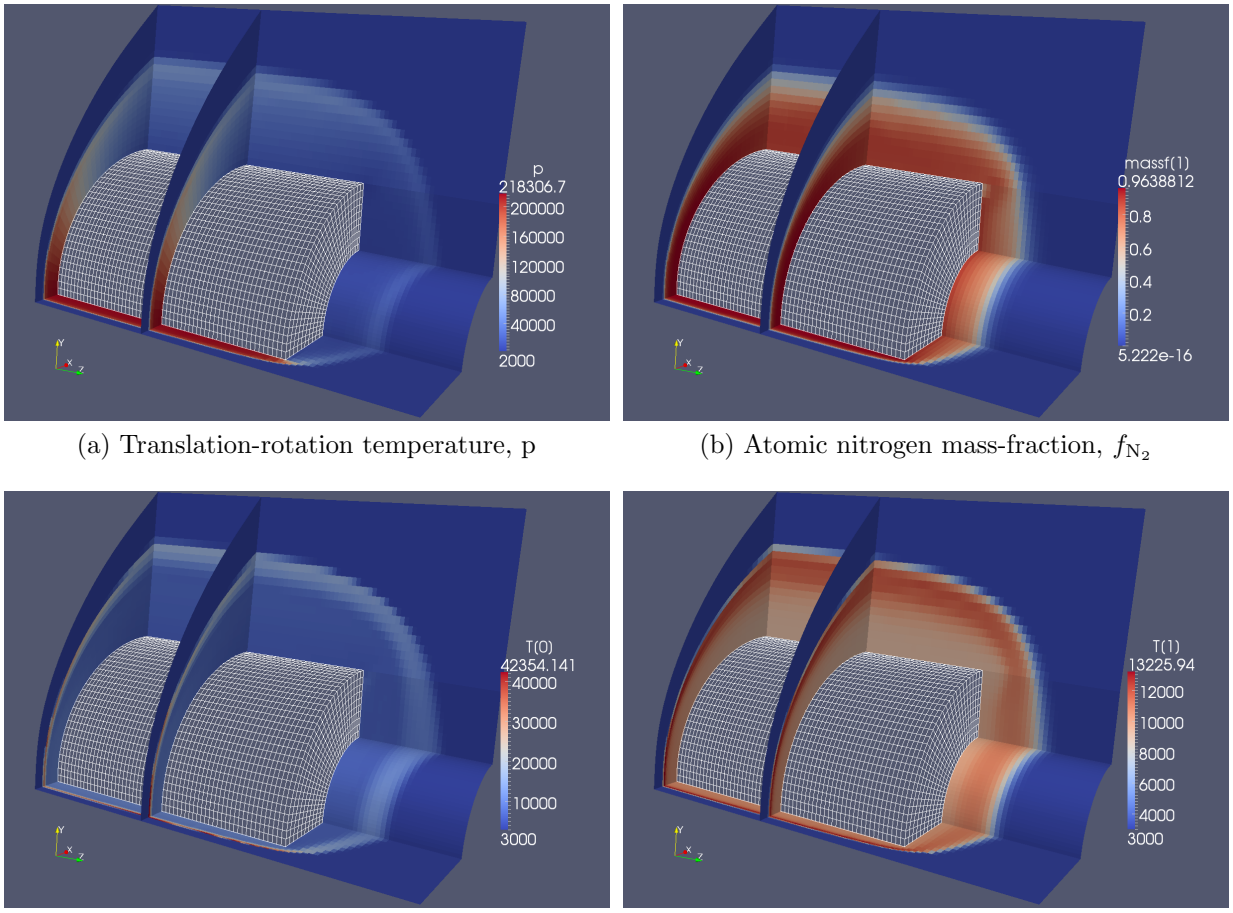
- It is well worth the bother to run this simulation on multiple processors. The elapsed time for the run with 4 MPI processes is 9193 seconds on geyser, a Dell server with 4×4 AMD cores. Of course, our MPI job used only 4 of those cores.

- 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.

53.2 Chemical and thermal nonequilibrium

Here we describe the finite-cylinder simulations with both chemical and thermal nonequilibrium. Specifically, a two-temperature thermal model as proposed by Park [49] is implemented. This means chemical reactions are permitted to occur at a finite-rate (chemical nonequilibrium), and the translation and rotation thermal modes are governed by one temperature T_{tr} and the vibration and electronic thermal modes by a separate temperature T_{ve}



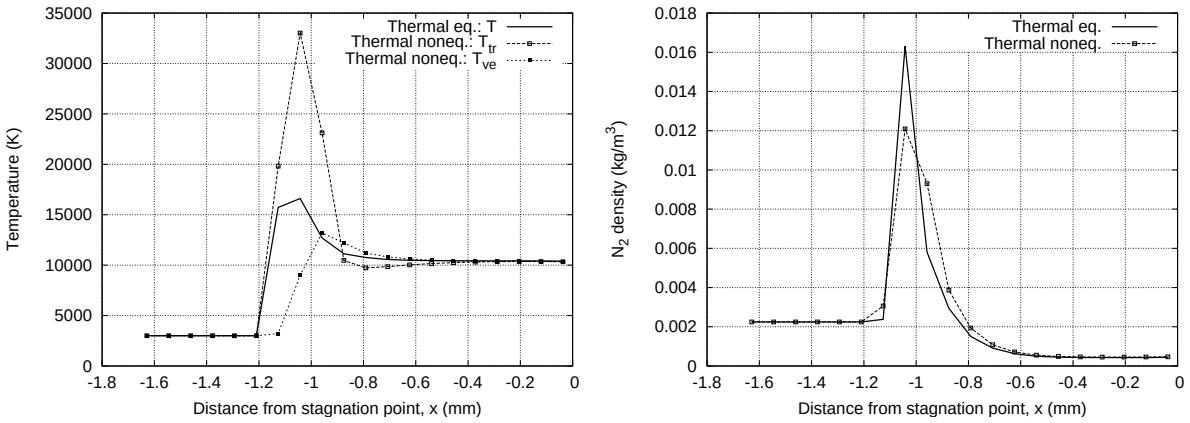
(a) Translation-rotation temperature, p

(b) Atomic nitrogen mass-fraction, f_{N_2}

(c) Translation-rotation temperature, T_{tr}

(d) Vibration-electron-electronic temperature, T_{ve}

Figure 137: Flow field contour plots from the thermal nonequilibrium simulation.



(a) Temperature profile

(b) Diatomic nitrogren density profile

Figure 138: Stagnation-line profile plots from the thermal nonequilibrium simulation.

53.2.1 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.
# DFP, 08-Dec-2011 port to thermal nonequilibrium

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='two temperature gas', species=['N2','N','N2_plus','N_plus','e_minus'])
set_reaction_scheme("nitrogen-5sp-6r.lua",reacting_flag=1)
set_energy_exchange_scheme("TV-TE_exchange.lua")
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/10
gdata.print_count = 1

print "Total number of blocks=", len(blk0.blks)+2
```

53.2.2 Reaction scheme file (.lua)

```
-- Author: Daniel F. Potter
-- Date: 29-Nov-2011
-- Place: DLR G ttingen , Germany
--
-- Two-temperature version of the ionising nitrogen
-- reaction scheme.
--
-- 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
--

scheme_t = {
    update = "chemical kinetic ODE MC",
    temperature_limits = {
        lower = 20.0,
        upper = 100000.0
    },
    error_tolerance = 0.000001
}

reaction{
   'N2 + N2 <=> N + N + N2',
   fr={'Park', A=7.0e21, n=-1.6, T_a=113200.0, p_name='N2', p_mode='vibration',
       s_p=0.3, q_name='N2', q_mode='translation'},
   ec={model='from CEA curves',iT=0}
}

reaction{
   'N2 + N <=> N + N + N',
   fr={'Park', A=3.0e22, n=-1.6, T_a=113200.0, p_name='N2', p_mode='vibration',
       s_p=0.3, q_name='N', q_mode='translation'},
   ec={model='from CEA curves',iT=0}
}

reaction{
   'N2 + e- <=> N + N + e-',
   fr={'Park', A=3.0e24, n=-1.6, T_a=113200.0, p_name='N2', p_mode='vibration',
       s_p=0.3, q_name='e_minus', q_mode='translation'},
   ec={model='from CEA curves',iT=0}
}

reaction{
   'N + N <=> N2+ + e-',
   fr={'Arrhenius', A=4.40e7, n=1.5, T_a=67500.0},
   ec={model='from CEA curves',iT=0}
}

reaction{
   'N2 + N+ <=> N2+ + N',
```

```
    fr={'Arrhenius', A=1.0e12, n=0.5, T_a=12200.0},
    ec={model='from CEA curves',iT=0}
}

reaction{
    'N + e- <=> N+ + e- + e-',
    fr={'Park', A=2.50e34, n=-3.82, T_a=168600.0, p_name='e_minus',
        p_mode='translation', s_p=1.0, q_name='NA', q_mode='NA'},
    ec={model='from CEA curves', iT=-1, species='e_minus', mode='translation'}
}
```

53.2.3 Energy exchange scheme file (.lua)

```
scheme_t = {
    update = "energy exchange ODE",
    temperature_limits = {
        lower = 20.0,
        upper = 100000.0
    },
    error_tolerance = 0.000001
}

ode_t = {
    step_routine = 'rkf',
    max_step_attempts = 4,
    max_increase_factor = 1.15,
    max_decrease_factor = 0.01,
    decrease_factor = 0.333
}

-- all VT exchange mechanisms identified by Park (1993)
-- all ET exchange mechanisms from Gnoffo (1989)

rates = {
    {
        mechanisms = {
            {
                type = 'VT_exchange',
                p_name = 'N2',
                relaxation_time = {
                    type = 'VT_MillikanWhite_HTC',
                    HTCS_model = {
                        type = 'Park',
                        sigma_dash = 3.0e-17
                    },
                    p_name = 'N2',
                    q_names = { 'N2', 'N' },
                    a_values = {  -1,   -1 },
                    b_values = {  -1,   -1 }
                }
            },
            {
                type = 'ET_exchange',
                relaxation_time = {
                    type = 'ET_AppletonBray',
                    ions = {
                        { c_name = 'N_plus', },
                    },
                    neutrals = {
                        { c_name = 'N', sigma_coefficients = { 5.0e-20, 0.0, 0.0 } },
                        { c_name = 'N2', sigma_coefficients = { 7.5e-20, 5.5e-24, -1.0e-28 } },
                    }
                }
            }
        }
    }
}
```

```
equilibriation_mechanisms = {}
```

53.2.4 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
```

53.2.5 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

# Plot temperature and N2 density profiles along the stagnation-line
# NOTE: thermal equilibrium solution needs to be present
gnuplot <<EOF
set term postscript eps enhanced "Helvetica" 20
set output "temperature_profiles.eps"
set size 1.0,1.0
set ylabel "Temperature (K)"
set xlabel "Distance from stagnation point, x (mm)"
set grid
set key top right
plot '../thermal-eq/stagnation-line.data' u (\$1*1000+7.5):25 w l lt 1 lw 3 t "Thermal eq.: T", \
     'stagnation-line.data' u (\$1*1000+7.5):25 w lp lt 2 lw 2 pt 4 ps 0.7 t "Thermal noneq.: T_{tr}", \
     'stagnation-line.data' u (\$1*1000+7.5):27 w lp lt 3 lw 2 pt 5 ps 0.7 t "Thermal noneq.: T_{ve}"

set output "N2_profiles.eps"
set key top right
set ylabel "N_2 density (kg/m^3)"
plot '../thermal-eq/stagnation-line.data' u (\$1*1000+7.5):(\$5*\$18) w l lt 1 lw 3 t "Thermal eq.", \
     'stagnation-line.data' u (\$1*1000+7.5):(\$5*\$18) w lp lt 2 lw 2 pt 4 ps 0.7 t "Thermal noneq."
EOF
epstopdf temperature_profiles.eps
epstopdf N2_profiles.eps
```

```
#! /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."
```

53.2.6 Notes

- The elapsed time for this simulation was 12895 seconds on 4 CPU's of the barrine cluster. On the same hardware the thermal equilibrium version of this simulation took 4130 seconds — a three-fold increase in computation time. This is to be expected as the implementation of a thermal nonequilibrium model introduces an additional conserved quantity to be accounted for (vibration-electron-electronic energy), and requires the ODE system for thermal energy exchange to be solved.

54 Spherically-blunted cone

An aeroshell-type model is shown in Figure 139. 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.

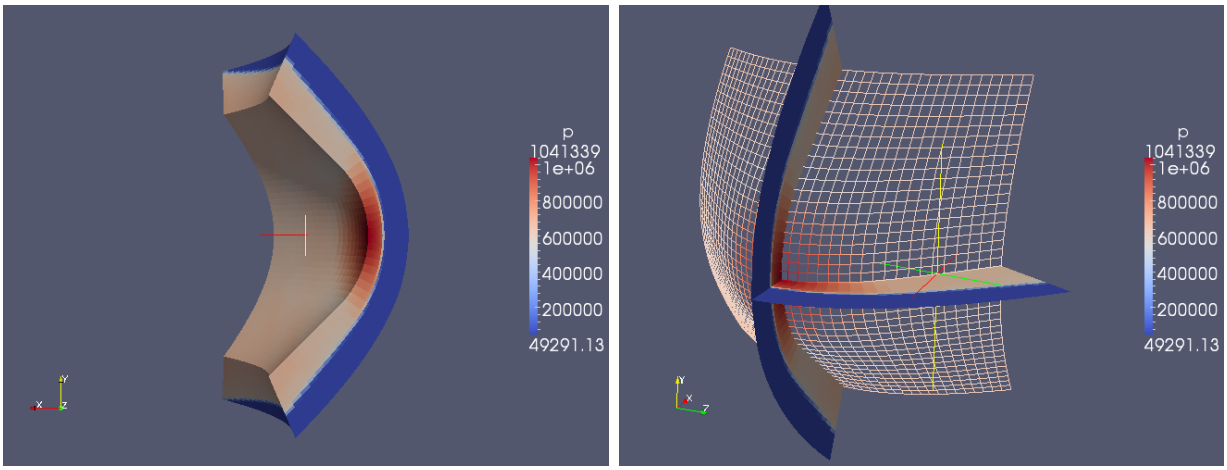


Figure 139: Views of the pressure field around a spherically-blunted cone. The left figure is the cell-average data for the entire block rendered as a coloured surface. The view is from behind the aeroshell surface. Only one half of the `RevolvedSurface` was used in the simulation. The right figure shows 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 relatively 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.

54.1 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")
```

55 Katsu's scramjet combustor and nozzle

The core of Katsuyoshi Tanimizu's scramjet model is shown in Figure 140. It was designed in 1994 by Prof. Ray Stalker and consists of 6 scramjet ducts distributed around a centrebody.

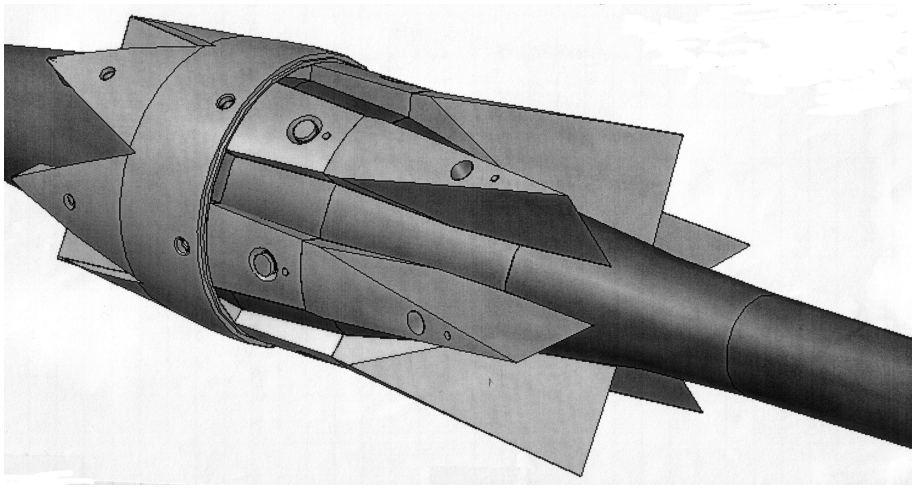


Figure 140: The core section of the scramjet model with the cowl removed from the combustor and nozzle sections to show the individual scramjet ducts between the dividing walls. The inlets are at the upper-left of the image. This image was scanned from a document provided by Katsu.

The geometry for only half of one duct is set up in the job script. We use a TriangePatch 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 pressure distribution across some of the surfaces is shown in Figure 142

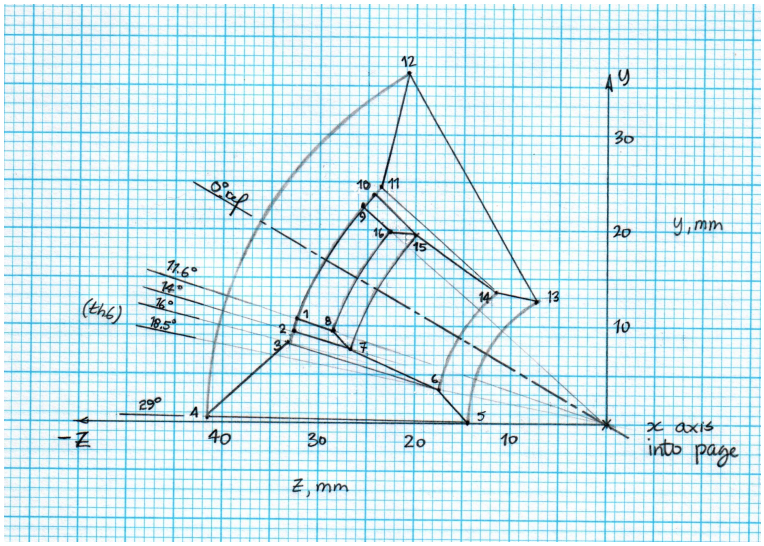


Figure 141: Front-view of the wireframe representation of one scramjet duct. The labelling of key points corresponds to that used in the job input file with the exception that points 9 through 16 were moved to the centre-plane of the duct (south surface).

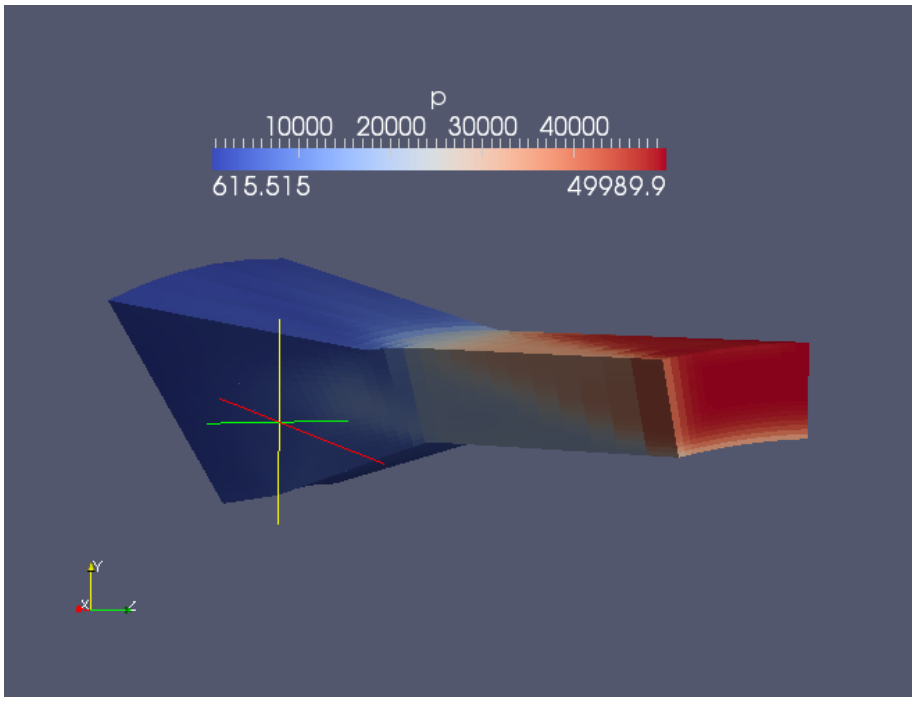


Figure 142: Pressure distribution on the inlet, combustor, nozzle, and cowl surfaces of the scramjet duct.

55.1 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...
```

56 Titan aeroshell using imported grids

Another aeroshell model is shown in Figure 143. 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 examining the grid planes as the indices were adjusted from one limit to another.

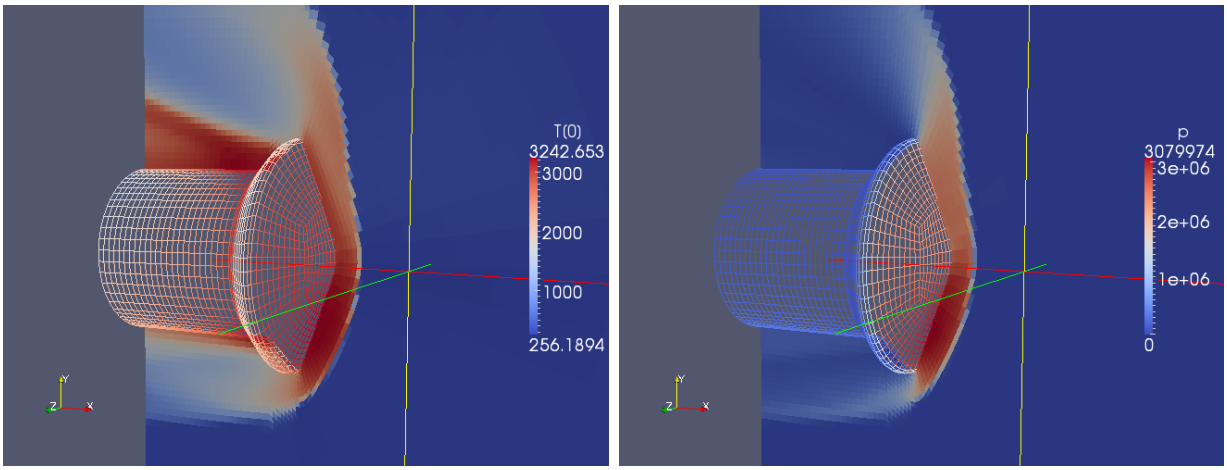


Figure 143: Views of the temperature and pressure fields around a Titan aeroshell. The surface grid is shown as a wire-frame rendering and a vertical slice (with solid colouring) is made through the flow field around the aeroshell.

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 grids 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.

56.1 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)
```

57 Couette Flow: 3D

This three-dimensional Couette flow case is provided by Jason Qin, as an extension of the two-dimensional case in Sec 47. The front and side views of the flow doamin are shown in Figure 144. Since we are going to monitor the velocity profile between the plates that are separated by a small distance in z direction, the grid has high resolution in that direction compared to the resolutions in x and y directions.

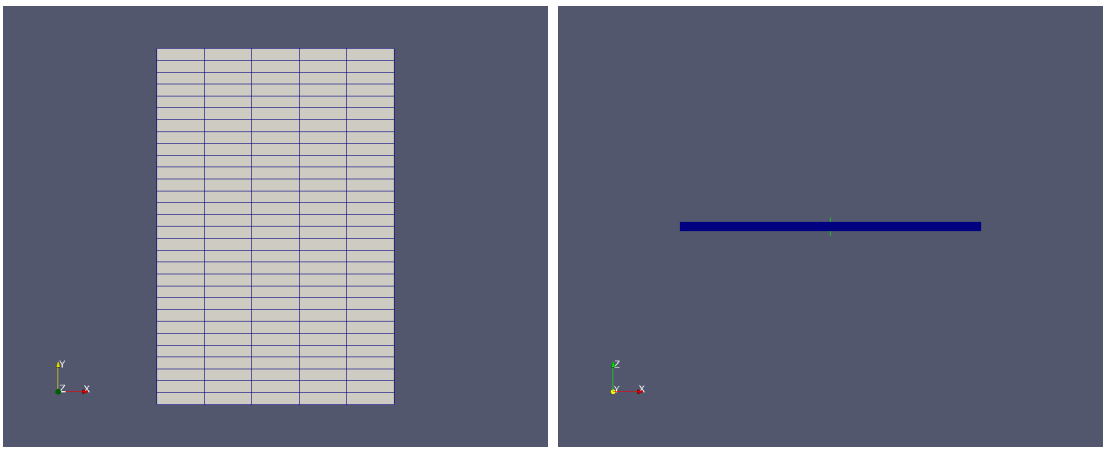


Figure 144: Front and side view of 3D couette flow.

57.1 Input script (.py)

The top surface is set as Moving-Wall boundary condition, while the BOTTOM surface Adiabatic-Wall. These are effectively the two plates that bound the flow. Slip-Wall boundary conditions were set both the the WEST and EAST surface, while the NORTH and SOUTH surfaces are connected manually with the aid of the function `connect_blocks_3D`. Since the NORTH and SOUTH boundary surfaces are not geometrically adjacent, the boolean parameter of `check_corner_locations` should be set to `False`, else the `connect_blocks_3D` will flag an error.

```
# couette.py
# Jason (Kan) Qin, November 2013

from math import pi, sin, cos

gdata.dimensions = 3
gdata.title = "pressure distribution in a thrust bearing chamber 3D"
print gdata.title

select_gas_model(model='ideal gas', species=['air'])
gdata.viscous_flag = 1
gdata.turbulence_model = "k_omega"
gdata.flux_calc = ADAPTIVE
gdata.max_time = 5.0e-3 # seconds
gdata.max_step = 30000
```

```
gdata.dt = 1.0e-10
gdata.dt_plot = 1.0e-3
gdata.dt_history = 1.0e-3

# Define flow conditions
p_exit = 0.1e6
v_trans = 130.0 ;

# Geometry
h_1 = 0.0 ;
h_2 = 3e-3 ;
r_1 = 0.0  ;
r_2 = 1e-1 ;
l_1 = 0.0 ;
l_2 = 1.5e-1 ;

def initial_flow(x, y, z):
    global h_2, p_exit, v_trans
    v = v_trans * z / h_2 # linear velocity profile
    return FlowCondition(p=p_exit, u=0.0, v=v, w=0.0).to_dict()

def makeSimpleBox(ini_x0, ini_x1, ini_y0, ini_y1, ini_z0, ini_z1):
   x0 = ini_x0 ; x1 = ini_x1 ;
   y0 = ini_y0 ; y1 = ini_y1 ;
   z0 = ini_z0 ; z1 = ini_z1 ;
   p0 = Vector(x0, y0, z0)
   p1 = Vector(x1, y0, z0)
   p2 = Vector(x1, y1, z0)
   p3 = Vector(x0, y1, z0)
   p4 = Vector(x0, y0, z1)
   p5 = Vector(x1, y0, z1)
   p6 = Vector(x1, y1, z1)
   p7 = Vector(x0, y1, z1)
   p01 = Line(p0, p1)
   p12 = Line(p1, p2)
   p32 = Line(p3, p2)
   p03 = Line(p0, p3)
   p45 = Line(p4, p5)
   p56 = Line(p5, p6)
   p76 = Line(p7, p6)
   p47 = Line(p4, p7)
   p04 = Line(p0, p4)
   p15 = Line(p1, p5)
   p26 = Line(p2, p6)
   p37 = Line(p3, p7)
   return WireFrameVolume(p01, p12, p32, p03, p45, p56, p76, p47, p04, p15, p26, p37)

# Define the blocks, boundary conditions and set the discretisaztion.
nx0 = 5 ; ny0 = 30 ; nz0 = 20 ;
c_0 = RobertsClusterFunction(1,1,1.0)
pvolume0 = makeSimpleBox(r_1,r_2,l_1,l_2,h_1,h_2)
cflist0 = [c_0,]*12 ;
blk_0 = Block3D(label="plate", nni=nx0, nnj=ny0, nnk=nz0,
               parametric_volume=pvolume0,
               cf_list=cflist0,
               fill_condition=initial_flow)

blk_0.set_BC("TOP", "MOVING_WALL", r_omega=[0.0,0.0,0.0],v_trans=[0.0,v_trans,0.0])
blk_0.bc_list[BOTTOM] = AdiabaticBC()
blk_0.set_BC("WEST","SLIP_WALL")
blk_0.set_BC("EAST","SLIP_WALL")

# the south face is connected with the north face
connect_blocks_3D(blk_0,blk_0,[(1,2),(5,6),(4,7),(0,3)],
                  reorient_vector_quantities=True,
                  nA=[0.0,1.0,0.0], t1A=[1.0,0.0,0.0],
                  nB=[0.0,1.0,0.0], t1B=[1.0,0.0,0.0],
                  check_corner_locations=False)

identify_block_connections()
```

57.2 Shell scripts

```
#!/bin/sh
# couette.sh

e3prep.py --job=couette --do-svg
e3post.py --job=couette --vtk-xml --tindx=0

time e3shared.exe --job=couette --run

e3post.py --job=couette --vtk-xml --tindx=last

e3post.py --job=couette --output-file=dudy0.dat --tindx=0 \
          --slice-list="0,1,:,0"
e3post.py --job=couette --output-file=dudy1.dat --tindx=last \
          --slice-list="0,1,:,0"

gnuplot <<EOF
set term postscript eps 20
set output "velocity.ps"
set title "Velocity profile along the height"
set ylabel "Height, m"
set xlabel "Velocity, m/s"
set yrange [0.0:0.0115]
set xrange [-10.0:110.0]
plot "dudy0.dat" using 6:2 with lines title "Initial value", \
     "dudy1.dat" using 6:2 with lines title "Steady state condition"
EOF
```

57.3 Results

The velocity profile along the height with different initial values are shown in Figure 145, respectively. The form of the initial velocity profile has a big effect on the computation time for this test case, with the uniform velocity requiring a long simulation time to reach steady state, compared with a linear initial velocity profile.

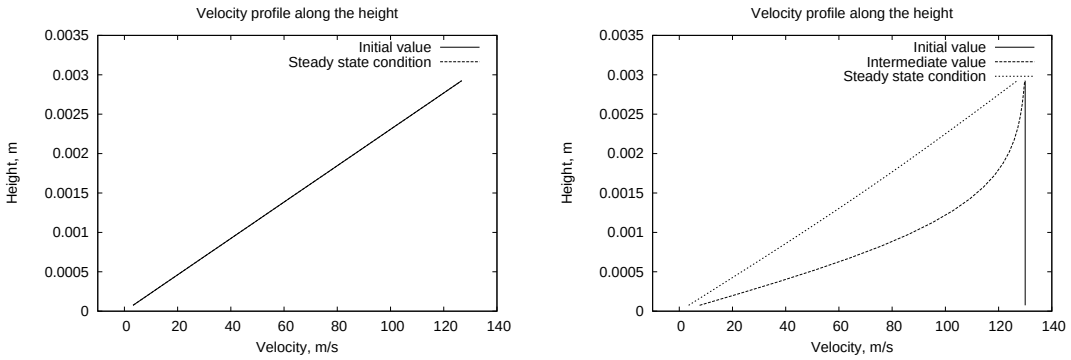


Figure 145: Velocity profile along the height: linear and uniform initial velocity profiles.

57.4 Notes

- None

58 Taylor Couette Flow

This test case, used to verify the nonzero rotational speed version of the Moving-Wall boundary condition, was provided by Jason Qin. It selects some examples of compressible Taylor-Couette flow from Ref. [50] for an annulus with inner radius 215.5 mm and gap width 3.1 mm. The axial extent of the annulus is 10 times the gap width. The outer cylindrical surface of the annulus (the housing) was fixed and the inner surface (the rotor) was moving with a rotational speed of 27600 rpm. Three different pressures are simulated to cover a range of cases, with and without Taylor vortices. Other parameters used for the simulations are shown in Table 2.

Table 2: Parameters for simulations, used match the experimental conditions reported in Ref.[50]

Case	intermediate pressure	high pressure	low pressure
Pressure, Pa	100	1000	10
Rotor Temperature, K	348	351	344
Stator Temperature, K	350	366	344
Taylor Number	17	181	3.6

58.1 Input script (.py)

To simulate this test case, only a small segment of the annulus is modelled, then periodic boundary conditions are applied to connect the ends of the segment. Note that the ends of the segment have different spatial orientations and this must be handled by the boundary condition. For the low and intermediate pressure cases, since the Taylor numbers are quite low, there is no vortices generated. In those cases, the grid is low resolution in z direction while, for high pressure case, the vortices might be generated in the gap, and it needs a high resolution grid in z direction (*i.e.* the axial direction of the rotor).

```
# taylor_couette.py
# Jason (Kan) Qin, December 2013

from math import pi, sin, cos, sqrt

gdata.dimensions = 3
gdata.title = "taylor couette flow"
print gdata.title

select_gas_model(model='ideal gas', species=['N2'])
gdata.viscous_flag = 1
gdata.turbulence_model = "k_omega"
gdata.flux_calc = ADAPTIVE
gdata.max_time = 500.0e-3           # seconds
gdata.max_step = 400000
```

```
gdata.dt = 1.0e-11
gdata.dt_plot = 1.0e-4
gdata.dt_history = 1.0e-4

# Define flow conditions
p_exit = 1000 ;
r_omega = 2*pi*27600.0/60.0 ;
T_1 = 351.0 ;
T_2 = 366.0 ;
theta = 60.0*pi/180 ;

# Geometry
r_1 = 0.2125 ;
g_width = 0.0031 ;
r_2 = r_1 + g_width ;
h_1 = 0.0 ;
h_2 = 10.0*g_width ;

initial = FlowCondition(p=p_exit, u=0.0, v=0.0, w=0.0, T=T_1)

def makeSimpleBox(ini_angular1, ini_angular2, ini_h1, ini_h2):
   from math import pi, sin, cos
   inih1 = ini_h1 ;
   inih2 = ini_h2 ;
   ini1 = ini_angular1 ;
   ini2 = ini_angular2 ;
   center_b = Node(0.0, 0.0, inih1)
   center_t = Node(0.0, 0.0, inih2)
   p0 = Vector(r_1*cos(ini1), r_1*sin(ini1), inih1)
   p1 = Vector(r_2*cos(ini1), r_2*sin(ini1), inih1)
   p2 = Vector(r_2*cos(ini2), r_2*sin(ini2), inih1)
   p3 = Vector(r_1*cos(ini2), r_1*sin(ini2), inih1)
   p4 = Vector(r_1*cos(ini1), r_1*sin(ini1), inih2)
   p5 = Vector(r_2*cos(ini1), r_2*sin(ini1), inih2)
   p6 = Vector(r_2*cos(ini2), r_2*sin(ini2), inih2)
   p7 = Vector(r_1*cos(ini2), r_1*sin(ini2), inih2)
   p01 = Line(p0, p1)
   p12 = Arc(p1, p2, center_b)
   p32 = Line(p3, p2)
   p03 = Arc(p0, p3, center_b)
   p45 = Line(p4, p5)
   p56 = Arc(p5, p6, center_t)
   p76 = Line(p7, p6)
   p47 = Arc(p4, p7, center_t)
   p04 = Line(p0, p4)
   p15 = Line(p1, p5)
   p26 = Line(p2, p6)
   p37 = Line(p3, p7)
   return WireFrameVolume(p01, p12, p32, p03, p45, p56, p76, p47, p04, p15, p26, p37)

nx = 25 ; ny = 80 ; nz = 200 ;
nbx = 1 ; nby = 40 ; nbz = 1 ;
c_0 = RobertsClusterFunction(1,1,1.0)

# North, East, South, West, Top, Bottom
mv = MovingWallBC(r_omega=[0.0,0.0,r_omega],v_trans=[0.0,0.0,0.0],Twall_flag=True,Twall=T_1)
ft = FixedTBC(Twall=T_2)
slip = SlipWallBC()

pvolume0 = makeSimpleBox(0.0*pi/180, 60.0*pi/180, h_1, h_2)
bclist0 = [None,ft,None,mv,slip,slip]
cflist0 = [c_0,]*12 ;
blk0 = SuperBlock3D(label="check", nni=nx, nnj=ny, nnk=nz,
               nbi=nbx, nbj=nby, nbk=nbz,
               parametric_volume=pvolume0,
               bc_list=bclist0,
               cf_list=cflist0,
               fill_condition=initial)

# South and North
connect_blocks_3D(blk0.blks[0][0][0],blk0.blks[0][-1][0],[(1,2),(5,6),(4,7),(0,3)],
                  reorient_vector_quantities=True,
                  nA=[0.0,1.0,0.0], t1A=[1.0,0.0,0.0],
```

```
                nB=[-sin(theta),cos(theta),0.0], t1B=[cos(theta),sin(theta),0.0],
                check_corner_locations=False)

identify_block_connections()
```

58.2 Shell scripts

```
#!/bin/sh
# prep.sh
e3prep.py --job=tc_flow_nitrogen --do-svg
```

```
#!/bin/bash -l
#PBS -S /bin/bash
#PBS -N tc_flow
#PBS -q workq
#PBS -l select=5:ncpus=8:NodeType=medium:mpiprocs=8 -A uq-XXX
#PBS -l walltime=40:00:00
echo "-----------------------------------------"
echo "Begin MPI job..."
date
cd $PBS_O_WORKDIR
mpirun -np 40 $HOME/e3bin/e3mpi.exe --job=tc_flow_nitrogen --run --max-wall-clock=150000 > LOGFILE
echo "End MPI job."
date
```

```
#!/bin/sh
# post-processing script
# post.sh

gnuplot <<EOF
set term postscript eps 20
set output "velocity.eps"
set title "axially averaged tangential velocity profile"
set xlabel "radial position"
set ylabel "velocity"
set xrange [0.0:1.0]
set yrange [0.0:1.0]
plot "average.txt" using 1:2 with lines title "Eilmer3", \
     "tangential.dat" using 1:2 with lines title "CTDNS"
EOF

gnuplot <<EOF
set term postscript eps 20
set output "temperature.eps"
set title "axially averaged temperature profile"
set xlabel "radial position"
set ylabel "temperature"
set xrange [0.0:1.0]
set yrange [350.0:400.0]
plot "average.txt" using 1:3 with lines title "Eilmer3", \
     "temperature.dat" using 1:2 with lines title "CTDNS"
EOF
```

58.3 Results

For the low pressure case, the velocity profile is roughly linear accross the narrow gap, and the temperature profile has a parabolic shape with maximum temperature near the center of the gap. Figure 146 shows the comparison results of velocity and temperature with different methods. The apparent difference is caused by the slip-wall boundary condition being considered in DSMC method and a no-slip boundary condition being used by Eilmer3. This is not too much of a problem because, with pressure increases, the real boundary condition more-closely approaches the no-slip condition.

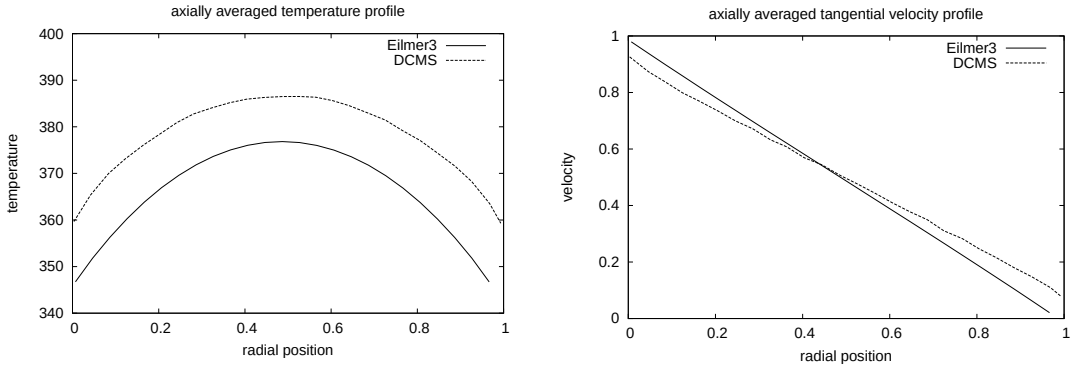


Figure 146: Comparison of temperature and velocity profiles in radial direction at low pressure condition.

For the intermediate pressure case, shown in Figure 147, there is a similar result. The profile for the tangential velocity is nearly linear and the temperature profile is nearly parabolic with a maximum slightly closer to the hotter wall as seen in The agreement between numercial schemes is now good.

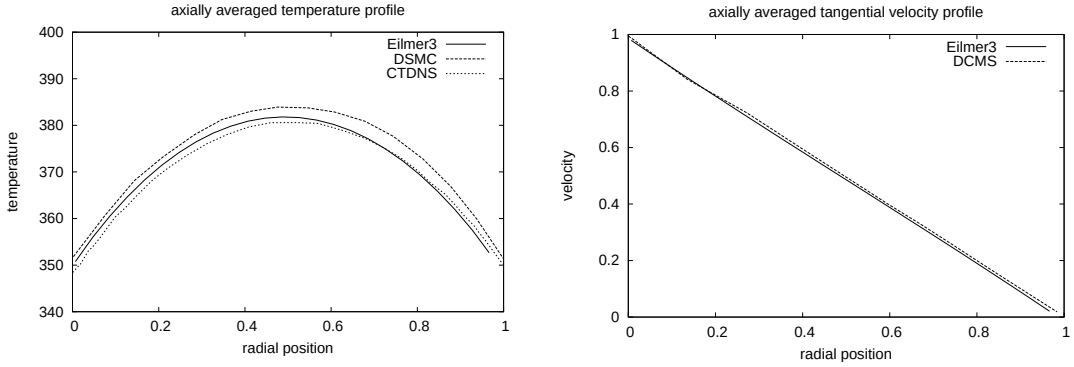


Figure 147: Comparison of temperature and velocity profiles in radial direction at intermediate pressure condition.

For the high pressure case, the Taylor number has exceeded a critical value and vortices, aligned with the surface velocity of the rotor, make the gap flow fully three dimen-

sional. Figure 148 shows velocity and temperature contours within the gap, the periodic structure being associated with the Taylor vortices.

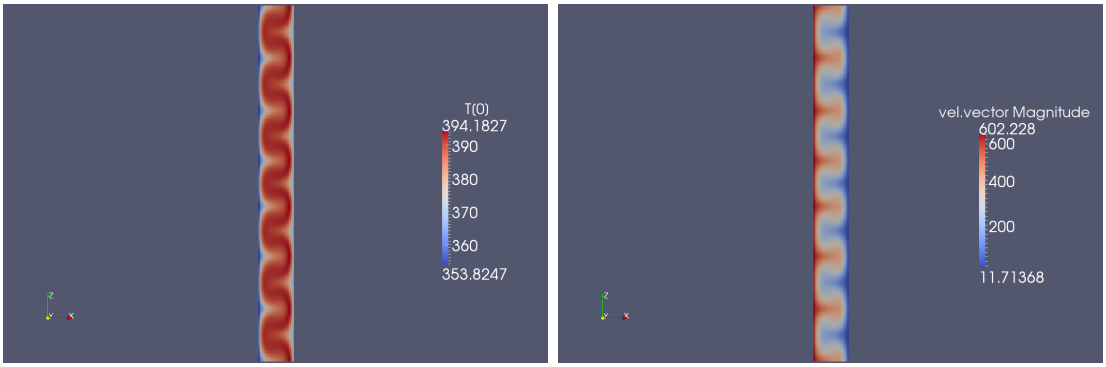


Figure 148: Temperature and velocity contours within the gap, at the high pressure condition. The left-most boundary is the rotor and the right-most surface is the housing wall.

The velocity profile (averaged over the axial direction) has changed to an "S"-shapes curve in Figure 149). This velocity profile characterizes a flow with a higher gradient at the walls, due to enhanced radial transport of fluid induced by the vortices.

The axially averaged temperature profile (seen in Figure 149) is much flatter than the parabolic profiles of the lower Taylor number cases. This averaged shape also exhibits steeper graidents at the walls, which induce a high heat flux. Again, these changes are due to the presence of vortices and the associated increase in radial transport across the gap.

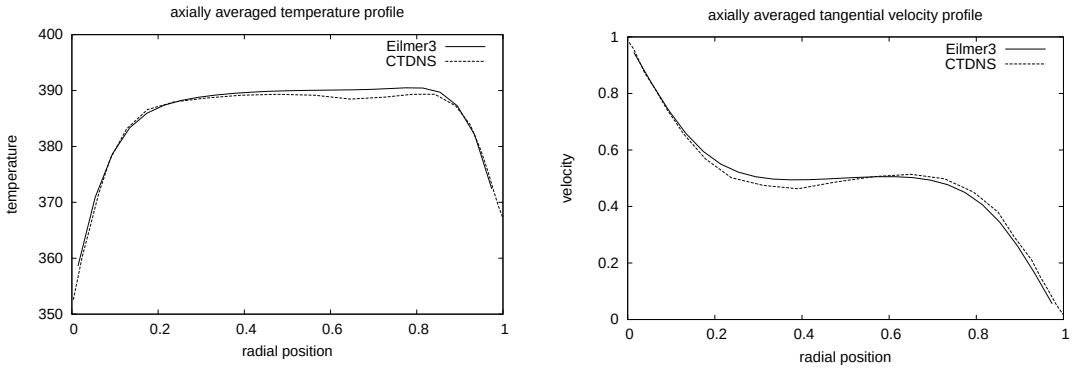


Figure 149: Comparison of averaged temperature velocity profiles in radial direction at high pressure condition.

58.4 Notes

- Python script for calculating axially averaged temperature and velocity profile.

```
#! /usr/bin/env python
# \file a_vt.py
#
# caculate the averaged velocity and temperature along the radial gap

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

print "\n\ncaculate the average temperature and velocity."

fileName = 'grid/t0000/tc_flow_nitrogen.grid.b0000.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/t0036/tc_flow_nitrogen.flow.b0000.t0036.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

# Caculate the averaged velocity and temperature along the radial gap
# South surface of block 0
fileName = "average.txt"
fout = open(fileName, "w")
j = 0
v_tan = 0.0
vel_tan = 0.0
T_tan = 0.0
Tem_tan = 0.0
for i in range(ni):
    for k in range(nk):
        pos_x = soln.data["pos.x"][i][j][k]
        pos_y = soln.data["pos.y"][i][j][k]
        r_g = sqrt(pos_x*pos_x + pos_y*pos_y)
        r_1 = ( r_g-0.2125 ) / 0.0031               # radial position
        vel_x = soln.data["vel.x"][i][j][k]
        vel_y = soln.data["vel.y"][i][j][k]
        T_tan = soln.data["T[0]"][i][j][k]          # temp temperature
        theta = atan(pos_y/pos_x)
        v_tan = vel_y*cos(theta) - vel_x*sin(theta)  # temp tangential velocity
        v_tan = v_tan / 614.0                       # change into nondimensionalized form
        vel_tan += v_tan                            # the sum of tangential velocity
        Tem_tan += T_tan                            # the sum of tempearture
    fout.write("%e %e %e\n" % (r_1, vel_tan/nk, Tem_tan/nk))
    v_tan = 0.0
    vel_tan = 0.0
    T_tan = 0.0
    Tem_tan = 0.0
fout.close()

print "done."
```

Part VI
References and Appendices

References

[1] W. Y. K. Chan, M. K. Smart, and P. A. Jacobs. Experimental validation of the T4 Mach 7.0 nozzle. School of Mechanical and Mining Engineering Report 2014/14, The University of Queensland, Brisbane, Australia, September 2014.

[2] 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.

[3] J. J. Quirk. A contribution to the great Riemann solver debate. *International Journal for Numerical Methods in Fluids*, 18(6):555–574, 1994.

[4] M. S. Liou and C. J. Steffen. A new flux splitting scheme. NASA Technical Memorandum 104404, 1991.

[5] D. I. Pullin. Direct simulation methods for compressible inviscid ideal-gas flow. *Journal of Computational Physics*, 34(2):231–244, 1980.

[6] M. N. Macrossan. The equilibrium flux method for the calculation of flows with non-equilibrium chemical reactions. *Journal of Computational Physics*, 80(1):204–231, 1989.

[7] Y. Wada and M. S. Liou. A flux splitting scheme with high-resolution and robustness for discontinuities. AIAA Paper 94-0083, January 1994.

[8] M. S. Liou. A sequel to AUSM, part II: AUSM+-up for all speeds. *Journal of Computational Physics*, 214:137–170, 2006.

[9] J. W. Maccoll. The conical shock wave formed by a cone moving at high speed. *Proceedings of the Royal Society of London*, 159(898):459–472, 1937.

[10] P. A. Jacobs. Single-block Navier-Stokes integrator. ICASE Interim Report 18, 1991.

[11] Ames Research Staff. Equations, tables and charts for compressible flow. NACA Report 1135, 1953.

[12] R. J. Hakkinen, I. Greber, L. Trilling, and S. S. Abarbanel. The interaction of an oblique shock wave with a laminar boundary layer. NASA Memorandum 2-18-59W, 1959.

[13] S. Mohammadian. Viscous interaction over convave and convex surfaces at hypersonic speeds. *Journal of Fluid Mechanics*, 55(1):163–175, 1972.

[14] M. S. Holden, T. P. Wadhams, J. K. Harvey, and G. V. Candler. Comparisons between DSMC and Navier-Stokes solutions on measurements in regions of laminar shock wave boundary layer interaction in hypersonic flows. AIAA Paper 2002-0435, January 2002.

[15] M. S. Holden and T. P. Wadhams. A database of aerothermal measurements in hypersonic flows in "building block" experiments for CFD validation. AIAA Paper 2003-1137, January 2003.

[16] M. MacLean and M. Holden. Validation and comparison of WIND and DPLR results for hypersonic, laminar problems. AIAA-Paper 2004-0529, AIAA, January 2004.

[17] I. Nompelis, G. V. Candler, and M. S. Holden. Effect of vibrational nonequilibrium on hypersonic double cone experiments. *A.I.A.A. Journal*, 41(11):2162–2169, 2003.

[18] J. D. Anderson. *Hypersonic and High Temperature Gas Dynamics.* McGraw-Hill, New York, 1989.

[19] J. M. Kendall. Experiments on supersonic blunt-body flows. Progress Report 20-372, Jet Propulsopn Laboratory, California Institute of Technology, Pasadena, California., February 1959.

[20] 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.

[21] K. Sawada and E. Dendou. Validation of hypersonic chemical equilibrium flow calculations using ballistic-range data. *Shock Waves*, 11:43–51, 2001.

[22] P. H. Rose and W. I. Stark. Stagnation point heat-transfer measurements in dissociated air. *Journal of the Aeronautical Sciences*, (February):86–97, 1958.

[23] 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.

[24] H. F. Lehr. Experiments on shock induced combustion. *Astronautica Acta*, 17:589–597, 1972.

[25] G. J. Wilson. Computation of steady and unsteady shock-induced combustion over hypervelocity blunt bodies. PhD thesis, Stanford University, California., December 1991.

[26] 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.

[27] R.W. Rutowski and D. Bershader. Shock tube studies of radiative transport in an argon plasma. *The physics of fluids*, 7(4):568–577, 1964.

[28] P.L. McDill, E.A. Brown, P.A. Ross, and O.A. Huseby. The performance of a buffered shock tube with area reduction. In A.M. Krill, editor, *Proceedings of the Second Symposium on Hypervelocity Techniques*, Advances in Hypervelocity Techniques,, pages 749–772, New York, 1962. Plenum Press.

[29] A.E. Kramida, Yu. Ralchenko, , J. Reader, and NIST ASD Team. NIST Atomic Spectra Database (version 5.1). [Online]. Available: http://physics.nist.gov/asd [Monday, 28-Oct-2013]. National Institute of Standards and Technology, Gaithersburg, MD., 2013.

[30] R.N. Gupta, J.M. Yos, R.A. Thompson, and K.-P. Lee. A Review of Reaction Rates and Thermodynamic and Transport Properties for an 11-Species Air Model for Chemical and Thermal Nonequilibrium Calculations to 30,000 K. Reference Publication 1232, NASA, August 1990.

[31] M.J. Wright, D. Bose, G.E. Palmer, and E. Levin. Recommended collision integrals for transport property computations, Part 1: Air species. *AIAA Journal*, 43(12):2558–2564, 2005.

[32] E. Levin and M.J. Wright. Collision integrals for ion-neutral interactions of air and argon. *Journal of Thermophysics and Heat Transfer*, 19(1):127–128, 2005.

[33] Mason. Transport properties of ionized gases. *The Physics of Fluids*, 10(8):1827–1832, 1967.

[34] J.-L. Cambier. Numerical simulations of a nonequilibrium argon plasma in a shock-tube experiment. AIAA Paper 91-1464, 1991.

[35] H. Petschek and S. Byron. Approach to equilibrium lonization behind strong shock waves in argon. *Annals of Physics*, 1(3):270 – 315, 1957.

[36] I.I. Glass and W.S. Liu. Effects of hydrogen impurities on shock structure and stability in ionizing monatomic gases. part 1. argon. *Journal of Fluid Mechanics*, 84(1):55–77, January 1978.

[37] J.P. Appleton and K.N.C. Bray. The conservation equations for a nonequilibrium plasma. *Journal of Fluid Mechanics*, 20(4):659–672, December 1964.

[38] M.Y. Jaffrin. Shock structure in a partially ionized gas. *Physics of Fluids*, 8:606, 1965.

[39] J.R. Howell and M. Perlmutter. Monte Carlo solution of thermal transfer through radiant media between gray walls. *Journal of Heat Transfer*, 86(1):116–122, 1964.

[40] M.F. Modest. The Monte Carlo method applied to gases with spectral line structure. *Numerical Heat Transfer*, 86(1):273–284, 1992.

[41] A. Wang and M.F. Modest. Spectral Monte Carlo models for nongray radiation analyses in inhomogeneous participating media. *International Journal of Heat and Mass Transfer*, 50(19-20):3877–3889, September 2007.

[42] M.F. Modest and S.C. Poon. Determination of three-dimensional radiative exchange factors for the space shuttle by monte carlo. ASME Paper 77-HT-49, 1977.

[43] W. Cunto, C. Mendoza, F. Ochsenbein, and C.J. Zeippen. TOPbase at the CDS. *Astronomy and Astrophysics*, 275(1):5–8, August 1993.

[44] H.R. Griem. *Spectral line broadening by plasmas*. Academic Press, New York, 1974.

[45] C. S. Park. Calculation of radiation from argon shock layers. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 28(1):29–40, 1982.

[46] C. S. Park. *Nonequilibrium hypersonic aerothermodynamics*. John Wiley and Sons, 1990.

[47] A. Kazakov and M. Frenklach. *Reaction Set DRM19*.

[48] T.J Poinsot and S.K Lelef. Boundary conditions for direct simulations of compressible viscous flows. *Journal of Computational Physics*, 101(1):104 – 129, 1992.

[49] C.S. Park. Assessment of two-temperature kinetic model for ionizing air. *Journal of Thermophysics and Heat Transfer*, 3(3):233–244, 1989.

[50] B. Larignon, K. Marr, and D. B. Goldstein. Monte Carlo and Navier-Stokes simulations of compressible Taylor-Couette flow. *A.I.A.A. Journal of Thermophysics and Heat Transfer*, 20(3):544–551, 2006.

[51] Mark G. Sobell. *A Practical Guide to Linux Commands, Editors and Shell Programming*. Prentice Hall, Upper Saddle River, New Jersey, 2005.

[52] R. Ierusalimschy, L. H. de Figueiredo, and W.C. Filho. Lua - an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[53] 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.

[54] 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.

[55] D. R. Mott. *New Quasi-Steady-State and Partial-Equilibrium Methods for Integrating Chemically Reacting Systems*. PhD thesis, University of Michigan, 1999.

[56] 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.

[57] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

A Instructions for installation and getting started

The latest version of this files should be in the `doc/sphinx/` directory of the package of source files.

```
Getting the codes and preparing to run them
===========================================

The code repository
-------------------
The codes are available for download from a Mercurial repository.
To make a clone of the repository::

  $ cd $HOME
  $ hg clone https://source.eait.uq.edu.au/hg/cfcfd3 cfcfd3

and provide the username "cfcfd-user@svn.itee.uq.edu.au" for authentication.
This takes about 40 seconds on campus at UQ.
It may take much longer, depending on your internet connection.

To see what's changed::

  $ cd cfcfd3
  $ hg incoming https://source.eait.uq.edu.au/hg/cfcfd3
  ...
  $ hg pull -u https://source.eait.uq.edu.au/hg/cfcfd3

Notes

#. You will need a password for any access.  Please ask.
#. You can read but not write with the "cfcfd-user" username.
#. Some usernames (cfcfd-dev@svn.itee.uq.edu.au) may push changesets back
   to the repository.  You will need to negotiate a developer role for this access.
#. Some gas models depend on the NASA CEA code or the NIST REFPROP library.
   If you want to use these models (and there is no look-up-table equivalent
   already available) you will need to obtain these codes and place them
   into the extern/ directory.
   They are not included as part of our cfcfd3 repository but the cfcfd3 makefiles
   will be aware of them if they are sitting in the extern/ directory.


Licence
-------
CFCFD program collection is a set of flow simulation tools for compressible fluids.
Copyright (C) 1991-2012 Peter Jacobs, Rowan Gollan, Daniel Potter,
Brendan O'Flaherty, Fabian Zander, Wilson Chan, Peter Blyton and
other members of the CFCFD group.

This collection is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or any later version.

This program collection is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License for more details.

You should have received a copy of the GNU-General-Public-License_
along with this program.  If not, see <http://www.gnu.org/licenses/>.

.. _GNU-General-Public-License: ./_static/gpl.txt


Your computational 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, libreadline-dev on Ubuntu)
* Python + the numpy, matplotlib and scipy extensions
* 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:

#. mercurial
#. gcc
#. gcc-c++
#. m4
#. openmpi
#. openmpi-devel
   (to use openmpi on Fedora,
   :ref:`the module must be loaded <label-openmpi-fedora>`)
#. gcc-gfortran
#. libgfortran.i686, glibc-devel.i686 and libgcc.i686
   (to compile the 32-bit CEA code on 64-bit Fedora)
#. swig
#. python-devel
#. numpy
#. python-matplotlib
#. scipy
#. readline-devel (for Lua)
#. popt-devel
#. sympy (to run the Method-of-Manufactured-Solutions test case for Eilmer3)

To a basic Ubuntu 10.04 (or any recent Debian derivative) installation,
you should add the following packages and their dependencies:

#. mercurial
#. g++
#. m4
#. mpi-default-dev
#. mpi-default-bin
#. gfortran
#. gfortran-multilib (for compiling 32-bit CEA2 on a 64-bit system)
#. swig
#. python-dev
#. python-numpy
#. python-matplotlib
#. python-scipy
#. libreadline-dev
#. libpopt-dev
#. libncurses5-dev
#. tk
#. bwidget
#. gnuplot
#. tcl-dev (if you want to build IMOC)
#. python-sympy (to run the Method-of-Manufactured-Solutions test case for Eilmer3)

Compiler versions
-----------------
Since March 2013, we have started using some of the C++11 features
such as range-based for loops and initializer expressions.
Because of this you will need a suitable C++ compiler.
```

```
For the GNU compiler collection, versions 4.6.3 and 4.8.0 are suitable.
Clang/LLVM versions 3.2 and later are also good.

Using the codes on MS-Windows
-----------------------------
The codes assemble most conveniently on a Linux/Unix-like environment.
They should also build and run within Cygwin (http://cygwin.com/), however,
it may be convenient to run a full linux installation within
VirtualBox (https://www.virtualbox.org/), on your MS-Windows computer.

Using the codes on Apple OSX
----------------------------
The codes can be compiled and run on OSX as this is a Unix based OS.
The Xcode development environment (https://developer.apple.com/xcode/)
should be downloaded and installed to provide Apple's versions of the
GNU Compiler Collection, Python and the make utility, amongst other
development tools.
popt, readline, SWIG and Tcl/Tk can either be installed from source
or via a package manager such as MacPorts (http://www.macports.org/) or
Fink (http://www.finkproject.org/).

Notes:

#. If possible, it is recommended to install these dependencies from source.
#. The required Python packages (numpy, scipy and matplotlib) are all available
   as pre-packaged binaries for OSX on sourceforge.net, although they can also
   be installed from source if necessary.
#. Ingo has had a good experience installing binary packages from MacPorts,
   the only subtly being the need to install swig and swig-python.

SSH access to the repository for developers
-------------------------------------------
Alternative access to the Mercurial repository for developers is possible via https.
You will need the password for the cfcfd-dev@svn.itee.uq.edu.au login.  Please ask.

::

  $ cd ~
  $ hg clone https://source.eait.uq.edu.au/hg/cfcfd3 cfcfd3
  $ cd cfcfd3/extern/
  $ hg clone https://source.eait.uq.edu.au/hg/cea2 cea2
  $ hg clone https://source.eait.uq.edu.au/hg/refprop refprop
```

```
Eilmer3
=======

Eilmer3 is our principal simulation code for 2D and 3D gas dynamics.
It is a research and education code, suitable for the exploration of
flows where the bounding geometry is not too complex.

.. figure:: _static/Kiock-Mach.png
   :align: center
   :scale: 30%

   Transonic flow through a plane turbine cascade (Kiock et al., 1986).
   Simulation by Peter Blyton, 2011.
   Visualization with Paraview.

Documententation (PDF)
----------------------

The full Eilmer3 User Guide and Example Book: pdf-user-guide_

.. _pdf-user-guide: ./pdf/eilmer3-user-guide.pdf

The Theory Book: pdf-theory-book_

.. _pdf-theory-book: ./pdf/eilmer3-theory-book.pdf
```

```
Slides from Fabian Zander's lecture introducing Eilmer3 to
MECH4480 students: zander-lecture-slides_

.. _zander-lecture-slides: ./pdf/mech4480_lecture.pdf

Typical build and run procedure
-------------------------------
The new 2D/3D code Eilmer3 is built from source into
an installation directory ``$HOME/e3bin/``.
A typical build procedure (using the default ``TARGET=for_gnu``) might be::

  $ cd $HOME/cfcfd3/app/eilmer3/build
  $ make install
  $ make clean

Or, if you want the MPI version of the code built as well::

  $ cd $HOME/cfcfd3/app/eilmer3/build
  $ make TARGET=for_openmpi install
  $ make clean

You may need to add the installation directory to your system's
search path to run Eilmer3.
On a recent Linux system, this could be done by adding the line::

  $ export PATH=${PATH}:${HOME}/e3bin

to the ``.bash_profile`` or ``.bashrc`` file in your home directory.

To access the Lua gas module from within the user-defined (Lua) functions,
or to use the REFPROP gas model, the following lines should also be added
to your bash configuration::

  $ export LUA_PATH=${HOME}/e3bin/?.lua
  $ export LUA_CPATH=${HOME}/e3bin/?.so
  $ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${HOME}/e3bin

If you wish to make use of the cfpylib functions
from your own stand-alone Python scripts,
it may be convenient to set the PYTHONPATH environment variable::

  $ export PYTHONPATH=${PYTHONPATH}:${HOME}/e3bin/

.. _label-nonstandard-install-path:

If you choose to install eilmer3 in a different location from the
default location (``$HOME/e3bin``), then you will need to
set an environment variable called ``E3BIN`` and point it to
the non-standard install directory. For example, if you installed
the executables and supporting scripts to: ``/work/e3bin`` then
you would set the following in your ``.bashrc``::

  $ export E3BIN=/work/e3bin

.. _label-openmpi-fedora:

For running on Fedora, also add the following::

  module load openmpi-i386
  # Or, for 64-bit:
  module load openmpi-x86_64

Then, try out the cone20-simple example::

  $ mkdir $HOME/work; cd $HOME/work; mkdir 2D; cd 2D
  $ mkdir cone20-simple; cd cone20-simple
  $ cp $HOME/cfcfd3/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.

Summary of lines for your ``.bashrc`` file::

  export E3BIN=${HOME}/e3bin
  export PATH=${PATH}:${E3BIN}
  export LUA_PATH=${E3BIN}/?.lua
  export LUA_CPATH=${E3BIN}/?.so
  export PYTHONPATH=${PYTHONPATH}:${E3BIN}
  export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${E3BIN}


Building and running on Mac OSX
-------------------------------
This is mostly the same as for a Linux machine but we provide a couple of
specific targets::

  $ make TARGET=for_macports_gnu install
  $ make TARGET=for_macports_openmpi install


Building and running on the Barrine cluster at UQ
-------------------------------------------------
The details of running simulations on any cluster computer will be
specific to the local configuration.  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.

Set up your environment by adding the following lines to your ``.bashrc`` file::

    module purge
    module load mercurial
    module load intel-cc-13
    module load intel-mpi
    export PATH=${PATH}:${HOME}/e3bin
    export LUA_PATH=${HOME}/e3bin/?.lua
    export LUA_CPATH=${HOME}/e3bin/?.so

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

To compile the MPI-version of the code, use the command::

    $ make TARGET=for_intel_mpi install

from the ``cfcfd3/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
```

```
    #PBS -l walltime=6:00:00
    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
```

This is the script input ``examples/eilmer3/2D/lehr-479/run_simulation.sh``.

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
```

Building and running the radiation transport solver

While a flowfield calculation with coupled radiation can be performed
via the single processor version of eilmer3 (e3shared.exe), the
radiation transport portion of such calculations can often take a very
long time to run. The obvious solution is to implement the radiation
transport calculation in parallel. Due to the non-local nature of the
radiation transport problem, however, for most radiation transport
models it is necessary to implement the parallelisation via the shared
memory multiprocessor approach. The radiation transport solver in
eilmer3 has therefore been written to make use of the OpenMP API. As
the Eilmer3 flowfield solver does not currently support an OpenMP
build, the radiation transport solver can be built as a separate
executable, e3rad.exe.

The typical build procedure for the OpenMP version of the radiation
transport solver using the GNU compiler is::

```
  $ cd $HOME/cfcfd3/app/eilmer3/build
  $ make TARGET=for_gnu_openmp e3rad
  $ make clean
```

Then, try out the radiating-cylinder example::

```
  $ mkdir $HOME/work; cd $HOME/work; mkdir 2D; cd 2D
  $ mkdir radiating-cylinder; cd radiating-cylinder
  $ cp $HOME/cfcfd3/examples/eilmer3/2D/radiating-cylinder/* .
  $ tclsh cyl.test
```

On the barrine cluster, the Intel compiler should be used for best performance::

```
  $ cd $HOME/cfcfd3/app/eilmer3/build
  $ make TARGET=for_intel_openmp e3rad
  $ make clean
```

It should be noted that the e3mpi.exe executable is able to run
radiation transport calculations in parallel when either the
"optically thin" or "tangent slab" models are implemented, however a
specific blocking layout is required for the "tangent slab" model.
See the radiatively coupled Hayabusa simulation in
$HOME/cfcfd3/examples/eilmer3/2D/hayabusa for an example of this
blocking layout.

When things go wrong

Eilmer3 is a complex piece of software, especially when all of the
thermochemistry comes into play. There will be problems buried in the

```
code and, (very) occasionally, you will expose them.  We really do
have some pride in this code and will certainly try to fix anything
that is broken, however, we do this work essentially on our own time
and that time is limited.

When you have a problem, there are a number of things that you can do
to minimize the duration and pain of debugging:

#. Check the repository and be sure that you have the most recent
   revision of the code.  This code collection is a work in progress
   and, in some cases, you will not be the only one hitting a blatant
   bug.  It is likely that we or someone else has hit the same problem
   and, if so, it may be fixed already.  The code changes daily in
   small ways.  This may sound chaotic, such that you should just stay
   with an old version, however, we do try hard to not break things.
   In general, it is safest to work with the lastest revision.

#. Put together a simple example that displays the problem.  This
   example should be as simple as possible so that there are not extra
   interactions that confuse us.

#. Provide a complete package of input files and output pictures.
   We should be able to run your simulation within a few minutes
   and see the same output.

#. Be prepared to dig into the code and identify the problem yourself.
   We appreciate all of the help that we can get.


Source Code Docs
----------------
The following documentation is tentative and experimental.
Use the PDF files above; they are the primary documents.

.. toctree::
   :maxdepth: 2

   eilmer3/e3prep
   eilmer3/e3post
   eilmer3/e3history
   eilmer3/e3cgns
   eilmer3/e3_flow
   eilmer3/e3_block
   eilmer3/e3_grid
   eilmer3/cgns_grid
   eilmer3/e3_defs
   eilmer3/bc_defs
   eilmer3/flux_dict
   eilmer3/e3_render

‘Doxygen documenation of C++ sources <http://mech.uq.edu.au/cfcfd/doxygen/group__eilmer3.html>‘_

Other Notes
-----------
On Xserver for Linux (especially Ubuntu):

* If Paraview crashes on exporting a bitmap image, try adding the line::

    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.

Transferring input files between machines
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
If you find you want to transfer just the input files between
```

```
machines, ignoring the generated output files, you can do this by
using the ``--exclude`` option for the ``rsync`` command. For example,
to transfer just the input files of a directory called ``my-sim`` on a
local machine to a remote machine, use::

 $ rsync -av --exclude=flow --exclude=grid --exclude=hist --exclude=heat \
 --exclude=plot my-sim/ remote:my-sim

If you find you are using this often, you can define an alias as
appropriate for your shell. In BASH, I add the following line to my
``.bashrc`` file::

 alias rsync-eilmer="rsync -av --progress --exclude=flow --exclude=grid \
 --exclude=hist --exclude=heat --exclude=plot"

Then I can use do the above transfer by issuing the following command::

 $ rsync-eilmer my-sim/ remote:my-sim
```

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 [51] 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 preceded by one or two dashes. These modify the behaviour 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 237.

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 237) 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.

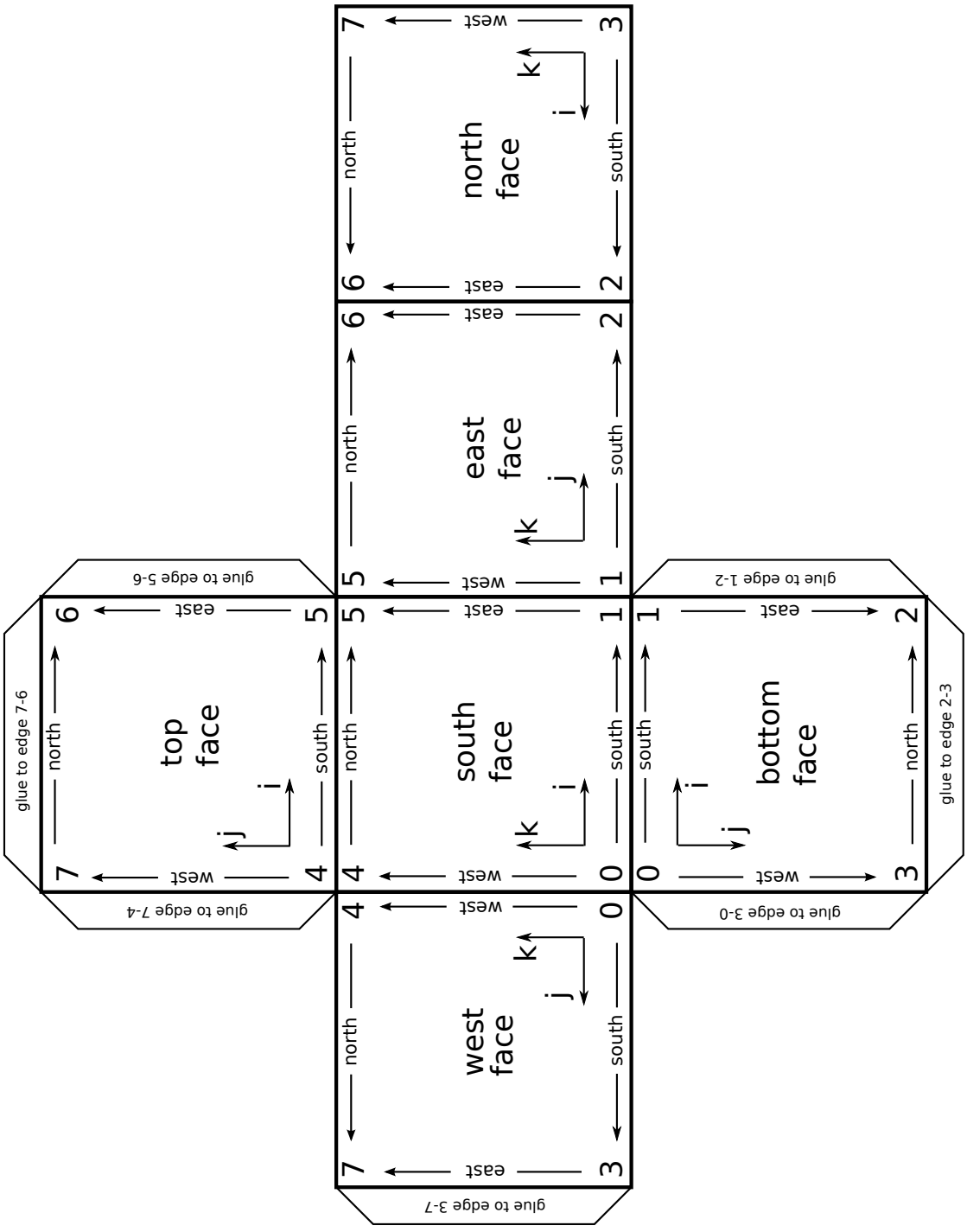
More specialized data objects can be defined via classes, and `e3prep.py` does exactly that. The name `gdata` is bound to an instance of the `GlobalData` class and contains many attributes that set the configuration of the flow solver. The user input file will use the already defined gdata object but 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.

When working in Python it is possible to see what options are available to you with a particular function or object using the `dir` command. This enables you to get a print out of the properties and functions associated with the object. For example create a node and see what the `dir` output is.

```
a = Node(0.0,0.1,label='a')
print dir(a)
['__add__', '__class__', '__del__', '__delattr__', '__dict__', '__div__', '__doc__', '__format__', '__getattr__',
'__getattribute__', '__hash__', '__iadd__', '__idiv__', '__imul__', '__init__', '__isub__', '__module__', '__mul__',
'__neg__', '__new__', '__pos__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__swig_destroy__', '__swig_getmethods__', '__swig_setmethods__',
'__weakref__', 'clone', 'copy', 'label', 'mirror_image', 'nodeList', 'norm', 'rotate_about_zaxis', 'str',
'this', 'transform_to_global', 'transform_to_local', 'translate', 'vrml_str', 'vtk_str', 'x', 'y', 'z']
```

The last items in this list are the different options available to a Node object that can be used within the prep file.

D Make your own debugging cube



E cfpylib modules

There are a number of modules that are useful for the definition of flow simulations but are not part of the core Eilmer code. These are available in a `cfpylib` Python library that may be imported into the user's input or postprocessing scripts. This library has become a bit of a catch-all for various utility modules and functions that don't fit directly into the main application source directories or the gas or geometry libraries. Documentation for the individual functions can be found online at http://cfcfd.mechmining.uq.edu.au/, under the link to `Libraries`. Alternatively, you may use Python's introspection facility or look at the source code directly.

E.1 Numerical Methods module

- `nm.collect_run_stats`: Run an executable a number of times and report.
- `nm.adapti`: Adaptive quadrature using Newton-Cotes 5- and 3-point rules.
- `nm.least_squares`: Fits a generalized polynomial basis to given data.
- `nm.line_search`: Implementation of an algorithm for optimization from Gerald and Wheatley.
- `nm.nelmin`: Nelder-Mead simplex minimization of a nonlinear (multivariate) function.
- `nm.ode`: Integrate a set of first-order ODEs.
- `nm.roberts`: Node distribution and coordinate stretching functions.
- `nm.secant_method`: Function solver, using the secant method.
- `nm.sode`: Integrate a set of stiff ODEs.
- `nm.stats`: Simple statistics for arrays of values. To replace those in scipy, just in case scipy is not installed.
- `nm.zero_solvers`: A small collection of function solvers, including bracketing.

E.2 Gas Dynamics module

- `gasdyn.billig`: Fred Billig's correlations for hypersonic shock-wave shapes. This module is shown completely in the following Section E.6.
- `gasdyn.ideal_gas_flow`: One-dimensional steady flow of an ideal gas. This module includes many small functions grouped into:

  - Isentropic flow relations.
  - 1D (Normal) Shock Relations.
  - 1-D flow with heat addition (Rayleigh-line).
  - Prandtl-Meyer functions.
  - Oblique-shock relations.
  - Taylor-Maccoll conical flow.

- `gasdyn.cea2_gas`: Thermodynamic properties of a gas mixture in chemical equilibrium. This module interfaces to the CEA code by writing a small input file, running the CEA code as a child process and then reading the results from the CEA plot file.

- `gasdyn.libgas_gas`: Access the gas models from the libgas library. This module gives you access to the same gas-model library as used in the main simulation program.

- `gasdyn.ideal_gas`: Thermodynamic properties of an ideal gas. This module provides a look-alike Gas class for use in the gas flow functions. Where ever cea2_gas works, so should this.

- `gasdyn.gas_flow`: Gas flow calculations using CEA2 or ideal Gas objects. The functions are generalized versions of those in gasdyn.ideal_gas_flow.

- `gasdyn.sutherland`: Sutherland form of viscosity and thermal conductivity for a few gases. Species available: Air, N2, O2, H2, CO2, CO, Ar.

E.3 Flow (house-keeping) module

- `flow.blockflow2d`: Pick up the flow data for mbcns2 block-structured grids.

- `flow.vtk_xml_writer`: Writing of BlockGrid2D and BlockFlow2D (mbcns2) objects to VTK XML files.

- `flow.tecplot_writer`: Writing of BlockGrid2D and BlockFlow2D (mbcns2) objects to Tecplot files.

E.4 Geometry module

- `geom.minimal_geometry`: A bare minimum geometry library to do some of the work required by Rowan's laura2vtk.py.

- `geom.svg_render`: Render a drawing in Scalable Vector Graphics format.

- `geom.transform_pyfunc`: Apply a matrix transformation to a Python function. The functions provided by this module are used to manipulate a python function prior to using the function to create a path with libprep3's PyFunctionPath. Available transformations are rotation and translation.

E.5 Utility module

- `util.flatten`: Function to flatten a nested list.
- `util.FortranFile`: Defines a file-derived class to read/write Fortran unformatted files.

E.6 Billig shock shape correlation

```
"""
billig.py: Fred 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

.. Author: PJ

.. Version: 19-June-2005
"""

from math import exp, sqrt, pow, tan
from ideal_gas_flow import beta_obl, beta_cone2

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.

    :param y: y-coordinate of the point on the shock wave
    :param M: free-stream Mach number
    :param theta: angle (in radians wrt free-stream direction)
                  of the downstream surface
```

```
    :param axi: (int) axisymmetric flag:

                | == 0 : cylinder-wedge
                | == 1 : sphere-cone

    :param 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)
    if axi == 1:
        # Use shock angle on a cone
        beta = beta_cone2(M, theta)
    else:
        # Use shock angle on a wedge
        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.

    :param x: x-coordinate of the point on the shock wave
    :param M: free-stream Mach number
    :param theta: angle (in radians wrt free-stream direction)
                  of the downstream surface
    :param axi: (int) axisymmetric flag:

                | == 0 : cylinder-wedge
                | == 1 : sphere-cone

    :param 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)
    if axi == 1:
    # Use shock angle on a cone
        beta = beta_cone2(M, theta)
    else:
        # Use shock angle on a wedge
        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."
```

F 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 [52]. The following subsections explain the requirements of configuration files for specific gas models.

F.1 User-defined gas model

The user may define their 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.[23] The job of the function will be to compute the required results based on the input arguments, typically this involves manipulating a supplied `Gas_data` table (see Table 3). The set of functions recognised by a 'user-defined' gas model, along with their arguments lists and return value lists, are given in Table 4. The mandatory functions are listed first,

[23] 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.

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*	dynamic 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

followed by the optional functions. A majority of the functions accept a `Gas_data` table as an argument and also return a `Gas_data` table. The fields in the `Gas_data` table are described in Table 3. Note that the fields for temperature (`T`), internal energy (`e`), species mass fractions (`massf`) and mass fractions per energy mode (`massf_mode`) are vector fields indexed from 0. So, to access what is commonly the translational temperature, one uses `Q.T[0]`. Similarly, the field for binary diffusion coefficients (`D_AB`) is a 2-dimensional array, also using indices beginning from 0. [24] 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 4.

[24] While typical Lua code uses 1-based indexing, the use of 0-based indexing was chosen here so that the user input is consistent with all of the other 0-based indexing used throughout `eilmer3`. Note that this means the Lua `#` operator for returning the length of an array will return the wrong result for the vector fields, and should not be used. Instead, `nsp` and `nmodes` are available globally in the module as they must be set by the user.

Table 4: 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, kg/-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 $\left(\frac{\partial T}{\partial \rho}\right)_p$.

Function	Arguments	Return values	Description
`dpdrho_const_T`	**Q**: `Gas_data` *pre-condition*: thermodynamic state variables are up-to-date	**dpdrho**: `float`	Return $\left(\frac{\partial p}{\partial \rho}\right)_T$.
`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.

F.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.
--
-- Notes:
--   20-Nov-2012 : Updated to compute thermal conductivity
--                 from Prandtl number
--

-- 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 Pr = 0.72

-- 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 thermal_conductivity(T)
   local mu = Sutherland_viscosity(T)
   local k = C_p*mu/Pr
   return
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[0] = Q.e[0]/C_v
   Q.p = Q.rho*R*Q.T[0]
   Q.a = sound_speed(gamma, R, Q.T[0])
   -- 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[0])
   Q.k[0] = thermal_conductivity(Q.T[0])
   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[0][0] = 0.0
   return Q
end
```

F.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 [53, 54]. 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.

F.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 5.

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-ions.lua.gz .`

Table 5: Description of pre-built look-up tables distributed with Elmer3

Pre-built table	Description
`cea-lut-air-ions.lua.gz`	Equilibrium air with ionisation. Useful for Earth reentry problems.
`cea-lua-co2-ions.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. Includes ionisation.
`cea-lut-kr.lua.gz`	Pure Krypton 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-ions.lua.gz`	A Martian atmosphere incluing trace species and ionisation.
`cea-lut-n2-ions.lua.gz`	Pure nitrogen in equilibrium with ionisation.
`cea-lut-titan-like.lua.gz`	A Titan-like atmosphere (N_2 and CH_4, no trace species). No ionisation is considered.
`cea-lut-titan-like-ions.lua.gz`	A Titan-like atmosphere which includes ionisation.

2. Add the following function call to your script:

```
select_gas_model(fname='cea-lua-co2-ions.lua.gz')
```

F.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 5. The tool `build-cea-lut.py` 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 [53, 54], and have that setup in your working area [25].

The `build-cea-lut.py` program has a lot of options. If you invoke it with out any options at all, you get the following text:

```
Begin build-cea-lut.py...
Usage: build-cea-lut.py [options]

Options:
  -h, --help            show this help message and exit
  -g GASNAME, --gas=GASNAME
```

[25] 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`.

```
                        name of built-in gas mixture
  -l, --list-gases      list available gas names and exit
  -c, --custom          build a custom gas model from reactants
  -b BOUNDS, --bounds=BOUNDS
                        bounds of the table in form
                        "T_min,T_max,log_rho_min,log_rho_max"
  -T T_FOR_OFFSET, --T-for-offset=T_FOR_OFFSET
                        Temperature (degree K) at which to evaluate the
                        internal energy offset.

  Custom gas options:
    -r REACTANTS, --reactants=REACTANTS
                        reactant fractions in dictionary form
    -o ONLYLIST, --only-list=ONLYLIST
                        limit species to this list
    -m, --moles         reactant fractions as mole fractions [default]
    -f, --massf         reactant fractions as mass fractions
    -n, --no-ions       excluding ions [default]
    -i, --with-ions     including ions

Example 1: build-cea-lut.py --gas=air5species
Example 2: build-cea-lut.py --custom --reactants="N2:0.79,O2:0.21" --only-list="N2,O2,NO,O,N"
Example 3: build-cea-lut.py --gas=air-ions --bounds="500,20000,-6.0,2.0"
Example 4: build-cea-lut.py --gas=co2 --T-for-offset=650.0 --bounds="1000.0,20000,-6.0,2.0"
Example 5: build-cea-lut.py --gas=co2-ions --T-for-offset=1000.0 --bounds="1000.0,20000,-6.0,2.0"

Sometimes CEA2 has problems and the table will fail to build.
The best approach to fixing the problem seems to be to raise
the lower temperatures, as shown in examples 3, 4 and 5 (above).
```

These options allow you to set bounds on the range of the table, select a gas model from a small library of prespecified gases or to make your own *custom* mixture. The available gases (as at end September 2013) are: "air", "air-ions", "air5species", "air7species", "air11species", "air13species", "n2", "n2-ions", "co2", "co2-ions", "mars-basic", "mars-trace", "mars-trace-ions", "jupiter-like", "titan-like", "titan-like-ions", "h2ne", "h2ne-ions", "ar", and "kr". If you make a custom mixture, you specify the reactants as a dictionary where the keys are species names, as recognised by the `CEA2` program. The `only_list` option can be used to restrict the allowable species in the gas mixture. If it is not specified, `CEA2` is free to choose which species are considered according to its own internal algorithm. To make equilibrium gas models that are consistent with a corresponding finite-rate kinetics model, it would probably be best to supply a value for the `only_list` option.

Upon successful execution of the `build-cea-lut.py`, you will have a compressed (gzipped) Lua file in your working directory. This 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.

G 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.[26]

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 dissociation 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.[27] 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.

[26] 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.

[27] 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.

G.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 pre-defined 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?)[28]

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.

G.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'
}
```

[28] 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.

G.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. These values are used to control the temperature limits at which reaction rate coefficients are evaluated. When the local temperature exceeds the limits (on either side), the rate is simply evaluated at the temperature corresponding to the exceeded limit. As pseudo-code:

```
if T > T_upper
  then T = T_upper
if T < T_lower
  then T = T_lower
eval_rate_coeff(T)
```

Note that these values are set as part of a subtable, `temperature_limits`. The example here shows the current default values if not set by the user.

```
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, unsurprisingly, 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 [55]
`'rkf'`	:	Runge-Kutta-Fehlberg method [56]
`'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.

H Thermal energy exchange mechanisms: specification by configuration file

For thermal nonequilibrium flow simulations, the user may wish to model a set of energy exchange mechanisms operating between the thermal modes. In a similar fashion as for chemical reactions (see Appendix G), thermal energy exchange mechanisms are described in a Lua input file prepared by the user.

As a first example, let's look at an input file for a two-temperature simulation of nitrogen flow. The fact that it is a two-temperature flow is not explicit in the the energy exchange file; this information appears in the accompanying gas model file. The two temperatures are a transrotational temperature (translational and rotational energy modes are assumed to be equilibriated at a common temperature) and a vibroelectronic temperature (the vibrational and electronic energy modes are assumed to equilibriated a common temperature, different from the transrotational temperature). The user-created energy exchange input file lists the mechanism and relaxation times which describe how these two temperature modes relax (or equilibriate) with one another. In this example, we just consider a V-T exchange: a mechanism for the vibrational energy mode to exchange energy with the translational energy mode. The input file is listed here.

```
mechanism{
   'N2 ~~ N2 : V-T',
   rt={'Landau-Teller-cf', A=7.12e-9, B = 124.07, C = 0.0}
}
```

There is only one mechanism listed here. What this says is that in the collision between a nitrogen molecule and another nitrogen molecule, the vibrational energy may be altered and the change in energy is soaked up in the translational mode. This energy exchange occurs at a particular rate which is controlled by a relaxation time. The relaxation time depends on the local thermodynamic state. In this case, the relaxation time is modelled as a curve fit to a Landau-Teller type relaxation. The parameters A, B and C control the shape of the curve fit and have been determined to a give a best fit to experimental measurement of the relaxation time. In this example, there is only one energy exchange mechanism. For certain gas mixtures, there may be several mechanisms of energy exchange amongst the various energy modes. Each of these mechanisms is listed in separate `mechanism` tables, and strictly speaking, there is also the facility to group families of mechanisms in one table (more on that later).

H.1 Overview of the input file format

The Lua programming language is used for the input data description. Any legal Lua code may appear in the energy exchange file. However, the user should not rename the

following special pre-defined functions:

1. `mechanism`
2. `scheme`
3. `ode_solver`

There are supplied default values for the selection of a `scheme` (how the energy exchange relaxation is computed) and the `ode_solver` (if used). These defaults should be adequate for the vast majority of cases. The bulk of the work for the user is usually specifying a set of appropriate `mechanism` entries. The format for a `mechanism` entry is discussed next.

H.2 Details of the `mechanism` table

The `mechanism` table consists of two mandatory fields, and an optional `list` field used in certain circumstances. The first mandatory field is unnamed and always appears first. It is a string describing the particular energy exchange mechanism. The second mandatory entry is a named field `rt` which stands for 'relaxation time'. This field is used to select the model for how the relaxation time of the particular energy exchange mechanism is computed. Thus, the minimal format of the mechanism table is:

```
mechanism{
  'mechanism string',
  rt={...}
}
```

If the `'mechanism string'` makes use of the symbol `(*list)`, then a `list` entry should also appear in the `mechanism` table.

The `'mechanism string'` is used to list which species and which energy modes are involved in a particular energy exchange mechanism. This string must conform to a strict syntax in order to be a valid description of an energy exchange system.[29] The general form of the mechanism string is:

```
'A ~~ <colliders> : modeA-modeC'
```

The first part of the string (before the colon), declares which main species is having one of its energy modes changed due to collisions with certain other species. So in this declaration, our attention is focussed on how a particular energy mode of species A is altered due to collisions with other particles. The second part of the string (after the

[29] For those with an interest in computer programming, the syntactical parsing of the mechanism string is an example of an embedded domain specific language.

colon) tells us which energy modes are affected. There should always be two modes affected: the first corresponds to a mode of species A and the second to a mode of the colliding species. The details and allowable values for the generic fields in the mechanism string are:

`A` is the name of a single species. This is the main species of interest. We are going to consider how collisions of other particles with this species affect the energy in one of its energy modes.

`<colliders>` is the list of colliding species which will affect the energy content of the main species `A`. There are four possible values allowed.

1. a single species name, e.g. `'O2'`
2. a bracketed list of species, e.g. `('O2', 'N2')`
3. the special keyword 'all' to denote collisions with all species in the mix, e.g. `(*all)`
4. the special keyword 'list' to denote collisions with a specific list of species, e.g. `(*list)`. If this value is used, a `list` field should appear in the mechanism table. Basically, this is used to instruct the parser to look in the mechanism table for a list of colliding species.

Options 2, 3 and 4 are means by which to group families of mechanisms into one entry. This can be used when a number of different B colliders all alter the energy state of the A molecule in the *same* way. Internally, the code will expand out the colliders list and treat each A-B interaction pair as a separate mechanism.

`modeA-modeC` This is a string which denotes which mode of collider A is altered during the collision and which mode of the other colliders is altered. The possible values for this string are:

`V-T` A vibration-translation energy exchange between vibrational mode of collider A and the translational energy of the colliding partners.

`V-V` A vibration-vibration energy exchange between the vibrational mode of collider A and the vibrational mode of another collider. Whenever this entry is present for a pair A-B, there should usually be a reciprocal `mechanism` listed. For example, a V-V exchange for N_2-O_2 should have a matching V-V exchange written for O_2-N_2.[30]

[30] The user might think that it is redundant having to specify two mechanisms for reciprocal pairs of V-V exchanges. There is a subtle reason for this: the relaxation times calculated for V-V exchanges are the relaxation time for an upper vibrational energy level of collider A to drop down a level due to collisions with collider B, and at the same time the vibrational energy level of collider B is raised a

`E-T` An electron-translation energy exchange. This is actually a translation-translation energy exchange. It is the exchange of translational energy of the electron species with the translational energy of the heavy particles.

When writing the mechanism string, the guiding rule is that it is written from the perspective of collider A. You are listing how collisions with other particles affect a certain energy mode of collider A.

Next we describe the `rt` field which is required as part of specifiying a mechanim. The `rt` field is used to select a model for the relaxation time related to the particular mechanism. For example, the Landau-Teller relaxation time model was selected in the first example by setting

```
rt={'Landau-Teller-cf', A=7.12e-9, B=124.07, C=0.0}
```

The value for the `rt` field is always a table. The first entry of this table is always a string which denotes a particular relaxation time model. The remaining key-value pairs in the table are specific to the chosen model. The relaxation time model must be appropriate for the type of mechanism. So for V-T exchanges, there is a certain set of relaxation time models available. For V-V exchanges, there is a different set of relaxation time models availables, as so on for other energy exchange mechanism types. The list of available relaxation time models and their required key-value pairs are grouped accodring to mechanism type in Table 7. Any keys which are enclosed in bracket [] are optional values. There will usually be a default method to compute the optional values if not supplied.

level. However, we have not looked at the relaxation time for the process of an upper vibrational level of B dropping due to collisions with A, and the accompanying promotion of the vibrational energy level of A. This will have a different relaxation time associated with the process, and so requires a separate `mechanism` entry.

Table 7: Relaxation time models for energy exchange mechanisms

Model	Format
	— for V-T exchanges —
Millikan-White $\tau = (1/p_{\text{bath}})\left(\exp(aT^{-1/3} - b) - 18.42\right)$	`{'Millikan-White', [a=...], [b=...]}` • `'Millikan-White'` appears first to name the model • `a` is a constant of the model. If not supplied it can be calculated based on the reduced mass (μ) of the colliders and the charateristic vibrational temperature (Θ_v) of collider A as $a = 1.16e^{-3}\sqrt{\mu}\Theta_V^{4/3}$. • `b` is a constant of the model. If not supplied, it can be computed based on the reduced mass (μ) as $b = 0.015\mu^{1/4}$.
Millikan-White with a high-temperature correction	`{'Millikan-White:HTCS', [a=...], [b=...], HTCS={}}` Parameters `a` and `b` as above. • `'Millikan-White:HTCS'` appears first to name the model • `HTCS` is a model for the high-temperature correction cross-section. Allowable values are: `'Park'` The user also supplies a value for `sigma_dash`. So the selection looks like `HTCS={'Park', sigma_dash=3.0e-17}`. `'Fujita'` In this case, no other parameters are required. The selection is `HTCS={'Fujita'}`.

Model	Format
Landau-Teller curve fit $\tau = (A/p_{bath}) \exp(B/T^{1/3} + C)$	{'Landau-Teller-cf', A=..., B=..., C=...} • 'Landau-Teller-cf' appears first to name the model • A is a constant of the model. • B is a constant of the model. • C is a constant of the model.
Schwartz-Slawsky-Herzfeld relaxation time model for V-T transfers	{'SSH-VT'} This model uses molecular parameters to compute the relaxation time for V-T transfers. No other information is required from the user.
	— *for V-V exchanges* —
Schwartz-Slawsky-Herzfeld relaxation time model for V-V transfers	{'SSH-VV'} This model uses molecular parameters to compute the relaxation time for V-V transfers. No other information is required from the user.
	— *for E-T exchanges* —
Appleton-Bray model for ions Appleton-Bray model for neutrals Appleton-Bray model for neutrals with two ranges	... ...

```
scheme_t = {...}

ode_t = {...}

rates = {...}

equilibriation_mechanisms = {...}
```

The `scheme_t` table defines the scheme that will be used to model the energy exchange update during a timestep. The table should have the following format:

```
scheme_t = {
    update ='energy exchange ODE',
    temperature_limits = {
        lower = 20.0,
        upper = 100000.0
    },
    error_tolerance = 0.000001
}
```

`update`
: A string defining the update method. Presently the only available option is `energy exchange ODE`, where the energy exchange update is modelled via solving a system of ordinary differential equations.

`temperature_limits`
: Specifies the range of *translational* temperatures where thermal energy exchange is permitted to occur. The fields `lower` and `upper` expect floating point values.

`error_tolerance`
: Although not currently used in the code, a floating point value is expected in this field.

The `ode_t` table defines parameters for controlling the ODE solver used during the energy exchange update. Note this has the same format as the `ode_solver` table in the chemistry input file described in Appendix G. The table should have the following format:

```
ode_t = {
    step_routine = 'rkf',
    max_step_attempts = 4,
    max_increase_factor = 1.15,
```

```
    max_decrease_factor = 0.01,
    decrease_factor = 0.333
}
```

`step_routine`

A string specifying the desired ODE stepping method. The available methods are:
`'qss'` : Mott's α-QSS method [55]
`'rkf'` : Runge-Kutta-Fehlberg method [56]
`'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 thermal 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.

The `rates` table lists the thermal energy exchange mechanisms to be considered for each thermal mode *except the primary mode*[31]. Therefore one entry is expected for a two temperature model, two entries for a three temperature model, etc. For a three temperature model, for example, where the list of thermal modes in the `gas-model.lua` file reads:

```
thermal_modes = { 'transrotational', 'vibrational', 'electronic' }
```

the table should have the following format:

```
rates = {
    {
```

[31]The energy of the primary thermal mode is solved for by enforcing the conservation of total energy during the thermal time-step.

```
        -- vibrational mode
        mechanisms = {...}
    },
    {
        -- electronic mode
        mechanisms = {...}
    }
}
```

where the first table entry is for the vibrational thermal mode, whilst the second table entry is for the electronic thermal mode. The `mechanisms` tables list the thermal energy exchange mechanisms to be applied to the respective thermal modes. The mandatory items for a `mechanisms` table entry are:

`type`
: A string specifying the type of energy exchange mechanism. The available types are:
  `'VT_exchange'` : Vibration-translation exchange
  `'ET_exchange'` : Electron-translation exchange

`relaxation_time`
: A table listing the parameters for the relaxation time model.

When specifying a `'VT_exchange'` mechanism, an additional field `'p_name'` that indicates the name of the vibrating species is required. A detailed description of the `relaxation_time` table will be available in a future version of this user guide. For the moment, please refer to the following example as a basic guide.

Below is the thermal energy exchange Lua input file for dissociating and ionising nitrogen described by the two temperature model (see Section 53.2 for an example simulation using this model). The gas consists of five species, namely N_2, N_2^+, N, N^+ and e^+, and two thermal modes, translation-rotation and vibration-electron-electronic. Two thermal energy exchange mechanisms are specified: vibration-translation exchange due to inelastic collisions with the N_2 molecule, and electron-translation exchange due to elastic collisions between free-electrons and heavy particles.

```
rates = {
    {
        mechanisms = {
            {
                type = 'VT_exchange',
                p_name = 'N2',
                relaxation_time = {
                    type = 'VT_MillikanWhite_HTC',
                    HTCS_model = {
```

```
                        type = 'Park',
                        sigma_dash = 3.0e-17
                    },
                    p_name = 'N2',
                    q_names = { 'N2', 'N' },
                    a_values = {  -1,   -1 },
                    b_values = {  -1,   -1 }
                }
            },
            {
                type = 'ET_exchange',
                relaxation_time = {
                    type = 'ET_AppletonBray',
                    ions = {
                        { c_name = 'N_plus', },
                    },
                    neutrals = {
                        { c_name = 'N', sigma_coefficients = { 5.0e-20, 0.0, 0.0 } },
                        { c_name = 'N2', sigma_coefficients = { 7.5e-20, 5.5e-24, -1.0e-28 } },
                    }
                }
            }
        }
    }
}
```

I User-defined functions for run-time customization

User-defined functions (UDFs) are callable functions written in Lua that are used to perform specialized and/or customized tasks.[32] These callable functions can be used for:

- specialized boundary conditions;
- the addition of custom source terms; and
- to perform special operations at the beginning and end of each timestep.

Some examples follow to give this idea a more concrete form. A specialized boundary condition might model mass injection from a porous boundary which is not presently available as a boundary condition in the simulation code. We use custom source terms when we are testing the code using the method of manufactured solutions (see Sections 42 and 43). The callable functions at the start and end of each timestep could be used to compute a special flow field variable.

I.1 Customizing the boundary conditions

Using a customized boundary condition requires two steps:

1. Selecting the `UserDefinedBC()` in the block setup.
2. Constructing a Lua file which defines the boundary condition behaviour.

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 minimum. 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.

When using the user-defined boundary conditions you need to instruct the code about what to do for the convective (inviscid) update and then, separately, for the viscous effects. The inviscid interaction at the boundary may be handled in one of two ways:

1. Defining a `ghost_cell()` function.
   In this case, you populate the properties of two ghost cells such that they give the desired inviscid effect at the wall. The ghost cells are abstract in that they do not

[32] 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.

exist in the simulated flow domain but do exist in the code data for each block boundary. They are used in the interpolation phase of the convective update, for cell faces that lie along the boundary. For the case of a solid wall, you use the `ghost_cell()` function and reflect the normal velocity. Examples of this are in the test cases.

2. Defining a `convective_flux()` function.
   This is an alternative to the `ghost_cell()` function and allows you to directly specify the convective flux. This function is only used if the `sets_conv_flux_flag` is set in the boundary condition. If it is set, the `convective_flux()` function will override anything in the `ghost_cell()` function thus causing the `ghost_cell()` function to have no effect (however, it is still needs to be present due to the way the implementation works).

The viscous effects at the boundary are also handled in one of two ways:

1. Defining an `interface()` function.
   In this case, you set the properties at the interface directly and, as part of the viscous update, the main code computes spatial derivatives from these specified flow properties. For example, you could set a temperature at the interface and zero velocity for a no-slip wall with the function called `interface()`. By doing this, you would not directly control the viscous heat flux into the flow directly, however, it would be controlled indirectly by setting the temperature.

2. Defining a `viscous_flux()` function.
   The other option is to specify the viscous flux directly at the boundary. The function inputs and outputs are identical to the `convective_flux()` function, except that the values for viscous fluxes of conserved quantities are returned. This option is convenient when something is directly known about the viscous flux effect at the boundary. For example, a heat flux at the boundary may be specified directly using this user-defined function. This function is only used if the `sets_visc_flux_flag` is set in the boundary condition, otherwise the code will just look to apply the `interface()` function.

Note that in an inviscid simulation, any user-specied viscous boundary effect functions are ignored: they are never called by the code.

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:

`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 (see below) 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 current process. This function may be called with indices which sample the properties in the ghost cells themselves. When this is the case, the flow properties in the ghost cells should not be relied on. The only useful data is the position (`x`, `y` and `z`) and the volume `vol`. These values are estimated by using a linear extrapolation from the nearby interior cells. The values of position and volume may be useful when setting the properties in the ghost cells (see for example the application in MMS case to give a first-order boundary condition).
`sample_i_face(jb,i,j,k)`	a function that returns a table of the flow state for a particular I-interface. The data is the same as that listed for the `ghost_cell` tables with the addition of `length`, the interface. This function is not likely to work for a MPI simulation, where only one block is visible to the current process.
`sample_j_face(jb,i,j,k)`	As for `sample_i_face()` except that the properties are returned for a J-interface.
`sample_k_face(jb,i,j,k)`	As for `sample_i_face()` except that the properties are returned for a K-interface.
`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 current process.

There are some additional convenience functions available to the user to compute or obtain values related to the gas model such as thermodynamic properties and transport coefficients. These are discussed in detail in Section I.4.

On being called at run time, the function `ghost_cell(args)` returns two Lua tables. It is the user writing the function who is responsible for constructing and returning these two 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`,...)

Note that the `ghost_cell` function is called once for every cell along the boundary, so be mindful of the possibility of repeating calculations that remain fixed across the full boundary. It may be efficient to do the calculation once, at the time the function is called for the first cell, and store the resulting data in global variables so that they are ready for use in subsequent calls.

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:

`massf`	table of `nsp` mass fractions. The zero entry, at least, must be specified.
`T`	table of `nmodes` temperatures to be set at interface, possibly a wall.
`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 all of the same attributes as for the call to the `ghost_cell` function. Additionally, `args` contains:

`dt`	the current global timestep, in seconds
`t_level`	an integer denoting the level within the explicit update
`area`	the interface area (at `t_level`, which is important for moving grid simulations)
`fs`	a table containing the present flow state data for the interface. Note that typically the user will provide new flow state data at the end of the function.

The flow state, `fs`, is table with the following flow properties:

`p`	pressure, Pa
`rho`	density, kg/m^2
`u,v,w`	velocity components in x,y,z-directions, m/s
`a`	sound speed, m/s
`mu`	molecular (dynamic) viscosity
`k`	a table of `nmodes` thermal conductivities
`mu_t`	turbulent viscosity
`k_t`	turbulent heat conduction coefficient
`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
`S`	shock-detector value (1 or 0)

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 functions `convective_flux(args)` and/or `viscous_flux(args)` that return a table specifying the interface fluxes, convective and viscous respectively, that are 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`,...)

A note on orientation of fluxes

When setting flux values, the user is responsible for giving the magnitude of flux that crosses normal to the boundary interface. As such, the user's function is given the com-

ponents of the interface normal vector in the Cartesian frame (`nx`, `ny`, `nz`) to aid in computing the correct flux magnitude for interfaces of arbitrary orientation. The positive sense for the unit normal is shown for two-dimensional boundaries in Figure 150. In words, the normals point inwards for the `WEST` and `SOUTH` boundaries, and the normals point outwards for `EAST` and `NORTH`. For example, if you are setting a flux that crosses the `NORTH` boundary and enters the domain, the magnitude of its value should be *negative* to indicate flux *into the domain*. The same holds for fluxes across the `EAST` boundary.

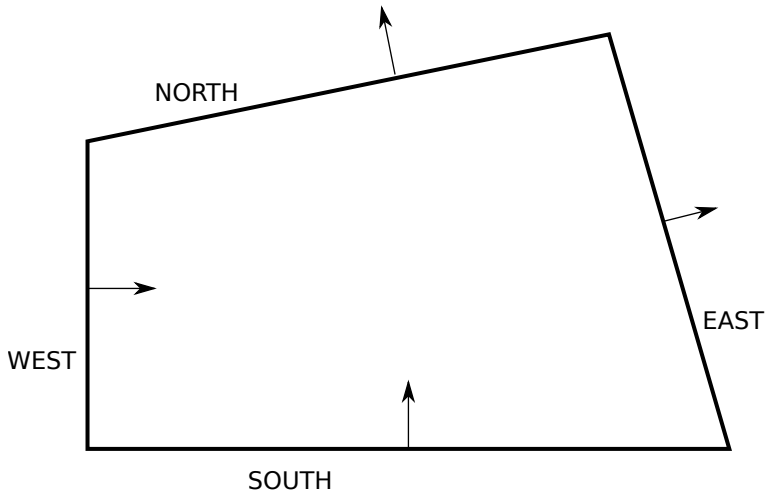


Figure 150: The positive sense of direction for unit normals at each of the boundaries in 2D.

The reason for this arrangement of face-normals is that, internal to the code, all `EAST` and `WEST` interfaces are part of the single array of `i-faces`. For `NORTH` and `SOUTH`, there is the single array of `j-faces` and, for `TOP` and `BOTTOM` faces, there is the array of `k-faces`. So, a single `i-face` will serve as the `EAST` face of one cell and the `WEST` face of the next cell to its right.

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`. These latter two functions are described in Section I.3. 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

The

Lua execution environment also includes information about the number of blocks and their configuration. We do not discuss this further here because this information is often not that useful for source vector specification. More details about the block information are given in Section I.3 where the `at_timestep_start()` and `at_timestep_end()` functions are discussed.

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
`mu`	gas viscosity
`T`	table of `nmodes` temperatures
`k`	table of `nmodes` thermal conductivities
`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

I.3 Callable functions at timestep start and timestep end

The callable functions at timestep start and timestep end differ from the user-defined boundary conditions and user-defined source terms in two key ways:

1. the functions are only called once on each timestep iteration; and

2. the functions are used to extract information from the flow field but cannot alter its state in anyway (nothing is returned to the C++ code)

In the case of the callable boundary conditions, the functions are called many times each timestep for each of the interfaces. Similarly, the callable source vector function is called once for every cell in the flow field. However, the user-defined functions `at_timestep_start()` and `at_timestep_end()` are only called once in each iteration.

As such, if the user would like to gather data from all cells, then the user is responsible for looping over those cells.

In the present implementation, we do not provide a mechanism to alter the state of the cells and boundaries in the flow field via the functions `at_timestep_start()` and `at_timestep_end()`. The callable functions are only intended to extract information from the flowfield.[33]

The global Lua environment for the `at_timestep_start()` and `at_timestep_end()` functions is the same as that for the user-defined source terms (see Section I.2). We did not introduce earlier the globally-set variables related to block information. These variables are:

`nblks`	number of blocks (in MPI process)
`blks`	array of block tables, each with block configuration

Note that the number of blocks is given on a per MPI rank basis and the list of blocks are the blocks associated with that rank.

One of the global variables, `blks`, is an array (or table in Lua speak) containing information about each of the blocks individually. To access the information in the second block, for example, you could interrogate the table `blks[1]`, using 1 for the second block because of 0-offset indexing. So each entry in `blks` is a table itself. Each table within `blks` contains the following information about a particular block:

`id`	the unique block id with the global collection of blocks
`nicells`	number of cells in i-direction
`imin`	index of minimum cell number in i-direction
`imax`	index of maximum cell number in i-direction
`njcells`	number of cells in j-direction
`jmin`	index of minimum cell number in j-direction
`jmax`	index of maximum cell number in j-direction
`nkcells`	number of cells in k-direction
`kmin`	index of minimum cell number in k-direction
`kmax`	index of maximum cell number in k-direction

When either of the timestep functions is called, it is passed a table of arguments from the flow solver. That table contains:

`t`	current simulation time in seconds
`step`	current step number in simulation

The brief example below shows how we could loop over all the cells in a flowfield and get a tally of the total mass at the start of the first timestep using the `at_timestep_Start()` function. This example appears in the file `udf-process.lua` as part of the example in Section 40.

```
function at_timestep_start(args)
```

[33]Note the cunning user could use all of the Lua callable functions to affect the flow field. For example, the user could extract certain information from the flow field using the `at_timestep_end()` hook, make a decision based on that information, write some data to a local file and then have a user-defined boundary condition pick up and act on that data. An early implementation of a conjugate heat transfer boundary condition was done in a similar manner to this.

```
    if (args.step ~= 0) then
        -- do nothing, just leave
        return
    end
    -- For the 0th step only
    mass = 0.0
    for ib=0,(nblks-1) do
        imin = blks[ib].imin; imax = blks[ib].imax
        jmin = blks[ib].jmin; jmax = blks[ib].jmax
        blk_id = blks[ib].id
        for j=jmin,jmax do
            for i=imin,imax do
                cell = sample_flow(blk_id, i, j, k)
                -- We are only given p and T
                -- so need to compute density
                -- using gas model
                Q = create_empty_gas_table()
                Q.p = cell.p
                Q.T = cell.T
                for isp=0,(nsp-1) do Q.massf[isp] = cell.massf[isp] end
                eval_thermo_state_pT(Q)
                rho = Q.rho
                -- Now we can compute mass in cell using volume of cell
                mass = mass + rho*cell.vol
            end
        end
    end
    print("Mass (kg) of gas in domain: ", mass)
    return
end
```

There's a little bit to digest in the example above. We'll begin with the `if`-statement. Remember that the `at_timestep_start()` is called for every timestep, which means we enter this piece of code on every iteration. However we only want to compute the mass at the very beginning of the simulation. So, the `if`-statement says that if we are *not* at step 0 (the beginning step), then do nothing and move on. The code only continues then in the case where the step number is equal to 0.

In the case where the step number is 0, we want to loop over all cells and tally the mass. To do that, we firstly need to know how many blocks there are in the simulation.

(Admittedly, we might know how many blocks there are already because we set the simulation up ourselves! However, by keeping the code general we can reuse it for other simulations without alterting the Lua code.) We can get the number of blocks from the global environment variabl (supplied by the C++ code) `nblks`. Then we loop over all blocks using the `ib` variable as a counter for the block index. Within any particular block, we want to loop over the simulation cells only, and exclude any ghost cells at the boundaries. The appropriate ranges for the simulation cells in each of the i-, j- and k-directions are given by the `min` and `max` variables within each block table. Having extracted those values, we can set up loops to visit every simulation cell in a block.

Be careful to note that the `sample_flow()` function requires the *global* block id. This is may not be the same as the variable `ib` in an MPI simulation where different processes work on collections of blocks. To ensure that we supply `sample_flow()` with the correct global block id, we retrieve that id value from the `blks` table and store it as `blk_id`.

In the inner most loop, we visit every cell and extract its density and volume so that we can compute the mass in the cell. We call the `sample_flow()` function to get the information of a single cell. To compute the density is a little complicated. We are only given pressure, temperature and species mass fractions. The provided gas model functions are used to compute density. For the moment, don't worry too much about the details of making the calculation to get density. These functions are explained later in Section I.4. The volume is easy to get: we extract directly from the cell as variable `vol`. In the last step, we compute the mass in this cell ($\rho \times V$) and add it to the total.

I.4 Helper gas model functions

There are a large number of functions provided by the gas module to the internal (C++) section of the code. For consistency with the internal gas model, a selection of the gas module functions are made available to the Lua run-time scripts. The names of these Lua-exposed functions match the internal C++ names very closely (and in fact, identically in most cases). The provided gas model functions are:

`create_empty_gas_table()`	Returns an empty `Gas_data` table with all entries set to 0.0 and appropriately sized internal arrays. This is useful to populate and pass to other functions which accept a `Gas_data` table.
`eval_thermo_state_pT(Q)`	A function that computes the thermodynamic state given the pressure and temperatures as set in the `Gas_data` table `Q`. The thermodynamic properties are updated and returned in place in the `Q` variable, that is, it is modified in place.
`eval_thermo_state_rhoe(Q)`	A function that computes the thermodynamic state given density and internal energy. The `Gas_data` table `Q` is modified in place.
`eval_thermo_state_rhoT(Q)`	A function that computes the thermodynamic state given density and temperatures. The `Gas_data` table `Q` is modified in place.
`eval_thermo_state_rhop(Q)`	A function that computes the thermodynamic state given density and pressure. The `Gas_data` table `Q` is modified in place.
`eval_sound_speed(Q)`	A function that computes the sound speed based on the supplied thermodynamic state in `Q`. The `Gas_data` table `Q` is modified in place such `Q.a` contains the computed sound speed value.
`eval_transport_coefficients(Q)`	A function that computes the transport coefficients, viscosity and thermal conductivities, based on the supplied gas state in `Q`. The `Gas_data` table `Q` is modified in place so that `Q.mu` and `Q.k[]` are up to date.
`eval_diffusion_coefficients(Q)`	A function that computes the diffusion coefficients for interacting species pairs based on the thermodynamic state in `Q`. The values in `Q.D[][]` are modified in place so that they are up to date.
`eval_Cv(Q)`	A function that returns (as a double) the mixture specific heat at constant volume (in J/(kg.K)) based on the supplied thermodynamic state in `Q`.
`eval_Cp(Q)`	A function that returns (as a double) the mixture specific heat at constant pressure (in J/(kg.K)) based on the supplied thermodynamic state in `Q`.
`eval_R(Q)`	A function that returns (as a double) the mixture gas constant (in J/(kg.K)) based on the supplied thermodynamic state in `Q`.
`eval_gamma(Q)`	A function that returns (as a double) the ratio of specific heats (non-dimensional) based on the supplied thermodynamic state in `Q`.

`molecular_weight(isp)`	A function that returns the molecular weight of species number `isp`. The units of molecular weight is returned in kg/mol because this is consistent with the internal units of the code. Note that the units of molecular weight listed on the Periodic Table and commonly used in textbook formulas is in g/mol. The returned value should be multiplied by 1000.0 to give g/mol.
`enthalpy(Q, isp)`	A function that returns the enthalpy in J/kg of species *isp*. The enthalpy is computed based on the supplied gas state in Q. When using the thermally perfect gas mix, the enthalpies of formation can be obtained by evaluating the enthalpy at $T = 298.15\,\mathrm{K}$.
`massf2molef(massf)`	A function that returns a table of mole fractions based on a supplied table of mass fractions. Note the table of supplied mass fractions must be the full size of the number of species in the gas model. Similarly, the returned mole fractions table has values for all participating species.
`molef2massf(molef)`	A function that returns a table of mass fractions based on a supplied table of mole fractions. Note the table of supplied mole fractions must be the full size of the number of species in the gas model. Similarly, the returned mass fractions table has values for all participating species.
`massf2conc(rho, massf)`	A function that returns a table of concentrations ($\mathrm{mol/m^3}$) based on a supplied density and table of mass fractions. Note the table of supplied mass fractions must be the full size of the number of species in the gas model. Similarly, the returned concentrations table has values for all participating species.
`conc2massf(rho, conc)`	A function that returns a table of mass fractions based on a supplied density and table of concentrations in $\mathrm{mol/m^3}$. Note the table of supplied concentrations must be the full size of the number of species in the gas model. Similarly, the returned mass fraction table has values for all participating species.
`species_rate_of_change(Q)`	A function that returns a table of the time rate of change of species concentrations based on the reaction scheme and current gas state and composition. The returned values have units of $\mathrm{mol/(m^3.s)}$. Note that the thermodynamic state and composition in the gas data `Q` must be filled with up-to-date values. You can ensure this by making an appropriate call to one of the thermo eval functions before passing `Q` to this function.

I.5 Notes on global variables and Lua interpreters

For each boundary condition that uses a `USER-DEFINED` boundary condition, an independent Lua interpreter is started. The global state in each of these interpreters (read boundary conditions) is kept between timesteps (*i.e.* the interpreter is reentrant). However, there is no way to communicate information internally from one Lua interpreter to another. There is a subtlety here. You could actually write just one Lua file as the boundary condition but set it on multiple boundaries however, you would need to make it smart enough to use the Eilmer-provided information to work out which boundary it was and then act accordingly. Remember that, although you might use the one file, it is running as an independent process for each boundary. Those independent processes will not share global state and cannot communicate.

An independent Lua interpreter is also started when using the global `udf_file` to supply `at_timestep_start()` and `at_timestep_end()` functions. A single interpreter is started to house both those functions and the global state in that interpreter is also reentrant.

J Hints for Solution Visualisation with ParaView

J.1 Plotting Streamlines and Streamtubes

The following steps can be used to visualise streamlines and streamtubes in ParaView.

1. Postprocess simulation results with the `--vtk-xml` flag as described in Section 3.7 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. Convert the cell data to point data (at the cell nodes) by applying the filter `Filters > Alphabetical > Cell Data to Point Data` and once again clicking `Apply`. For multi-block simulations, the user must also apply the filters `Group Data Sets` and `Merge Blocks` as described in Section 52.

4. Now the streamlines 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 55 is illustrated in Figure 151.

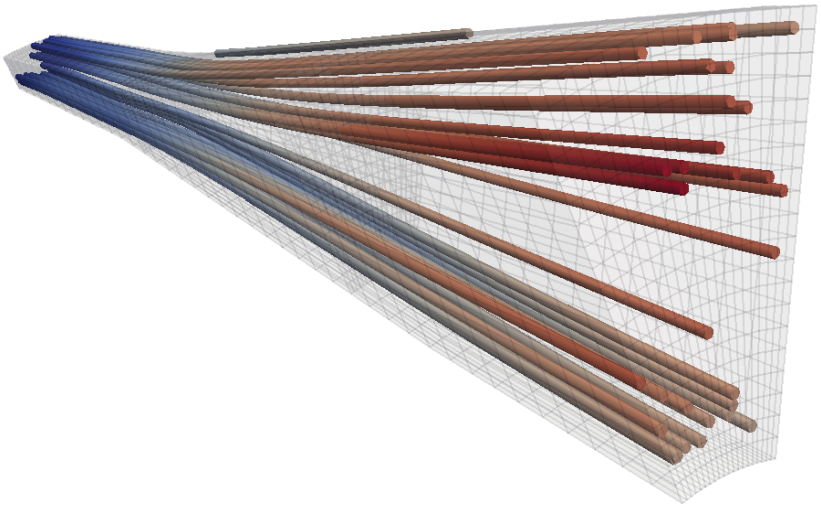


Figure 151: Streamtubes passing through Katsu's scramjet combustor and nozzle.

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 152. 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.

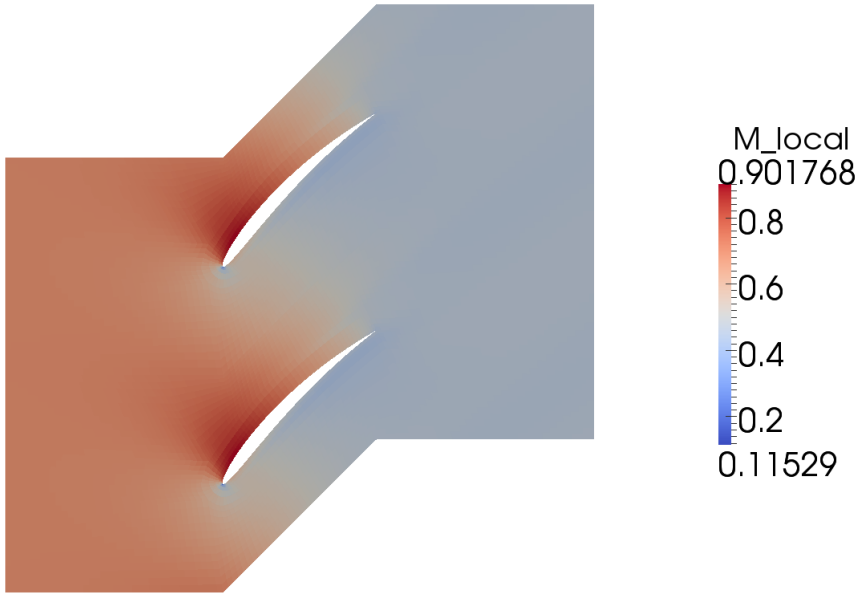


Figure 152: Standard Configuration 10 Mach field illustrating correct operation of periodic boundary condition.

K Load balancing MPI simulations

Consider a parallel simulation with 16 blocks which you wish to run on 16 processes. Due to the geometry, the 16 blocks are not of equal size. For example, 2 of the blocks are twice as large (have twice the number of cells) as the other 14 blocks. When running this simulation in parallel over 16 processes, there is a degree of inefficiency. The code needs to synchronise the exchange of block-boundary data at the end of every timestep, so the 14 smaller blocks are spending roughly 50% of their compute time just waiting on the two larger blocks to complete their calculations. This type of inefficiency is not a big deal on your own machine with 16 cores but it will make you unpopular on large shared resource machines such as shared-memory supercomputers and supercomputing clusters. To alleviate some of this inefficiency, it's possible to run Eilmer3 in a mode where several blocks are handled by just one MPI process. In the example here, we would assign several of the smaller blocks to just one MPI process. In the end, we could use 9 MPI processes instead of 16, placing each of the two large blocks on their own MPI process and then assigining two of the smaller blocks to each of the remaining MPI processes. We do this with an MPImap file which is explained below.

The other case in which we might want to use the MPImap feature is when we are running a simulation with many blocks (on the order of 100s of blocks) which exceeds the number of processes available. In this case, the only way to start a parallel simulation is by using the MPImap mode to assign multiple blocks to one MPI process.

Using the MPImap feature is a three-step process. First, we prepare an Eilmer3 simulation as usual, using `e3prep.py`. Second, a mapping file is built with the `e3loadbalance.py` program. Third, when running `e3mpi.exe` the mapping file is supplied as an option using the `mpimap=` flag. As an example, consider the simulation included in `examples/eilmer3/3D/load-balance-test`. This contains a GridPro grid with 27 blocks. After running `e3prep.py`, we are ready to use the load-balance program to map blocks to processes. We wish to run this simulation on 16 MPI processes, so first we call `e3loadbalance.py` to build an MPImap file:

```
> e3loadbalance.py --job=test -n 16
```

We provide two options: we give the base file name to the `--job` option and supply the desired number of MPI processes with the `-n` option. The number of MPI processes should be less than (or equal to) the number of blocks. After running this, an `mpimap` file is created. In this case, it's called `test.mpimap`. The contents of that file are shown here. It is an INI-type file which lists which blocks have been assigned to which MPI process (called 'ranks' in the MPI terminology).

```
[global]
nrank = 16
[rank/0]
nblock = 1
blocks = 13
[rank/1]
nblock = 1
blocks = 9
[rank/2]
nblock = 1
blocks = 11
[rank/3]
nblock = 1
blocks = 15
[rank/4]
nblock = 1
blocks = 16
[rank/5]
nblock = 1
blocks = 14
[rank/6]
nblock = 2
blocks = 5 19
[rank/7]
nblock = 2
blocks = 6 21
[rank/8]
nblock = 2
blocks = 7 23
[rank/9]
nblock = 2
blocks = 8 24
[rank/10]
nblock = 2
blocks = 10 26
[rank/11]
nblock = 2
blocks = 0 12
[rank/12]
nblock = 2
blocks = 1 17
[rank/13]
nblock = 2
blocks = 3 18
[rank/14]
nblock = 2
blocks = 2 25
[rank/15]
nblock = 3
blocks = 4 20 22
```

Now to run this simulation, we would invoke `e3mpi.exe` in the following manner:

```
> mpirun -np 16 e3mpi.exe --job=test --mpimap=test.mpimap --run
```

Note the new option `--mpimap=` which we haven't seen before. This supplies the name of the mapping file to use.

The advantage to having `e3loadbalance.py` as a separate step is that you can re-configure how you want your blocks mapped to MPI processes without re-prepping the simulation. Say you had a 200-block simulation but could only reliably get 16 CPUs on a cluster machine one week, then you could build a mapping file for 16 MPI processes. If

later on, you want to re-run the same simulation but there are now 64 CPUs available, you could rebuild the mapping file. Just build a new mapping file for 64 MPI processes and supply that to the `e3mpi.exe` command.

The algorithm used to do the load balancing does not guarantee the optimal arrangement for mapping of blocks to MPI processes, but it can be shown that it gives a very good load balancing for minimal computational expense [57]. The optimal arrangement is possible to compute by brute force (trying every combination of block arrangement for processes) but that is computationally very expensive. There is an extra option for the `e3loadbalance.py` program which will give some measure of the quality of the load balancing based on the selected number of processes. The program will actually sweep over a range of numbers of processes. For example, let's see how the load balancing looks for this 27 block case if we vary the number of processes from 2 to 27.

```
> e3loadbalance.py --job=test -n 16 --sweep-range=2:27
```

The results of this sweep are written to the file `load-balance.dat`. It's a simple text file with four columns of data: (1) number of processes; (2) Δ_{cells}, the difference between the process with the largest number of cells to the process with the smallest number of cells; (3) packing quality, computed as $1.0 - \frac{L_{\max} - L_{\min}}{L_{\max}}$ where L is the load (based on number of cells assigned to a process); and (4) estimated speedup, computed as $\frac{L_{\text{total}}}{L_{\max}}$. The contents of this file are displayed here:

```
# nprocs Delta_cells  packing-quality  speedup
002  32     0.993610      1.993610
003  64     0.980952      2.971429
004  48     0.980892      3.974522
005  64     0.968504      4.913386
006  64     0.962617      5.831776
007  240     0.850000      6.240000
008  416     0.740000      6.240000
009  576     0.640000      6.240000
010  704     0.560000      6.240000
011  800     0.500000      6.240000
012  896     0.440000      6.240000
013  976     0.390000      6.240000
014  1072     0.330000      6.240000
015  1120     0.300000      6.240000
016  1184     0.260000      6.240000
017  1216     0.240000      6.240000
018  1280     0.200000      6.240000
019  1312     0.180000      6.240000
020  1344     0.160000      6.240000
021  1376     0.140000      6.240000
022  1440     0.100000      6.240000
023  1440     0.100000      6.240000
024  1472     0.080000      6.240000
025  1472     0.080000      6.240000
026  1536     0.040000      6.240000
027  1536     0.040000      6.240000
```

Note that there is no benefit to choosing more than 7 processes for this simulation. We

see that beyond 7, the speedup remains constant and that the packing quality starts to drop rapidly. What this essentially means is that we have got to the point where one large block (or possibly several) is dominating the load balancing. This large block is given one process to itself, the remaining blocks are smaller such that when combined they are still not as large as the largest block. By increasing more processes, you are effectively only sharing the smaller blocks around more processes. You are still limited by the one large block dominating the load. If this really is limiting achieving a good load-balancing, then the solution is to divide that block at the gridding stage. It might be possible to subdivide the block using the `SuperBlock` option in Eilmer3.

L Radiation transport models

A variety of radiation transport models are implemented in Eilmer3:

- Optically thin model
- Tangent slab model
- Modified discrete transfer model
- Photon Monte-Carlo model

The radiation transport model is defined the `transport_data` table of the radiation Lua input file. Note that all radiation transport models also require a radiation spectral model to run. See § 8 of the Photaura Users Guide (http://cfcfd.mechmining.uq.edu.au/pdf/photaura-users-guide.pdf) for a detailed explanation of how to setup a radiation spectral model via the tools provided in the cfcfd3 radiation library.

L.1 Optically thin model

The optical thin radiation transport model is selected by setting the `transport_model` field in the `transport_data` field to `"optically thin"`. The following code snippet gives an example of selecting and defining the parameters for the optically thin model:

```
transport_data = {
   transport_model = "optically thin",
   spectrally_resolved = true
}
```

A description of the Lua input fields for the optically thin radiation transport model is given in Table 8.

Table 8: Description of Lua input fields for the optically thin radiation transport model

Field	Type	Description
`spectrally_resolved`	*bool*	Flag to request a spectrally resolved or unresolved determination of the radiative power density

L.2 Tangent slab model

The tangent slab radiation transport model is selected by setting the `transport_model` field in the `transport_data` field to `"tangent slab"`. No other input parameters need to be set. The following code snippet gives an example of selecting and defining the parameters for the tangent slab model:

```
transport_data = {
    transport_model = "tangent slab"
}
```

L.3 Modified discrete transfer model

An implementation of the modified discrete transfer radiation transport model is selected by setting the `transport_model` field in the `transport_data` field to `"discrete transfer"`. The following code snippet gives an example of selecting and defining the parameters for the discrete transfer model:

```
transport_data = {
    transport_model = "discrete transfer",
    nrays = 32,
    clustering = "by volume",
    binning = "opacity",
    N_bins = 10
}
```

A description of the Lua input fields for the modified discrete transfer radiation transport model is given in Table 9.

Table 9: Description of Lua input fields for the modified discrete transfer radiation transport model

Field	Type	Description
`nrays`	*int*	Number of rays emitted per cell and per frequency interval
`clustering`	*string*	Ray clustering: `by volume`, `by area` or `none`
`binning`	*string*	Binning model: `opacity`, `frequency` or `none`
`N_bins`	*int*	Number of bins (does not need to be set if `binning = "none"`)

L.4 Photon Monte-Carlo model

The photon Monte-Carlo radiation transport model is selected by setting the `transport_model` field in the `transport_data` field to `"monte carlo"`. The following code snippet gives an example of selecting and defining the parameters for the photon Monte-Carlo model:

```
transport_data = {
    transport_model = "monte carlo",
    nrays = 512,
    clustering = "by area",
    absorption = "partitioned energy"
}
```

A description of the Lua input fields for the photon Monte-Carlo radiation transport model is given in Table 10. Note that here `nrays` is the total number of rays emitted per cell, whereas for the discrete transfer model `nrays` is the number of rays emitted per cell per frequency interval.

Table 10: Description of Lua input fields for the photon Monte-Carlo radiation transport model

Field	Type	Description
`nrays`	*int*	Number of rays emitted per cell
`clustering`	*string*	Ray clustering: `by volume`, `by area` or `none`
`absorption`	*string*	Absorption model: `standard`, or `partitioned energy`

Index