Peter A. Jacobs and Rowan J. Gollan

The Eilmer 4.0 flow simulation program:

Guide to the transient flow solver

including some examples to get you started.

February 7, 2024

Technical Report 2017/26
School of Mechanical & Mining Engineering
The University of Queensland

Abstract

Eilmer is a program for the simulation of transient, compressible flow in two and three spatial dimensions. It has a preparation mode that can be used to set up a database of simulation parameters, a multiblock grid defining the flow domain and an initial flow field. These items are then used as a starting point for the main simulation which computes a series of snapshots of the evolving flow. A postprocessing mode can extract and reformat flow data of interest.

Eilmer is available as source code from https://github.com/gdtk-uq/gdtk and is related to the larger collection of compressible flow simulation codes found at https://gdtk.uqcloud.net/.

This user's 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.

Contributors

Peter A. Jacobs, Rowan J. Gollan, Kyle Damm, Nick Gibbons, Daryl Bond, Ingo Jahn and Anand Veeraragavan, as co-chief gardeners

with contributions from a cast of many, including:
Ghassan Al'Doori, Nikhil Banerji, Justin Beri, Peter Blyton, Viv Bone, Jamie Border, Arianna Bosco, Djamel Boutamine, Laurie Brown, James Burgess, David Buttsworth, Wilson Chan, Eric Chang, Sam Chiu, Chris Craddock, Brian Cook, Jason Czapla, Tim Cullen, Andrew Dann, Andrew Denman, Zac Denman, Luke Doherty, Elise Fahy, Antonia Flocco, Delphine Francois, James Fuata, David Gildfind, Richard Goozeé, Sangdi Gu, Birte Haker, Stefan Hess, Jonathan Ho, Hans Hornung, Jimmy-John Hoste, Carolyn Jacobs, Juanita Jacobs, Chris James, Ian Johnston, Reid Jones, Ojas Joshi, Xin Kang, Joshua Keep, Rory Kelly, Rainer Kirchhartz, Jens Kunze, Sam Lamboo, Will Landsberg, Alexis Lefevre, Cor Lerink, Steven Lewis, Yu (Daisy) Liu, Kieran Mackle, Pierre Mariotto, Tom Marty, Matt McGilvray, David Mee, Carlos de Miranda-Ventura, Christine Mittler, Luke Montgomery, Heather Muir, Jan-Pieter Nap, Brendan O'Flaherty, Reece Otto, Austen Pane, Andrew Pastrello, Paul Petrie-Repar, Jorge Sancho Ponce, Daniel F. Potter, Jason (Kan) Qin, Deepak Ramanath, Andrew Rowlands, Michael Scott, Umar Sheikh, Daniel Smith, Tamara Sopek, Sam Stennett, Ben Stewart, Phillip Swann, Joseph Tang, Katsu Tanimizu, Nils Temme, Augustin Tibère–Inglesse, Pierpaolo Toniato, Matthew Trudgian, Paul van der Laan, Tjarke van Jindelt, Jaidev Vesudevan, Han Wei, Mike Wendt, Brad (The Beast) Wheatley, Vince Wheatley, Lachlan Whyborn, Adriaan Window, Hannes Wojciak, Fabian Zander, Mengmeng Zhao

Over the years, 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.

Acknowledgement
This document was mostly prepared while PAJ was on Special Studies Program at Laboratoire EM2C, CNRS UPR288, Centrale-Supélec, Paris.

Contents

1

Introduction

1.1 Compressible flow simulation and the Eilmer code

Eilmer code is a program for the numerical simulation of transient, compressible gas flows in two and three dimensions. This program answers 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 as solid no-slip walls, inflow surfaces and outflow surfaces. To help with the setup of this domain, the code collection includes a preparation mode that can be used to set up a database of simulation parameters, a blocked, body-fitted mesh defining the flow domain and an initial flow-field specification. This preparation mode includes a mesh generator that can accept a description of the flow domain in terms of boundary surfaces and then generate the blocked mesh of finite-volume cells. Within each block, the underlying grid of cells may be structured or unstructured. The mesh and initial flow state can then be used as a starting point for the main simulation mode which computes a series of snapshots of the evolving flow. Finally, a rudimentary but versatile postprocessing mode makes the flow data available for further analysis.

We have arranged the code as a *programmable* program, with a user-supplied input script (written in Lua) providing the configuration for any particular simulation exercise. Our target audience is the advanced student of gas dynamics, possibly an undergraduate student of engineering but, more likely, a postgraduate student or academic colleague wanting to simulate gas flows as part of their study. A laundry list of features in the code includes:

- Eulerian/Lagrangian description of the flow (finite-volume, 2D axisymmetric or 3D).
- Transient, time-accurate updates, and optionally implicit updates for steady flow.
- Shock capturing plus shock fitting boundary.

- Multiple-block, structured and unstructured grids.
- Parallel computation in a shared-memory context.
- High-temperature nonequilibrium thermochemistry.
- GPU acceleration of the finite-rate chemistry.
- Dense-gas thermodynamic models and rotating frames of reference for turbomachine modelling.
- Turbulence models.
- Conjugate heat transfer to solid surfaces and heat flow within solid objects.
- MHD simulation for a single-fluid plasma.
- Import of GridPro and SU2 grids for complex flow geometries.

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 Eilmer code 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.

1.2 History of the code

In developing Eilmer, our focus has been on producing an open source code that is designed to be a simple access point for teaching students about computational fluid dynamics. The aspects of CFD that interest us include both the application of a simulation code and the development of the code itself.

Eilmer 4.0 is a complete reimplementation of the best bits of the Eilmer3 program. In turn, Eilmer3 was a derivative of the code mbcns2 which was an experiment in writing the mb_cns code in C++. Once it was determined that (despite the programming pain) there were clear benefits in using C++, our three-dimensional flow code Elmer[1] 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. This was a large code base, in a difficult language for us (since we are mechanical engineers by trade) so, when a viable alternative language became available, we jumped ship and rebuilt the best parts of Eilmer3 in the D programming language (`http://dlang.org`) coupled with the Lua scripting language (`http://www.lua.org`).

[1] Yes, the spelling changed in recent years, to avoid a name clash with the finite-element code from Finland.

The code is named after Eilmer of Malmesbury[2], with the spelling chosen to avoid a naming clash with the Elmer finite-element code from Finland.[3]

1.3 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/eilmer
Study the example scripts carefully; some of the interesting bits of the documentation are embedded within them.

For a description of the methods coded into `Eilmer`, see the companion report [2] and the paper [3] which cover the gas-dynamic formulation. More detailed introductions to the various components of the overall simulation package can be found in the following documents:

- a brief introduction to the D-language implementation of the Eilmer code [4]
- a guide to the gas model and basic thermochemistry package [5]
- a guide to the reacting-gas model with finite-rate kinetics [6]
- a guide to the look-up table gas model [7]
- a guide to the geometric modelling package [8]
- a guide to the shock-fitting boundary condition [9]

1.4 License

For the source code, we use the GNU General Public License 3. Please see the file `gpl.txt` in the source tree. For the documentation, such as this user guide, we use the Creative Commons Attribution-ShareAlike 4.0 International License.

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 papers on the Eilmer code:

> Jacobs, P.A. and Gollan, R.J. (2016). Implementation of a Compressible-Flow Simulation Code in the D Programming Language. *Advances of Computational Mechanics in Australia* Volume 846, pages 54–60, in the series *Applied Mechanics and Materials* (DOI: 10.4028/www.scientific.net/AMM.846.54)

> 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(1):19-57 (DOI: 10.1002/fld.3790)

[2] https://en.wikipedia.org/wiki/Eilmer_of_Malmesbury
[3] http://www.csc.fi/elmer

2

Getting started

The core solver and its modules are mainly written in the D programming language for speed and the benefits of compile-time checking. The pre- and post-processing modes make use of the Lua scripting language so that we get flexibility and convenient customization. There is also a little Tcl/Tk used in the automated testing scripts.

2.1 Prerequisite environment and assumed knowledge

Our main development environment is Linux but the programs can be deployed on Linux, flavours of Unix such as MacOS-X, and MS-Windows. The main requirement is that the D language compiler and the Tcl interpreter be available. The source code of the Lua interpreter is included in the Eilmer source code repository. If you are not accustomed to working with Unix/Linux, have a look at Appendix A for a brief introduction to working on the command line.

Beyond our expectations of your computing environment, we also assume that your mathematics, science or engineering background adequately prepares you for CFD analysis. In particular, we assume that you have a working knowledge of geometry, calculus, mechanics, and thermo-fluid-dynamics, at least to a second- or third-year university level. With the Eilmer code, we try to make the analysis of compressible, reacting flow accessible and reliable; we cannot make it trivial.

2.2 Getting the source code

The full source code for Eilmer and a set of examples can be found in a public repository on GitHub. To get your own copy, use the Git revision control client to clone the repository with something like the following command:

```
$ git clone https://github.com/gdtk-uq/gdtk.git gdtk
```

and within a couple of minutes, depending on the speed of your network connection, you should have your own copy of the full source tree and the complete repository history.

2.3 Building and installing the programs

Once you have cloned this repository, all that is required is a Linux environment with a fairly recent D compiler and a C compiler (for building the Lua interpreter). We recommend the LDC compilers.

Going into the `gdtk/src/eilmer` directory you will find a single `makefile` that allows the build to proceed with the command `make install`. The executable program and supporting files will be installed into the directory `$HOME/gdtkinst/` by default.

2.4 Running the program

For running the program, environment variables may be set for the bash shell. On a recent Ubuntu system, put the following commands put into your `.bash_aliases` file:

```
export DGD_REPO=${HOME}/gdtk
export DGD=${HOME}/gdtkinst
export PATH=${PATH}:${DGD}/bin
export DGD_LUA_PATH=${DGD}/lib/?.lua
export DGD_LUA_CPATH=${DGD}/lib/?.so
```

Setting the variable `DGD_REPO` may be handy if you have cloned your copy of the repository to somewhere other than `$HOME/gdtk/`.

The actual running of a simulation is done in stages.

1. Prepare the grids and initial flow configuration by interpreting your Lua input script.
2. Run the main simulation program, starting from the prepared initial flow state and allowing the flow field to develop in time, writing the resulting flow field states at particular times.
3. Postprocess the simulation data to extract particular data of interest.

Of course, this description is too superficial to actually expect that you will be able to run a simulation with no further instruction. If you are keen, it's now time to try the tutorial example in the following chapter.

3

A first simulation with Eilmer

Let's start with a simple-to-imagine flow of ideal air over a sharp-nose of a supersonic projectile. Figure 3.1 is a reproduction of Fig. 3 from Maccoll's 1937 paper [10] 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.

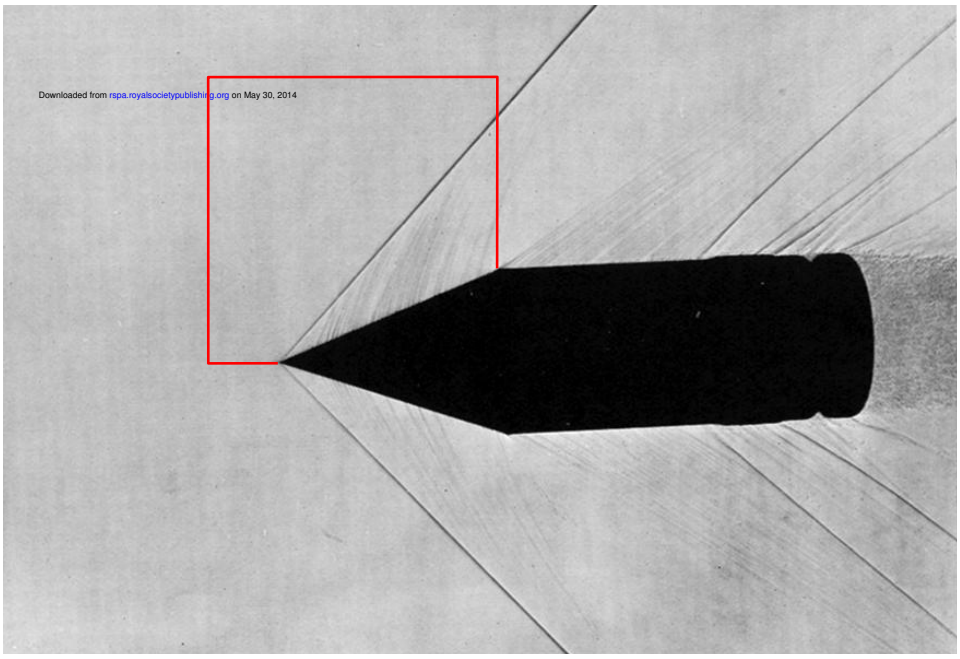


Figure 3.1: A two-pound projectile in flight. A conical shock is attached to the sharp nose of the projectile. This photograph was published by Maccoll in 1937. The red lines have been added to demark the region of gas flow for which we will set up our simulation.

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.

3.1 The simulation

To build our simulation, we abstract the boxed region from Figure 3.1 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.

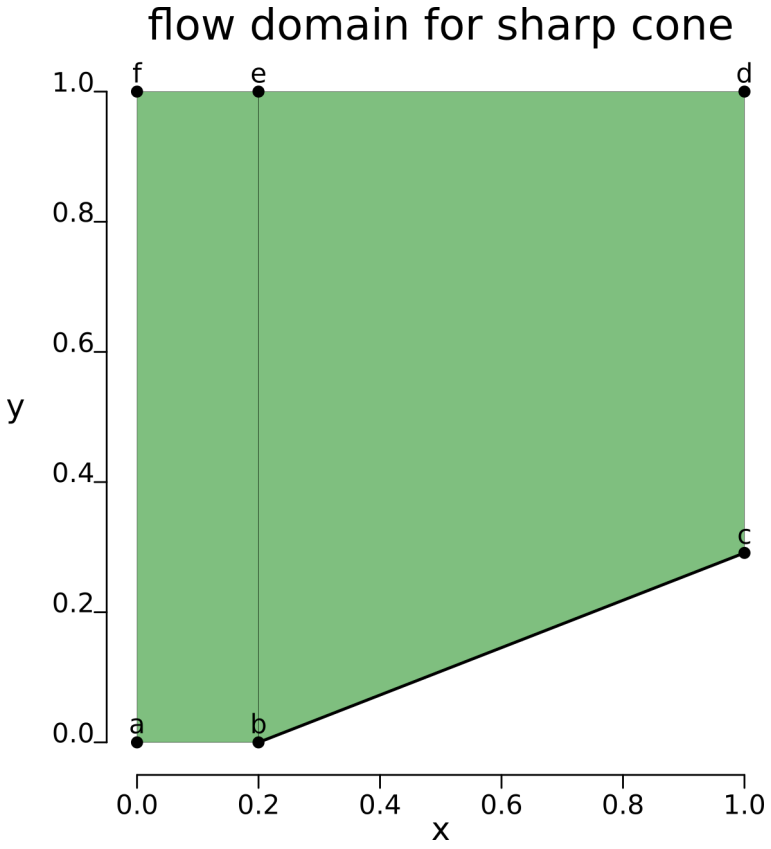


Figure 3.2: Schematic diagram of the geometry for a cone with 20 degree half-angle. The thick dark line represents the cone surface and the green coloured region represents the gas domain. Boundary conditions will be added such that gas flows into the domain on the left (west) boundary and out on the right (east) boundary. The north and south boundaries will be set as walls with slip. This SVG figure was generated as a sketch at preparation time.

Despite Figure 3.1 being a good motivator for this simulation, the free-stream conditions of $p_\infty = 95.84\,\text{kPa}$, $T_\infty = 1103\,\text{K}$ and $V_\infty = 1000\,\text{m/s}$ are actually related to the shock-over-ramp test problem in the original ICASE Report [11] 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.

3.1.1 Running the simulation

Assuming that you have the program executable files built and accessible on your system's search PATH, as described in Chapter 2, use the following commands:

```
$ mkdir ~/temporary-work
$ cd ~/temporary-work
$ rsync -av ~/dgd/examples/eilmer/2D/sharp-cone-20-degrees/sg/ .
```

to set up a work space that is separate to your copy of the source code tree. The $ character represents the command prompt for your system. That way you can do what you like within the work space and then just remove it when you are finished. The rsync command should have made a copy of the essential files for this example in your newly constructed workspace, so you don't really need to type in the content of files discussed below.

The first task in starting our simulation is to prepare an input file for the building of a simple gas model for air. We might call this file ideal-air.inp and it should have the two lines:

```
1 model = "IdealGas"
2 species = {'air'}
```

We now use this input file to prepare the actual gas-model definition file with the command:

```
$ prep-gas ideal-air.inp ideal-air-gas-model.lua
```

The result will be the generation of the Lua file ideal-air-gas-model.lua that contains the detailed specification of an ideal-air gas model.

To generate the gas-model file, the prep-gas program uses a database to collect the detailed thermodynamic properties of the requested gas species. The content of the gas-model file is very much more detailed than the input file but it is just a Lua script and can be inspected with a text editor. As you build more simulations, you may value having the gas-model for a particular simulation being fully documented in this manner. The thermochemical module within Eilmer is very flexible and there are many possible gas models that you might use.

The next thing that you should do in preparing a new simulation is to construct a description of your flow conditions and flow domain as a Lua input script that will be fed to the preparation stage of the flow solver. As is done in all of the televised cooking shows, we will make use of one that was prepared a little earlier and saved as the file cone20.lua. The details of the script content will be examined in the next section but, for now we will proceed to make use of our precooked script.

The first phase of the simulation calculation is to generate a set of grid and initial flow-state files. This is done with the command:

```
$ e4shared --prep --job=cone20
```

which should result in a pair of grid files in the subdirectory `grid/t0000/` and a pair of initial flow-state files in subdirectory `flow/t0000/`. Note that each of the long-format command-line options starts with two dashes.

We can now start the computation of the evolution of the flow field with the command:[1]

```
$ e4shared --run --job=cone20 --verbosity=1 --max-cpus=2
```

and, within a minute or so, you should end up with the flow solution files accumulated in subdirectories of `flow/`. The time-evolution of the flow field is computed for 5 ms (with 833 time steps being required) and while the time-stepping is happening, some status messages should be appearing on the console. Every 20 time steps, a message is printed to indicate the current time step, the current simulation time, the size of the time step (`dt`), the wall-clock time elapsed since the program started (`WC`), the estimated wall-clock time to the final simulation time (`WCtFT`) and the estimated wall-clock time to the maximum allowed number of steps (`WCtMS`). The unit of time for all of these values is seconds. For example, let us take a look at what is printed[2] for step number 540.

```
Step= 540 t= 3.123e-03 dt= 6.003e-06 WC=3 WCtFT=1.7 WCtMS=13.7
```

We can interpret this as follows. At step 540, Eilmer has simulated 3.123 ms of physical time for gas to flow over the cone, and at that step it was using a timestep of 6.003 μs. At step 540, the program has been running for 3 seconds. We have two estimates of when the program will finish and whichever it reaches first, wins — the program will stop. Those two estimates are that: (1) the program will finish in 1.7 seconds when it reaches the requested simulated time of 5 ms; (2) the program will finish in 13.7 seconds when it reaches the requested step number of 3000 steps. In this case, it reaches the maximum simulation time first.

After the simulation calculation is finished, you will probably want to view the flow field data with a visualization program. The command:

```
$ e4shared --post --job=cone20 --vtk-xml \
          --add-vars="mach,pitot,total-p,total-h"
```

will pick up the final frame of the flow solution and write a set of VTK files in XML format. Note that the backslash character at the end of the first line above indicates that the line is not yet complete and there is more to come. Both lines are thus effectively a single logical line. The `--add-vars` option adds some derived variables that are not part of the default solution data. Here, these extra variables are Mach number, Pitot pressure, total pressure and total enthalpy. The postprocessing program defaults to selecting the last time snapshot, however, you may select a different instance to plot with the `--tindx-plot` option.

[1] The option `--max-cpus=2` indicates that the program is allowed to make use of two cores of the computer. It's rare these days to have less than a couple of CPU cores available on your workstation, so you should be pleased to make good use of them.

[2] Note that the numerical details will change, depending on the version of code that you run and the performance of the computer that you are using.

The VTK plot files (all in the `plot` subdirectory) may be viewed with Paraview and nicely-coloured images can be made. Even if your boss doesn't believe your analysis, bright colours are always convincing.

The commands discussed above to prepare, run and postprocess the simulation have been gathered together into the `run.sh` shell script, which can be invoked to run all phases of the simulation in one pass. This is an example of a minimal shell script that tries to execute all of the commands unconditionally. Ideally, you should either run the commands interactively and check the result of each or you should program your shell script to check the return status of each command and then conditionally proceed with each subsequent command. That is a little more sophisticated than we wish to discuss at the moment but learning to do this will be valuable when you write shell scripts that are to be used in a batch system.

This particular example is small enough that running each stage interactively is convenient and, having done so, let's return to examining the content of the input script.

3.1.2 Input script (.lua)

The Lua input script contains the details that specify our simulation. We need a gas model, some flow conditions, a region to be defined and meshed and some boundary conditions specified. There are also decisions to be made about configuration options such as the size of the time step.

```
1 -- cone20.lua
2 -- Simple job-specification file for e4prep -- for use with Eilmer4
3 -- PJ & RG
4 -- 2015-02-24 -- adapted from the Python version of cone20
5
6 -- We can set individual attributes of the global data object.
7 config.title = "Mach 1.5 flow over a 20 degree cone."
8 print(config.title)
9 config.dimensions = 2
10 config.axisymmetric = true
11
12 -- The gas model is defined via a gas-model file.
13 nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
14 print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
15 initial = FlowState:new{p=5955.0, T=304.0, velx=0.0}
16 inflow = FlowState:new{p=95.84e3, T=1103.0, velx=1000.0}
17
18 -- Demo: Verify Mach number of inflow and compute dynamic pressure.
19 print("inflow=", inflow)
20 print("T=", inflow.T, "density=", inflow.rho, "sound speed= ", inflow.a)
21 print("inflow Mach number=", 1000.0/inflow.a)
22 print("dynamic pressure q=", 1/2*inflow.rho*1.0e6)
23
24 -- Set up two quadrilaterals in the (x,y)-plane by first defining
25 -- the corner nodes, then the lines between those corners.
26 a = Vector3:new{x=0.0, y=0.0}
27 b = Vector3:new{x=0.2, y=0.0}
28 c = Vector3:new{x=1.0, y=0.29118}
```

```
29 d = Vector3:new{x=1.0, y=1.0}
30 e = Vector3:new{x=0.2, y=1.0}
31 f = Vector3:new{x=0.0, y=1.0}
32 ab = Line:new{p0=a, p1=b} -- lower boundary, axis
33 bc = Line:new{p0=b, p1=c} -- lower boundary, cone surface
34 fe = Line:new{p0=f, p1=e}; ed = Line:new{p0=e, p1=d} -- upper boundary
35 af = Line:new{p0=a, p1=f} -- vertical line, inflow
36 be = Line:new{p0=b, p1=e} -- vertical line, between quads
37 cd = Line:new{p0=c, p1=d} -- vertical line, outflow
38 quad0 = makePatch{north=fe, east=be, south=ab, west=af}
39 quad1 = makePatch{north=ed, east=cd, south=bc, west=be, gridType="ao"}
40 -- Mesh the patches, with particular discretisation.
41 nx0 = 10; nx1 = 30; ny = 40
42 grid0 = StructuredGrid:new{psurface=quad0, niv=nx0+1, njv=ny+1}
43 grid1 = StructuredGrid:new{psurface=quad1, niv=nx1+1, njv=ny+1}
44 -- Define the flow-solution blocks.
45 blk0 = FluidBlock:new{grid=grid0, initialState=inflow}
46 blk1 = FluidBlock:new{grid=grid1, initialState=initial}
47 -- Set boundary conditions.
48 identifyBlockConnections()
49 blk0.bcList[west] = InFlowBC_Supersonic:new{flowState=inflow}
50 blk1.bcList[east] = OutFlowBC_Simple:new{}
51
52 -- add history point 1/3 along length of cone surface
53 setHistoryPoint{x=2*b.x/3+c.x/3, y=2*b.y/3+c.y/3}
54 -- add history point 2/3 along length of cone surface
55 setHistoryPoint{ib=1, i=math.floor(2*nx1/3), j=0}
56
57 -- Do a little more setting of global data.
58 config.max_time = 5.0e-3  -- seconds
59 config.max_step = 3000
60 config.dt_init = 1.0e-6
61 config.cfl_value = 0.5
62 config.dt_plot = 1.5e-3
63 config.dt_history = 10.0e-5
64
65 dofile("sketch-domain.lua")
```

The first thing to be aware of with your input script is that it is part of a larger Lua program and that the interpreter for this Lua program is embedded within the main `e4shared` simulation program. On seeing the `--prep` flag, the `e4shared` program loads a number of Lua interfaces to the D-language part of the program and then calls the Lua interpreter to do the first part of the Lua program. This sets up a number of services that are made available to your input script that is then processed by the embedded Lua interpreter.

Single-line comments in Lua start with a double-dash and continue to the end of the line. We have started the input script with a few such comments to remind us of the intended simulation and who is to blame for writing the file. The first real command is on line 7, where we set the simulation title as a string. The overall configuration of the simulation is contained in a global configuration class that appears in the Lua script as the table `config`. Behind this table is the D-language global configuration class that stores the data in the D-language domain. Accessing entries in the

`config`, calls up functions that access the corresponding attributes in the D-language domain.

Note that you have access to the full capabilities of the Lua interpreter. On line 8, we use the Lua function `print` to print the value of `config.title`. On lines 9 and 10, we continue to set a couple more configuration options. Boolean values may be specified as `true` or `false`.

The gas model is central to the calculations in the simulation. On line 13, we tell the program where to find the specification for the gas model. This specification is another Lua script that sets relevant tables of parameters for the thermodynamic and transport properties of the gas. Recall that we built the file `ideal-air-gas-model.lua` with the `prep-gas` program, as discussed on p.9. The companion report [5] on the gas model and its programming interface provides more details. On configuring the gas model, the `setGasModel` function returns 3 values: the number of species, the number of nonequilibrium thermal energy modes and a reference to the gas model object. These items may be handy for making calculations within your input script.

Once a gas model has been set, you may then create `FlowState` objects. On lines 15 and 16, we construct two such objects, using the convention described in the Programming in Lua book [12]. Note the use of the colon rather than a dot before the word `new`, and note the use of braces to construct a table of parameters that are given to the `new` function. When constructing objects within the input script, we have mainly chosen a notation that has all of the elements as named attributes in a single table. For these particular `FlowState` constructors, we have omitted some of the parameters, such as `vely` and `velz`, which take on default values. These particular parameters have default values of zero. If a mandatory item is missing, the function called by `new` should complain, telling you specifically what was required.

For the gas-dynamic part of the `Eilmer` code, we try to consistently use SI-MKS units. Point coordinates are specified in metres, time in seconds, velocities in m/s, pressures in Pa and temperatures on the thermodynamic scale K.

To demonstrate the use of arbitrary calculations within the input script, lines 19–22 show how to dip into the FlowState table for the inflow, print some of the parameter values describing the flow condition, and use them to compute the derived quantities of Mach number and dynamic pressure. Eilmer is a multi-species and multi-temperature flow code at heart, but the presently specified gas model for ideal air has only one species and no nonequilibrium thermal modes. When setting flow conditions earlier, the `FlowState` constructor is aware of the single-species specification of this particular gas model and will internally generate an appropriate table from the supplied and default parameters. To access the static temperature value in order to print it on line 20, we can ask for the `T` element as `inflow.T` (as shown) or as `inflow["T"]`. Lines 21 and 22 compute the inflow Mach number as $\frac{V_\infty}{a}$ and dynamic pressure $\frac{1}{2}\rho V_\infty^2$ using the gas state data and the known inflow velocity. The printed results appear on the standard output as the preparation stage is run.

Having a gas model and a set of flow conditions, let's turn our attention to the construction of the flow domain. The geometric complexity of the domain outlined in Figures 3.1 and 3.2 is small, so we will use the geometry and meshing functions built into `Eilmer`.

Lines 26 to 31 start the geometric description by defining a few points of interest using the `Vector3` constructor. Note that the coordinate values are supplied as named items in the table passed to the `new` method of the `Vector3` class and that the unspecified `z` component defaults to zero. As shown in Figure 3.2, these points will become the corner points for a pair of quadrilateral patches over which the meshes and the blocks will be defined, so it is convenient to assign them to simple names. Point `b` is the tip of the cone, located at $x = 0.2\,\text{m}$. Point `c`, at $x = 1.0\,\text{m}$ indentifies the base of the cone in the x, y-plane. We manually compute the y-coordinate of point `c` as $0.8 \times \tan 20^\circ$. The points `d`, `e` and `f` will be used to define the upper (`north`) boundary of the flow domain and we set their y-coordinates at $1.0\,\text{m}$.

Once we have the corner points, we proceed to construct some line segments on script lines 32 through 37. The first point on the line segment is identified as item `p0` and the final point as `p1`. These correspond to parameter values $t = 0.0$ and $t = 1.0$ respectively. The `Line` class is a derived class of the base class `Path`. Later, you will be defining more sophisticated `Path` objects such as Arcs, Bezier curves and splines, and each of these `Path` objects will be defined over the same parameter range $0.0 \leq t \leq 1.0$. The variable names for the constructed lines have been chosen simply to reflect the end points of each segment. Almost any names will suffice[3], however, careful naming will lessen confusion as your script grows.

Lines 38 and 39 use the `makePatch` function to assemble the line segments as the edges of two-dimensional patches. The patch assigned to `quad0` will have default transfinite interpolation while the patch assigned to `quad1` will interpolate points in physical space with the aid of a background mesh defined using Knupp's robust elliptic grid generator [13].

The flow simulation requires that the flow domain be specified as meshes of finite-volume cells. So far, we have a description of the domain as patches of space. On line 41, we define three variables that represent the number of cells that we want along the mesh directions and, on lines 42 and 43, we construct a mesh of cells for each quadrilateral patch. Note that the numbers of vertices in each mesh direction is supplied to the mesh constructor. The number of vertices in each direction is one more than the number of cells that we wish to have.

The flow solution is defined over blocks of cells. On lines 45 and 46, we construct the blocks by associating them with a mesh and an initial flow state. Here, we use a uniform flow state over each block but it is also possible to specify a varying flow state. Later examples will show you how to do that by using a Lua function that you define in your input script. For now, we set the initial flow state upstream of the cone nose to be the same as the inflow condition because we expect nothing interesting to happen in that block. In the block adjacent to the cone surface, we set the initial flow condition to be a fairly low pressure, quiescent gas. The upstream flow will drive a shock through this block and establish a flow over the cone.

To complete the flow domain definition, we need to specify the boundary conditions that drive the flow solution. Two-dimensional structured-grid blocks, as used here, have boundaries labelled `north`, `east`, `south` and `west`. If we don't specify a

[3]The list of global symbol names in the Lua environment is given in Appendix B.7. You should avoid these as names for your objects.

particular boundary condition for each particular block boundary, a `WallBC_WithSlip` condition will be assumed. One option that we have is to specify the boundary conditions when the `FluidBlock` constructor is called, supplying the boundary conditions as a table. A different approach is used in this example. First, on line 48, we ask the program to automatically identify connected blocks. This is done as a brute-force search for matching corner points. If the grids are positioned in space such that the corner points coincide for a pair of boundaries, as would be the case here for the `east` boundary of block 0 and the `west` boundary of block 1, an `ExchangeBC_FullFace` is applied to the corresponding boundaries with the other boundary specified as the exchange partner. This effectively stitches the flow domain together along this common boundary edge. Lines 49 and 50 assign individual boundary condition objects for inflow and outflow, respectively.

Sometimes we are interested in the history of the flow at particular points and would like more detail of that history than would be recorded in the set of snapshots of the entire flow field. We can identify particular cells for this history recording as shown in lines 53 and 55. Line 53 specifies the particular cell via a spatial position. The nearest containing cell to this position is identified. Line 55 directly specifies the block and the `i` and `j` indices (within that block) of another cell.

The final preparations are to set a few more configuration parameters in lines 58 to 63. The simulation calculation will be terminated when either the simulation time exceeds `max_time` or step reaches `max_step`, whichever happens first. Usually, you would like your simulation to run to a particular time, however, there are many occasions when it is good to limit the number of steps. Limiting the maximum number of steps to a fairly small value may save you lots of waiting when trying to set up new simulations.

On line 60, we specify the initial time step that we would like the simulation to start with. The code then takes over and adjusts the time step to some allowable value, guided by the `cfl_value` specified on line 61. Every few steps, the code will scan all cells, looking for the shortest time for a signal to cross any cell. For the convective terms, this is the time of a pressure wave to traverse a cell. The shortest time found is then multiplied by the `cfl_value` and the result is used as the time step for the whole simulation.

The simulation program will write snapshots of the entire flow field every `dt_plot` period. This period is specified as seconds in *simulation* time, as opposed to *wall-clock* time that you are watching go by as the calculation proceeds. The history data for the few selected points (discussed three paragraphs above) is usually with a shorter period, `dt_history`.

The final thing that we do in this input script is to call yet another Lua script to make the SVG sketch that is rendered in Figure 3.2. We will show the content of this sketch script later.

3.2 Results and postprocessing

Figure 3.3 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.

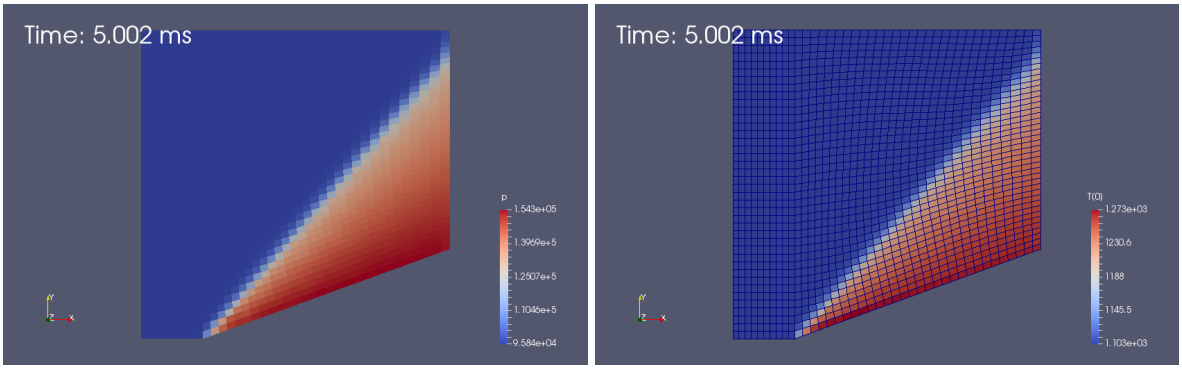


Figure 3.3: Pressure and temperature fields for a low-resolution simulation of flow over a cone with 20 degree half-angle. The temperature field plot also included the mesh.

The plots have been produced with `Paraview`, by opening up the `plot/cone20.pvd` file. The time stamp in the upper 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.

The shock displayed in the pressure field shows features that are characteristic of a flow solution produced by a "shock-capturing" code such as `Eilmer`. 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.[4] Also, when following a line that crosses the shock, a small number of cells are passed 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 3.4. The high-resolution solution is looking clean but the computational cost, in terms of calculation time, has gone up from a few seconds to more than an hour.

Since Eilmer is a simulation program, it starts with some initial (but possibly variable) flow states 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 3.5 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 `config.dt_plot = 1.5e-3`.

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

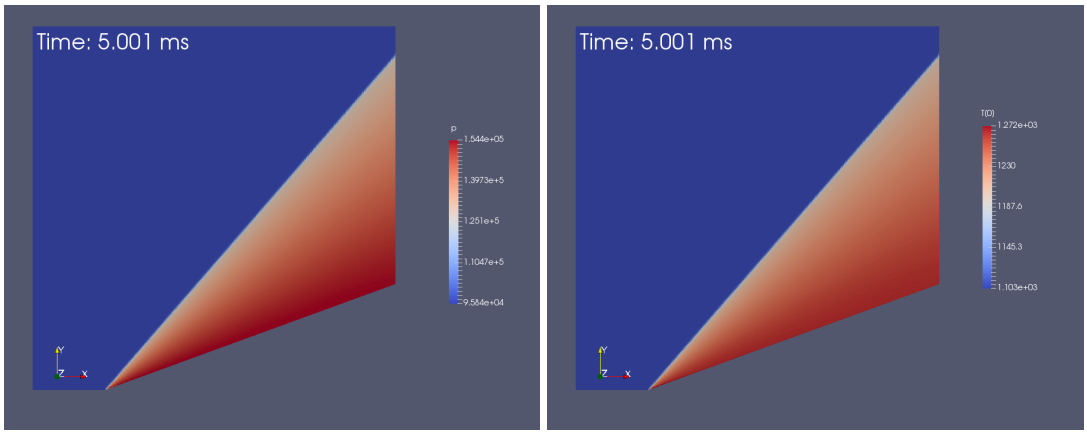


Figure 3.4: Pressure and temperature fields for a mesh with 8 times more resolution in each direction.

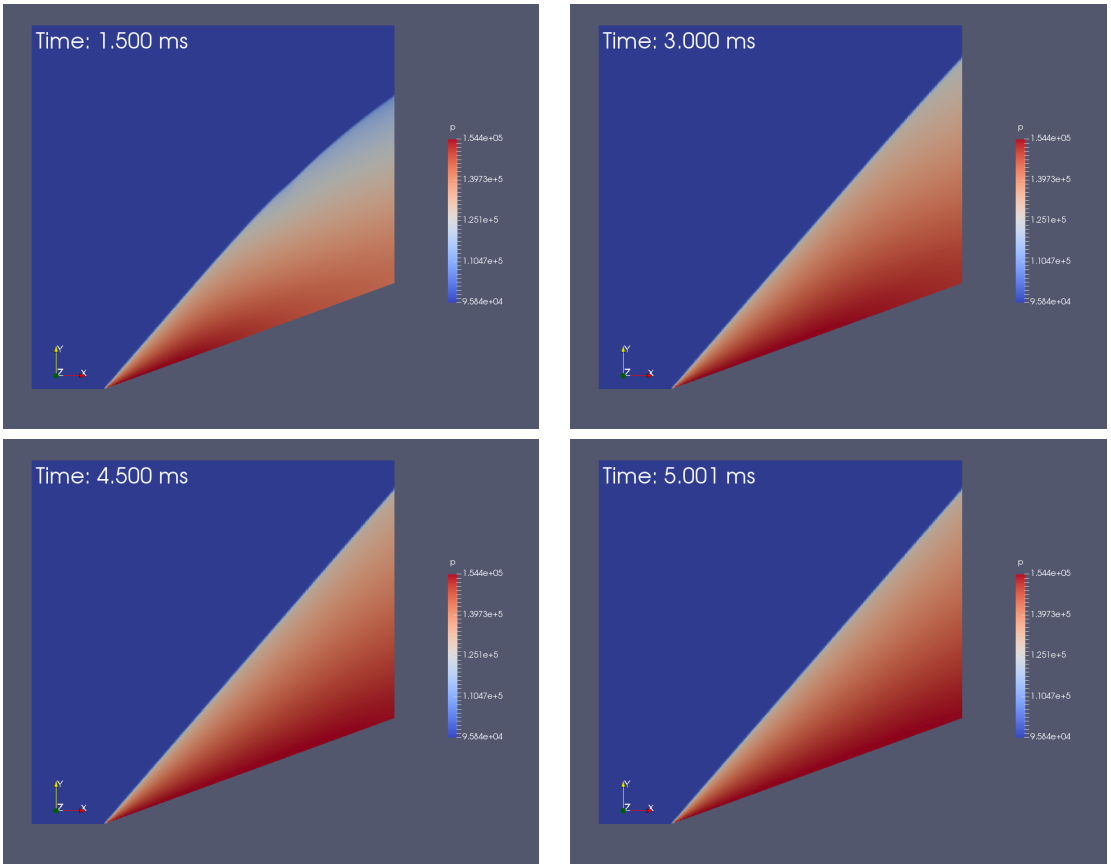


Figure 3.5: Evolution of the pressure field, times as indicated.

Although not obvious in Figure 3.5, 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 pressure on the cone surface and, by setting history points, we have arranged `Eilmer` to occasionally write out the flow properties in a couple of cells on the cone surface. We then use an Awk program (`cp.awk`) to filter the history file, extracting the time (column 1) and static pressure (column 10), writing a simple data file with the time in milliseconds in column 1 and the coefficient of pressure in column 2. New users might like to learn about the Awk language. It is very convenient for writing filter programs and a brief introduction is given in Appendix C.

```
# cp.awk
# Scan a history file, picking out pressure and scaling it
# to compute coefficient of pressure.
#
# PJ, 2016-09-22
#
BEGIN {
    Rgas = 287.1; # J/kg.K
    p_inf = 95.84e3; # Pa
    T_inf = 1103; # K
    rho_inf = p_inf / (Rgas * T_inf)
    V_inf = 1000.0; # m/s
    q_inf = 0.5 * rho_inf * V_inf * V_inf
    print "# rho_inf=", rho_inf, " q_inf=", q_inf
    print "# t,ms cp"
}

$1 != "#" {
    t = $1; p = $10
    print t*1000.0, (p - p_inf)/q_inf
}

END {}
```

From Chart 5 in NACA Report 1135 [14], 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 3.6 shows the pressure coefficient for the history point located two thirds along from the nose to the base of the cone. Note the sudden rise as the shock structure driven by the free-stream flow arrives at this history location on 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.

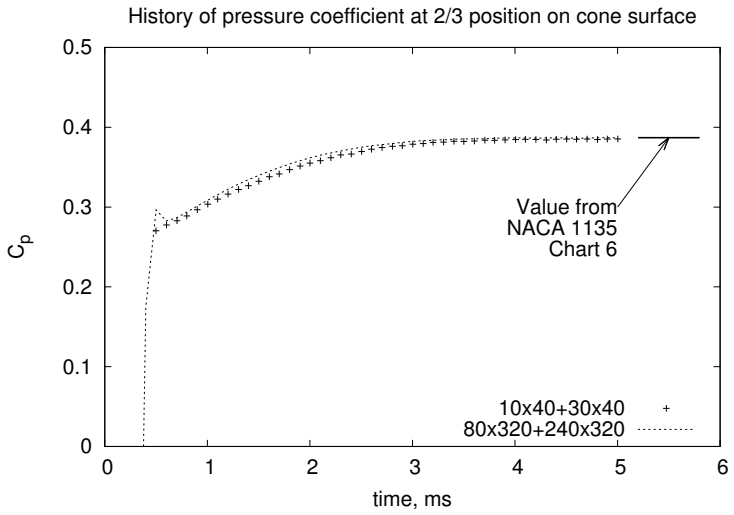


Figure 3.6: Evolution of the coefficient of pressure at the cone surface for flow over a cone with 20 degree half-angle for two mesh resolutions.

The commands to produce the plot in Figure 3.6 are:

```
#!/bin/bash
# plot.sh
# Compute coefficient of pressure for the history point
# and plot it against the previously computed high-res data.
#
# PJ, 2016-09-22
#
awk -f cp.awk hist/cone20-blk-1-cell-20.dat > cone20_cp.dat
gnuplot plot_cp.gnuplot
```

and the GnuPlot commands are:

```
set term postscript eps enhanced 20
set output "cone20_cp.eps"
set style line 1 linetype 1 linewidth 3.0
set title "History of pressure coefficient at 2/3 position on cone surface"
set xlabel "time, ms"
set ylabel "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 "80x320+240x320" with lines
```

3.3 Accessing the field data for specialized postprocessing

Beyond the usual slice-and-dice type of postprocessing that is provided by the post-processing mode of `e4shared`, it may be useful to do specialized calculations on the flow data by providing a custom postprocessing script (in Lua) that can do arbitrary calculations. This script has full access to the flow solution data that can be picked up by `e4shared` and, being a Lua script, has full access to the capabilities of the Lua interpreter.

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 built into the main program.

```
1 -- ideal_shock_angle.lua
2 -- Invoke with the command line:
3 -- $ e4shared --custom-post --script-file=ideal_shock_angle.lua
4 V1=1000.0; p1=95.84e3; T1=1103.0; theta=math.rad(20.0)
5 beta = idealgasflow.beta_cone(V1, p1, T1, theta)
6 print("beta=", math.deg(beta), "degrees")
```

Although we could have specified the input values as literals to the `beta_cone` function call, the assignment to variables and the use of those names in the function call makes the script somewhat self-documenting. The full set of functions available in the `idealgasflow` table is listed in Appendix D.1.

The `estimate_shock_angle.lua` script (below) uses the data reading and storage capability provided by the `FlowSolution` class (line 8 in the script) that is available in the custom post-processing mode of `e4shared`. Access to the flow data for a particular cell is provided by the method `get_cell_data` (line 37), which returns the data in the form of a table. Note that a colon is used to access the method for the `FlowState` object bound to `fsol`.

```
1 -- estimate_shock_angle.lua
2 -- Invoke with the command line:
3 -- $ e4shared --custom-post --script-file=estimate_shock_angle.lua
4 -- PJ, 2015-10-20
5 --
6 print("Begin estimate_shock_angle")
7 nb = 2
8 fsol = FlowSolution:new{jobName="cone20", dir=".", tindx=4, nBlocks=nb}
9 print("fsol=", fsol)
10
11 function locate_shock_along_strip()
12    local p_max = ps[1]
13    for i = 2, #ps do
14       p_max = math.max(ps[i], p_max)
15    end
16    local p_trigger = ps[1] + 0.3 * (p_max - ps[1])
17    local x_old = xs[1]; local y_old = ys[1]; local p_old = ps[1]
18    local x_new = x_old; local y_new = y_old; local p_new = p_old
19    for i = 2, #ps do
20       x_new = xs[i]; y_new = ys[i]; p_new = ps[i]
21       if p_new > p_trigger then break end
```

```
22       x_old = x_new; y_old = y_new; p_old = p_new
23    end
24    local frac = (p_trigger - p_old) / (p_new - p_old)
25    x_loc = x_old * (1.0 - frac) + x_new * frac
26    y_loc = y_old * (1.0 - frac) + y_new * frac
27    return
28 end
29
30 xshock = {}; yshock = {}
31 local nj = fsol:get_njc(0)
32 for j = 0, nj-1 do
33    xs = {}; ys = {}; ps = {}
34    for ib = 0, nb-1 do
35       local ni = fsol:get_nic(ib)
36       for i = 0, ni-1 do
37          cellData = fsol:get_cell_data{ib=ib, i=i, j=j}
38          xs[#xs+1] = cellData["pos.x"]
39          ys[#ys+1] = cellData["pos.y"]
40          ps[#ps+1] = cellData["p"]
41       end
42    end
43    locate_shock_along_strip()
44    if x_loc < 0.9 then
45       -- Keep only the good part of the shock.
46       xshock[#xshock+1] = x_loc
47       yshock[#yshock+1] = y_loc
48    end
49 end
50
51 -- Least-squares fit of a straight line for the shock
52 -- Model is y = alpha0 + alpha1 * x
53 sum_x = 0.0; sum_y = 0.0; sum_x2 = 0.0; sum_xy = 0.0
54 for j = 1, #xshock do
55    sum_x = sum_x + xshock[j]
56    sum_x2 = sum_x2 + xshock[j]*xshock[j]
57    sum_y = sum_y + yshock[j]
58    sum_xy = sum_xy + xshock[j]*yshock[j]
59 end
60 N = #xshock
61 alpha1 = (sum_xy/N - sum_x/N * sum_y/N) / (sum_x2/N - sum_x/N * sum_x/N)
62 alpha0 = sum_y/N - alpha1 * sum_x/N
63 shock_angle = math.atan(alpha1)
64 sum_y_error = 0.0
65 for j = 1, N do
66    sum_y_error = sum_y_error+math.abs((alpha0+alpha1*xshock[j])-yshock[j])
67 end
68 print("shock_angle_deg=", shock_angle*180.0/math.pi)
69 print("average_deviation_metres=", sum_y_error/N)
```

The function `locate_shock_along_strip` (lines 11–28) searches for a significant pressure jump in a strip of cells along the i-index direction of the structured blocks. Lines 33–42 set up a strip of data at a particular j-index. Once the coordinates of the jump locations are stored in tables `xshock` and `yshock`, a least-squares fit of a linear model is performed in lines 53–62. The command for invoking the script is shown in the comment on line 3. It is generally a good idea to document your commands

to performing a simulation and the associated postprocessing activities in comments like this or in shell scripts. You will need to be reminded of the details so you might as well make these notes machine readable or executable.

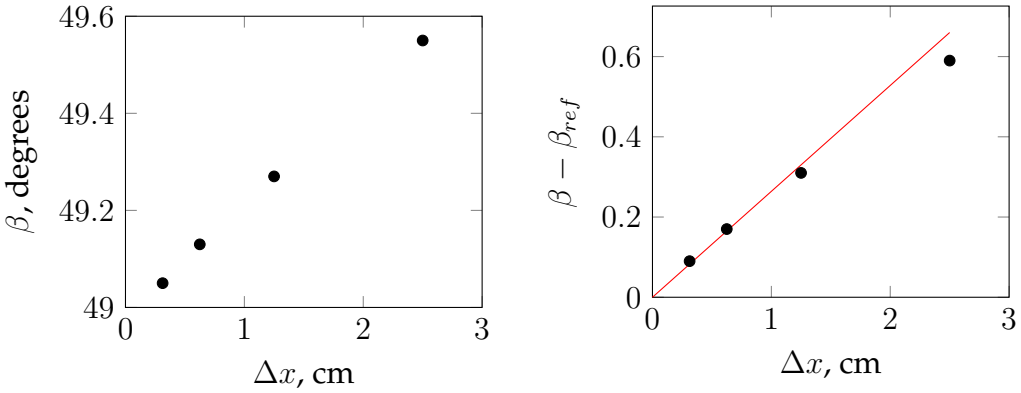


Figure 3.7: Convergence of the shock angle and its error with mesh refinement. $\beta_{ref} = 48.96^o$.

3.4 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 3.7 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 as though we can get as close as we wish.

3.5 Other notes on this first example

- Run time for this simple simulation is approximately 8 seconds for 853 steps on a Surface Pro 3 with a pair of Intel Core i5-4300U processors. With the blocks not being well balanced, we do not make best use of both cores. To make best use of your multi-core workstation, try to arrange blocks with reasonably equal numbers of cells.
- This `cone20.lua` input script really has full access to the Lua interpreter built into the simulation program. Be careful!
- Lua 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 program when processing your configuration. Where this might happen in a non-obvious way is in the importing of foreign modules or packages (to do something interesting in your script). Also, without explicitly declaring a name as local, you will be referring to a global variable in your first assignments. See Appendix B.7 for the list of global symbols defined when your input script starts to be processed.

3.6 Parametric modelling using Lua

Let's rework the simulation to explore the gas-dynamics a little more and also make use of the parametric capabilities of the Lua 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 inflow 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.

3.6.1 Input script (.lua)

```
1 -- conep.lua
2 -- Parametric setup for sharp-cone simulation.
3 -- PJ & RG
4 -- 2016-09-23 -- adapted from cone20.lua
5
6 -- We can set individual attributes of the global data object.
7 config.dimensions = 2
8 config.axisymmetric = true
9
10 -- The gas model is defined via a gas-model file.
11 nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
12 print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
13 initial = FlowState:new{p=5955.0, T=304.0, velx=0.0}
14 -- Compute inflow from Mach number.
15 inflow_gas = FlowState:new{p=95.84e3, T=1103.0}
16 M = 1.5
17 Vx = M * inflow_gas.a
18 print("inflow velocity Vx=", Vx)
19 print("dynamic pressure q=", 1/2*inflow_gas.rho*Vx*Vx)
20 inflow = FlowState:new{p=95.84e3, T=1103.0, velx=Vx}
21 print("T=", inflow.T, "density=", inflow.rho, "sound speed= ", inflow.a)
22
23 -- Parameters defining cone and flow domain.
24 theta = 20 -- cone half-angle, degrees
25 L = 0.8    -- axial length of cone, metres
26 rbase = L * math.tan(math.pi*theta/180.0)
27 x0 = 0.2   -- upstream distance to cone tip
28 H = 1.0    -- height of flow domain, metres
29 config.title = string.format("Mach %.1f flow over a %.1f-degree cone.",
30                              M, theta)
```

```
print(config.title)

-- Set up two quadrilaterals in the (x,y)-plane by first defining
-- the corner nodes, then the lines between those corners.
a = Vector3:new{x=0.0, y=0.0}
b = Vector3:new{x=x0, y=0.0}
c = Vector3:new{x=x0+L, y=rbase}
d = Vector3:new{x=x0+L, y=H}
e = Vector3:new{x=x0, y=H}
f = Vector3:new{x=0.0, y=H}
ab = Line:new{p0=a, p1=b} -- lower boundary, axis
bc = Line:new{p0=b, p1=c} -- lower boundary, cone surface
fe = Line:new{p0=f, p1=e}; ed = Line:new{p0=e, p1=d} -- upper boundary
af = Line:new{p0=a, p1=f} -- vertical line, inflow
be = Line:new{p0=b, p1=e} -- vertical line, between quads
cd = Line:new{p0=c, p1=d} -- vertical line, outflow
quad0 = makePatch{north=fe, east=be, south=ab, west=af}
quad1 = makePatch{north=ed, east=cd, south=bc, west=be, gridType="ao"}
-- Mesh the patches, with particular discretisation.
dx = 1.0/40
nx0 = math.floor(x0/dx); nx1 = math.floor(L/dx); ny = math.floor(H/dx)
grid0 = StructuredGrid:new{psurface=quad0, niv=nx0+1, njv=ny+1}
grid1 = StructuredGrid:new{psurface=quad1, niv=nx1+1, njv=ny+1}
-- Define the flow-solution blocks.
blk0 = FluidBlock:new{grid=grid0, initialState=inflow}
blk1 = FluidBlock:new{grid=grid1, initialState=initial}
-- Set boundary conditions.
identifyBlockConnections()
blk0.bcList[west] = InFlowBC_Supersonic:new{flowState=inflow}
blk1.bcList[east] = OutFlowBC_Simple:new{}

-- add history point 1/3 along length of cone surface
setHistoryPoint{x=2*b.x/3+c.x/3, y=2*b.y/3+c.y/3}
-- add history point 2/3 along length of cone surface
setHistoryPoint{ib=1, i=math.floor(2*nx1/3), j=0}

-- Do a little more setting of global data.
config.max_time = 5.0e-3  -- seconds
config.max_step = 3000
config.dt_init = 1.0e-6
config.cfl_value = 0.5
config.dt_plot = 1.5e-3
config.dt_history = 10.0e-5

dofile("sketch-domain.lua")
```

3.7 Exploring the gas dynamics

First, we'll repeat our earlier simulation of a 20-degree half-angle cone, but this time built from a parameterized script. Second, we'll explore what happens with the flow over a 32-degree half-angle cone. The focus on this section is on that second case.

Repeating our simulation of a 20-degree half-angle cone, Figure 3.8 shows essentially the same flow field 5 milliseconds after flow start as Figure 3.3. It has the same straight, attached shock and same range of pressures displayed.

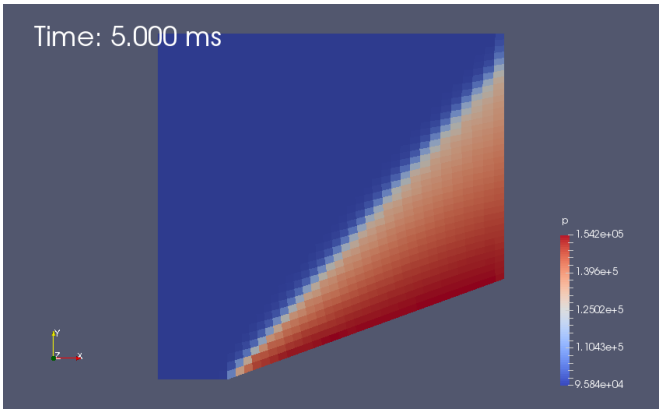


Figure 3.8: Pressure field for the low-resolution simulation of Mach 1.5 flow over a cone with 20 degree half-angle. This is the parametric setup but produces the same simulation as the original setup.

Looking up the conical shock charts in NACA-1135 [14], 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 preparation program and main simulation program, with the calculations to get the appropriate velocity already encoded within the user input script. Figure 3.9 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 `WallBC_WithSlip` 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 3.10 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.

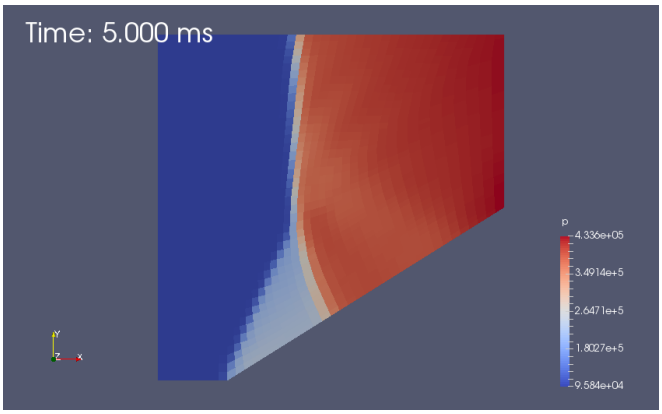
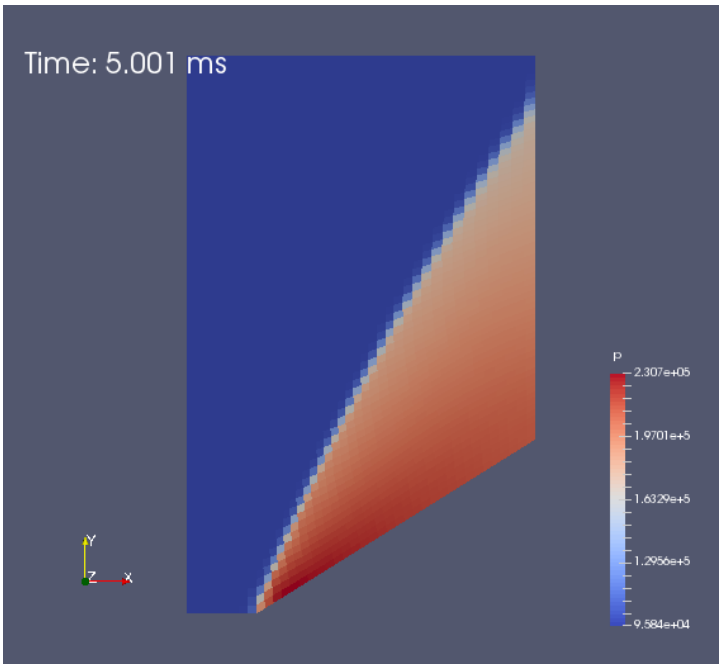
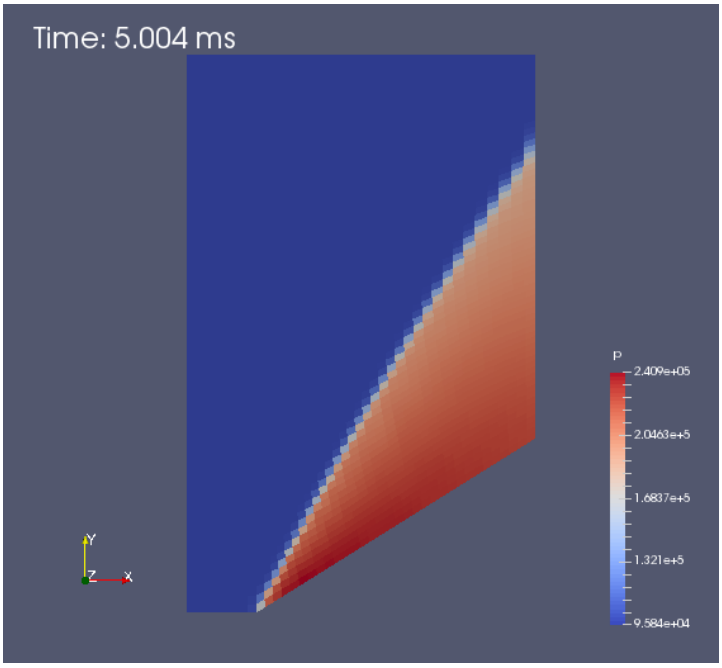


Figure 3.9: Pressure field for the low-resolution simulation at 5 ms of Mach 1.5 flow over a cone with 32 degree half-angle.



(a) Mach 1.5 inflow



(b) Mach 1.6 inflow

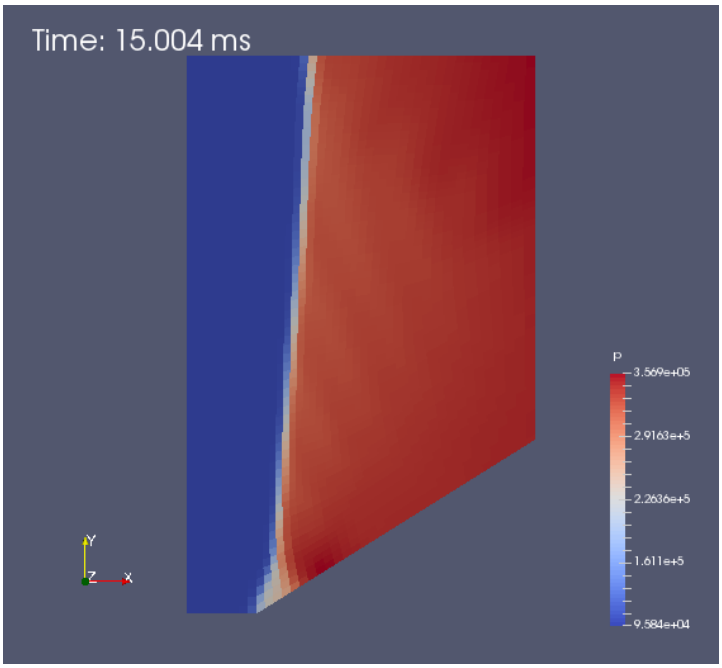
Figure 3.10: Pressure field for the low-resolution simulation at 5 ms of flow over a cone with 32 degree half-angle in a larger flow domain, $H = 1.6$.

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 most convenient pub conducive to the study of high quality multi-block grids. 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?

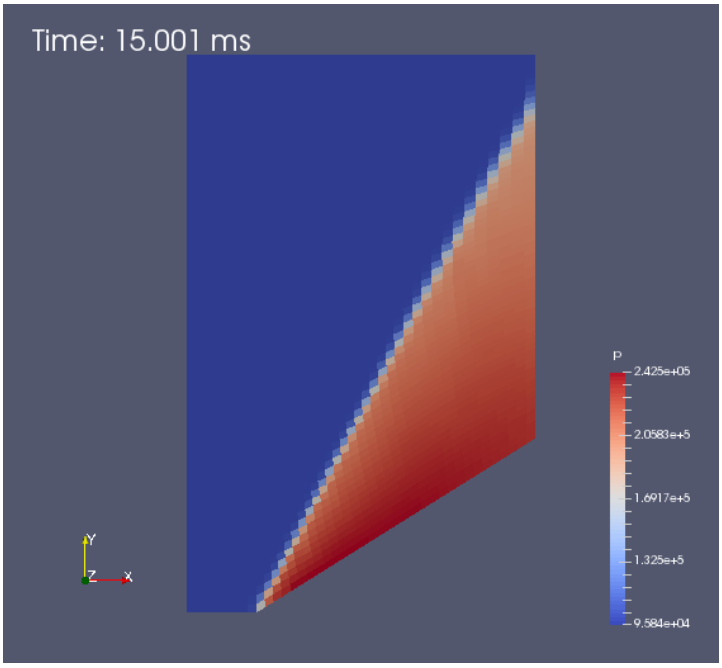
Approximately 5 minutes later, you see the results shown in Figure 3.11 and you wish that you had left for the pub some time earlier. The Mach 1.6 shock looks good and a little straighter, as it should, but the Mach 1.5 case is not showing the desired result. Why, with such a small difference in inflow specification should there be such a big difference? And, why does that difference seem to come from downstream?

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 of the plot files (as shown in line 7 of the `run.sh` script on page 10), be sure to include `mach` in the list of variables that you want added to the flow solution and then produce the plots shown in Figure 3.12.

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 `OutFlowBC_Simple` boundary condition applied to the outflow boundary works by copying flow data from just inside the boundary to the ghost cells just outside the boundary. This simple procedure does not handle subsonic flow across the boundary very well at all, and results in the whole simulation not being a good representation of the physical situation. A good fix is to alter the flow domain, so that the outflow is mostly supersonic.

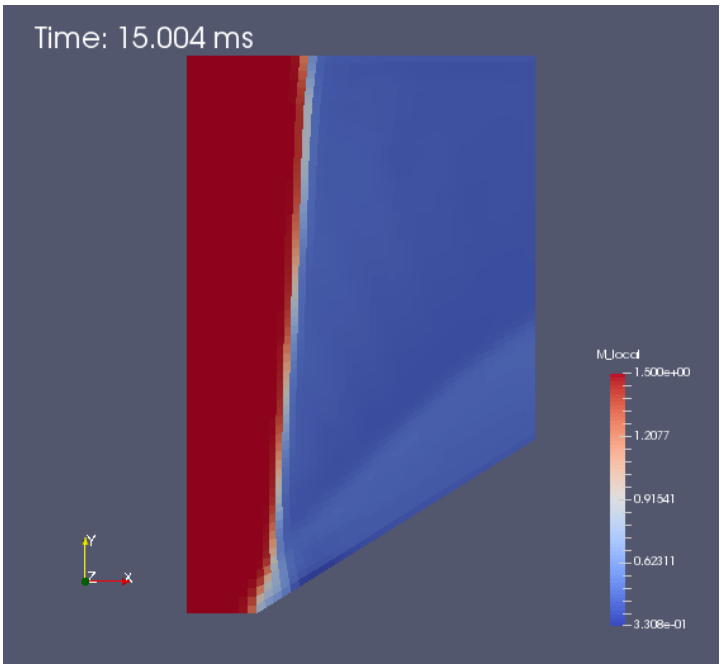


(a) Mach 1.5 inflow

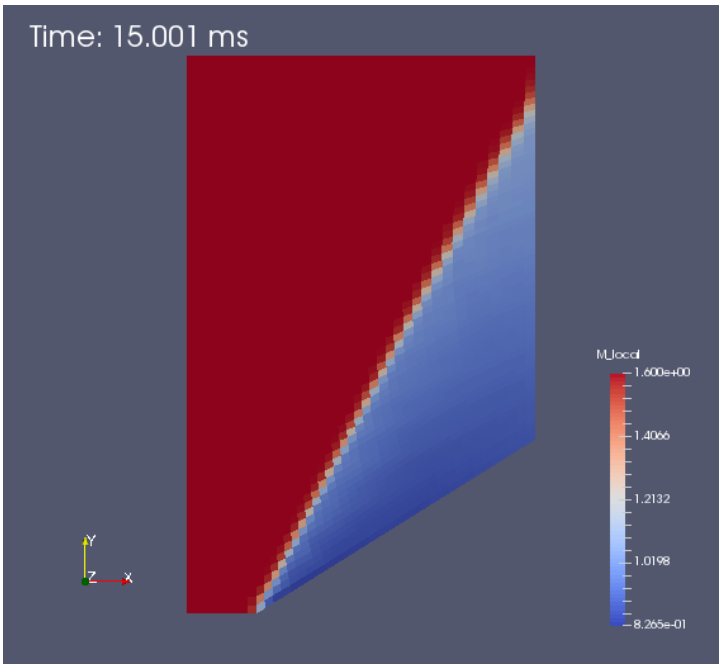


(b) Mach 1.6 inflow

Figure 3.11: 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, $H = 1.6$.



(a) Mach 1.5 inflow, The full range 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 3.12: 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, $H = 1.6$.

3.8 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 as shown in Figure 3.13 (and known as `quad2`, `grid2` and `blk2` in the following input script) allows the flow to regain supersonic flow conditions before crossing the outflow boundary.

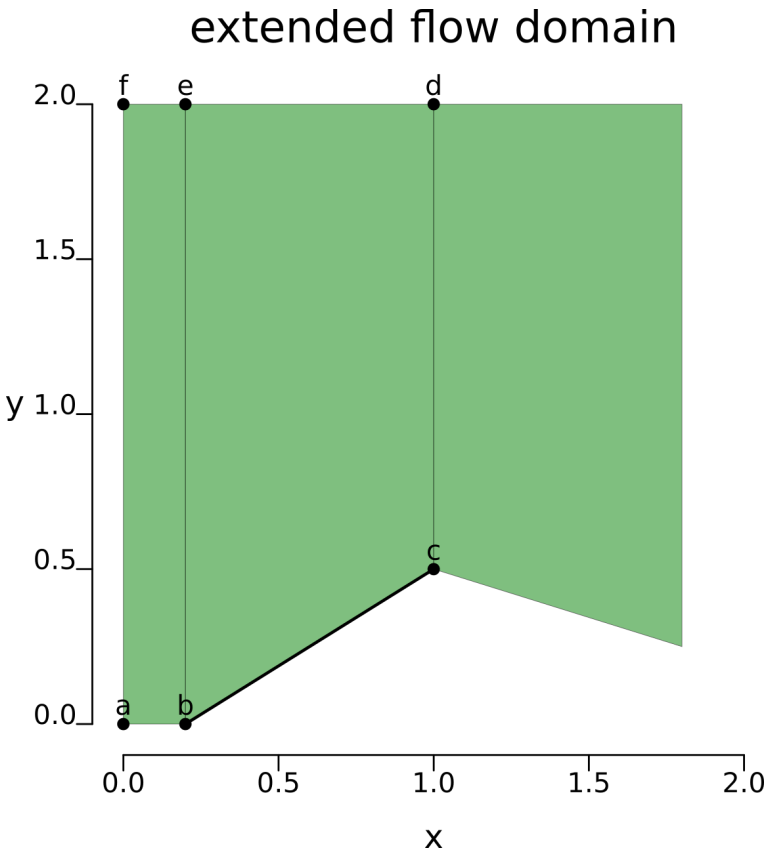


Figure 3.13: Schematic diagram of the extended geometry for a cone with 20 degree half-angle.

3.8.1 Input script (.lua)

```
1 -- conepe.lua
2 -- Parametric, extended setup for sharp-cone simulation.
3 -- PJ & RG
4 -- 2016-09-23 -- adapted from conep.lua
5
6 -- We can set individual attributes of the global data object.
7 config.dimensions = 2
8 config.axisymmetric = true
9
```

```
10 -- The gas model is defined via a gas-model file.
11 nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
12 print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
13 initial = FlowState:new{p=5955.0, T=304.0, velx=0.0}
14 -- Compute inflow from Mach number.
15 inflow_gas = FlowState:new{p=95.84e3, T=1103.0}
16 M = 1.5
17 Vx = M * inflow_gas.a
18 print("inflow velocity Vx=", Vx)
19 print("dynamic pressure q=", 1/2*inflow_gas.rho*Vx*Vx)
20 inflow = FlowState:new{p=95.84e3, T=1103.0, velx=Vx}
21 print("T=", inflow.T, "density=", inflow.rho, "sound speed= ", inflow.a)
22
23 -- Parameters defining cone and flow domain.
24 theta = 32 -- cone half-angle, degrees
25 L = 0.8    -- axial length of cone, metres
26 rbase = L * math.tan(math.pi*theta/180.0)
27 x0 = 0.2   -- upstream distance to cone tip
28 H = 2.0    -- height of flow domain, metres
29 config.title = string.format("Mach %.1f flow over a %.1f-degree cone.",
30                               M, theta)
31 print(config.title)
32
33 -- Set up two quadrilaterals in the (x,y)-plane by first defining
34 -- the corner nodes, then the lines between those corners.
35 a = Vector3:new{x=0.0, y=0.0}
36 b = Vector3:new{x=x0, y=0.0}
37 c = Vector3:new{x=x0+L, y=rbase}
38 d = Vector3:new{x=x0+L, y=H}
39 e = Vector3:new{x=x0, y=H}
40 f = Vector3:new{x=0.0, y=H}
41 ab = Line:new{p0=a, p1=b} -- lower boundary, axis
42 bc = Line:new{p0=b, p1=c} -- lower boundary, cone surface
43 fe = Line:new{p0=f, p1=e}; ed = Line:new{p0=e, p1=d} -- upper boundary
44 af = Line:new{p0=a, p1=f} -- vertical line, inflow
45 be = Line:new{p0=b, p1=e} -- vertical line, between quads
46 cd = Line:new{p0=c, p1=d} -- vertical line, outflow
47 quad0 = makePatch{north=fe, east=be, south=ab, west=af}
48 quad1 = makePatch{north=ed, east=cd, south=bc, west=be, gridType="ao"}
49 -- extend the flow domain
50 xend = x0 + 2*L
51 quad2 = CoonsPatch:new{p00=c, p10=Vector3:new{x=xend, y=rbase/2},
52                        p11=Vector3:new{x=xend, y=H}, p01=d}
53 -- Mesh the patches, with particular discretisation.
54 dx = 1.0/40
55 nx0 = math.floor(x0/dx); nx1 = math.floor(L/dx); ny = math.floor(H/dx)
56 grid0 = StructuredGrid:new{psurface=quad0, niv=nx0+1, njv=ny+1}
57 grid1 = StructuredGrid:new{psurface=quad1, niv=nx1+1, njv=ny+1}
58 grid2 = StructuredGrid:new{psurface=quad2, niv=nx1+1, njv=ny+1}
59 -- Define the flow-solution blocks.
60 blk0 = FluidBlock:new{grid=grid0, initialState=inflow}
61 blk1 = FluidBlock:new{grid=grid1, initialState=initial}
62 blk2 = FluidBlock:new{grid=grid2, initialState=initial}
63 -- Set boundary conditions.
64 identifyBlockConnections()
65 blk0.bcList[west] = InFlowBC_Supersonic:new{flowState=inflow}
66 blk2.bcList[east] = OutFlowBC_Simple:new{}
```

```
67
68 -- add history point 1/3 along length of cone surface
69 setHistoryPoint{x=2*b.x/3+c.x/3, y=2*b.y/3+c.y/3}
70 -- add history point 2/3 along length of cone surface
71 setHistoryPoint{ib=1, i=math.floor(2*nx1/3), j=0}
72
73 -- Do a little more setting of global data.
74 config.max_time = 30.0e-3  -- seconds
75 config.max_step = 15000
76 config.dt_init = 1.0e-6
77 config.cfl_value = 0.5
78 config.dt_plot = 1.5e-3
79 config.dt_history = 10.0e-5
80
81 dofile("sketch-domain-extended.lua")
```

3.8.2 Final results

For a domain height $H = 2$, Figure 3.14 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.

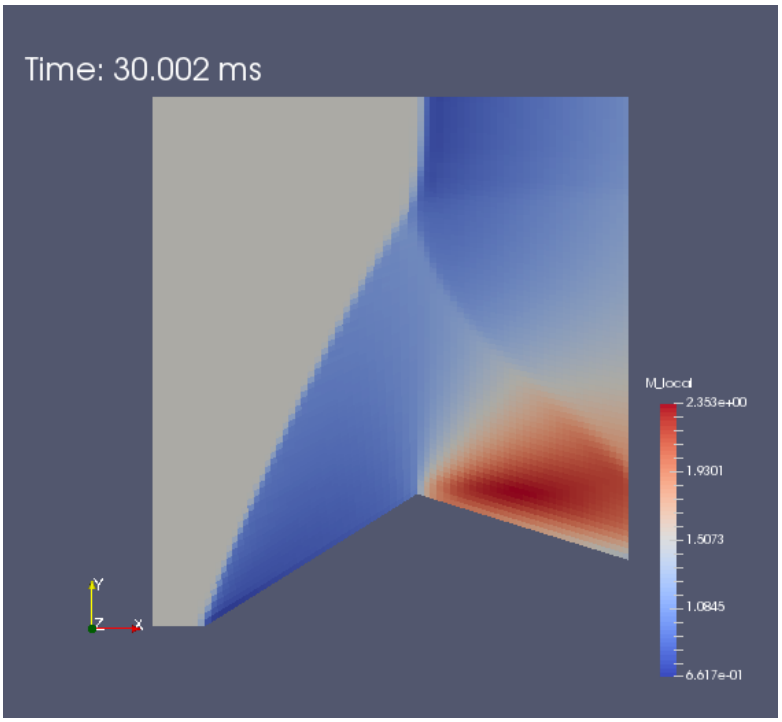


Figure 3.14: Mach number field for the low-resolution simulation at 30 ms of Mach 1.5 flow over a cone with 32 degree half-angle. Flow domain height $H = 2$.

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 3.15 appears to be clean and mostly free from obvious boundary induced problems. The `OutFlowBC_Simple` 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.

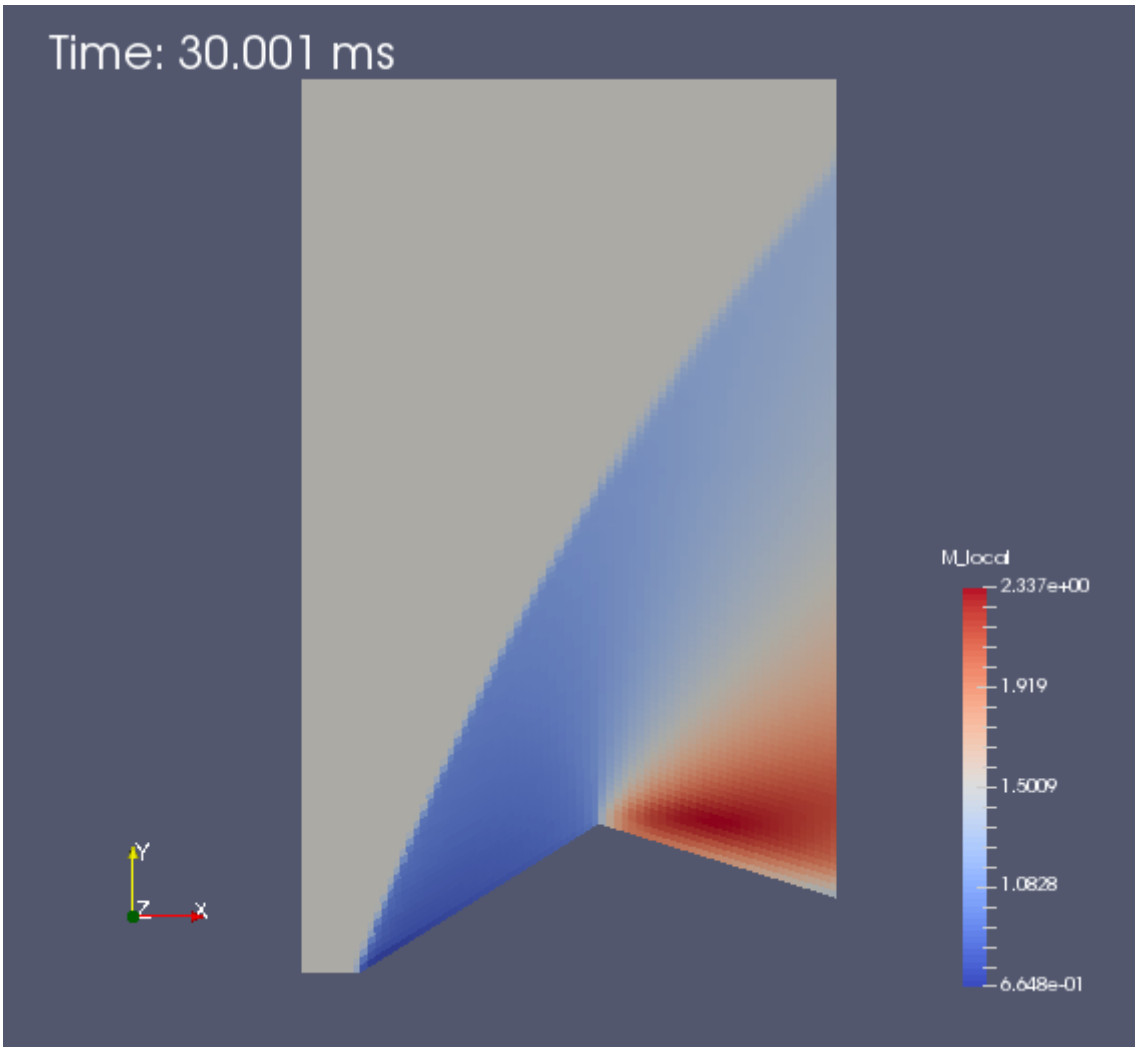


Figure 3.15: Mach number field for the low-resolution simulation at 30 ms of Mach 1.5 flow over a cone with 32 degree half-angle. Flow domain height $H = 3$.

Here endeth the lesson.[5]

[5] https://www.churchofengland.org/prayer-worship/worship/book-of-common-prayer/the-order-for-morning-prayer.aspx

4

Guide to using Eilmer

4.1 Running simulations

Setting up a simulation job is mostly an exercise in writing a text-based description of a gas model, your flow domain and its boundary conditions. This *input script* is presented to the preparation phase as a Lua source file, with the extension ".lua". Once you have prepared your job specification as an input script using your favourite text editor, the simulation data is generated by the `Eilmer` program 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 in the companion report [8]) are adequate, and often convenient because you will not need any other grid preparation tools. For complex geometries, you may find it better to import either block-structured or unstructured grids from a specialized gridding tool such as `Gridpro` or `Pointwise`.

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

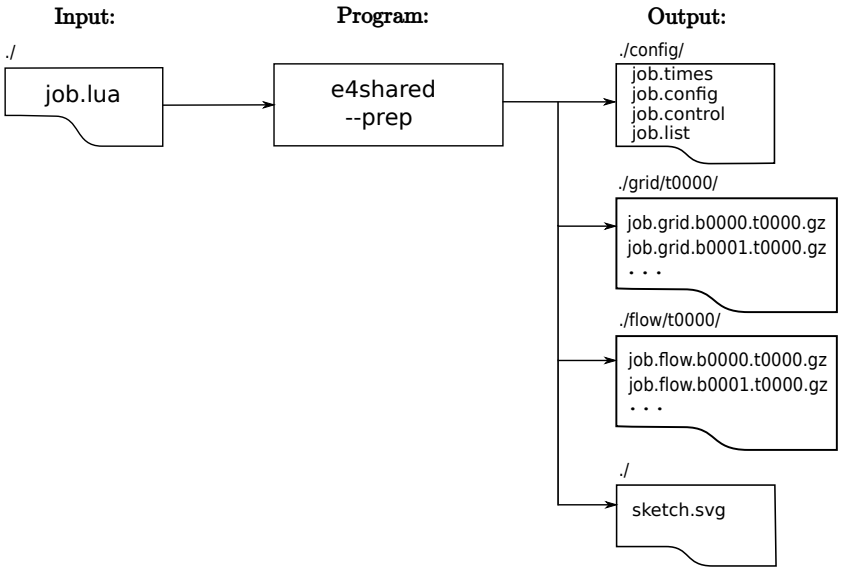
3 Reformat the flow solution data to produce files suitable for a data viewing program such as Paraview or GNU-Plot.

4.1.1 Job preparation phase

The preparation phase of running a simulation is implemented as a special mode of the main simulation program. On seeing the `--prep` command option the program loads all of the Lua wrapped classes that might be useful for defining geometric objects, grids, thermochemical models and flow conditions. It then loads a Lua script that sets up a number of classes and functions that will be used to help build boundary conditions and fluid blocks. Finally it loads your user input script that is identified via the `--job` command option, constructs the objects defined by your script and writes a set of grids, initial flow files and a JSON description of the configuration and control parameters.

Create the geometry definition and a grid with the command

```
$ e4shared --prep --job=job
```



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*`.lua` with the `.lua` extension, indicating that the script is written in Lua. The files from the preparation stage are:

- *job*`.config`: A database of configuration parameters in JSON format. 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 completely regenerating this file. Be careful when manually editing any JSON file; the parsers are not forgiving.

- *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 JSON 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. This parameter is found toward the end of the file. Other control parameters are marked with ‡ in Section 4.10.

- *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*`.list`: A list of block numbers, the type of grid and the label (which may be a default value) given to the block. Block numbering starts from zero.

- `sketch.svg`: Sometimes it is convenient to see a graphical representation of the flow domain and boundary conditions. There is a small set of rendering functions available for rendering geometric objects such as paths, surfaces and volumes to a scalable-vector-graphics file. 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.

- one grid-data file for each block: *job*`.grid.b0000.t0000.gz`, *job*`.grid.b0001.t0000.gz`, ... containing the grid of finite-volume cells. The grids are written as compressed text files in a relatively simple format. The spatial coordinates for points within each file are associated with cell vertices of the structured grid. Note that the grid and flow files are written to subdirectories within the job directory.

- one flow-data file for each block: *job*`.flow.b0000.t0000.gz`, *job*`.flow.b0001.t0000.gz`, ... containing the initial flow state within each of the finite-volume cells.

Note that machine-generated configuration files are written to the `./config/` subdirectory while the grid and flow data files are written to subdirectories `./grid/` and `./flow/`, respectively. For a fixed-grid simulation, 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. For a moving-grid simulation, there will be a grid directory with the grid locations at each output time.

The data is written as plain text compressed using the "gzip" format, hence the ".gz" extension. The details of the data layout are documented in the source code. Look for the functions read_grid, write_grid, read_solution and write_solution in the source code files. A command such as

```
$ grep -n read_solution *.d
```

may be a good way to get started with finding your way around the source code. You can also uncompress any of the output files and then read them with a standard text editor. Look at the first few of lines of a flow file to see what data elements are written for each cell. The format is somewhat self-describing with variable names appearing on the fifth line. Remember that the units for the data are SI-MKS.

4.1.2 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 postprocessing program:

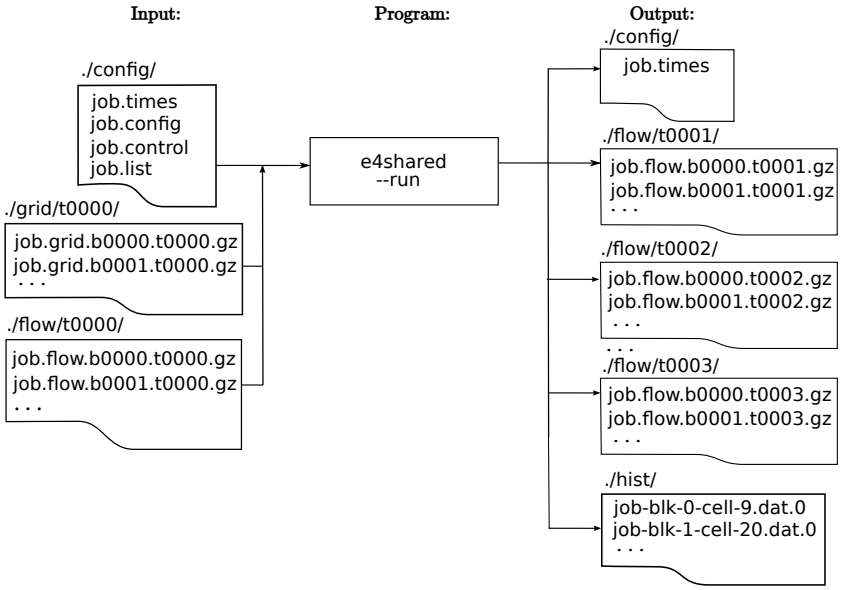
```
$ e4shared --post --job=job --tindx-plot=0 --vtk-xml
```

and then pick up the resulting files for inspection with Paraview. The generated files will appear in the `plot/` subdirectory. Look ahead to Sec. 4.1.5 for a more complete discussion of the postprocessing stage.

4.1.3 Running the simulation

Run the simulation code to produce flow data at subsequent times.[1]

```
$ e4shared --job=job --run
```



The output files are:

- *job*`.flow.b`*nnnn*`.t`*mmmm*`.gz`: 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*`-blk-`*n*`-cell-`*m*`.dat.0`: Data at particular "history points" and at times requested. This data is typically used to simulate the signals recorded by pressure and heat-transfer sensors mounted on model surfaces. The first time a simulation job is run, the history files will have a trailing index of `.0`. When restarting or re-running a simulation, Eilmer will open new history files with the trailing index incremented to be one more than the most recent, existing files. To produce a single data file from a collection of history files for a particular cell, just use the Linux `cat` command to concatenate their content. If you are running a simulation from the start multiple times and you do not want the files generated from previous runs, you will need to manually remove the history files before each run.

[1] 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 the time step is reported as being excessively large. Choose a suitably small value for `dt_init` and try again.

- *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 is useful for automating some of the postprocessing operations.

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

```
$ e4shared --help
Eilmer 4.0 compressible-flow simulation code.
Revision: d50a10ec
Compiler-name: dmd
Build-flavour: debug
Capabilities: multi-species-gas multi-temperature-gas MHD turbulence.
Parallelism: Shared-memory
Usage: e4shared/e4mpi/... [OPTION]...
Argument:                           Comment:
------------------------------------------------------------------------------
  --job=<string>                    file names built from this string
  --verbosity=<int>                 defaults to 0

  --prep                            prepare config, grid and flow files
  --no-config-files                 do not prepare files in config directory
  --no-block-files                  do not prepare flow and grid files for blocks
  --only-blocks="blk-list"          only prepare blocks in given list

  --run                             run the simulation over time
  --tindx-start=<int>|last|9999     defaults to 0
  --next-loads-indx=<int>           defaults to (final index + 1) of lines
                                    found in the loads.times file
  --max-cpus=<int>                  (e4shared) defaults to 8 on this machine
  --threads-per-mpi-task=<int>      (e4mpi) defaults to 1
  --max-wall-clock=<int>            in seconds, default 5days*24h/day*3600s/h
  --report-residuals                write residuals to file config/job-residuals.txt

  --post                            post-process simulation data
  --list-info                       report some details of this simulation
  --tindx-plot=<int>|all|last|9999|"1,5,13,25"   defaults to last
  --add-vars="mach,pitot"           add variables to the flow solution data
                                    (just for postprocessing)
                                    Other variables include:
                                    total-h, total-p, enthalpy, entropy, molef, conc,
                                    Tvib (for some gas models)
  --ref-soln=<filename>             Lua file for reference solution
  --vtk-xml                         produce XML VTK-format plot files
  --binary-format                   use binary within the VTK-XML
  --tecplot                         write a binary szplt file for Tecplot
  --tecplot-ascii                   write an ASCII (text) file for Tecplot
  --tecplot-ascii-legacy            write an ASCII (legacy, text) file for Tecplot
  --plot-dir=<string>               defaults to plot
  --output-file=<string>            defaults to stdout
  --slice-list="blk-range,i-range,j-range,k-range;..."
                                    output one or more slices across
                                    a structured-grid solution
  --surface-list="blk,surface-id;..."
                                    output one or more surfaces as subgrids
  --extract-streamline="x,y,z;..."  streamline locus points
  --track-wave="x,y,z(,nx,ny,nz);..."
                                    track wave from given point
                                    in given plane, default is n=(0,0,1)
  --extract-line="x0,y0,z0,x1,y1,z1,n;..."
                                    sample along a line in fluid domain
  --extract-solid-line="x0,y0,z0,x1,y1,z1,n;..."
                                    sample along a line in solid domain
  --compute-loads-on-group=""       group tag
  --probe="x,y,z;..."               locations to sample flow data
  --output-format=<string>          gnuplot|pretty
  --norms="varName,varName,..."     report L1,L2,Linf norms
```

```
  --region="x0,y0,z0,x1,y1,z1"       limit norms calculation to a box

  --custom-script | --custom-post    run custom script
  --script-file=<string>             defaults to "post.lua"

  --help                             writes this long help message
--------------------------------------------------------------------------------
```

Most of these command-line options are for the postprocessing stage and only a few are used when running the main simulation. You can see them grouped just below the `--run` option. By default, the starting value for `tindx-start` will be zero and the program tries to use as many CPU cores as are available, up to the number of blocks. The default limitation on wall-clock time is the very large value of 5 days. Mostly, you only need to pay attention to this option when using a shared batch system which may place an upper limit on your run time. Set `max-wall-clock` to something less than the batch system limit to ensure that the program will stop time-stepping and write solution files before the batch system terminates your job abruptly.[2]

4.1.4 Restarting a simulation

By default, the simulation program picks up the flow solution for `tindx` equal to 0 but it can be told to pick up any other `tindx` snapshot. To pick up a solution and continue, it is probably best to do a little house-keeping, checking the state of the simulation at the end of run, then editing the *job*`.control` file and changing the parameters `dt_init`, `max_time` and `max_steps` to suitable values. Do not run the preparation stage again, else it will write 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-start` value on the command line for your restart. For example:

```
$ e4shared --job=name --tindx-start=5 --run
```

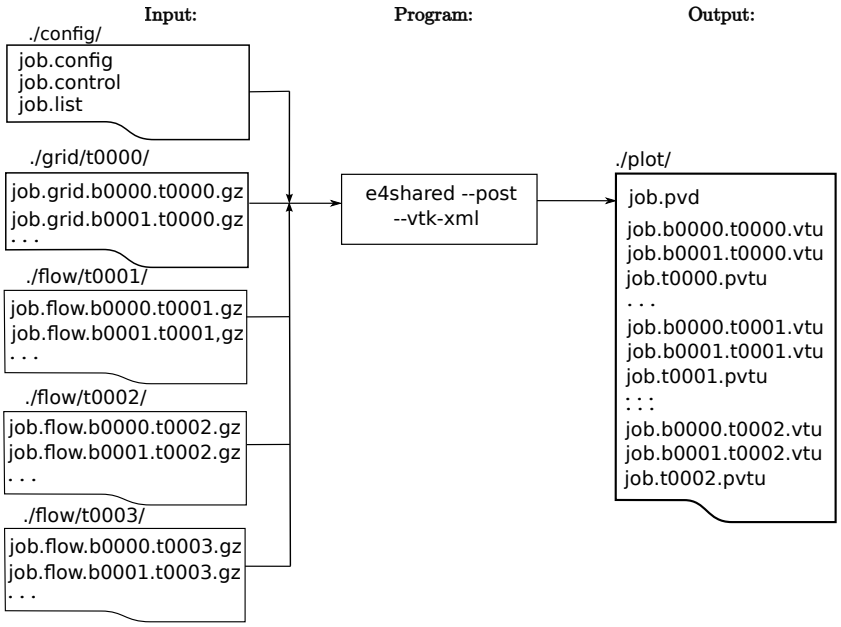
Also, with restarts, be careful that you have consistent modelling requirements and settings. Restarting a laminar simulation as a turbulent simulation with the $k-\omega$ model would lead to inconsistent data. It may be better to start a new job and use `FlowSolution` objects (see Section 4.5) to pick up the old data. Note that your old and new solutions need to have consistent data, such as number of chemical species, etc. `FlowSolution` works with the data available in the old solution and is not smart enough to fill in missing values.

4.1.5 Postprocessing

Postprocessing of the simulation data is the most unstructured of the simulation activities and it is difficult to provide a comprehensive description of the things that you will do. Some hint as to the scope of post-processing activities is given by the fact that most of the command-line options listed by the `--help` option have something to do with extracting data from a previously completed simulation.

[2] On simulations that run for multiple days, we suggest that you request a `max-wall-clock` that is 10 minutes (600 seconds) *less* than the time you request from the batch system. This final 10 minutes is usually ample time for the simulation to write out the final data and finish cleanly, even on a busy cluster.

We provide a postprocessing mode, `--post` 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[3].



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:

```
$ e4shared --post --job=job --vtk-xml --tindx-plot=all
```

To do more sophisticated postprocessing, the command-line options can be combined in fairly complex ways; some experimentation on the part of the user may be required to get the desired effect. The options, however, can be divided into a number of subsets.

Data loading options:

- `--job=<string>` specifies the root name of the solution files. The config, control and list files are scanned to pick up information about the solution. This includes information about the gas model which is initialized and available for use.

- `--tindx-plot=<int>|all|last|9999` 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`.

Data addition options:

[3]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

- `--add-vars`, add the named variables 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 Eilmer native flow solution file and must be reconstructed by the postprocessing phase. The available flow variables are Mach number (`"mach"`), Pitot pressure (`"pitot"`), total enthalpy (`"total-h"`), total pressure (`"total-p"`), specific entropy (`"entropy"`), mole fractions of chemical species (`"molef"`), and molar concentrations of chemical species (`"conc"`). If a number of these variables are desired, these can be joined together in a list with commas separating the names.

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 into 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.

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 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;..."` extracts subsets of the data. A 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 or $ to indicate the end of the available range. 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 eastern strip of cells from block 0 in a 2D structured-grid simulation, you would use the string `"0,$,:,0"` because we want the 0th block, the *i* = *max* ($) position, all cells in the *j*-direction (:), and 0 for the *k*-direction in a 2D grid.

- `--surface-list="blk,surface-id;..."` extracts data on a set of surfaces from the full flow field and writes them as VTK files. The output-file option is used to specify a base file name for constructing the names of the set of VTK files. The `surface-id` may be an integer or one of `north`, `east`, `south`, `west`, `top` or `bottom` for a structured grid.

- `--extract-line="x0,y0,z0,x1,y1,z1,N"` generates a list of up to `N` sampled points between the specified end points. The sampled data is written for

the enclosing-cell centre for each sample point, with repeated cells being omitted.

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

Data manipulation and summary options:

- `--ref-solution=<Lua-script>` compares the flow solution with that supplied in a Lua script. The difference is output. This means that the values recorded in the output files are the differences between the reference solution and the simulated solution. The differences are only computed for those field variables that are given in the reference solution. For example, if the reference solution provides a value for density then the `rho` output field will have the difference value.

- `--report-norms` returns a dictionary of norms for all of the flow variables. The available norms are `L1`, `L2`, and `Linf` (maximum magnitude). This must be used in conjunction with a supplied `ref-solution`.

- `--region="x0,y0,z0,x1,y1,z1"` limits the computation of the norms to a particular box.

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 the program. It is also worth noting that, by default, `e4shared` does not write much to the console while it is 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 specifying the `--custom-script` mode and a Lua script file with `--script-file`. The program starts, loads all of the Lua-bound packages and then executes the specified script. There are classes for picking up whole flow solutions (i.e. `FlowSolution`) and accessing any part of the grid or flow data. A couple of specific applications that show the writing of a custom postprocessing script are:

- estimating the angle of the shock in the axisymmetric flow over a cone (Section 3.2).

- finding the location of the bow shock for the finite cylinder simulation (Section 5.2).

Although we introduce this mode in the postprocessing section, it can be more generally useful. For example, you may be interested in doing some simple gas-dynamic calculations using the functions in Appendix D. These functions are loaded into the Lua interpreter started within `e4shared` and are available for use by the Lua code in your script file.

4.2 Input script overview

Because your specification script, *job*`.lua`, becomes a part of that program when it runs, it is worth the effort to learn just enough Lua to be dangerous. The web site `https://www.lua.org` is a good starting point for learning about the Lua programming language and the older edition of the text "Programming in Lua"[12], which is available online, is a good read and has everything that you need to successfully write good Lua scripts.

After doing some initialization, the program executes your script file and assembles the geometry and flow specification data into a form that can be given to the main simulation code. The advantage of this approach is that you have the full capability of the Lua 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 Lua control structures to make repetitive definitions much more concise. Additionally, you may use Lua comments and print statements to add documentation to the script file. An input script usually does the following:

1. Sets a gas model.
2. Optionally, creates geometric elements to assist in defining the boundary representation of the gas domain. This is in the form of patches (in 2D) or volumes (in 3D).
3. Discretizes these patches or volumes as grids of finite-volume cells.
4. Creates fluid blocks based on these grids by assigning boundary conditions and initial flow state.
5. Sets some simulation control parameters.

Most examples in this manual do just these things, however, it is possible to do much more.

4.3 Specifying a thermochemical model

The thermochemical models are provided by the `gas` module [5]. This is a D-language module with a Lua interface so that its objects and methods can be accessed from the user's input script. For the moment, we'll just tell you how to set the gas model for perfect air. Start by placing the following text:

```
model = "IdealGas"
species = {"air"}
```

into a file called `ideal-air.inp` and run the following command:

```
$ prep-gas ideal-air.inp ideal-air-gas-model.lua
```

to produce the file `ideal-air-gas-model.lua` which contains the fully-specified gas model. Note that you don't type the `$` command prompt that we show above.

To make use of this gas model within your job input script, use the line:

```
nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
```

This will initialize the gas model within the program and return the number of species, the number of energy modes and a reference to the gas model object. You don't need to assign these returned values as shown here, but you may find it convenient to have access to them for calculations or just for information in the preparation phase. For the ideal gas model, as shown here, the number of species is 1 and the number of nonequilibrium energy modes is zero.

For even more sophisticated gas models, the line shown above is all that is needed within your job script to initialize the gas model. Of course, you will have done all of the detailed work to set up your sophisticated model and have the details in a corresponding Lua script. All of the gory details are in the gas package documentation [5] but there are a couple of the examples to study later in the present manual, however, we will be restricting our discussion to gas models that have either frozen internal modes or equilibrium internal modes, such that a single temperature is sufficient to compute the gas thermal energy.

4.3.1 Finite-rate chemical kinetics

Simulations involving nonequilibrium chemistry require an extra input file describing the participating gas species and their reactions. Preparation of this file is described in the companion report [5].

4.4 Defining flow conditions

Because Eilmer is a flow *simulation* code, initial gas flow conditions need to be specified throughout the domain. Also, depending on your flow domain, 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, construct a `FlowState` object[4] as:

fs = `FlowState:new{p=`p, `T=`T, `massf=`mf, `T_modes=`T_modes, `quality=`q,
`velx=`v_x, `vely=`v_y, `velz=`v_z,
`tke=`tke, `omega=`ω, `mu_t=`μ_t, `k_t=`k_t,
`Bx=`β_x, `By=`β_y, `Bz=`β_z, `psi=`ψ`}`

For a single-temperature gas model, only two of the field values are required. These are:

- p: pressure in Pa.
- T: temperature in degrees K.

The other fields may be optionally specified.

[4] The `FlowState` class is defined in source file prep.lua.

- T_modes: For gas models with multimodal energies, these are the corresponding temperatures, provided in an array. If they are not provided, equilibrium will be assumed and each assigned the value T. For a gas model with only one temperature, ignore this field.
- mf: mass fractions of the component species. If there is only one species in a gas model, a default value of 1.0 will be assumed. If you do provide a field value, it is to be provided in a table of species names with mass fraction values. For the ideal air example above, you could provide `massf={air=1.0,}`. If a species name is not present in a multispecies model, the corresponding mass fraction is assumed to be 0.0. Note that the mass fractions supplied must sum to 1.0, within a tolerance of 1.0×10^{-6}.
- v_x, v_y, v_z: velocity components in m/s. These default to 0.0.
- q: quality of a two-phase mixture. Default value is 1.0 (*i.e.* all gas phase).
- tke: turbulent kinetic energy per unit mass in m^2/s^2 or J/kg, default value 0.0.
- ω: turbulence vorticity in 1/s, default value 1.0.
- μ_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 $C_p\mu_t/Pr_t$.
- β_x, β_y, β_z: components of the magnetic field in Tesla. These default to 0.0.
- ψ: divergence cleaning parameter for MHD calculations.

In the Lua environment, FlowState objects are tables that give access to the full internal state, including derived quantities such as density and sound speed. The full collection of fields is:

- `gm`: GasModel object associated with this FlowState,
- `nSpecies`: number of chemical species in the gas model,
- `speciesNames`: array of strings giving the names,
- `nModes`: number of other internal energy modes, [5],
- `p`: pressure in Pa,
- `T`: temperature in degrees K,
- `T_modes`: array of other internal temperatures,
- `quality`: fraction of gas phase,
- `massf`: table of named mass-fraction values,

[5] For a single-temperature gas model this number will be zero. For multitemperature gas models, the number will indicate how many internal energy modes, beyond transrotational.

- `a`: sound speed in m/s,
- `rho`: density in kg/m^3,
- `mu`: dynamic viscosity in Pa.s,
- `k`: thermal conductivity in W/(m.K),
- `tke`: turbulent kinetic energy in J/kg,
- `omega`: turbulence vorticity in 1/s,
- `mu_t`: turbulence viscosity in Pa.s,
- `k_t`: turbulence conductivity in W/(m.K),
- `velx`, `vely`, `velz`: velocity components in m/s,
- `Bx`, `By`, `Bz`, `psi`, `divB`: magnetic field parameters.

The table also contains a GasState object, `Q`. Note that the FlowState objects are defined in the context of a gas model, so you need to have called `setGasModel` (as shown in Section 4.3) before constructing any FlowState objects.

4.5 Using flow conditions from other simulations

For custom postprocessing, you will need to be able to pick up the grid and flow data for the simulation and be able to inspect any particular cell, however, there are occasions in preparing a new simulation where you might like to use flow data from an old simulation. This data might be used as initial conditions for some or all of your blocks in your new simulation. A typical example is to restart a simulation with a finer, or otherwise changed, mesh. In any case, you may construct a `FlowSolution` object as:

fsol = `FlowSolution:new{jobName="`*job*`", dir="`*myDir*`",`
`tindx=`*tindx*`, nBlocks=`*nb*` }`

where the named fields and their possible values are:

- `jobName`: the root file name that will be used to access the individual flow and grid files that hold the solution data.
- `dir`: the directory where we'll find our existing solution files. Commonly, it will be the current directory, so you may specify "."
- `nBlock`: number of blocks in the solution data set.
- `tindx`: the time index to select 0..9999. Do not specify with leading zeros because there is the potential to confuse decimal and octal numbers.

Having constructed the FlowSolution object, there are a number of methods to provide access to the data.

- $fsol$:`find_enclosing_cell{x=`x`, y=`y`, z=`z`}` returns a table with fields `ib` and `i`, for the block index and the cell index, respectively. Note that the single index for the cell works in the unstructured-grid context and the structured-grid context. If any of the `x`, `y` or `z` fields are not supplied, a value of 0.0 is assumed. If the method fails to find an enclosing cell, nil is the return value for both `ib` and `i`.

- $fsol$:`find_enclosing_cells_along_line{p0=`$\vec{p}_0$`, p1=`$\vec{p}_1$`, n=`n`}` returns an array of tables, one for each located cell centre. Each table has fields `ib` and `i` giving the block index and the cell index, respectively. The $\vec{p}_0$ and $\vec{p}_1$ points may be specified as tables of labelled coordinates. If any of the `x`, `y` or `z` coordinates are not supplied, a value of 0.0 is assumed. Alternatively, the points may be supplied as Vector3 objects. The number of sample points is specified as n, which may be set to be quite large so as to be sure to pick up all cells along the line. The process of compiling the array of cells will eliminate duplicate entries.

- $fsol$:`find_nearest_cell_centre{x=`x`, y=`y`, z=`z`}` returns a table with fields `ib` and `i`, for the block index and the cell index, respectively. If cells are long and slender, this method might return the indices of a neighbouring cell, rather than those of the enclosing cell (if it exists).

- $fsol$:`get_nic(`ib`)` returns the number of cells in the i-direction for block ib.

- $fsol$:`get_njc(`ib`)` returns the number of cells in the j-direction for block ib.

- $fsol$:`get_nkc(`ib`)` returns the number of cells in the k-direction for block ib.

- $fsol$:`get_var_names()` returns a table with the variable names for the cell data as strings.

- $fsol$:`get_cell_data{fmt=`$dataFormat$`, ib=`ib`, i=`i`, j=`j`, k=`k` }` returns the cell data for a particular cell.
  The $dataFormat$ value may be `"Plotting"` or `"FlowState"`, the difference being in the names of the fields within the returned table. When dipping into an unstructured-grid block, omit the `j` and `k` entries because only the `i` index may be specified. You can access a structured-grid block with a single `i` index, or with all three indices, if you wish. The single-index access works well with the result from `find_enclosing_cell` or `find_nearest_cell_centre` methods.

- $fsol$:`get_vtx{ib=`ib`, i=`i`, j=`j`, k=`k` }` or ($fsol$:`vtx{ib=`ib`, i=`i`, j=`j`, k=`k` }`) returns a Vector3 value representing the position of the vertex. Values of unspecified fields for this method default to zero.

- $fsol$:`get_sgrid{ib=`ib`}` returns the `StructuredGrid` object for the specified block.

4.6 Representation of the flow domain

Now that we have a gas model and a way to specify flow conditions, we need to define the spatial domain of the flow, together with a set of boundary conditions. The domain is specified as a grid of finite-volume cells, either structured or unstructured, shaped to fit the boundaries. If you are going to set up the boundary-representation of your flow domain using the library of geometric primitives provided with Eilmer, the companion report [8] describes the details. Alternatively, you may already have a favourite grid generator. Either way, we now assume that we have the flow domain defined as one or more grids of cells and we are ready to make the connection to the initial flow state and the boundary conditions.

4.6.1 Fluid blocks built on structured grids

For a structured-grid in 2D, each fluid block is a region bounded by 4 edges, labelled north, east, south and west. We are looking at a plan-view of a 2D flow domain in Figure 4.1. The i and j indices are related to the r and s parametric coordinates used within the geometric functions and the corner points are identified by their (r, s) coordinates. These corner points are used in the search to determine block connectivity if the fluid 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 relatively simple interpolations.

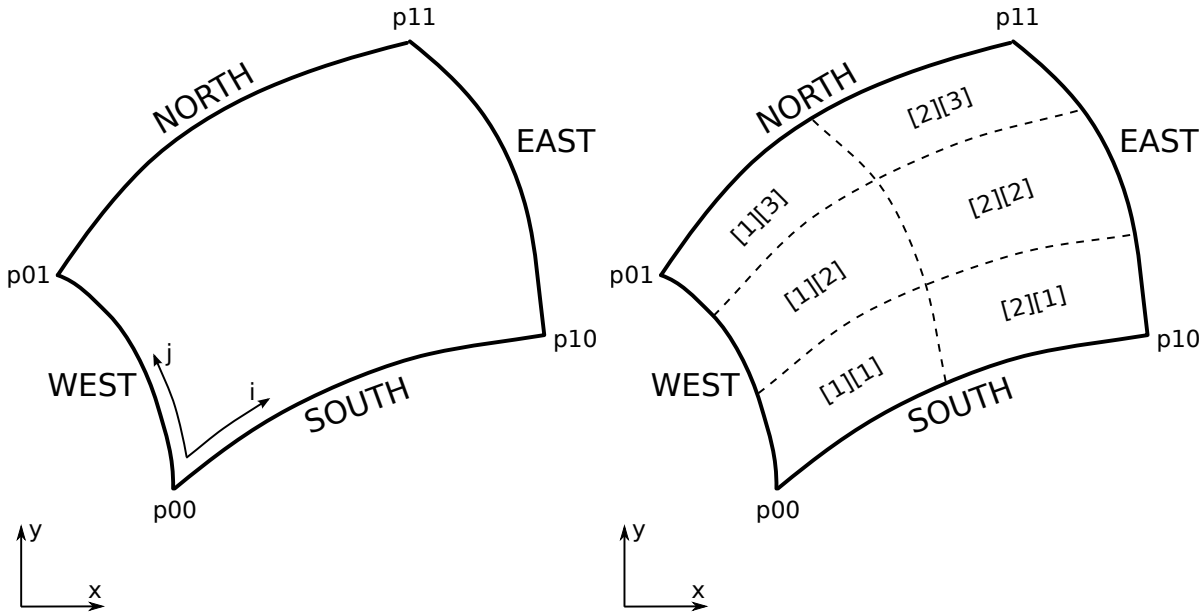


Figure 4.1: A two-dimensional flow domain containing the structured mesh for a single FluidBlock object (left) and a collection of sub-blocks defined via a FluidBlock Array (right). The orientations of the bounding paths are important: west and east paths progress with parametric coordinate, s, from south to north; south and north edges progress with parametric coordinate, r, from west to east.

In 3D, life is just that bit more complicated with each block defined by 6 boundaries (north, east, south, west, top and bottom) fitted to the actual surfaces of the domain. Figure 4.2 shows the "index-space" view with cell indices i,j and k corresponding to the r, s and t parametric coordinates used within the geometric functions. The corner vertices of the block are numbered 1 through 7, as shown.

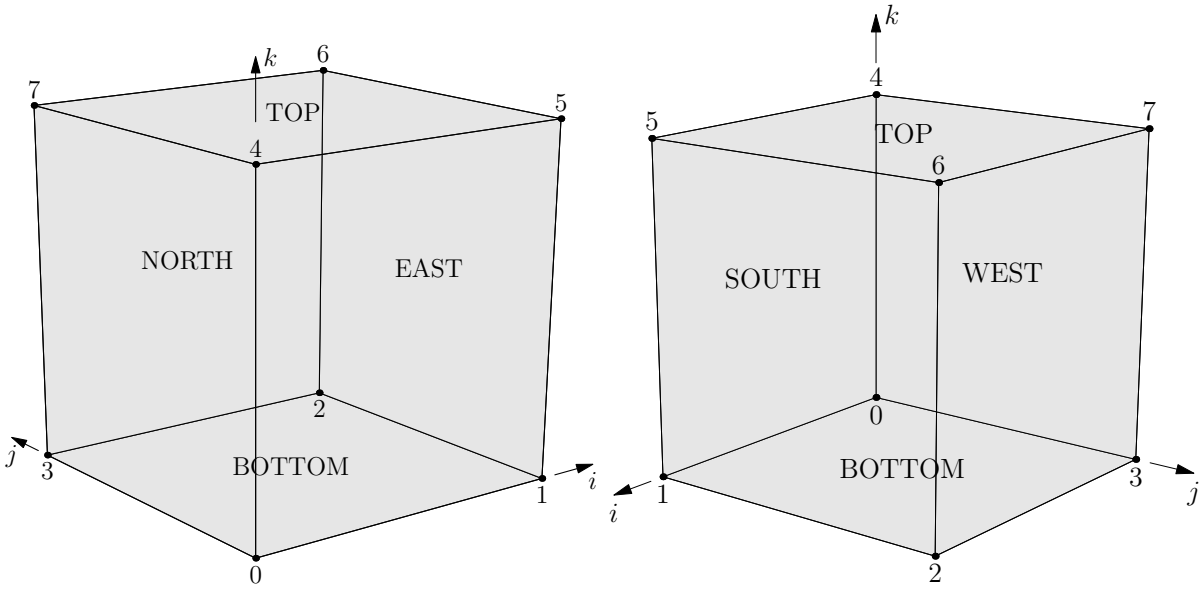


Figure 4.2: Two views of the hexahedral block containing a 3D structured mesh. These figures are ambiguous but each is supposed to show a hollow box with the *far* surfaces in each view being labelled. The *near* surfaces are transparent and unlabelled. To get your hands on an unambiguous representation, build the debugging cube drawn in the Appendix of Report [8].

To define a structured-grid fluid block in your input script, construct an `FluidBlock` object as:

my_block = `FluidBlock:new{grid=`*grid*`, initialState=`fs`, omegaz=`ω_z`, bcList=`*bcList*`, label=`*tagString*`, active=`*aFlag* }

where the assignment to the variable *my_block* allows easy referencing of the block at later times, say, for adding boundary conditions. The values bound to the field names in the constructor table represent[6]:

- *grid*: a `StructuredGrid` object that has been previously constructed (or imported) as described in Chapter 3 of Report [8].
- fs: a `FlowState` (see Section 4.4) or Lua function object that, given coordinates x,y and z, returns the FlowState as a table. This function may be derived from a `FlowSolution` object, as described toward the end of Section 4.5.
- ω_z: angular speed (in rad/s about the z-axis) for the rotating frame of reference. Useful for the calculation of flow in turbomachines. Default value is zero.

[6]The definitive source is, of course, the `FluidBlock` class definition in `prep.lua`.

- *bcList*: a table of named entries, bound to boundary condition objects. The field names are the boundary surface names (north, east, south, west, top, bottom). Look forward to Section 4.7 for a discussion of the available boundary conditions. If a particular boundary has no assigned condition, the default is `WallBC_WithSlip`. You don't have to supply any or all of the boundary conditions to the FluidBlock constructor. You may attach boundary conditions to the block boundaries at a later point in your script.

- *tagString*: a label for use in postprocessing. Internally, the simulation program only cares for the block's index in the array of blocks that it holds, however, this string will be written to the job`.list` file and may be a convenient way to identify particular blocks.

- *aFlag*: Defaults to `true`, such that the block is actively updated during the simulation. Although it would be unusual for you to be setting this field manually, the block-marching mode of the simulation program uses this flag to selectively update blocks as part of its overall solution strategy. See the `config.block_marching` flag in Section 4.10.

When defining large domains and running simulations on a multiprocessor computer, it may be convenient to define many FluidBlock objects with one call. The function call for this situation is:

my_fba = `FBArray:new{grid=`*grid*, `initialState=`*fs*, `omegaz=`ω_z,
`bcList=`*bcList*, `nib=`n_i, `njb=`n_j, `nkb=`n_k`}`

which contains `blockArray`, a multidimensional array of FluidBlock objects that may be subscripted [i][j][k] in 3D and [i][j] in 2D to access the individual FluidBlock objects. Note that each index starts with the Lua default of 1. The single grid passed in is subdivided into $n_i \times n_j \times n_k$ sub-grids. Be careful with numbers of cells in any direction which do not divide into equal sections. You may not get the block alignment that you assume. Internally, the sub-blocks should be returned with the newly-formed interior boundaries connected together with `ExchangeBC_FullFace` boundary conditions. The peripheral boundaries will inherit from the *bcList* passed in, or default to `WallBC_WithSlip` boundary conditions. If you don't specify the number of blocks to make in each direction, the default value is 1.

When assembling large numbers of blocks for complex geometries, there is a function `identifyBlockConnections(`*blockList*, *excludeList*, *tolerance*`)` that performs a brute-force search for all adjacent blocks and attaches `ExchangeBC_FullFace` boundary conditions for pairs of faces 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 an array of references to the blocks that you do want to search as *blockList*. You may also supply an array of references for blocks that should be excluded. If you don't supply this argument, an empty array is assumed for *excludeList*. The default tolerance for the colocation of vertices is 1.0e-6.

Be aware that the `identifyBlockConnections()` function is oblivious to 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 `connectBlocks()` 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. To manually connect a pair of blocks, use the function: `connectBlocks(`A, $face_A$, B, $face_B$, $orientation$`)` where A and B are references to the individual `FluidBlock` objects and $face_A$ and $face_B$ are their adjoining faces (`north`, `east`, `south`, `west`, `top` or `bottom`) and $orientation$ is an integer value as described in the Lua file `blk_conn.lua`. It is important to supply the $orientation$ only for 3D. For example, the list of vertex pairings {{3,2}, {7,6}, {6,7}, {2,3}} specifies a north-to-north connection with orientation 0. In 2D, there is only one orientation that is valid for each possible connection so you don't need to specify it.

4.6.2 Fluid blocks built on unstructured grids

It is also possible to define fluid blocks over unstructured-grids and even have them part of a simulation that is mainly built upon structured-grid blocks. Each unstructured grid can be viewed as a bag of cells without any globally-structured indexing scheme; the cells being simply numbered 0 through $n_{cells} - 1$. The region defined by the cells will be bounded by one or more *boundary-sets* of cell faces and boundary conditions are assigned to these boundary sets. Cell faces that are in a particular boundary-set will have the corresponding boundary condition applied during the simulation.

To define an unstructured-grid fluid block in your input script, call the `FluidBlock` constructor as:

my_ublk = `FluidBlock:new{grid=`$grid$`, initialState=`fs`, omegaz=`ω_z`,`
`bcList=`$bcList$`, bcDict=`$bcDict$`,`
`label=`$tagString$`, active=`$aFlag$ `}`

where the assignment to the variable my_ublk allows easy referencing of the block at later times. The values bound to the field names in the constructor table represent[7]:

- $grid$: an `UnstructuredGrid` object that has been previously constructed (or imported) as described in Chapter 3 of Report [8].
- fs: a `FlowState` (see Section 4.4) or Lua function object that, given coordinates x,y and z, returns the FlowState as a table.
- ω_z: angular speed (in rad/s about the z-axis) for the rotating frame of reference. Useful for the calculation of flow in turbomachines. Default value is zero.
- $bcList$: an array of boundary condition objects that will be bound to the boundary-sets, in order of appearance. Since it is probably difficult to make this mapping manually, you will more likely use the following field.

[7] The definitive source is, again, the `FluidBlock` class definition in `prep.lua`.

- *bcDict*: a table with named boundary-condition objects. The names in this table are matched against the tag strings for the boundary-sets in the underlying unstructured-grid object. Look forward to Section 4.7 for a discussion of the available boundary conditions. If a particular boundary has no assigned condition, the default is `WallBC_WithSlip`. You don't have to supply any or all of the boundary conditions to the FluidBlock constructor. Also, you may attach boundary conditions to the block boundaries at a later point in your script.

- *tagString*: a label for use in postprocessing. Internally, the simulation program only cares for the block's index in the array of blocks that it holds, however, this string will be written to the job`.list` file and may be a convenient way to identify particular blocks.

- *aFlag*: Defaults to `true`, such that the block is actively updated during the simulation.

When making connections between unstructured-grid blocks and other (possibly structured-grid) blocks, your only option is to apply `ExchangeBC_MappedCell` boundary-conditions to the corresponding boundary-sets of the blocks.

4.7 Boundary conditions

Boundary conditions in the Eilmer code are composite objects that apply effects to the ghost-cell and interface data at various points during the update for each time step. Each boundary condition object holds four lists of effects which you may specify manually or you may just pick a pre-assembled boundary condition that sets up the lists. Constructors for pre-assembled boundary condition classes include the following.

4.7.1 Walls

We cannot think of any flows of engineering interest that do not have at least one wall that bounds the gas-flow region and interacts with the gas flow. If you specify no boundary conditions at all when constructing a `FluidBlock`, the preparation program will actually apply `WallBC_WithSlip` conditions at all boundaries.

- `WallBC_WithSlip:new{label=`*tagString*`, group=`*tag*`}` where we want a solid wall with no viscous effects. This is the default boundary condition where no other condition is specified.

- `WallBC_NoSlip_FixedT:new{Twall=`T_{wall}`, label=`*tagString*`, group=`*tag*`}` where we want viscous effects to impose a no-slip velocity condition and a fixed wall temperature. We need to set `config.viscous = true` (see Section 4.10, Viscous effects) to make this boundary condition effective.

- `WallBC_NoSlip_UserDefinedT:new{Twall=`$fileName$`, label=`*tagString*`, group=`*tag*`}` where we want a wall with an arbitrary temperature profile, specified via a user defined function defined in the file $fileName$. Internally, this file

is passed to a UserDefinedInterface boundary effect which expects a function called $interface$, as described in Appendix E.

- `WallBC_NoSlip_Adiabatic:new{label=`$tagString$`, group=`tag`}` 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 `config.viscous = true` to make this boundary condition effective.

- `WallBC_TranslatingSurface_FixedT:new{Twall=`T_{wall}`, v_trans=`$\vec{v}_{trans}$`, label=`$tagString$`, group=`tag`}` where we want viscous effects to impose a specified translating velocity condition and a fixed wall temperature. By *translating* we mean that the (flat) wall is moving tangential to the block boundary. The value for `v_trans` may be specified as a table of three named (x,y,z) components. We need to set `config.viscous = true` to make this boundary condition fully effective. An example of use is Couette flow between moving plates.

- `WallBC_TranslatingSurface_Adiabatic:new{v_trans=`v_{trans}`, label=`$tagString$`, group=`tag`}` where we want viscous effects to impose a specified translating velocity at the wall but where there is no heat transfer. Otherwise, similar considerations to `WallBC_TranslatingSurface_FixedT`, described above.

- `WallBC_RotatingSurface_FixedT:new{Twall=`T_{wall}`, r_omega=`$\vec{\omega}$`, centre=`$\vec{p}$`, label=`$tagString$`, group=`tag`}` where we want viscous effects to impose a specified translating velocity condition and a fixed wall temperature. By *rotating* we mean that the cylindrical wall is moving tangential to the block boundary. Values for `r_omega` and `centre` may be specified as tables of three named (x,y,z) components giving the angular-velocity and a point on the axis of rotation. The actual velocity of a point on the wall is then given by the vector expression $\vec{\omega} \times (\vec{r} - \vec{c})$, 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. An example of use is the surface of a shaft in a journal bearing. We need to set `config.viscous = true` to make this boundary condition fully effective.

- `WallBC_RotatingSurface_Adiabatic:new{r_omega=`$\vec{\omega}$`, centre=`$\vec{p}$`, label=`$tagString$`, group=`tag`}` where we want viscous effects to impose a specified translating velocity at the wall but where there is no heat transfer. Otherwise, similar considerations to `WallBC_RotatingSurface_FixedT`, described above.

4.7.2 In-flow

Often, the flow domain in your analysis is part of a bigger flow domain. To abstract your smaller region of interest, you will apply in-flow and out-flow conditions at the edges of the abstracted flow domain that are not walls. In-flow boundary conditions, as the name suggests, will drive gas into the flow domain.

- `InFlowBC_Supersonic:new{flowState=`fs`, x0=`x`, y0=`y`, z0=`z`, r=`r`, label=`$tagString$`, group=`tag`}` where we want to specify the inflow condition, fs, that gets copied into the ghost cells each time step. fs is a FlowState object,

as described in Section 4.4. By default, $r = 0$ and a uniform inflow condition is assumed, as would be appropriate for simulations of free flight. To enable modelling of flows that have been produced by a wind tunnel or shock tunnel with a conical nozzle, you may specify the origin (x, y, z) of the virtual source flow and a non-zero radial distance `r` (from that origin) at which the nominal flow condition is specified. When used in this mode, only the x-component of the specified nominal flow velocity is used. On being called to provide data for each specific ghost cell, the location of the ghost cell is used to determine a specific flow condition as a perturbation of the nominal condition. The velocity will be computed to have appropriate axial and radial components. Default values for x, y and z are zero.

- `InFlowBC_StaticProfile:new{fileName=`$fileName$`, match=`$matchString$`, label=`$tagString$`, group=`tag`}` where we want to specify an inflow condition that might vary in a complicated manner across the boundary. Data for the flow condition, on a per-cell basis, is contained in the specified file. It may be that the file is obtained from an earlier simulation, with a post-processing option like `--extract-line` used to write the file entries. Matching of the ghost cells to particular entries in the data file is controlled by $matchString$, where the default is to match to the nearest location on all three coordinates of the ghost-cell position `match="xyz-to-xyz"`. Other possible values are:
  - `"xyA-to-xyA"` For 2D or 3D simulations, don't care about z-component of position.
  - `"AyA-to-AyA"` For 2D or 3D simulations, care about the y-component of position only.
  - `"xy-to-xR"` Starting with a profile from a 2D simulation, map it to a radial profile in a 3D simulation, considering the x-component of the position of the ghost cells.
  - `"Ay-to-AR"` Starting with a profile from a 2D simulation, map it to a radial profile in a 3D simulation, ignoring the x-component of the position of the ghost cells.

- `InFlowBC_Transient:new{fileName=`$string$`, label=`$tagString$`, group=`tag`}` where we want to specify the time-varying inflow condition at the boundary. Data for the inflow condition, at particular time instants and assumed uniform across the full boundary, is contained in the specified file. The user needs to write this file according to the expected format encoded in the `FlowHistory` class, found toward the end of `flowstate.d`. Each data line will have the following space-delimited items:
  time, vel_x, vel_y, vel_z, p, T, mass-fractions, T_{modes} if any.

- `InFlowBC_ConstFlux:new{flowState=`fs`, x0=`x`, y0=`y`, z0=`z`, r=`r`, label=`$tagString$`, group=`tag`}` where we want to specify directly the fluxes of mass, momentum and energy across the boundary faces. The fluxes are computed from the supplied flow condition. See `InFlowBC_Supersonic` for notes on the virtual source flow option.

- `InFlowBC_ShockFitting:new{flowState=`fs`, x0=`x`, y0=`y`, z0=`z`, r=`r`, label=`*tagString*`, group=`*tag*`}` where we want to have the inflow boundary be the location of a bow shock. The fluxes across the boundary are computed from the supplied flow condition and the boundary velocities are set to follow the shock. See `InFlowBC_Supersonic` for notes on the virtual source flow option. Note that we need to set `config.moving_grid = true`, select an appropriate gas-dynamic update scheme for the moving grid, and have all of the blocks with the shock-fitting boundary as part of a single `FBArray`.

- `InFlowBC_FromStagnation:new{stagnationState=`fs`,`
  `fileName=`*string*`,`
  `direction_type="normal",`
  `direction_x=1.0,direction_y=0.0,direction_z=0.0,`
  `alpha=0.0,beta=0.0,`
  `mass_flux=0.0,relax_factor=0.10,`
  `label=`*tagString*`, group=`*tag*`}`
  where we want a subsonic inflow with a particular stagnation pressure and temperature and a velocity *direction* at the boundary. (Note that many of the fields are shown with their default values, so you don't need to specify them.) 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 whose components are `direction_x`, `direction_y` and `direction_z`
  - `"radial"` radially-in through a cylindrical surface using flow angles `alpha` and `beta`, or
  - `"axial"` axially-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. Note that when we select a nonzero mass flux, we no longer control the stagnation pressure. This will be adjusted to give the desired mass flux. The value for `relax_factor` adjusts the rate of convergence for this feedback mechanism. Note, that for multi-temperature simulations, all of the temperatures are set to be the same as the transrotational temperature. 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.

  When `mass_flux` is zero and `fileName` is left as the default empty string, the specified FlowState, fs, is used as a constant stagnation condition. This may

be modified by a user-defined function if `fileName` is a non-empty string that give the name of a Lua script containing a function with the name `stagnationPT` On every boundary condition application, this function receives a table of data (including the current simulation time) and returns values for stagnation pressure and temperature. Here is a minimal example:

```
function stagnationPT(args)
   -- print("t=", args.t)
   p0 = 500.0e3 -- Pascals
   T0 = 300.0 -- Kelvin
   return p0, T0
end
```

The intention is that the user may program the stagnation pressure as more interesting functions of time.

4.7.3 Out-flow

If you have an in-flow boundary condition, you will likely require one or more out-flow boundary conditions to avoid your flow domain being just an accumulator. When truncating a larger flow domain to make your simulation flow domain, select a location for your out-flow boundary conditions to be as far as you can reasonably afford from the parts of the flow domain where you are taking measurements.

- `OutFlowBC_Simple:new{label=`*tagString*`, group=`*tag*`}` is an alias for `OutFlowBC_SimpleFlux:new{}`.

- `OutFlowBC_SimpleFlux:new{label=`*tagString*`, group=`*tag*`}` where we want a (mostly) supersonic outflow condition. It should work with subsonic outflow as well, however, remember that you are deliberately ignoring information that may propagate into the domain from the real (physical) region that you have truncated. The outflow flux is determined from the flow state in the cell just inside the boundary. If the velocity in that cell tries to produce an influx of mass, the flux calculation switches to that of an impermeable wall.

- `OutFlowBC_SimpleExtrapolate:new{xOrder=0, label=`*tagString*`, group=`*tag*`}` where we want a (mostly) supersonic outflow condition. Flow data is effectively copied (`xOrder=0`) or linearly-extrapolated (`xOrder=1`) from just inside the boundary to the ghost cells just outside the boundary, every time step. In subsonic flow, this can lead to physically invalid behaviour. If you encounter strange flow behaviour that seems to start at this boundary and propagate upstream into your flow domain, try extending your simulated flow domain such that you eventually have an outflow boundary across which nothing exciting happens.

- `OutFlowBC_FixedP:new{p_outside=1.0e5, label=`*tagString*`, group=`*tag*`}` where we want something like `OutFlowBC_Simple` but with a specified back pressure. This can be analogous to a vacuum pump that removes gas at the boundary to maintain a fixed pressure in the ghost cells.

- `OutFlowBC_FixedPT:new{p_outside=1.0e5, T_outside=300.0, label=`*tagString*`, group=`*tag*`}` is like `OutFlowBC_FixedP`, above, but also sets the temperature in the ghost cells.

4.7.4 Inter-block exchange

In a simulation with more than one block of cells, the flow solver mostly deals with the blocks independently. At each stage of an update, the flow solution is stitched together at the inter-block boundaries by exchanging flow data across those boundaries.

- `ExchangeBC_FullFace:new{otherBlock=nil, otherFace=nil, orientation=-1, reorient_vector_quantities=false, Rmatrix={1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0}, label=`*tagString*`, group=`*tag*`}` Usually, this boundary condition is applied implicitly, by calling the function `identifyBlockConnections`, for cases where one structured-grid 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 one 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. Note that this boundary condition is only for structured-grid blocks. If one or both of the blocks to be joined is based on an unstructured-grid, you will need to use the following `MappedCell` flavour of the exchange boundary condition.

- `ExchangeBC_MappedCell:new{transform_position=false, c0=Vector3:new{x=0.0,y=0.0,z=0.0}, n=Vector3:new{x=0.0,y=0.0,z=1.0}, alpha=0.0, delta=Vector3:new{x=0.0,y=0.0,z=0.0}, list_mapped_cells=false, reorient_vector_quantities=false, Rmatrix={1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0}, label=`*tagString*`, group=`*tag*`}` is something like the `ExchangeBC_FullFace` boundary condition but with a mapping of destination(ghost)-cell location to source-cell location. It allows us to stitch boundaries together, even if the cells do not align, one-for-one. The position of the source cell is computed by taking the position of the ghost cell, computing the solid-body rotation of `alpha` radians about the axis `n` through the point `c0`, then adding a displacement `delta`. This will accommodate general rigid-body transformations.

4.7.5 User-defined

These are our "get out of jail" boundary conditions. They allow you to do just about anything you wish and, because of the lack of constraint, are somewhat involved to describe. See Appendix E for the more complete story.

- `UserDefinedGhostCellBC:new{fileName=`*string*`, label=`*tagString*`, group=`*tag*`}` 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, written in the Lua programming language, and provided in the specified file. See Section E.1.1 in Appendix E for a more complete description.

- `UserDefinedFluxBC:new{fileName=`*string*`, funcName=`*string*`, label=`*tagString*`, group=`*tag*`}` allows the user to define the interface convective-fluxes at run time. This is done via a function defined by the user, written in the Lua programming language, and provided in the specified file. If the user does not specify the function name, `convectiveFlux` is used as the default name. See Section E.1.2 in Appendix E for a more complete description.

- `ExchangeBC_FullFacePlusUDF:new{otherBlock=nil, otherFace=nil, orientation=-1, reorient_vector_quantities=false, Rmatrix={1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0}, fileName=`*string*`, label=`*tagString*`, group=`*tag*`}`
  There are cases where you might conditionally want to exchange block-boundary data or do something else altogether. This boundary condition allows that by first doing a FullFace exchange of data and then calling upon your user-defined functions (as for `UserDefinedGhostCellBC`) to conditionally overwrite the data. This turns out to be a convenient way to implement diaphragm models for shock-tunnel simulations. Note that this boundary condition can work across MPI tasks but is only for structured-grid blocks.

Note that all boundary conditions have optional `label` and `group` fields that have empty default values. These may be used to group boundary surfaces symbolically.

4.7.6 Binding boundary conditions to block faces

In Section 4.6.1 (page 50), it was shown that the boundary conditions could be specified as a table of `BoundaryCondition` objects passed to the constructor of `FluidBlock` objects. This is preferred. Alternatively, `BoundaryCondition` objects can be assigned individually to elements of the `bcList` attribute after block construction. For example:

```
blk_0:bcList[west] = InFlowBC_Supersonic:new{flowState=fs}
blk_1:bcList[east] = OutFlowBC_Simple:new{}
```

When using the `FBArray:new` constructor it is far more convenient to use the `bcList` style of boundary condition specification as this will take care of all the boundary conditions on the subdivided blocks automatically.

The same can be done for `FluidBlock` objects based on unstructured grids but you will have to use the numerical index of the boundary set. It is probably easiest to supply the boundary-condition objects as a table with named entries at the time of constructing the block. See the example in Section 5.3 (page 92) for a guide to setting up such a table.

4.8 Special zones

Zones of special conditions may be defined within the flow domain as rectangular (2D) or regular hexahedral (3D) patches which are specified by two diagonally-opposite corners (`p0` and `p1`). There are three special classes in the present code: reaction, ignition and turbulence.

For example, we could specify `ReactionZone:new{p0=`$\vec{a}$`, p1=`$\vec{b}$`}` where the corners of the zone are given by the `Vector3` values $\vec{a}$ and $\vec{b}$. In a flow with an active reaction scheme, this type of zone makes it possible to selectively allow reactions to proceed, or not. If the centre of a cell lies within the reaction zone, the finite-rate chemistry is allowed to proceed, else the species fractions are maintained constant. Maintaining the species fractions constant effectively "freezes" the reactions. In a two-dimensional simulation, `p0` corresponds to `p00` in Figure 4.1 (on page 49) while `p1` corresponds to `p11`, at the opposite corner of the patch. In three-dimensional simulation, `p0` corresponds to `p0` in Figure 4.2 (on page 50) while `p1` corresponds to `p6` at the diagonally-opposite vertex of the hexahedral block. If no reaction zones are specified and a reaction scheme is active, then reactions are permitted for the entire flow field. An example application of this type of zone is the simulation of the student ramjet[8] where the inflow for the whole simulation domain included the fuel mixture but we wanted reactions to proceed only in the ramjet combustor.

An effective method to trigger chemical reactions is to use `IgnitionZone:new{p0=`$\vec{a}$`, p1=`$\vec{b}$`, T=`T_{ig}`}` where temperature, T_{ig}, controls the reaction rate used for chemical reactions, without effecting the gas temperature in the flow field. The rate-controlling temperature is used to evaluate the chemical reaction rates only within the physical extents of the ignition zone. The effect of this zone can be limited in time by specifying a nonzero values for `config.ignition_time_start` and `config.ignition_time_stop`. While the zone is active, the reaction rates within the zone are altered. The rate-controlling temperature is typically set to an artificially inflated value to promote ignition. For example, a value of 2000 K is effective in igniting certain compositions of a methane/air mixture.

Also, when running turbulent flow simulations, the turbulence effects can be localized using `TurbulentZone:new{p0=`$\vec{a}$`, p1=`$\vec{b}$`}` The turbulence model (say, the $k-\omega$ model) is active throughout the flow but its effect on the flow field is masked outside of any defined turbulent zones. 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 `TurbulentZone`. If there are no such defined regions, the whole flow field is allowed to have nonzero turbulence viscosity.

4.9 History points

A history point is a location in the flow field at which data is written to a file at a frequency different to (and possibly much higher than) the frequency that full flow field is written. The files for history points are found in the `hist/` subdirectory and are identified by their associated block and cell indices. A history point can be located

[8] See examples/eilmer/2D/ramjet-student-design in the source code repository.

by its Cartesian coordinates using the function call:
`setHistoryPoint{x=`x`, y=`y`, z=0.0}`
where the z-coordinate is shown taking its default value of zero. Alternatively, the point can be located via its block and cell indices as:
`setHistoryPoint{ib=`b`, i=`i`, j=`j`, k=0}`
The frequency of writing the history data is controlled by `config.dt_history`, as described in the following section on configuration parameters.

4.10 Simulation configuration and control parameters

A number of other parameters can be set in order to configure and control the behaviour of the simulation. These parameters are mainly collected into the `config` table[9] which is accessible to the user's input script. Grouped by theme, the possible attributes and their default values include[10]:

4.10.1 Geometry

- `dimensions=2`: The number of geometric dimensions (2 or 3).
- `axisymmetric=false`: If set `true`, the two-dimensional, axisymmetric geometry will have the x-axis as the axis of symmetry. Two-dimensional, planar geometry is the default. Geometries should be constructed above (and possibly including) the x-axis, so, with positive y ordinates.

4.10.2 Time stepping

The code tries to automatically adjust the size of the time-step, such that the numerical integration process remains stable for all cells. Sometimes this is difficult and you will have to alter one or more of the following parameters.

- `cfl_value=0.5`‡: The CFL number is the ratio of the time-step divided by the time for the fastest signal to cross a cell. The time-step adjustment process tries to set the time-step for the overall simulation to a value such that the maximum CFL number for any cell is at this `cfl_value`. If you are having trouble with a simulation that has lots of sudden flow field changes, decreasing the size of `cfl_value` may help.
- `gasdynamic_update_scheme=predictor-corrector`:
  The options are: `'euler'`,`'pc'`,`'predictor-corrector'`,`'midpoint'`, `'classic-rk3'`,`'tvd-rk3'`,`'denman-rk3'`,`'moving-grid-1-stage'`, `'moving-grid-2-stage'`.
  Note that `'pc'` is equivalent to `'predictor-corrector'`. If you want time-accurate solutions, use a two- or three-stage stepping scheme, otherwise, Euler

[9] The `config` table is a view into the `GlobalConfig` class defined in the D-language part of the simulation code. Although many of the attributes are discussed here, see the source code for that class for a full list of attributes. You will find it in the file `src/eilmer/globalconfig.d`.

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

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. Note that you may use dashes or underscores when spelling the scheme names.

- `fixed_time_step=false`‡: Normally, we allow the time step size to be determined from cell conditions and CFL number.
  Setting `fixed_time_step=true` forces the time step size to be unchanged from `dt_init`.

- `dt_init=1.0e-3`‡: Although the computation of the time step size is automatic, there might be cases where this process does not select a small enough value to get the simulation started stably. For the initial step, the user may override the computed value of time step by assigning a suitably small value to `dt_init`. This will then be the initial time step (in seconds) that will be used for the first few steps of the simulation process. Be careful to set a value small enough for the time-stepping to be stable. Since the time stepping is synchronous across all parts of the flow domain, this time step size should be smaller than half of the smallest time for a signal (pressure wave) to cross any cell in the flow domain. If you are sure that your geometric and boundary descriptions are correct 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=1.0e-3`‡: Maximum allowable time step (in seconds). 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.

- `viscous_signal_factor=1.0`‡: By default, the full viscous effect for the signal calculation will be used 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=false`‡: The default action for structured grids is to use different cell widths in each index direction. Setting `stringent_cfl=true` will use the smallest cross-cell distance in the time-step size check.

- `cfl_count=10`‡: The number of time steps between checks of the size of the time step. This check is expensive so we don't want to do it too frequently but, then, we have to be careful that the flow does not develop suddenly and the time step become unstable.

- `max_time=1.0e-3`‡: The simulation will be terminated on reaching this value of time.

- `max_step=100`‡: The simulation will be terminated on reaching this number of time steps. You will almost certainly want to use a larger value, however, a small value is a good way to test the starting process of a simulation, to see that all seems to be in order.

- `dt_plot=1.0e-3`‡: The whole flow solution will be written to disk when this amount of simulation time has elapsed, and then again, each occasion the same increment of simulation time has elapsed.

- `dt_history=1.0e-3`‡: The history-point data will be written to disk repeatedly, each time this increment of simulation time has elapsed. To obtain history data, you will also need to specify one or more history points.

4.10.3 Block marching

- `block_marching=false`: Normal time iteration proceeds on all blocks simultaneously, however, such a time-marching calculation may be very expensive computationally. Setting `block_marching=true` enables a sequencing of the time integration such that at any one instant, only two slices of blocks are being integrated. The i-direction is the marching direction and the assumed dominant (supersonic) flow direction. The blocks are assumed to be in a regular array with a fixed number of blocks in the j- and k-directions to the entire flow domain.

- `nib=1`, `njb=1`, `nkb=1`: are the number of blocks in each index direction. To make the best use of block marching, you should have `nib` set to a fairly large number. Since the array of blocks is assumed regular, you cannot have very complicated geometries. Simple ducts, nozzles and plates are the intended applications. As seen in the examples, it may be convenient to define the full domain with one or more calls to the constructor `FBArray:new`. There is a restriction that the overall flow domain be assembled as a single structured array of `FlowBlock` objects.

- `propagate_inflow_data=false`: By default, the integration begins in each set of blocks from the initial gas state set up in the preparation phase of the simulation. Some advantage may be gained following integration of the first block slices by initializing subsequent block slices with the downstream (east boundary) flow states. Setting `propagate_inflow_data=true` propagates these data across each new block slice, before the integration process for the slice begins.

- `save_intermediate_results=false`: Usually, a single set of solution files (after marching over all block slices) is all that is required. Sometimes, when debugging a troublesome calculation, it may be useful to have a solution written after the time-integration process for each pair of block slices. Set this parameter `true` to get these intermediate solutions written.

4.10.4 Spatial reconstruction/interpolation

- `interpolation_order=2`: Before applying the flux calculator, high-order reconstruction is applied. Setting `interpolation_order=1` results in no reconstruction of intra-cell flow properties.
- `apply_limiter=true`: By default, we apply a limiter to the flow-field reconstruction.
- `extrema_clipping=true`: By default, we do extrema clipping at end of each scalar-field reconstruction. Setting `extrema_clipping=false` suppresses clipping.
- `thermo_interpolator="rhou"`: String to choose the set of interpolation variables to use in the interpolation, options are `"rhou"`, `"rhop"`, `"rhoT"` and `"pT"`.

4.10.5 Flux calculator

- `flux_calculator="adaptive_hanel_ausmdv"`: Selects the flavour of the flux calculator. Options are:
  - `"efm"`: A cheap and very diffusive scheme by Pullin and Macrossan [15, 16]. 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 with low-diffusion for supersonic flows.[17].
  - `"adaptive_efm_ausmdv"` A blend [18] 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 subsonic and supersonic flow. The blend is controlled by the parameters `compression_tolerance` and `shear_tolerance` that are described below.
  - `"ausm_plus_up"`: Implemented from the description in Ref. [19]. 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 Harten-Lax-vanLeer-Einfeldt (HLLE) scheme. It is somewhat dissipative and is the only scheme usable with MHD terms.
  - `"adaptive_hlle_ausmdv"` As for `"adaptive_efm_ausmdv"` but with the dissipative scheme being the HLLE flux calculator.
  - `"hanel"` The Hanel-Schwane-Seider scheme, from their 1987 paper. It is also dissipative and is somewhat better behaved than our EFM implementation.

  - "adaptive_hanel_ausmdv" As for "adaptive_efm_ausmdv" but with the dissipative scheme being the Hanel-Schwane-Seider flux calculator.
  - "roe" The Phil Roe's classic linearized flux calculator.
  - "adaptive_hlle_roe" A blend of Roe's low-dissipation scheme and the more dissipative HLLE flux calculator.

  The default adaptive scheme is a good all-round scheme that uses AUSMDV away from shocks and Hanel-Schwane-Seider flux calculator near shocks.

- compression_tolerance=-0.30: The 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 an adaptive flux calculator is used and the shock detector is triggered, the more-dissipative flux calculation will be used in place of the default low-dissipation calculation. A value of -0.05 seems OK for the Sod shock tube and sharp-cone inviscid flow simulations, however, a higher value is needed for cases with viscous boundary layers, where it is important to not have too much numerical dissipation in the boundary layer region.

- shear_tolerance=0.20: The value of the relative tangential-velocity change (normalised by local sound speed) across a cell-interface that suppresses the use of the high-dissipation flux calculator even if the shock detector indicates that a high dissipation scheme should be used within 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=0.01: representative Mach number for the free stream. Used by the ausm_plus_up flux calculator.

4.10.6 Viscous effects

- viscous=false: If set true, viscous effects will be included in the simulation.

- separate_update_for_viscous_terms=false: If set true, the update for the viscous transport terms are done separately to the convective terms. By default the updates are done together in the gas-dynamic update procedure.

- viscous_delay=0.0: 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=0.01: The per-time-step increment of the viscous effects, once t > viscous_delay.

- mass_diffusion_model="none": Controls the molecular diffusion of individual species in a multi-species, viscous, laminar calculation. The only available model is "ficks_first_law", which specifies the form of the diffusion fluxes, but leaves open different possibilities for the diffusion coefficient. Note that in a turbulent simulation, this parameter is ignored, and species diffuse

based on the `turbulence_schmidt_number`. This is because turbulent diffusion is typically much larger than laminar diffusion, and so some calculation time can be saved by ignoring the latter.

- `diffusion_coefficient_type="none"`: Controls the formulas used to compute the diffusion coefficient in a `mass_diffusion_model="ficks_first_law"` simulation. The simplest option is `constant_lewis_number`, which will work with any multi-species gas. The value of the Lewis number can be set using `lewis_number`. A somewhat more flexible option is `species_specific_lewis_numb` which requires the Lewis number for each species to be specified in the gas model. The Thermally Perfect Gas model is an example of a gas model that has this information. The highest fidelity diffusion coefficient model is `binary_diffusion`, which asks the gas model to compute a diffusion coefficient for each species against each other species and average them together, typically using collision integrals. Again, the Thermally Perfect Gas model is an example of a gas model with this capability, though the Two Temperature Air model is also equipped for this calculation.

- `lewis_number=1.0`: Sets the ratio between heat transport and diffusive mass transport in a simulation using `constant_lewis_number`.

- `turbulence_model="none"`: String specifying which model to use. Options are: `"none"`,`"k_omega"`,`"spalart_allmaras"`,`"spalart_allmaras_edwards"`.

- `turbulence_prandtl_number=0.89`

- `turbulence_schmidt_number=0.75`

- `max_mu_t_factor=300`: The turbulent viscosity is limited to laminar viscosity multiplied by this factor.

- `transient_mu_t_factor=1.0`

4.10.7 Thermo-chemistry

- `reacting=false`: Set to `true` to activate the finite-rate chemical reactions.

- `reactions_file="chemistry.lua"`: File name for reaction scheme configuration.

- `reaction_time_delay=0.0`: Time after which finite-rate reactions are allowed to start.

- `T_frozen=300.0`: 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.

4.10.8 Miscellaneous parameters

- `title="Eilmer simulation"`: The title string that may appear in a number of places. For example, in plots made during the postprocessing stage.
- `adjust_invalid_cell_data=false`: Usually, you will want the flow solver to provide its best estimate for your flow, however, there are flow situations for which the flow solver will not compute physically valid flow data. If you encounter a difficult flow situation and are prepared to fudge over a few cells, then set this parameter to `true` and `max_invalid_cells` to a non-zero value.
- `max_invalid_cells=0`: The maximum number of bad cells that will be tolerated on decoding conserved quantities. If this number is exceeded, the simulation will stop.
- `report_invalid_cells=true`: If you are stuck with having to fudge over cells, you probably will want to know about them until, of course, that you don't. Set this parameter to `false` to silence the reports of bad cells being fudged over.
- `apply_bcs_in_parallel=true`: This will be the fastest calculation, however, some boundary conditions, such as the shock-fitting need to cooperate across blocks and so will have race conditions if applied in parallel. If your simulation has such a boundary condition, set this parameter to `false` to favour safety above speed.
- `udf_source_terms=false`: Set to `true` to apply user-defined source terms, as supplied in a Lua file.
- `udf_source_terms_file="dummy-source-terms.txt"`: Name of the Lua file for the user-defined source terms.
- `print_count=20`‡: Number of time steps between printing status information to the console.
- `control_count=10`: Number of time steps between re-parsing the job`.control` file. If the job`.control` has been edited, then the new values are used after reparsing.
- `MHD=false`: Set to `true` to make MHD physics active.

4.11 Notes on the layout of your input script

The goal of your input script is to define one or more `FluidBlock` objects that have suitable boundary conditions and initial gas state. The order in which you construct the objects and set configuration variables is somewhat defined by the input needed by the constructor for each class of object. To define a `FluidBlock`, you need a `Grid`, either imported or constructed from geometric elements, and a `GasState`. To define a `GasState`, you need to have set the master `GasModel`. Thus a typical input script will set the `GasModel` and then proceed to define one or more `GasState` objects. Geometry construction and the creation or importing of Grid objects can be done

independently of the `FlowState` construction, so it does not matter if this phase is done before or after the `FlowState` specification. The `FluidBlock` construction comes after.

Most of the `config` variable settings are just for use at run time of the simulation. If you want a value other than the default value, you can set them at any point in the input script. There are, however, a few `config` variables that set context that is important within your input script. These include `dimensions`, `viscous`, `grid_motion` and `turbulence_model`. If you want a value different from the default value, be sure to set it early in your script.

There are many configuration variables and their definitive reference is the flow solver's source code. Do not be afraid to open up the file `globalconfig.d` and browse the definitions of those variables within the `GlobalConfig` class. There are documentation comments embedded in the source code that do not appear in this guide. As a general rule, only set a value in your input script if you want something other than the default value or if you want to explicitly document the setting via your script. Assigning many `config` variables to their default values in your input script will just add clutter.

4.12 MPI simulations

Once you are successful with your first couple of simulations, your computational ambition is likely to grow and the use of the MPI flavour of the code, `e4mpi`, is of interest. The only extra configuration that the MPI code requires is the distribution of the FluidBlocks to MPI tasks. If you do not care for the detail, the default arrangement will be to assign each FluidBlock to its own MPI task. If you want something different for your time dependent simulation, maybe to fit more comfortably within your workstation's capabilities, you may explicitly arrange the distribution by calling:

```
mpiDistributeBlocks{ntasks=3, dist="load-balance",
                    preassign={[0]=1}}
```

Here, we have specified that we want the FluidBlocks to be distributed across 3 MPI tasks, with FlowBlock[0] being assigned to MPI task 1. The options for the distribution algorithm of the remaining FlowBlocks are `"round-robin"` and `"load-balance"`, with the default being `"load-balance"`. This call to `mpiDistributeBlocks` should be made *after* all FluidBlocks have been defined in your input script.

For block-marching calculations, the distribution of blocks should be such that blocks in a ribbon along the marching direction (the i-index direction) should be assigned to the same MPI task. If you have constructed your blocks with `FBArray:new`, there is a call:

```
mpiDistributeFBArray{fba=my_fba, ntasks=njb*nkb}
```

that will assign the blocks of `my_fba` correctly. If you do not call the function directly, the preparation program will arrange the assignments for you, with the default number of tasks shown above. Of course, you will need to be aware of that default so that you can specify the correct number of tasks when you subsequently start up the simulation with `mpirun`. You may specify fewer MPI tasks but the number should be

such that a j,k-slice of blocks should distribute neatly across that number of tasks. In the example simulation of a two-dimensional channel with a bump, the flow domain is constructed of 192 blocks in a $48 \times 4 \times 1$ array. For that case `ntasks=2` or `4` would be suitable.

5

More example simulations

With some confidence that the code is working correctly and a knowledge of the manual postprocessing arrangements shown in the tutorial example (Chapter 3), you are ready to try to simulate flows that are a bit more "realistic". The following sections look at two examples, that are more demanding.

The first is a flat-plate, in a supersonic flow, with a laminar boundary layer. The interesting behaviour occurs where an oblique shock interacts with the boundary layer. This is a two-dimensional flow simulation with a simple ideal-gas thermochemical model and simple flow domain.

The second example is the bluff-body flow of a high-enthalpy gas over a cylinder. The thermochemical model of reacting nitrogen is more sophisticated and the description of the 3D domain requires more Lua code to set up.

5.1 Oblique shock boundary layer interaction.

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 5.1. This flow image was taken as part of an experimental campaign [20] in a continuous flow wind tunnel at MIT. The flow is from left to right in the image. The plate with the boundary layer of interest is 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 [20], shear-stress data indicated that the boundary layer remained laminar after the interaction.

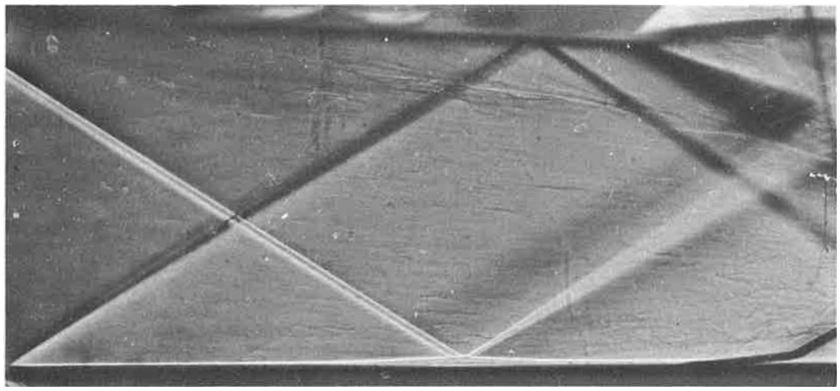


Figure 5.1: Schlieren image of the Mach 2 flow over a flat plate taken from Fig.6b in Reference [20]. Flow is from left to right, with the leading edge of the flat plate close to the bottom left corner of the picture. The leading edge of the shock generator is out of view, near the top left corner of the picture. The interesting region for the shock-wave boundary-layer interaction is roughly midway along the plate at the bottom of the picture.

Although the behaviour of laminar compressible-flow boundary layers on flat plates is well predicted via simple theories, the addition of an impinging shock makes the analysis significantly more difficult. The flow complexity increases while the defining flow geometry remains very simple. Metaphorically speaking, this is a good nut to crack with our CFD hammer.

5.1.1 Organising the simulation

To get the simulation started, prepare an input file for building a simple gas model for air. As we had done in the tutorial example, we might call this file `ideal-air.inp` and it should have the two lines:

```
1 model = "IdealGas"
2 species = {'air'}
```

We now use the input file to prepare the actual gas-model definition file with the command:

```
$ prep-gas ideal-air.inp ideal-air-gas-model.lua
```

where the `$` character represents the command prompt for your system. The result, if successful, will be the generation of the Lua file `ideal-air-gas-model.lua`. Of course, in your own work, you should choose file names that are descriptive of the problem at hand. Reusing these names is fine, if they are appropriate and not misleading. With your gas model organised, you now have the larger task of defining your flow and flow domain.

Figure 5.2 shows the region, as modelled for simulation. The instrumented flat plate (located along the lower boundary and labelled ADIABATIC) starts at $x = 0$ and 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 (along the upper, inclined boundary) 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. This allows us to estimate the deflection angle for the specified pressure rise across the reflected shock using just the oblique-shock relations for an ideal gas.

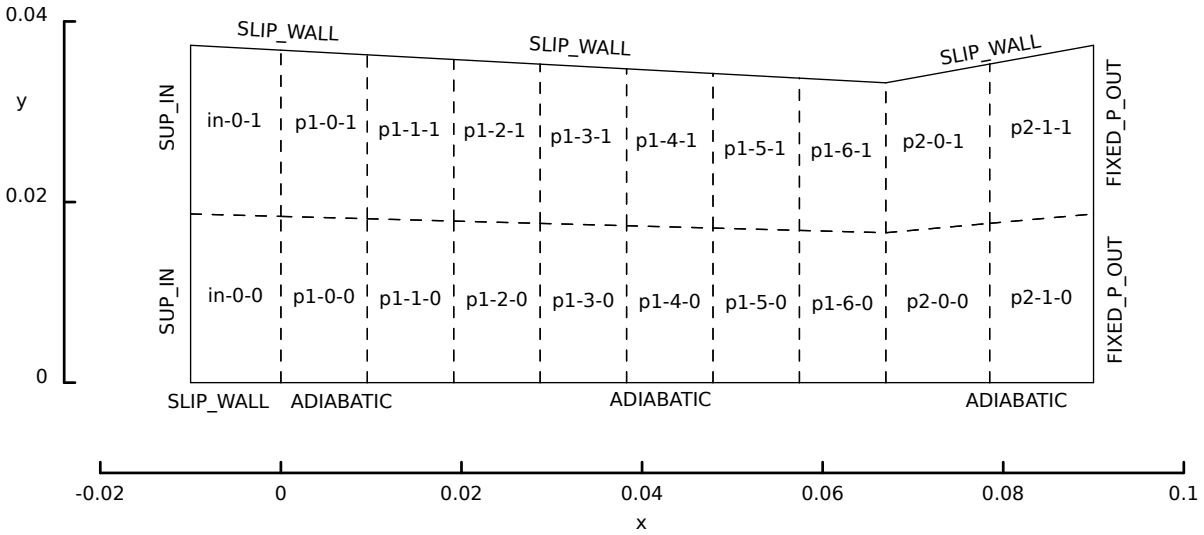


Figure 5.2: Schematic view of the simulated flow region for the shock-wave interaction with a laminar boundary layer.

Looking at the photograph (Figure 5.1), and extrapolating the surface of the shock-generator plate back to where it would intersect the shock, gives us a location for

the leading edge of the shock generator plate that is upstream of the leading edge of the instrumented flat plate. Since it is convenient to work with a box-like flow domain, we include the region from the tip of the shock-generator plate to the tip of the instrumented flat plate in the simulated flow domain. We use a SLIP_WALL boundary condition along the lower boundary of that part of the flow domain, that is, for $x < 0$.

Using the ideal-gas flow functions built into the simulation program, the following script computes the combined pressure rise across the incident and reflected oblique shocks as 1.4. With a minute or two of trial and error fiddling, the shock generator deflection angle was estimated as being 3.09^o.

```
-- double-oblique-shock.lua
-- Estimate pressure rise across a reflected oblique shock.
-- PJ, 01-May-2013, 2016-11-01 for Lua version
-- $ e4shared --custom-post --script-file=double-oblique-shock.lua
--
print("Begin...")
M1 = 2.0
p1 = 1.0
g = 1.4
print("First shock:")
delta1 = 3.09 * math.pi/180.0
beta1 = idealgasflow.beta_obl(M1,delta1,g)
p2 = idealgasflow.p2_p1_obl(M1,beta1,g)
M2 = idealgasflow.M2_obl(M1,beta1,delta1,g)
print("   beta1=", beta1, "p2=", p2, "M2=", M2)
--
print("Reflected shock:")
delta2 = delta1
beta2 = idealgasflow.beta_obl(M2,delta2,g)
p3 = p2 * idealgasflow.p2_p1_obl(M2,beta2,g)
M3 = idealgasflow.M2_obl(M2,beta2,delta2,g)
print("   beta2=", beta2, "p3=", p3, "M3=", M3)
print("Done.")
```

Note the command for running the script actually starts the simulation program in the custom post-processing mode and specifies what Lua script to run.

```
$ e4shared --custom-post --script-file=double-oblique-shock.lua
```

The results, as written to the console, are:

```
First shock:
   beta1= 0.56869987213562      p2= 1.1867698723259     M2= 1.8891863108079
Reflected shock:
   beta2= 0.60486005952261      p3= 1.4001003974295     M3= 1.7811889520851
```

5.1.2 Input script (.lua)

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 three `FBArray:new` calls to split the region into 20 individual blocks as shown in Figure 5.2. 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 [20], 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.

```
-- swbli.lua
-- Anand V, 10-October-2015 and Peter J, 2016-11-02
-- Model of Hakkinen et al's 1959 experiment.

config.title = "Shock Wave Boundary Layer Interaction"
print(config.title)
config.dimensions = 2

-- Flow 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

nsp, nmodes = setGasModel('ideal-air-gas-model.lua')
print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
inflow = FlowState:new{p=p_inf, velx=u_inf, T=T_inf}

-- Flow domain.
--
--   y
--   ^   a1---b1---c1---d1   Shock generator
--   |   |    |    |    |
--   |   | 0  |  1 | 2  |   patches
--   |   |    |    |    |
--   0   a0---b0---c0---d0   Flat plate with boundary layer
--
--             0---> x
mm = 1.0e-3 -- metres per mm
-- Leading edge of shock generator and inlet to the flow domain.
L1 = 10.0*mm; H1 = 37.36*mm
a0 = Vector3:new{x=-L1, y=0.0}
a1 = a0+Vector3:new{x=0.0,y=H1}
-- Angle of inviscid shock generator.
alpha = 3.09*math.pi/180.0
tan_alpha = math.tan(alpha)
-- Start of flat plate with boundary layer.
b0 = Vector3:new{x=0.0, y=0.0}
b1 = b0+Vector3:new{x=0.0,y=H1-L1*tan_alpha}
-- End of shock generator is only part way long the plate.
L3 = 67*mm
c0 = Vector3:new{x=L3, y=0.0}
c1 = c0+Vector3:new{x=0.0,y=H1-(L1+L3)*tan_alpha}
```

```
-- End of plate, and of the whole flow domain.
L2 = 90.0*mm
d0 = Vector3:new{x=L2, y=0.0}
d1 = d0+Vector3:new{x=0.0,y=H1}
-- Now, define the three patches.
patch0 = CoonsPatch:new{p00=a0, p10=b0, p11=b1, p01=a1}
patch1 = CoonsPatch:new{p00=b0, p10=c0, p11=c1, p01=b1}
patch2 = CoonsPatch:new{p00=c0, p10=d0, p11=d1, p01=c1}
--
-- Discretization of the flow domain.
--
-- We want to cluster the cells toward the surface of the flat plate.
-- where the boundary layer will be developing.
rcf = RobertsFunction:new{end0=true,end1=true,beta=1.1}
factor = 4 -- We'll scale discretization off this value
ni0 = math.floor(20*factor); nj0 = math.floor(80*factor)
grid0 = StructuredGrid:new{psurface=patch0, niv=ni0+1, njv=nj0+1,
                           cfList={east=rcf,west=rcf}}
grid1 = StructuredGrid:new{psurface=patch1, niv=7*ni0+1, njv=nj0+1,
                           cfList={east=rcf,west=rcf}}
grid2 = StructuredGrid:new{psurface=patch2, niv=2*ni0+1, njv=nj0+1,
                           cfList={east=rcf,west=rcf}}
--
-- Build the flow blocks and attach boundary conditions.
--
blk0 = FBArray:new{grid=grid0, initialState=inflow, nib=1, njb=2,
                   bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                           north=WallBC_WithSlip:new{},
                           south=WallBC_WithSlip:new{}}}
blk1 = FBArray:new{grid=grid1, initialState=inflow, nib=7, njb=2,
                   bcList={south=WallBC_NoSlip_Adiabatic:new{},
                           north=WallBC_WithSlip:new{}}}
blk2 = FBArray:new{grid=grid2, initialState=inflow, nib=2, njb=2,
                   bcList={south=WallBC_NoSlip_Adiabatic:new{},
                           north=WallBC_WithSlip:new{},
                           east=OutFlowBC_FixedPT:new{p_outside=p_inf,
                                                      T_outside=T_inf}}}
identifyBlockConnections()

config.gasdynamic_update_scheme = "classic-rk3"
config.flux_calculator = 'adaptive'
config.viscous = true
config.spatial_deriv_calc = 'divergence'
config.cfl_value = 1.0
config.max_time = 5.0*L2/u_inf -- time in flow lengths
config.max_step = 200000
config.dt_init = 1.0e-8
config.dt_plot = config.max_time/10
```

5.1.3 Running the simulation

To get the simulation started, prepare the grids and initial flow state using the command:

```
$ e4shared --prep --job=swbli
```

This may take a little while because there are many more cells in this simulation than we had used for the tutorial example. We are trying to capture the development of a boundary layer and, to do that accurately, we have to pay the computational cost of using high-resolution grids. After a minute or so, depending on the speed of your computer, you should have the grid files in the subdirectory `grid/t0000` and the initial flow files in subdirectory `flow/t0000`.

We can now start the computation of the evolution of the flow field with the command:

```
$ e4shared --run --job=swbli --verbosity=1 --max-cpus=4
```

You should soon see the usual console output of a simulation proceeding to take time steps and reporting its progress toward reaching a final time. Be patient because this simulation is much more demanding than the initial tutorial exercise. Even if you are working on a big multi-core machine, go and have dinner and return in about 5-7 hours to check the state of the simulation.

But, just before you go, start up the `htop` monitor program[1] on your computer. It gives a good overview of the processes that are running and what processor utilization you are getting. On a little HP laptop computer with 4 AMD cores, htop shows greater then 99% utilization of all 4 processors, once the initialization phase is finished and the time-stepping phase has begun. This simulation is nicely load balanced and makes good use of a multi-core computer. As well as the processor resources used, htop also shows that this simulation occupies a little under 3GBytes of memory as it runs. You probably want to have at least 8GB of RAM in your computer to run interesting simulations.

Coming back to your computer, you see that the time-evolution of the flow field has been computed for about 876 μs (with 11176 time steps being required). You can then generate the flow solution data for display in Paraview with the command:

```
$ e4shared --post --job=swbli --tindx-plot=all --vtk-xml \
           --add-vars="mach,pitot,total-p,total-h"
```

This could all be typed onto one line, omitting the backslash continuation character.

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 `swbli.control` file as described in Section 4.1.4 and setting `max_time = 1.751e-03` and `dt_init = 8.0e-08` then running the command:

```
$ e4shared --run --job=swbli --tindx-start=last --max-cpus=4
```

Given the hours that have passed since preparing the gas-model file for this example, it's probably time for a sleep. Running calculations overnight, or even over several days, is a fairly common activity for someone doing CFD analyses.

[1] You may need to install htop manually, using your operating system's package manager.

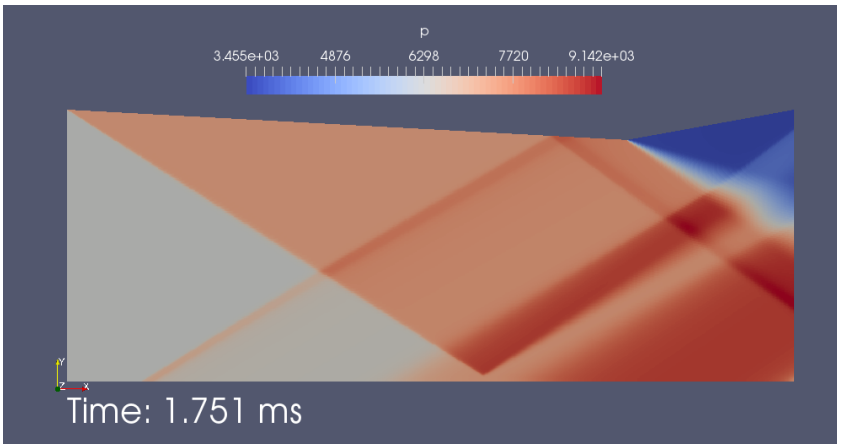
5.1.4 Simulation results

Figure 5.3 shows some of the flow field data at t=1.751 ms after flow start. The magnitude of the gradients of density (Fig. 5.1.4) are also shown as an approximation to the Schlieren image of Figure 5.1. 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 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 recompression 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.

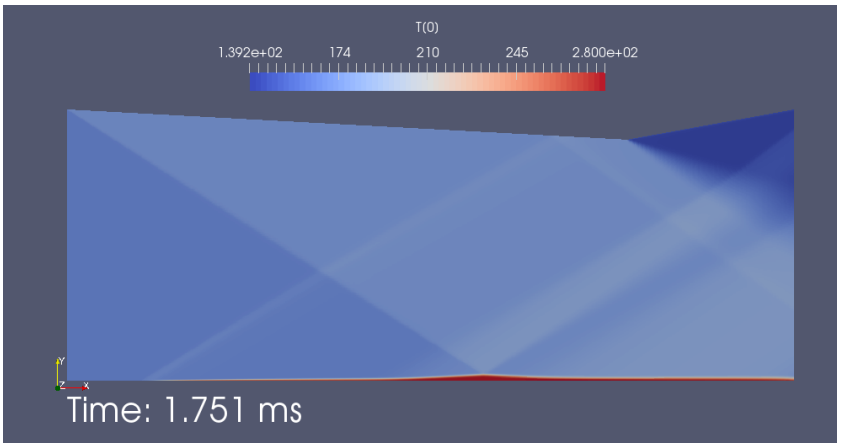
The real proof of success is in comparison with the experimental data. Figure 5.4 shows the pressure and shear-stress along the plate. The simulation has done a reasonable job of estimating the pressure distribution right through the separation zone. Features that look a little wrong include the viscous 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 5.1.5. 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.

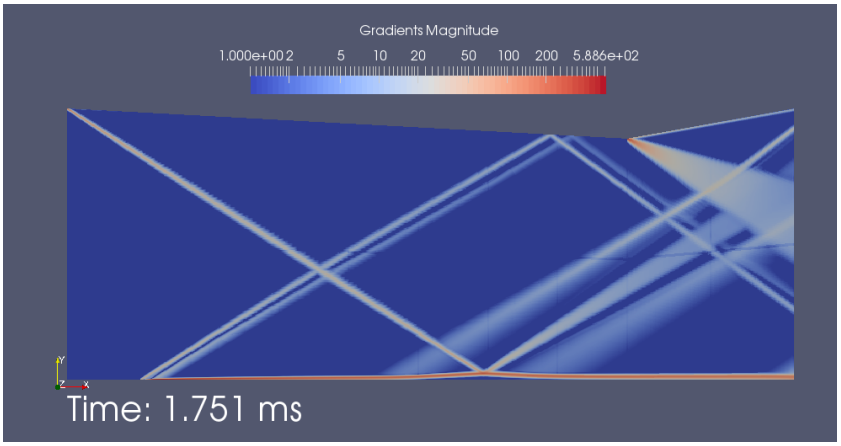
Note that the influence of the flat plate boundary layer on the pressure in the region near the plate is small but measureable. With a free-stream pressure of 6.205 kPa specified at the inflow plane, we see 6.28 kPa in the pressure data leading into the shock-interaction region. For the free-stream conditions 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.



(a) Pressure field.

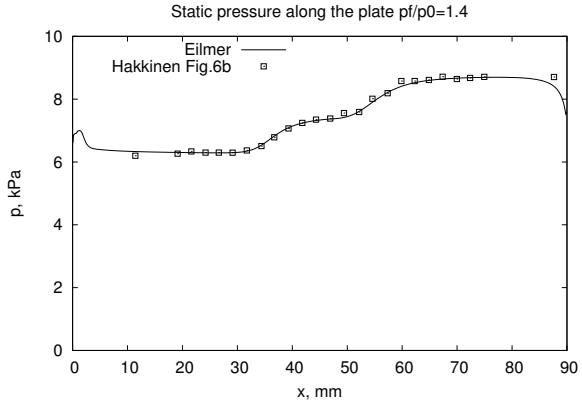


(b) Temperature field.

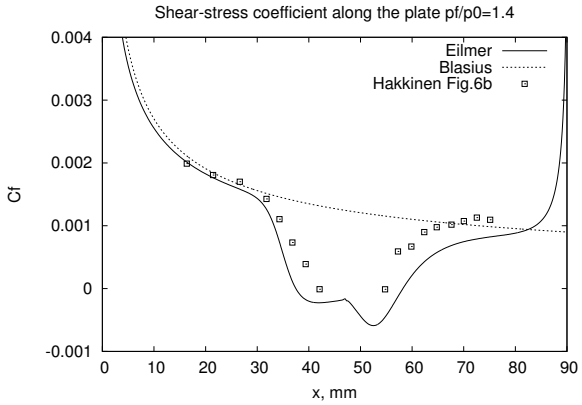


(c) Gradient of density field.

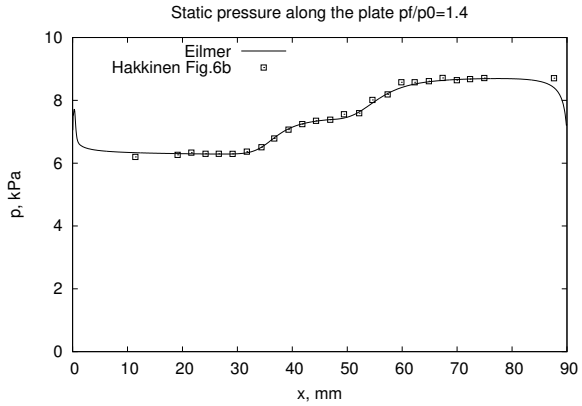
Figure 5.3: Computed flow field at t=1.751 ms.



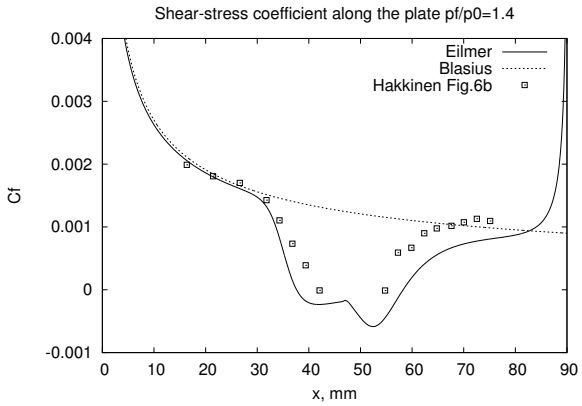
(a) Pressure (factor=4).



(b) Shear stress (factor=4).



(c) Pressure (factor=8).



(d) Shear stress (factor=8).

Figure 5.4: Distribution of pressure and shear along the plate at t=1.751 ms.

5.1.5 Postprocessing for shear stress

To get estimates of the pressure and shear stress along the plate, we can extract the flow data for the cells against the south boundaries for all blocks along the plate. This can be done with the following command:[2]

```
$ e4shared --post --job=swbli --tindx-plot=last --add-vars="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"
```

where the string given to the `slice-list` option picks out the row of j=0 and k=0 (*i.e.* south boundary) cells for every block that sits against the plate (*i.e.* blocks 2, 4, 6, 8, 10, 12, 14, 16, and 18).

The x-position and static pressure for each cell will be found in columns 1 and 9 of the output file `bl.data`. Thus, the pressure along the surface can be directly plotted. The shear stress, however, needs to be computed from the cell data that are available in the file `bl.data`. The AWK script below does that job.

```
# compute-shear.awk
# Invoke with the command line:
# $ awk -f compute-shear.awk bl.data > shear.data
#
# PJ, 2016-11-01
#
BEGIN {
    rho_inf = 0.1315 # kg/m**3
    velx_inf =  514.0 # m/s
    T_inf = 164.4 # K
    # Sutherland expression for viscosity
    mu_ref = 1.716e-5; T_ref = 273.0; S_mu = 111.0
    mu_inf = (T_inf/T_ref)*sqrt(T_inf/T_ref)*(T_ref+S_mu)/(T_inf+S_mu)*mu_ref
    print("# x(m)  tau_w(Pa)  Cf   y_plus")
}

$1 != "#" {
    x = $1; y = $2; rho = $5; velx = $6; mu = $11; k = $12
    dvelxdy = (velx - 0.0) / y # Assuming that the wall is straight down at y=0
    tau_w = mu * dvelxdy    # wall shear stress
    Cf = tau_w / (0.5*rho_inf*velx_inf*velx_inf)
    if (tau_w > 0.0) abs_tau_w = tau_w; else abs_tau_w = -tau_w;
    vel_tau = sqrt(abs_tau_w / rho) # friction velocity
    y_plus = vel_tau * y * rho / mu
    Rex = rho_inf * velx_inf * x / mu_inf
    Cf_blasius = 0.664 / sqrt(Rex)
    print(x, tau_w, Cf, Cf_blasius, y_plus)
}
```

This particular script was used to filter the `bl.data` file, picking up data from every line that did not start with a sharp "#" character, computing an estimate of the shear stress, the friction coefficient, the theoretical laminar (Blasius) shear stress and y^+, and printing these results to standard output. The variable names of the form `$1`,

[2] Even using a small font, the string specifying the slice-list runs over the end of the line in this report. You should not have any problem putting it all onto one line in the usual command windows on Linux.

for example, pick out the data from a particular column on each line captured by the filter. The following command was used to apply our AWK filter program to the `bl.data` file and redirect the output to the file `shear.data` for subsequent plotting.

```
$ awk -f compute-shear.awk < bl.data > shear.data
```

5.2 Flow of nitrogen over a cylinder of finite length

This example is relevant to Troy Eichmann's X2 experiments [21] 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.

The flow domain shown is made up of 4 block-structured grids as shown in Figure 5.5 and number of the surface grids are indicated in Figure 5.6 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.

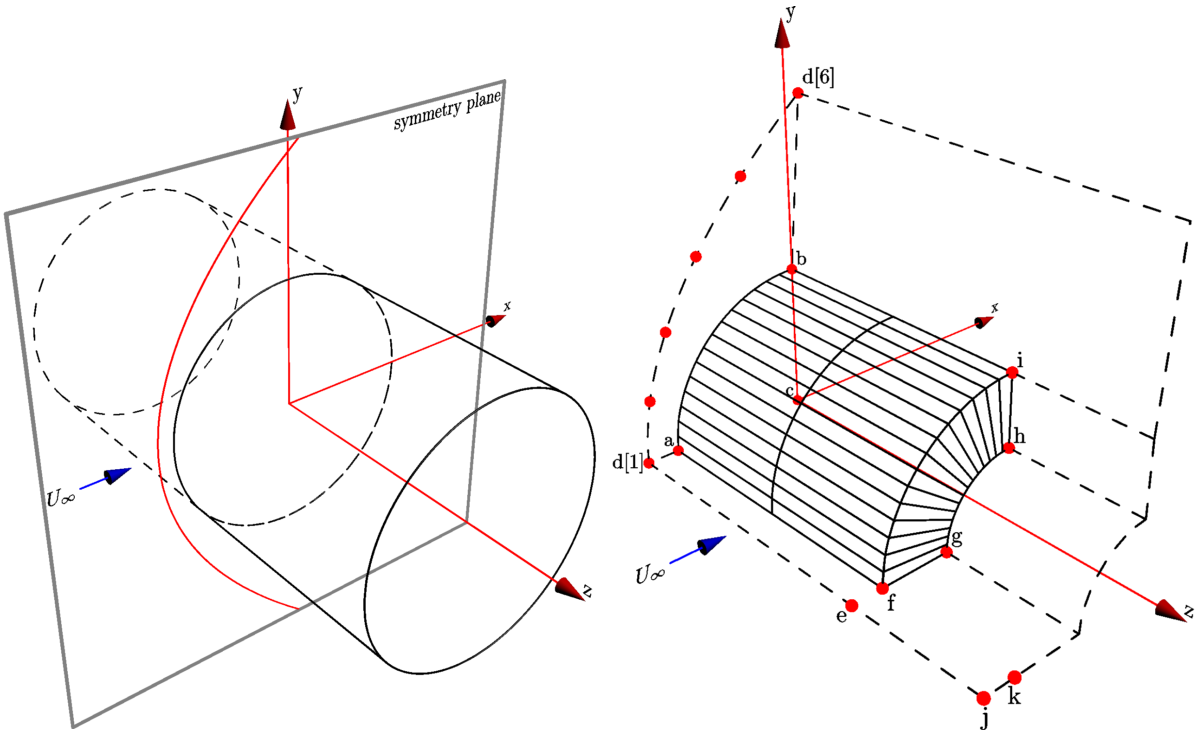


Figure 5.5: Left: full cylinder with the expected shock location scribed on the symmetry plane. Right: layout of finite-cylinder simulation with one-quarter of forward-facing half of the cylinder surface shown as wire-frame. Some of the edges of the flow domain are shown dashed and the labelled nodes 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 the 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. Here we describe a finite-cylinder simulation with single-temperature chemical nonequilibrium. This means chemical reactions are permitted to occur at a finite rate (chemical nonequilibrium), but all internal energy modes of the gas 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 pressure and temperature fields (Figure 5.7). 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.

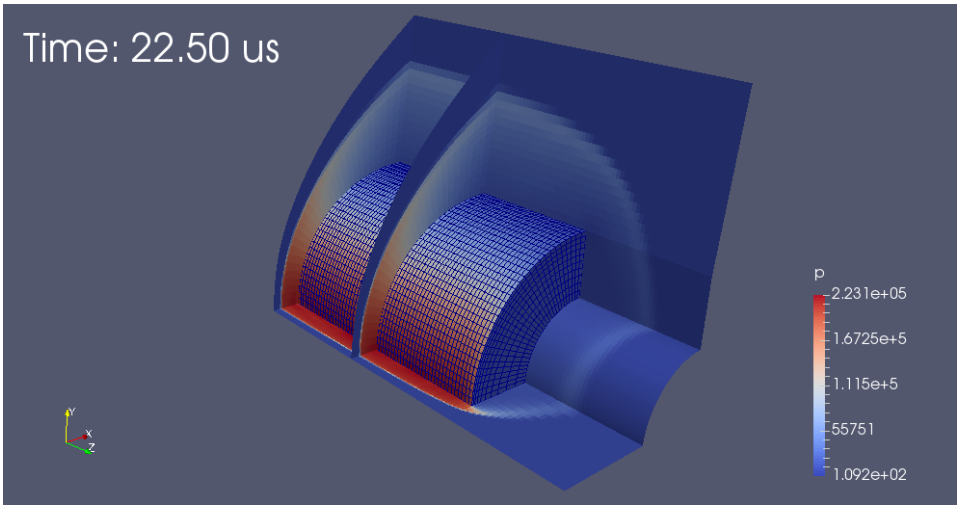


Figure 5.6: A selection of surface grids from the finite-cylinder simulation with chemical nonequilibrium, shown as wire-frame on the cylinder surface and coloured by pressure in the flow field. This PNG figure was generated with Paraview using block surfaces extracted from final solution file.

This case is quite difficult for both the flow solver and defects can be seen in the solution around the flat end of the cylinder, where there is a very strong expansion combined with a strong shear. These defects are visible in the temperature field with a checkered pattern of low temperatures. Sometimes you will find that, without replacing bad cell data in a region like this, your simulation will not run. Our get-out-of-jail options include `adjust_invalid_cell_data` in combination with `max_invalid_cells` on lines 148 and 149 of the input script.[3]

Despite the flow calculation problems over the edge of the cylinder, the forebody flow looks to be reliably computed and the shock stand-off distance is 1.2 mm near the midplane of the cylinder.

This simulation can make good use of multiple processors. The elapsed time for the run with 4 CPUs is a little over an hour on a HP Pavillion laptop with 4 AMD cores. To double the grid resolution (as one might want to do for a convergence study), would require a factor of 8 increase in memory and 16 in CPU cycles. If you are planning to do calculations of any reasonable complexity, it is worth your while to invest in a computer with a good number of CPU cores.

[3] As of 2021, these options are not required for this particular example. However, we find that the expansion-tube laboratory people continue to push flow speeds and flow conditions that break the Eilmer code and will occasionally need these options.

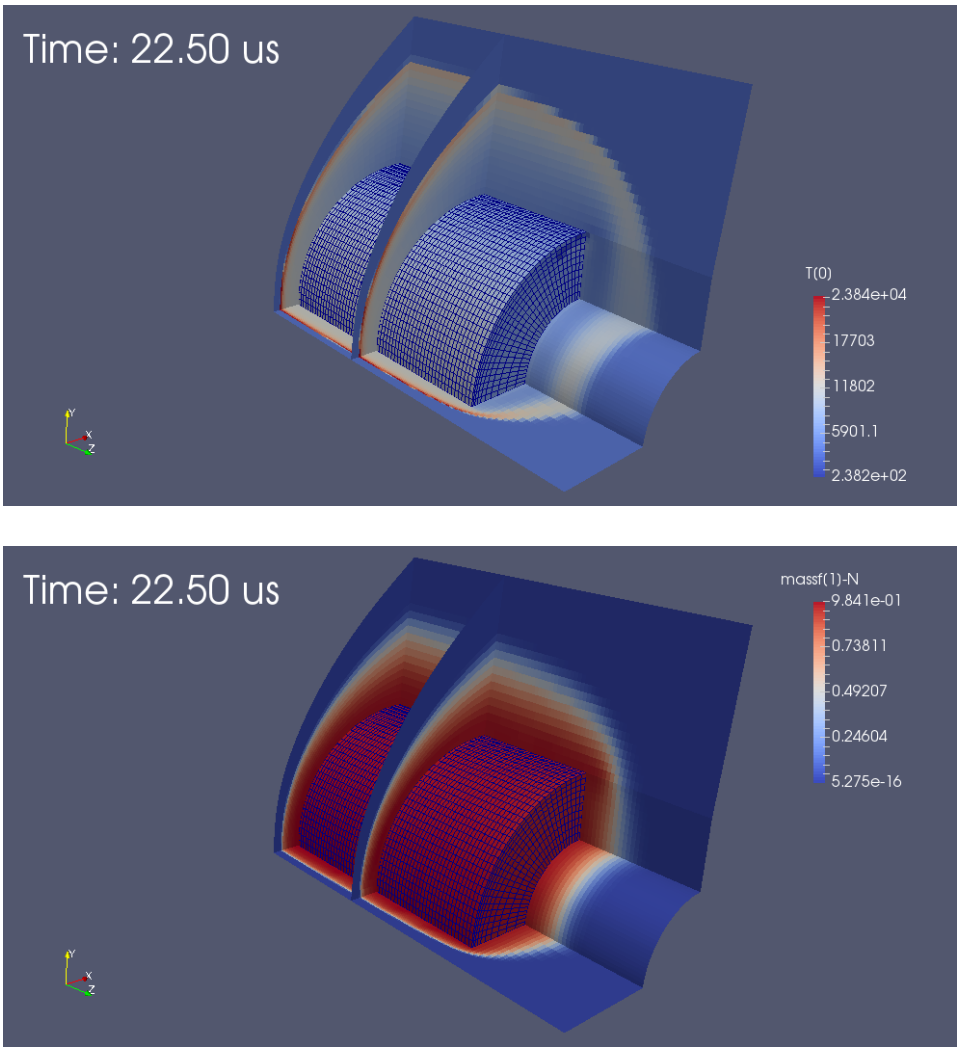


Figure 5.7: Static temperature and mass fraction of nitrogen atoms in the flow field from the chemical nonequilibrium simulation.

Input script (.lua)

```
-- cyl.lua
-- Troy and Tim's finite-length cylinder in dissociating nitrogen flow.
--
-- PJ & RJG 2016-10-16 built from the eilmer3/3D/finite-cylinder
-- and eilmer4/2D/cylinder-dlr-n90

config.dimensions = 3
D = 15.0e-3  -- diameter of cylinder, metres
L = 2.0 * D  -- (axial) length of full cylinder, will be halved later

-- Free-stream properties
T_inf = 3000.0  -- degrees K
p_inf = 2000.0  -- Pa
V_inf = 10.0e3  -- m/s
config.title = string.format("Cylinder L/D=%g in N2 at u=%g m/s.", L/D, V_inf)
print(config.title)

nsp, nmodes, gm = setGasModel('nitrogen-2sp.lua')
print("GasModel set nsp= ", nsp, " nmodes= ", nmodes)

-- Compute inflow Mach number.
Q = GasState:new{gm}
Q.p = p_inf; Q.T = T_inf; Q.massf = {N2=1.0}
gm:updateThermoFromPT(Q); gm:updateSoundSpeed(Q)
print("T=", Q.T, "density=", Q.rho, "sound speed= ", Q.a)
M_inf = V_inf / Q.a
print("M_inf=", M_inf)

inflow = FlowState:new{p=p_inf, T=T_inf, velx=V_inf, massf={N2=1.0}}
initial = FlowState:new{p=p_inf/3, T=300.0, massf={N2=1.0}}

config.reacting = true
config.reactions_file = 'e4-chem.lua'

-- Build geometry from body and flow parameters
Rc = D/2.0   -- radius of cylinder

a = Vector3:new{x=-Rc}; b = Vector3:new{y=Rc}; c = Vector3:new{x=0.0, y=0.0}

-- 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.
dofile("billig.lua")
M_inf = 8.9566
print("Points on Billig's correlation.")
xys = {}
for i,y in ipairs({0.0, 0.5, 1.0, 1.5, 2.0, 2.5}) do
   x = x_from_y(y*Rc, M_inf, 0.0, false, Rc)
   xys[#xys+1] = {x=x, y=y*Rc}  -- a new coordinate pair
   print("x=", x, "y=", y*Rc)
end

-- Scale the Billig distances, depending on the expected behaviour
-- relative to the gamma=1.4 ideal gas.
```

```
55 local b_scale = 1.1 -- for ideal (frozen-chemistry) gas
56 if config.reacting then
57     b_scale = 0.87  -- for finite-rate chemistry
58 end
59 d = {} -- will use a list to keep the nodes for the shock boundary
60 for i, xy in ipairs(xys) do
61    -- the outer boundary should be a little further than the shock itself
62    d[#d+1] = Vector3:new{x=-b_scale*xy.x, y=b_scale*xy.y, z=0.0}
63 end
64 print("front of grid: d[1]=", d[1])
65
66 -- Extent of the cylinder in the z-direction to end face.
67 zshift = Vector3:new{z=L/2.0}
68 c2 = c + zshift
69 e = d[1] + zshift
70 f = a + zshift
71 g = Vector3:new{x=-Rc/2.0, y=0.0, z=L/2.0}
72 h = Vector3:new{x=0.0, y=Rc/2.0, z=L/2.0}
73 i = Vector3:new{x=0.0, y=Rc, z=L/2.0}
74 -- the domain is extended beyond the end of the cylinder
75 zshift2 = Vector3:new{z=Rc}
76 j = e + zshift2
77 k = f + zshift2
78
79 -- ...then lines, arcs, etc, that will make up the domain-end face.
80 xaxis = Line:new{p0=d[1], p1=a} -- first-point of shock to nose of cylinder
81 cylinder = Arc:new{p0=a, p1=b, centre=c}
82 shock = ArcLengthParameterizedPath:new{underlying_path=Spline:new{points=d}}
83 outlet = Line:new{p0=d[#d], p1=b}  -- top-point of shock to top of cylinder
84 domain_end_face = CoonsPatch:new{south=xaxis, north=outlet,
85                                  west=shock, east=cylinder}
86
87 -- ...lines along which we shall extrude the domain-end face
88 yaxis0 = Line:new{p0=d[1], p1=e}
89 yaxis1 = Line:new{p0=e, p1=j}
90
91 -- End-face of cylinder
92 xaxis = Line:new{p0=f, p1=g}
93 cylinder = Arc:new{p0=f, p1=i, centre=c2}
94 inner = Arc:new{p0=g, p1=h, centre=c2}
95 outlet = Line:new{p0=i, p1=h}
96 cyl_end_face = CoonsPatch:new{south=xaxis, north=outlet,
97                               west=cylinder, east=inner}
98 yaxis2 = Line:new{p0=f, p1=k}
99
100 over_cylinder = SweptSurfaceVolume:new{face0123=domain_end_face,
101                                        edge04=yaxis0}
102 outside_cylinder = SweptSurfaceVolume:new{face0123=domain_end_face,
103                                          edge04=yaxis1}
104 beside_cylinder = SweptSurfaceVolume:new{face0123=cyl_end_face,
105                                         edge04=yaxis2}
106
107 -- Build discrete grids.
108 -- We choose a basic discretization and scale others from it.
109 nr = 20                    -- number of cells radially
110 nc = math.floor(1.5 * nr) -- number of cells circumferentially
111 na = math.floor(L/D * nc) -- number of cells along the cylinder
```

```
na1 = nc                  -- cells off the end of the cylinder
nr2 = math.floor(nr/2)    -- cells toward the cylinder axis
-- Adjust the cluster functions by trying various values.
cf0 = RobertsFunction:new{end0=true, end1=false, beta=1.5}
grid0 = StructuredGrid:new{pvolume=over_cylinder,
                           cfList={edge03=cf0, edge47=cf0},
                           niv=nr+1, njv=nc+1, nkv=na+1}
grid1 = StructuredGrid:new{pvolume=outside_cylinder,
                           cfList={edge03=cf0, edge47=cf0},
                           niv=nr+1, njv=nc+1, nkv=na1+1}
cf1 = RobertsFunction:new{end0=true, end1=false, beta=1.05}
cf2 = RobertsFunction:new{end0=true, end1=false, beta=1.6}
grid2 = StructuredGrid:new{pvolume=beside_cylinder,
                           cfList={edge01=cf1, edge32=cf2,
                                   edge45=cf1, edge76=cf2},
                           niv=nr2+1, njv=nc+1, nkv=na1+1}

-- Use the grids to define some flow blocks.
-- Note that we divide up the biggest grid to make better use
-- of our multiple cpu machine.
blk0 = FBArray:new{grid=grid0, initialState=initial,
                   bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                           north=OutFlowBC_Simple:new{}},
                   nkb=math.floor(L/D)}
blk1 = FluidBlock:new{grid=grid1, initialState=initial,
                      bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                              north=OutFlowBC_Simple:new{}}}
blk2 = FluidBlock:new{grid=grid2, initialState=initial,
                      bcList={east=OutFlowBC_Simple:new{},
                              north=OutFlowBC_Simple:new{}}}
identifyBlockConnections()

-- Set a few more config options
config.flux_calculator = "adaptive"
config.thermo_interpolator = "pT"
config.adjust_invalid_cell_data = true
config.report_invalid_cells = false
config.max_invalid_cells = 10
-- config.cfl_count = 3
config.gasdynamic_update_scheme="euler"
my_max_time = Rc/V_inf * 30
print("max_time=", my_max_time)
config.max_time = my_max_time
config.max_step = 40000
config.dt_init = 1.0e-10
config.cfl_value = 0.5
config.dt_plot = my_max_time/10
```

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
--   11-Aug-2015 - updated for dlang module

--[[
Config{
   tightTempCoupling = true
}
--]]

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

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

Shell script

```
#!/bin/bash
# prep.sh
prep-gas nitrogen-2sp.inp nitrogen-2sp.lua
prep-chem nitrogen-2sp.lua nitrogen-2sp-2r.lua e4-chem.lua
e4shared --prep --job=cyl
```

```
#!/bin/bash
# run.sh
e4shared --run --job=cyl --verbosity=1 --max-cpus=4
```

Postprocessing program

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

# Create a VTK plot file of the steady full flow field.
e4shared --post --job=cyl --tindx-plot=last --vtk-xml

# Pull out the cylinder surfaces.
e4shared --post --job=cyl --tindx-plot=last --output-file=cylinder \
    --add-vars="mach" --surface-list="0,east;1,east;3,bottom"

# Now pull out some block surfaces that show cross-sections of the flow field.
e4shared --post --job=cyl --tindx-plot=last --output-file=interior \
    --add-vars="mach" \
    --surface-list="0,bottom;1,bottom;0,north;1,north;2,north;3,north;0,south;1,sou

# Stagnation-line flow data
e4shared --post --job=cyl --tindx-plot=last --output-file=stagnation-line.data \
    --add-vars="mach" --slice-list="0,:,0,0" \
```

```
-- locate-bow-shock.lua
-- Invoke with the command line:
-- $ e4shared --custom-post --script-file=locate-bow-shock.lua
--
-- PJ, 2016-10-24, updated for Eilmer4
--
print("Locate a bow shock by its pressure jump.")
print("Start by reading full flow solution.")
fsol = FlowSolution:new{jobName="cyl", dir=".", tindx=10, nBlocks=4}
print("fsol=", fsol)

function locate_shock_along_strip()
   local p_max = ps[1]
   for i = 2, #ps do
      p_max = math.max(ps[i], p_max)
   end
   local p_trigger = ps[1] + 0.3 * (p_max - ps[1])
   local x_old = xs[1]; local p_old = ps[1]
   local x_new = x_old; local p_new = p_old
   for i = 2, #ps do
      x_new = xs[i]; p_new = ps[i]
      if p_new > p_trigger then break end
      x_old = x_new; p_old = p_new
   end
   local frac = (p_trigger - p_old) / (p_new - p_old)
   x_loc = x_old * (1.0 - frac) + x_new * frac
   return
end

-- 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.
-- Block 0 contains the stagnation point and the bottom surface is
```

```
-- the plane of symmetry that cuts the cylinder half-way along its axis.
-- The south boundary is the plane of symmetry that cuts the cylinder
-- along its axis. Supersonic flow comes in from the west boundary
-- and exits from the north boundary. The east boundary is the
-- cylinder surface.
local xshock = {}; local yshock = {}
local ib = 0
local nk = fsol:get_nkc(0)
for k = 0, nk-1 do
   xs = {}; ys = {}; ps = {}
   local j = 0
   local ni = fsol:get_nic(ib)
   for i = 0, ni-1 do
      cellData = fsol:get_cell_data{ib=ib, i=i, j=j, k=k}
      xs[#xs+1] = cellData["pos.x"]
      ps[#ps+1] = cellData["p"]
   end
   locate_shock_along_strip()
   xshock[#xshock+1] = x_loc
   yshock[#yshock+1] = y_loc
   if #xshock >= 6 then break end
end

x_sum = 0.0
for _,x in ipairs(xshock) do x_sum = x_sum + x end
x_average = x_sum / #xshock
print("Average x-location=", x_average)
D = 15.0e-3 -- cylinder diameter
delta = -x_average - D/2
print("shock displacement=", delta*1000.0, "mm")
print("delta/R=", delta/(D/2))
```

5.3 Revisiting the flow over a sharp cone.

As a final example, let us return to the sharp-cone flow explored in the first tutorial example of Section 3, but introduce simulation on unstructured grids as an important new feature of Eilmer. This capability allows the convenient simulation of flows in geometrically-complex flow domains. Although the two-dimensional geometry of the flow domain over the cone is not complex, it is familiar and it serves to show the basic arrangement for computing a flow field on a collection of unstructured grids.

5.3.1 Input script (.lua)

Since all of the geometry modelling and grid construction was done by Kyle Damm in Pointwise mesh generation package, there is not much required in the input script.

```
1 -- cone20.lua
2 -- Unstructured Grid Example -- for use with Eilmer4
3 -- 2015-11-08  PeterJ, RowanG, KyleD
4
5 config.title = "Mach 1.5 flow over a 20 degree cone -- Unstructured Grid."
6 print(config.title)
7 config.dimensions = 2
8 config.axisymmetric = true
9
10 nsp, nmodes, gm = setGasModel('ideal-air-gas-model.lua')
11 print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
12 initial = FlowState:new{p=5955.0, T=304.0}
13 inflow = FlowState:new{p=95.84e3, T=1103.0, velx=1000.0}
14
15 -- Define the flow domain using an imported grid.
16 grids = {}
17 for i=0,3 do
18    fileName = string.format("cone20_grid%d.su2", i)
19    grids[i] = UnstructuredGrid:new{filename=fileName, fmt="su2text"}
20 end
21 my_bcDict = {INFLOW=InFlowBC_Supersonic:new{flowState=inflow},
22              OUTFLOW=OutFlowBC_Simple:new{},
23              SLIP_WALL=WallBC_WithSlip:new{},
24              INTERIOR=ExchangeBC_MappedCell:new{list_mapped_cells=true},
25              CONE_SURFACE=WallBC_WithSlip:new{}
26             }
27 blks = {}
28 for i=0,3 do
29    blks[i] = FluidBlock:new{grid=grids[i], initialState=inflow,
30                             bcDict=my_bcDict}
31 end
32
33 -- Do a little more setting of global data.
34 config.max_time = 5.0e-3  -- seconds
35 config.max_step = 3000
36 config.dt_init = 1.0e-6
37 config.cfl_value = 0.5
38 config.dt_plot = 1.5e-3
39 config.dt_history = 10.0e-5
40
```

```
41 setHistoryPoint{x=1.0, y=0.2} -- nose of cone
42 setHistoryPoint{x=0.201, y=0.001} -- base of cone
```

The main tasks are to:

- select a gas model (line 10),
- define the initial and inflow conditions (lines 12 and 13),
- import the grids from external files (line 16 to 20) and
- build the flow blocks from the imported grids (lines 27 to 30), attaching boundary conditions to them.

Note that the keys used for the boundary condition table (lines 21 to 26) need to match the tags within the SU2 grid files. These tags could be agreed upon at grid-generation time or you could extract them from the grid files with the following command:

```
$ grep MARKER_TAG *.su2
```

To apply the boundary conditions, supply the dictionary of boundary conditions to the `FluidBlock` constructor as the `bcDict` item (line 29).

5.3.2 Results and postprocessing

The simulation runs to 5 milliseconds in 1641 steps and requires about 80 seconds wall clock, while using the 4 cores of a HP Pavillion laptop. Figure 5.8 shows the flow field, with the shock appearing a bit noisier than for the structured-grid simulation but otherwise essentially straight. The post-processing script loads up the data for this time and samples the flow field across a number of straight lines, looks for the pressure jump along each sample line and accumulates the arrays of shock coordinates. Finally, on lines 58 and later, a linear model is fitted to this collection of shock points (using a least-squares error critereon). The angle of the line fitted to these points is 49.22^o and the average deviation is 2.8 mm.

```
1 -- estimate_shock_angle.lua
2 -- Invoke with the command line:
3 -- $ e4shared --custom-post --script-file=estimate_shock_angle.lua
4 -- PJ, 2016-11-13
5 --
6 print("Begin estimate_shock_angle for the unstructured-grid case.")
7 nb = 4
8 fsol = FlowSolution:new{jobName="cone20", dir=".", tindx=4, nBlocks=nb}
9 print("fsol=", fsol)
10
11 function locate_shock_along_strip()
12    local p_max = ps[1]
13    for i = 2, #ps do
14       p_max = math.max(ps[i], p_max)
15    end
16    local p_trigger = ps[1] + 0.3 * (p_max - ps[1])
17    local x_old = xs[1]; local y_old = ys[1]; local p_old = ps[1]
18    local x_new = x_old; local y_new = y_old; local p_new = p_old
```

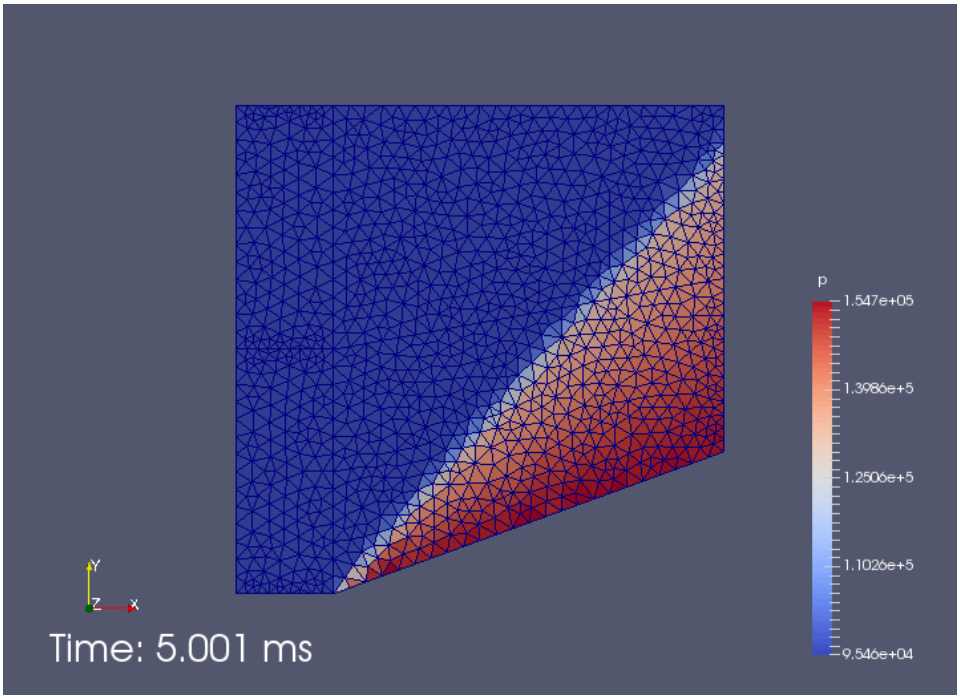


Figure 5.8: Pressure field for a low-resolution unstructured-grid simulation of flow over a cone with 20 degree half-angle.

```
19    for i = 2, #ps do
20       x_new = xs[i]; y_new = ys[i]; p_new = ps[i]
21       if p_new > p_trigger then break end
22       x_old = x_new; y_old = y_new; p_old = p_new
23    end
24    local frac = (p_trigger - p_old) / (p_new - p_old)
25    x_loc = x_old * (1.0 - frac) + x_new * frac
26    y_loc = y_old * (1.0 - frac) + y_new * frac
27    return
28 end
29
30 xshock = {}; yshock = {}
31 for j = 1, 45 do
32    local y = j*0.02
33    xs = {}; ys = {}; ps = {}
34    local cellsFound = fsol:find_enclosing_cells_along_line{p0={x=0.0,y=y},
35                                                            p1={x=1.0,y=y},
36                                                            n=100}
37    print("number of cells found=", #cellsFound)
38    for i,indices in ipairs(cellsFound) do
39       cellData = fsol:get_cell_data{ib=indices.ib, i=indices.i}
40       xs[#xs+1] = cellData["pos.x"]
41       ys[#ys+1] = cellData["pos.y"]
42       ps[#ps+1] = cellData["p"]
43    end
44    locate_shock_along_strip()
45    if x_loc < 0.9 then
46       -- Keep only the good part of the shock.
47       xshock[#xshock+1] = x_loc
```

```
48        yshock[#yshock+1] = y_loc
49     end
50  end
51
52  --[[
53  for j = 1, #xshock do
54     print("shock point j=", j, xshock[j], yshock[j])
55  end
56  --]]
57
58  -- Least-squares fit of a straight line for the shock
59  -- Model is y = alpha0 + alpha1 * x
60  sum_x = 0.0; sum_y = 0.0; sum_x2 = 0.0; sum_xy = 0.0
61  for j = 1, #xshock do
62     sum_x = sum_x + xshock[j]
63     sum_x2 = sum_x2 + xshock[j]*xshock[j]
64     sum_y = sum_y + yshock[j]
65     sum_xy = sum_xy + xshock[j]*yshock[j]
66  end
67  N = #xshock
68  alpha1 = (sum_xy/N - sum_x/N * sum_y/N) / (sum_x2/N - sum_x/N * sum_x/N)
69  alpha0 = sum_y/N - alpha1 * sum_x/N
70  shock_angle = math.atan(alpha1)
71  sum_y_error = 0.0
72  for j = 1, N do
73     sum_y_error = sum_y_error+math.abs((alpha0+alpha1*xshock[j])-yshock[j])
74  end
75  print("shock_angle_deg=", shock_angle*180.0/math.pi)
76  print("average_deviation_metres=", sum_y_error/N)
77  print("Done.")
```

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 Technical Report 2014/14, The University of Queensland, Brisbane, Australia, September 2014.

[2] Peter A. Jacobs and Rowan J. Gollan. The Eilmer 4.0 flow simulation program: Formulation of the transient flow solver. School of Mechanical and Mining Engineering Technical Report 2018/03, The University of Queensland, Brisbane, Australia, April 2018.

[3] R. J. Gollan and P. A. Jacobs. About the formulation, verification and validation of the hypersonic flow solver Eilmer. *International Journal for Numerical Methods in Fluids*, 73(1):19–57, 2013.

[4] Peter Jacobs and Rowan Gollan. Implementation of a compressible-flow simulation code in the D programming language. In *Advances of Computational Mechanics in Australia*, volume 846 of *Applied Mechanics and Materials*, pages 54–60. Trans Tech Publications, 9 2016.

[5] R. J. Gollan and P. A. Jacobs. The Eilmer 4.0 flow simulation program: Guide to the basic gas models package, including gas-calc and the API. School of Mechanical and Mining Engineering Technical Report 2017/27, The University of Queensland, Brisbane, Australia, February 2018.

[6] R. J. Gollan and P. A. Jacobs. The Eilmer 4.0 flow simulation program: Reacting gas thermochemistry with the thermally-perfect-gas model. School of Mechanical and Mining Engineering Technical Report 2018/04, The University of Queensland, Brisbane, Australia, March 2020.

[7] James M. Burgess. Adaptive look-up table gas model for Eilmer4. School of Mechanical and Mining Engineering Technical Report 2016/18, The University of Queensland, Brisbane, Australia, March 2016.

[8] Peter A. Jacobs, Rowan J. Gollan, and Ingo Jahn. The Eilmer 4.0 flow simulation program: Guide to the geometry package, for construction of flow paths. School of Mechanical and Mining Engineering Technical Report 2017/25, The University of Queensland, Brisbane, Australia, February 2018.

[9] Kyle Damm. Shock fitting mode for Eilmer. School of Mechanical and Mining Engineering Technical Report 2016/15, The University of Queensland, Brisbane, Australia, February 2016.

[10] 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.

[11] P. A. Jacobs. Single-block Navier-Stokes integrator. ICASE Interim Report 18, 1991.

[12] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, 2006.

[13] P. M. Knupp. A robust elliptic grid generator. *Journal of Computational Physics*, 100(2):409–418, 1992.

[14] Ames Research Staff. Equations, tables and charts for compressible flow. NACA Report 1135, 1953.

[15] D. I. Pullin. Direct simulation methods for compressible inviscid ideal-gas flow. *Journal of Computational Physics*, 34(2):231–244, 1980.

[16] 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.

[17] Y. Wada and M. S. Liou. A flux splitting scheme with high-resolution and robustness for discontinuities. AIAA Paper 94-0083, January 1994.

[18] J. J. Quirk. A contribution to the great Riemann solver debate. *International Journal for Numerical Methods in Fluids*, 18(6):555–574, 1994.

[19] M. S. Liou. A sequel to AUSM, part II: AUSM+-up for all speeds. *Journal of Computational Physics*, 214:137–170, 2006.

[20] 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.

[21] T. N. Eichmann, T. J. McIntyre, A. I. Bishop, S. Vakata, and H. Rubinsztein-Dunlop. Three-dimensional effects on line-of-sight visualization measurements of supersonic and hypersonic flow over cylinders. *Shock Waves*, 16(4–5):299–307, 2007.

[22] Mark G. Sobell. *A Practical Guide to Linux Commands, Editors and Shell Programming.* Prentice Hall, Upper Saddle River, New Jersey, 2005.

[23] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language.* Pearson, 1988.

A

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 [22] 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, two or three 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 commands together so that the output from one command becomes the input for another. Once you understand the system, customised commands can be built rather simply in this way. The following tables summarize a number of commands that you are likely to find useful while using Eilmer.

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' *.lua`	Find the string `ideal` in all of the Lua 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*.

B

A little bit of Lua

We use the Lua language as the format of the user's input scripts. It makes a very versatile input format that enables a lot of automation when setting up complex simulations, however, it demands a bit of understanding from the user. Our first advice is to get a cup of your favourite beverage and sit down for an hour or so to read the introductory sections of the *Programming in Lua* book [12]. The first edition of this book, which has all that you need, is available online https://www.lua.org/pil/contents.

That investment of time will be repaid many-fold but, if you are in a hurry to get some flow simulations going, this section may be just enough to get started. To try out bits of Lua code within the context of Eilmer but without the concerns of setting up a full simulation, you can process an arbitrary Lua script with the command:

```
$ e4shared --custom-post --script-file=myscript.lua
```

where `myscript.lua` is the name of the file containing your Lua code.

B.1 Basics and syntax

Short comments start with a double dash `--` and finish at the end of the line. Long comments may span multiple lines by using `[[ ... ]]` to hold the lines together as a single string and then comment the whole string with `--`. For example:

```
--[[ Long, multiple-line comments
can be used to cut sections of code
out of your script.]]
```

Commands (or statements) are the lines in you script that actually get things done. For example:

```
-- Compute the area of a circle.
radius = 2.0
area = math.pi * radius^2
print("For radius=", radius, "area=", area)
```

Identifiers or names, as in the example above, can be any string of letters, digits or underscores. Names cannot begin with a digit but they can begin with an underscore. If you see name consisting of a single underscore, it is probably being used as a dummy variable. And, when we talk about variables, we are talking about names bound to data. The types of data that you will be manipulating include:

- numbers
- strings
- booleans
- `nil`
- functions
- tables
- userdata

Numbers are floating-point values. Strings of characters may be delimited by double or single quote characters. Boolean values are either `true` or `false`. In a boolean context, only `false` and `nil` are effectively false. Other values, such as `0` (the number zero), are effectively true. This might be surprising, especially if you have a C-programming background.

The special value `nil` represents nothing and is different from all other types. If we try to access a variable that is not initialized, the value is `nil`. We might say that the name is bound to nothing.

Functions and tables will be discussed below while userdata is a type representing C data structures and implemented via the C programming interface.

B.2 Operators and expressions

An expression is a combination of operands (data) and operators that results in a single value of a certain type. The expression `math.pi * radius^2` from the previous code snippet is an example. Names that appear in such expressions evaluate to the value that was bound to them.

The assignment operator = is used to bind names to data values, as was done for the `radius` and `area` variables. In Lua, variables can contain any type. The type is an attribute of the value.

B.3 Tables

Almost all complex data types in Lua are constructed as some form of table. A table literal is delimited by braces `{}` and entries can have numeric indices or string keys. For example:

```
t1 = {} -- an empty table
t2 = {["x"]=1.0, [1]=2.0, ["1"]=3.0} -- [1] is different to ["1"]
t3 = {1.0, 2.718, 3.142} -- an array
```

To access items in the table, use square brackets. For example `t2[1]` would evaluate to the number `2`, while `t3[1]` would evaluate to the number `1`. There is a shorthand for indexing items with string keys, so that `t2["x"]` and `t2.x` are equivalent and evaluate to the number `1`. If you try to access a nonexistent entry, you obtain the `nil` value.

B.4 Functions

Functions are a way to abstract blocks of your code. They are first-class objects that can be created as anonymous and then assigned to a variable or returned from another function. Here is a simple function that computes a parabolic curve.

```
myParabola = function(s)
   local x = s
   local y = s^2
   return x, y
end
xx, yy = myParabola(0.5)
print("for s=", 0.5, "xx=", xx, "yy=", yy)
```

The call operator parentheses `()` following the name `myParabola` invokes the function. The numeric value `0.5` is passed as the only argument and assigned to the local variable `s` within the function. The commands within the body of the function are then executed.

Note that a function can return multiple values and those values can be assigned, in parallel, to multiple variables. An example of this multiple assignment in use for setting up a simulation can be seen in the setting of the gas model in the introductory simulation of a sharp-nosed cone on page 11. There we catch the returned number of chemical species, number of internal energy modes, and a reference to the GasModel object, and assign these three items to three variables.

Note, also, the prefix `local` when assigning to the `x` and `y` variables in the function. Variables assigned without a local prefix will be global variables. Sometimes this default behaviour will be useful but it can be surprising if you are coming to Lua from another programming language.

B.5 Object-based programming

The underlying D-language objects used in your simulation are wrapped in interface code and presented to your script in the Lua domain with an object-based notation that is described in one of the later chapters of the `Programming in Lua` book. Here, we will just provide an example of constructing a couple of points in 3D space and then using those points to define a line segment.

```
a = Vector3:new{x=0.0, y=0.0}
b = Vector3:new{x=0.2, y=0.0}
ab = Line:new{p0=a, p1=b}
print("line ab=", ab)
print("midpoint is at ", ab(0.5))
```

Note the colon : characters before the `new` method name. These indicate that the methods are bound to particular objects. Note, also, that the data given to each new method are contained within single table literal. When calling a function and providing a single argument that happens to be a table, you may omit the call operator parentheses `()`.

B.6 Control statements

Processing of your script statements proceeds sequentially unless a control statement says otherwise. Code blocks are delimited with key words, usually terminated by the keyword `end`. Lua has the usual compound statement constructs to control the flow of execution of your script.

Selection of one code block or another another is controlled by the if-then-else-end construct. For example, to conditionally execute some code, you could write:

`if` *condition* `then`
    code-block
`end`

To select between alternative code blocks, the `else` and `elseif` keywords are available. For example:

`if` *condition-1* `then`
    code-block-1
`elseif` *condition-2* `then`
    code-block-2
`else`
    code-block-3
`end`

Code blocks can be repetitively executed using the `while` and `for` loop constructs. The while loop looks like:

`while` *condition* `do`
    code-block
`end`

and there are two flavours of for loop. The numerical for loop to print the odd integers up to 9 might look like the following example:

```
for i=1, 9, 2 do
   print(i)
end
```

and the generic for loop is good for iterating through table entries:

```
t = {["pi"]=3.14, e=2.71}
```

```
for k, v in pairs(t) do
   print("key=", k, "value=", v)
end
```

For an array of items, there is the `ipairs` iterator to work with the same generic for loop. It provides the index for each value, as so:

```
t = {"a", 3.142, "b", 2.718}
for i, v in ipairs(t) do
   print("index=", i, "value=", v)
end
```

B.7 Global symbols in the Lua environment

Before your Lua input script is processed by Eilmer, there are a number of symbols inserted into the global name space of the Lua interpreter. These are for you use in constructing flow conditions and your description of the flow domain, and for setting simulation configuration parameters. This list of names can be obtained from within your script with the line

```
for k,_ in pairs(_G) do print(k) end
```

This prints all of the names found in the global namespace table, `_G`. To avoid unpleasant surprises, you should not be rebind any of these names to other objects in your script.

C

Understanding an AWK script

We use the AWK programming language [23] as a programmable filter for text files. When an AWK program processes input files, it splits the text into records and fields. Records, by default, are separated by newlines. Fields, within a record, are separated by whitespace. For each record, the fields may then be selected for use in calculations and displayed.

Here, again, is the AWK script used to extract the pressure history for the sharp-cone simulation back in Section 3.2. The line numbers in the left margin are not part of the script.

```
1  # cp.awk
2  # Scan a history file, picking out pressure and scaling it
3  # to compute coefficient of pressure.
4  #
5  # PJ, 2016-09-22
6  #
7  BEGIN {
8      Rgas = 287.1; # J/kg.K
9      p_inf = 95.84e3; # Pa
10     T_inf = 1103; # K
11     rho_inf = p_inf / (Rgas * T_inf)
12     V_inf = 1000.0; # m/s
13     q_inf = 0.5 * rho_inf * V_inf * V_inf
14     print "# rho_inf=", rho_inf, " q_inf=", q_inf
15     print "# t,ms cp"
16 }
17
18 $1 != "#" {
19     t = $1; p = $10
20     print t*1000.0, (p - p_inf)/q_inf
21 }
22
23 END {}
```

It is used with the command line:

```
$ awk -f cp.awk hist/cone20-blk-1-cell-20.dat > cone20_cp.dat
```

to scan the history data file `hist/cone20-blk-1-cell-20.dat` and write its pressure-coefficient results, via standard-output redirection, to the file `cone20_cp.dat`.

Other than comments, that start with a `#` character and continue until the end of the line, the AWK program above consists of pattern-action statements of the form
pattern `{` *action* `}`
As the input text file is scanned, each line is checked against the patterns and any matching pattern has its action performed. A missing pattern always matches and a missing action results in the whole line being printed.

The `BEGIN` and `END` patterns are special and their actions are performed, unsurprisingly, at the beginning of program execution (before any records are processed) and at the end of program execution (following the processing of all records). We use the `BEGIN` action (lines 8 through 16) to set up a number of named constants and print a header to the standard output. This header is convenient for later reminding us of the meaning of the computed numbers.

The action that does the interesting work starts on line 18 with a pattern that matches, so long as the first field (denoted by `$1`) is not equal to the string consisting of the single character `#`. We often use this character in our data files to indicate a comment rather than data because GNUPlot accepts data mixed with such comments. If the pattern matches, the record is then acted upon. First, field 1 is assigned to the variable `t`, representing time in seconds, and field 10 is assigned to the variable `p`, representing static pressure in Pascals. We scale the time to milliseconds and convert the static pressure into a pressure coefficient, normalized by the previously-computed value of dynamic pressure. The `print` sends the results to standard output, which we had redirected to our results file `cone20_cp.dat`.

There is much more to the AWK programming language. You can have other forms of patterns based on regular expressions and start,stop pairs. Your action may include arbitrarily complex calculations using programming constructs such as `if` statements and loops. When your AWK code starts to grow in complexity, you will probably want to define your own special-purpose functions. These will be introduced with the keyword `function` and be of the form:
`function` *name* `(` *parameter-list* `)` `{` *statements* `}`
You also have access to special variables such as `NR`, the current record number, and `NF`, the number of fields in the current record. There is a slightly more complex script for computing shear stress on the flat plate in Section 5.1.5.

D

Functions for simple gas flows

From within the Lua environment of the main program, you have access to a number of functions for computing simple gas flow relations. These functions might be handy when setting flow conditions during simulation preparation, within user-defined boundary conditions or during post-processing activities.

D.1 Ideal gas

The first set of functions is for simple flow situations of an ideal gas. See page 20 for an example of using one of the functions. All of the functions are contained in the single table `idealgasflow`, however, they are grouped below according to flow situation.

D.1.1 Simple isentropic flow

- `A_Astar(`M, $g = 1.4$`)` returns the area ratio, $\frac{A}{A^*}$, for a given Mach number M and ratio of specific heats, g. If you don't provide a value for g, the default value of 1.4 will be used.
- `T0_T(`M, $g = 1.4$`)` returns the total-temperature to static-temperature ratio, $\frac{T_0}{T}$.
- `p0_p(`M, $g = 1.4$`)` returns the total-pressure to static-pressure ratio, $\frac{p_0}{p}$.
- `r0_r(`M, $g = 1.4$`)` returns the density ratio, $\frac{\rho_0}{\rho}$.

D.1.2 Normal shock relations

- `m2_shock(`M_1, $g = 1.4$`)` returns the Mach number following shock processing. The frame of reference has the shock stationary, with the incoming flow being supersonic, with Mach number M_1.
- `r2_r1(`M_1, $g = 1.4$`)` returns the density ratio, $\frac{\rho_2}{\rho_1}$, across the shock. State 2 is the shock-processed state.
- `u2_u1(`M_1, $g = 1.4$`)` returns the velocity ratio, $\frac{u_2}{u_1}$, across the shock.
- `p2_p1(`M_1, $g = 1.4$`)` returns the pressure ratio, $\frac{p_2}{p_1}$, across the shock.

- `T2_T1(`M_1, $g = 1.4$`)` returns the static-temperature ratio, $\frac{T_2}{T_1}$, across the shock.
- `p02_p01(`M_1, $g = 1.4$`)` returns the total-pressure ratio, $\frac{p_{02}}{p_{01}}$, across the shock.
- `DS_Cv(`M_1, $g = 1.4$`)` returns the normalized entropy change, $\frac{s_2 - s_1}{C_v}$, across the shock.
- `pitot_p(`M_1, $g = 1.4$`)` returns the Pitot pressure of the stream (following shock processing).

D.1.3 1D flow with heat addition

- `T0_T0star(`M, $g = 1.4$`)` returns the total-temperature ratio, $\frac{T_0}{T_0^*}$, for a given Mach number M and ratio of specific heats, g. T_0 is the local total-temperature value and T_0^* is the total temperature at the hypothetical critical point where enough heat has been added for the Mach number to be 1.
- `M_Rayleigh(`T_r, $g = 1.4$`)` returns the local Mach number for a given total-temperature ratio, $T_r = \frac{T_0}{T_0^*}$.
- `T_Tstar(`M, $g = 1.4$`)` returns the static-temperature ratio, $\frac{T}{T^*}$.
- `p_pstar(`M, $g = 1.4$`)` returns the static-pressure ratio, $\frac{p}{p^*}$.
- `r_rstar(`M, $g = 1.4$`)` returns the density ratio, $\frac{\rho}{\rho^*}$.
- `p0_p0star(`M, $g = 1.4$`)` returns the total-pressure ratio, $\frac{p_0}{p_0^*}$.

D.1.4 Supersonic turning flow (isentropic)

- `PM1(`M, $g = 1.4$`)` returns the Prandtl-Meyer function ν (in radians) for a given Mach number.
- `PM2(`ν, $g = 1.4$`)` returns the Mach number for a given Prandtl-Meyer value (in radians).
- `MachAngle(`M`)` returns the Mach angle μ in radians.

D.1.5 Oblique shock relations

- `beta_obl(`M_1, θ, $g = 1.4$, $tol = 1.0e-6$`)` returns the oblique shock angle, β (in radians), with respect to the undeflected free-stream direction. Required input includes the free-stream Mach number, M, and the flow deflection angle, θ (in radians).
- `beta_obl2(`M_1, $\frac{p_2}{p_1}$, $g = 1.4$`)` returns the shock wave angle, given free-stream Mach number and static-pressure ratio across the oblique shock.
- `theta_obl(`M_1, β, $g = 1,4$`)` returns stream deflection angle, θ (in radians), given shock angle, β (in radians).
- `M2_obl(`M_1, β, θ, $g = 1.4$`)` returns the Mach number behind the oblique shock.

- `r2_r1_obl(`M_1, β, $g = 1.4$`)` returns the density ratio, $\frac{\rho_2}{\rho_1}$, across the oblique shock.

- `Vn2_Vn1_obl(`M_1, β, $g = 1.4$`)` returns the normal-velocity ratio, $\frac{V_{n2}}{V_{n1}}$, across the oblique shock.

- `V2_V1_obl(`M_1, β, $g = 1.4$`)` returns the flow speed ratio, $\frac{V_2}{V_1}$, across the oblique shock.

- `p2_p1_obl(`M_1, β, $g = 1.4$`)` returns the static-pressure ratio, $\frac{p_2}{p_1}$, across the oblique shock.

- `T2_T1_obl(`M_1, β, $g = 1.4$`)` returns the static-temperature ratio, $\frac{T_2}{T_1}$, across the oblique shock.

- `p02_p01_obl(`M_1, β, $g = 1.4$`)` returns the total-pressure ratio, $\frac{p_{02}}{p_{01}}$, across the oblique shock.

D.1.6 Conical shock flow

- `theta_cone(`V_1, p_1, T_1, β, $R = 287.1$, $g = 1.4$`)` returns four values, θ_c, V_c, p_c, T_c, that specify the cone surface angle and the surface values of velocity, pressure and temperature, respectively. Required input includes the free-stream velocity, V_1, the static pressure, p_1, static temperature, $T1$, and the conical shock wave angle, β (in radians), with respect to the free-stream direction.

- `beta_cone(`V_1, p_1, T_1, θ, $R = 287.1$, $g = 1.4$`)` returns the shock wave angle, β (in radians), given the free-stream conditions (in SI units) and cone surface angle, θ (in radians).

- `beta_cone2(`M_1, θ, $R = 287.1$, $g = 1.4$`)` returns the shock wave angle, β (in radians), given the free-stream Mach number and cone surface angle, θ (in radians).

D.2 General gas

The second set of functions is for simple flow situations of a more general gas, using one of the built-in gas models. These functions operate on GasState objects that have been constructed in the context of a GasModel.

D.2.1 Steady isentropic flow

- `state1, V = gasflow.expand_from_stagnation(state0, p_over_p0)` returns the flow state (1) from a given stagnation condition (0) and pressure ratio. `V` is the velocity of the expanded gas, assuming isentropic, steady-state processing.

- `state1, V = gasflow.expand_to_mach(state0, mach)` returns the expanded-gas flow state at a particular Mach number.

- `state0 = gasflow.total_condition(state1, V1)` returns the stagnation condition given a particular free-stream condition, assuming isentropic, steady-state processing.
- `state2pitot = gasflow.pitot_condition(state1, V1)` returns the Pitot-probe stagnation condition. The free-stream flow (1) may be supersonic, such that a normal shock will form between the free-stream and the stagnation region.
- `state2, V2 = gasflow.steady_flow_with_area_change(state1, V1, A2_over_A1, tol)`
  returns the flow state resulting from a steady isentropic process associated with a change in stream-tube area. The area change is specified as the ratio $\frac{A_2}{A_1}$ and there is a default value of 10^{-4} for `tol`.

D.2.2 Unsteady isentropic flow

- `state2, V2 = gasflow.finite_wave_dp(state1, V1, characteristic, p2, steps)`
  returns the state (2) following an unsteady isentropic process to a new pressure `p2`. The process follows either the `"cplus"` or `"cminus"` characteristic. There is a default value of 100 for the number of steps.
- `state2, V2 = gasflow.finite_wave_dp(state1, V1, characteristic, V2_target, steps, Tmin)`
  returns the state (2) following an unsteady isentropic process to a new velocity `V2_target`. The process follows either the `"cplus"` or `"cminus"` characteristic. There is a default value of 100 for the number of steps. `Tmin` has a default value of 200.0 K, and the stepping process will be terminated if the gas temperature falls below this value.

D.2.3 Normal shock relations

- `state2, V2, Vg = gasflow.normal_shock(state1, Vs, rho_tol, T_tol)`
  returns the conditions following shock processing. The frame of reference has the shock stationary, with the incoming flow (1) being supersonic, with velocity, `Vs`. The velocity, `Vg`, is the velocity of the post-shock (2) gas in the laboratory frame, in which the shock has travelled past with velocity `Vs` into quiescent gas.
- `V1, V2, Vg = gasflow.normal_shock_p2p1(state1, p2p1)` returns the pre- and post-shock velocities, given the pressure ratio across the shock.
- `state5, Vr = gasflow.reflected_shock(state2, Vg)` returns the post-reflected-shock conditions, given the flow condition following the incident shock. Both velocities are in the laboratory frame, where the initial (1) and final (5) gas velocities are zero.

D.2.4 Oblique shock relations

- `state2, theta, V2 = gasflow.theta_oblique(state1, V1, beta)`
  returns the post-shock conditions (2) for a given free-stream condition (1) and

shock-wave angle, `beta`, (in radians) with respect to the free-stream flow direction. The angle, `theta`, is the associated flow deflection angle, also in radians.

- `beta = gasflow.theta_oblique(state1, V1, theta)` returns the shock-wave angle (in radians) with respect to the free-stream (1) flow direction.

D.2.5 Example of use

The following example shows the use of the gas flow functions to compute the approximate test flow conditions for one of the operating conditions for the T4 shock tunnel. The CEA gas model is used, with a 13-species air specification, and processing is performed state-to-state.

```
model = "CEAGas"

CEAGas = {
  mixtureName = 'air13species',
  speciesList = {"N2","O2","Ar","N","O","NO","Ar+","NO+","N+","O+","N2+","O2+","e-"},
  reactants = {N2=0.7811, O2=0.2095, Ar=0.0093},
  inputUnits = "moles",
  withIons = true,
  trace = 1.0e-6
}
```

```
-- reflected-shock-tunnel.lua
-- Run with a command like:
-- $ e4shared --custom-post --script-file=reflected-shock-tunnel.lua
-- PJ, 2017-11-12
print("Compute the test-flow conditions for a shot in the T4 shock tunnel.")
--
print("shock-tube fill conditions")
gm = GasModel:new{'cea-air13species-gas-model.lua'}
state1 = GasState:new{gm}
state1.p = 125.0e3; state1.T = 300.0
gm:updateThermoFromPT(state1)
print("state1:"); printValues(state1)
--
print("normal shock, given shock speed")
Vs = 2414.0
state2, V2, Vg = gasflow.normal_shock(state1, Vs)
print("V2=", V2, "Vg=", Vg)
print("state2:"); printValues(state2)
--
print("reflected shock")
state5, Vr = gasflow.reflected_shock(state2, Vg)
print("Vr=", Vr)
print("state5:"); printValues(state5)
--
print("Expand from stagnation (with ratio of pressure to match observation)")
state5s, V5s = gasflow.expand_from_stagnation(state5, 34.37/59.47)
print("V5s=", V5s, " Mach=", V5s/state5s.a)
print("state5s:"); printValues(state5s)
print("(h5s-h1)=", gm:enthalpy(state5s) - gm:enthalpy(state1))
```

```
30 --
31 print("Expand to throat condition (Mach 1.0001)")
32 state6, V6 = gasflow.expand_to_mach(state5s, 1.0001)
33 print("V6=", V6, " Mach=", V6/state6.a)
34 print("state6:"); printValues(state6)
35 --
36 print("Mach 4 nozzle expansion to test-flow condition.")
37 state7, V7 = gasflow.steady_flow_with_area_change(state6, V6, 27.0)
38 print("V7=", V7, " Mach=", V7/state7.a)
39 print("state7:"); printValues(state7)
```

E

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 that are not part of the standard flow solver. These callable functions can be used:

- as specialized boundary conditions;
- for the addition of custom source terms;
- to prescribe grid motion; and
- to perform special operations at the beginning and end of each time step.

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. The callable functions at the start of each time step could be used to compute a special flow field variable.

Each of your customizations will be Lua code, typically a function of the form:

```
function specifiedName(args)
   -- Do something interesting,
   -- perhaps making use of args,
   -- then return a value.

   return requiredValue
end
```

The main simulation code sets up `args`, specific to the time and place of the call. It then requests the Lua interpreter to execute the function `specifiedName`, passing `args`. Your Lua code within the function performs whatever custom calculation is needed and returns `requiredVal`. The simulation code then makes use of `requiredVal` in its subsequent calculations. Of course, you may have other supporting code in your Lua script. The entire script file is interpreted at start-up, so variables and functions may be defined, files may be read and tables set up. You have the full capability of the Lua interpreter at your service.

E.1 Customizing the boundary conditions

Users can affect both convective and diffusive boundary conditions. The customized boundary conditions may supply ghost-cell flow states and interface values or fluxes directly. Using a customized boundary condition requires two steps:

1. Setting the `UserDefinedGhostCellBC` or `UserDefinedFluxBC` boundary condition (page 58) in the FluidBlock setup.

2. Constructing a Lua file which defines the boundary condition behaviour.

E.1.1 Setting ghost cells and interface properties

When the user's (Lua) input script constructs a new `UserDefinedGhostCellBC` boundary condition, a Lua file name is specified. This Lua file is interpreted at the time of boundary-condition construction and it needs to define the Lua function `ghostCells()`, at a minimum. For a viscous simulation, you will also need to define the Lua function `interface()`. These particular functions are later called, every time the boundary condition is applied during the simulation. In fact, they are called for every interface along a boundary; the functions work interface-by-interface. As well as providing the *expected* functions, the Lua file may contain whatever supporting code that the user wishes to include. 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.

To set up a user-defined boundary condition 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 by defining a `ghostCells(args)` function. In this case, you populate the properties of two ghost cells such that they give the desired convective flow effect at the boundary. The ghost cells are abstract in that they do not exist in the simulated flow domain but do exist in the program's data for each block boundary. They are used in the interpolation phase of the convective update, for cell faces that lie along the boundary. The function returns two tables of flow data. The first table is for the inner ghost cell, the one closest to the domain edge. The second table is for the outer ghost cell.

The viscous effects at the boundary are handled by 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. Note that, in an inviscid simulation, any user-specified viscous boundary effect functions are ignored; they are never called by the code.

Ghost cells

On being called at run time, the function `ghostCells(args)` returns two Lua tables. The user writing the function 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 farther away from the boundary face. If you do not wish to set the ghost-cell data from your user-defined function, you may return *empty* tables. This may sound a strange thing to do, but there are situations where you would conditionally set the ghost-cell data when it has previously been set by an exchange action. (See the `ExchangeBC_FullFacePlusUDF` boundary condition on page 59.)

Items that are supplied in `args` table include some state information for the update, some geometric information for the ghost cells, the boundary interface, adjacent interior cell:

t the current simulation time, in seconds

dt the current time-step size, in seconds

timeStep integral time step count

gridTimeLevel integral value indicating the phase of the update for a moving grid

flowTimeLevel integral value indicating the phase of the flow update

boundaryId integral value indicating the which boundary within the block we are working on For a structured-grid block it may be convenient to use the symbolic indices `north`,...

x,y,z coordinates of the midpoint of the interface, in m

csX,csY,csZ direction cosines for unit normal of the interface

csX1,csY1,csZ1 direction cosines for first tangent of the interface

csX1,csY1,csZ1 direction cosines for second tangent of the interface

i,j,k indices of the interior cell adjacent to the interface

gc0x,gc0y,gc0z coordinates of the centre of the first ghost cell

gc1x,gc1y,gc1z coordinates of the centre of the second ghost cell

If you want information on the flow state within the adjacent interior cell, use the `sampleFluidCell` service function to get the data shown on page 129. Items to appear in the returned tables are:

p gas pressure in Pa (required)

T trans-rotational temperature in Kelvin (required)

T_modes array of temperatures associated with other internal energy modes. This is only required if `n_modes` is nonzero.

massf table of named mass fractions. This is only required if `n_species` is greater then 1. For a single-species simulation, a value of 1.0 for the only species is the default.

velx,vely,velz velocity components in x,y,z-directions in m/s. The default value for any component is 0.0.

tke turbulent kinetic energy per unit mass, in J/kg. Default is 0.0

omega ω for the $k-\omega$ turbulence model, in 1/s. Default is 1.0.

S shock-detector value (1 or 0). Default is 0.

Other gas thermodynamic data, such as internal energy and sound speed, are determined via the gas model.

Note that your `ghostCells` 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.

Interfaces

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:

p gas pressure in Pa

T trans-rotational temperature in Kelvin

T_modes array of temperatures associated with other internal energy modes.

massf table of named mass fractions.

velx,vely,velz velocity components in x,y,z-directions in m/s.

tke turbulent kinetic energy per unit mass, in J/kg.

omega ω for the $k-\omega$ turbulence model, in 1/s.

mu_t turbulence viscosity, in Pa.s

k_t turbulent heat conduction coefficient

Note that not all parameters may be needed, so the calling D code treats all parameters as optional and extracts from your table only the entries that it can find. These supplied values overwrite the corresponding values in the FlowState object at the boundary face. If you do not wish to set the interface data from your user-defined function, you may return an *empty* table. On entry to the function, `args` contains all of the same attributes as for the call to the `ghostCells` function. Additionally, `args` contains:

t the current simulation time, in seconds

dt the current time-step size, in seconds

timeStep integral time step count

gridTimeLevel integral value indicating the phase of the update for a moving grid

flowTimeLevel integral value indicating the phase of the flow update

boundaryId integral value indicating the which boundary within the block we are working on For a structured-grid block it may be convenient to use the symbolic indices `north`,...

x,y,z coordinates of the midpoint of the interface, in m

csX,csY,csZ direction cosines for unit normal of the interface

csX1,csY1,csZ1 direction cosines for first tangent of the interface

csX1,csY1,csZ1 direction cosines for second tangent of the interface

i,j,k indices of the interior cell adjacent to the interface

Remember that 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. This allows a useful pattern where, for example, flow profile data could be read from files at set up time and stored in a table that then is available for subsequent calls to `ghostCells` and `interface`.

E.1.2 Directly specifying fluxes

Instead of specifying ghost-cell and interface data that are used internally by the simulation code to compute fluxes of mass momentum and energy across the boundary, the user may provide a function, `convectiveFlux(args)`, that returns a table specifying the combined (convective and viscous) interface fluxes. The name `convectiveFlux` is configurable via the `funcName` parameter in the call to the boundary condition constructor `UserDefinedFluxBC:new{}`. (See page 59.)

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

species table of `nsp` species mass fluxes.

and the input `args` table contains:

t the current simulation time, in seconds

dt the current time-step size, in seconds

timeStep integral time step count

gridTimeLevel integral value indicating the phase of the update for a moving grid

flowTimeLevel integral value indicating the phase of the flow update

boundaryId integral value indicating the which boundary within the block we are working on

x,y,z coordinates of the midpoint of the interface, in m

csX,csY,csZ direction cosines for unit normal of the interface

csX1,csY1,csZ1 direction cosines for first tangent of the interface

csX1,csY1,csZ1 direction cosines for second tangent of the interface

i,j,k indices of the interior cell adjacent to the interface

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 components 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 on structured grids in Figure E.1. 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.

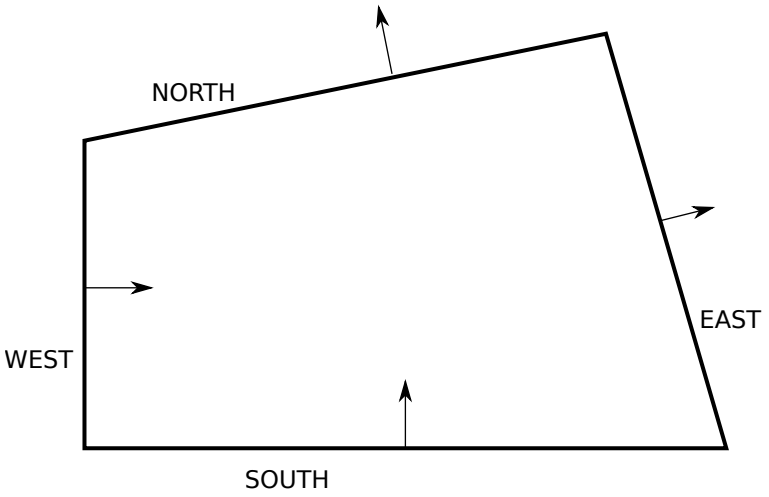


Figure E.1: The positive sense of direction for unit normals at each of the boundaries in 2D, using a structured-grid block.

The reason for this arrangement of face-normals in structured grids 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.

E.2 Source terms

User-defined source terms *add* to the internally-computed source terms. These terms specify the rate of addition of each quantity on a per-unit-volume basis. They can be applied to any/all conservation equations but, to activate the addition of these terms, you need to set the configuration options:

```
config.udf_source_terms = true
config.udf_source_terms_file = 'my_source_terms.lua'
```

Within the specified Lua file, you will need to provide a function `sourceTerms(t, cell)` that is called at each stage in the update over a time step, for every cell in the domain. Here `t` is the current simulation time and `cell` is a table containing the cell data, as returned by the `sampleFluidCell` function (See Section E.5.5). Added to that table are the items

blkId index of the block containing the cell

i,j,k indices of the cell. Note that these indices are in storage space, with `imin`≤`i`≤`imax`.

The table of values that your function returns may include the entries:

mass mass rate of addition per unit volume of the cell

momentum_x x-direction momentum flux per unit volume

momentum_y y-direction momentum flux per unit volume

momentum_z z-direction momentum flux per unit volume

total_energy rate of energy addition per unit volume

species table of `nsp` named species mass additions or a single value if `nspecies`==1.

energies table of `nmodes` energy fluxes.

omega rate of addition of ω for the $k-\omega$ turbulence model

tke rate of addition of turbulent kinetic energy

nuhat rate of addition of Spalart-Allmaras field variable

E.3 Grid motion

Grid motion is controlled through the movement of the vertices of the grid. The most flexible arrangement is that you supply the velocities for all vertices in the domain via the function `assignVtxVelocities(t, dt)`. Vertices with zero velocity may be skipped. To activate the grid-motion features, you need to set the following configuration options in your simulation input script:

```
content.config.gasdynamic_update_scheme = 'moving_grid_1_stage'
config.grid_motion = 'user_defined'
config.udf_grid_motion_file = 'grid-motion.lua'
```

For the `gasdynamic_update_scheme`, you could also choose the alternative stepper `'moving_grid_2_stage'` to get a second-order update scheme.

Within your assignVtxVelocities function, you may use the following functions for getting the current positions of individual vertices and setting their velocities. Your Lua function needs to supply velocities of the vertices, not positions.

pos = getVtxPosition(blkId, i, j, k) returns the position of the vertex as a table of named Cartesian coordinates [x, y, z]

x, y, z = getVtxPositionXYZ(blkId, i, j, k) returns the position of the vertex as three floating-point values

setVtxVelocity(vel, blkId, vtxId) sets the velocity vector for vertex `vtxId` in block `blkId`. The argument `vel` is to be provided as a `Vector3` object. This works for both structured and unstructured grids.

setVtxVelocityXYZ(velx, vely, velz, blkId, vtxId) sets the velocity components for vertex `vtxId` in block `blkId`. The velocity components are expected to be provided as floating-point values. This works for both structured and unstructured grids.

setVtxVelocity(vel, blkId, i, j) sets the velocity vector for vertex `i, j` in block `blkId` in a two-dimensional structured grid.

setVtxVelocityXYZ(velx, vely, velz, blkId, i, j) sets the velocity components for vertex `i, j` in block `blkId` in a two-dimensional structured grid.

setVtxVelocity(vel, blkId, i, j, k) set the velocity vector for vertex `i, j, k` in block `blkId` in a three-dimensional structured grid.

setVtxVelocityXYZ(velx, vely, velz, blkId, i, j, k) set the velocity components for vertex `i, j, k` in block `blkId` in a three-dimensional structured grid.

For setting the vertex velocities of whole blocks or whole domains at once, such as for rigid-body motion, there are functions for setting the velocities of all vertices in a block:

setVtxVelocitiesForDomain(vel) sets the one velocity vector for all vertices in all blocks in the domain.

setVtxVelocitiesForBlock(blkId, vel) sets the one velocity vector for all vertices in block `blkId`.

setVtxVelocitiesForBlockXYZ(blkId, velx, vely, velz) sets the one set of velocity components for all vertices in block `blkId`.

setVtxVelocitiesForRotatingBlock(blkId, omega) sets rotational speed `omega` (rad/s) for rotation about the z-axis.

setVtxVelocitiesForRotatingBlock(blkId, omega, point) sets rotational speed `omega` (rad/s) for rotation about the (0,0,1)-axis through `point`

setVtxVelocitiesByCorners(blkId, p0vel, p1vel, p2vel, p3vel) sets the velocity vectors for block with corner velocities specified by four corner velocities. This works for both 2D and 3D meshes. In 3D there is no variation in the k direction.

setVtxVelocitiesByCorners(blkId, p0vel, p1vel, p2vel, p3vel, p4vel, p5vel, p6vel, p7vel) sets the velocity vectors for block with corner velocities.

E.4 Coordination

If you provide a file name for `config.udf_supervisor_file` in your input script, the flow simulation program will call the `atTimestepStart(t, step, dt)` function that you define in that Lua file. This function differs from the boundary condition functions and source-term function because it called only once, at the start of the update process over the time step, and it does not pass any data back into the simulation. It can however be used for coordination of other user-defined functions by sampling flow field data, running external processes and writing files that the other Lua interpreters can read in the course of their activities.

After calling your `atTimestepStart` and on returning to the D-domain, there are a couple of house-keeping actions that may affect other parts of the simulation. The first is that a check is made for the Lua variable `dt_override`. If it is non-nil, its value will override the current timestep. This is useful for situations where you have made significant and sudden changes, such as rupturing the diaphragm in a shock-tube simulation, and it would help to immediately adjust the time step. Note that, on subsequent calls to your `atTimestepStart` function, you will need to explicitly set `dt_override=nil` to avoid the subsequent timesteps from being overridden. The second thing that is done is to copy the current values of the `userPad` array and propagate these values to all of the other active Lua interpreters, as discussed in Section E.5.1.

There are also points in the main simulation loop where the user-defined Lua functions `atTimestepEnd(t, step, dt)` and `atWriteToFile(t, step, dt)` are called, if they are defined in your supervisor script. The time of writing to file refers to the point at which the flow-solution data has been written to disk. This may happen several times during the course of a simulation and may be a good point to invoke an external program that needs current data for some sort of interaction with the flow field. Ingo Jahn's transient solid mechanics solver is a good example of a external program being coupled to the flow solver.

E.5 Helper variables, functions and modules

There are certain conveniences the code provides to your user-defined functions and these are set as global variables in your Lua interpreter. For example, the code sets up a reference to the D-language gas model for use from your Lua functions. It also provides some information about the block data, specific to the block where you apply the user-defined boundary condition.

E.5.1 Variables

If the `userPad` array is specified as having nonzero length in the job's input script, then the array will be set up to be available in the global name-space of each Lua interpreter. The array holds only floating-point numbers and the elements are indexed from 1, in the usual Lua convention. What you use them for is entirely up to you, however, the intent is to provide communication between the (possibly many) Lua interpreters without the need to resort to file writing and reading from within your Lua functions.

In the input script, you may set the length and some, or all, of the array values. Elements, for which an initial value is not provided, get a zero value. For example:

```
config.user_pad_length = 6
user_pad_data = {1.0, 3.14}
```

This data may be accessed from the Lua script as

```
x = userPad[1]
y = userPad[2]
```

Since the array is just a table in the Lua name-space, you may assign new values to the elements. At return from the `atTimestepStart` function call, the `userPad` content is copied back into the D-language domain and broadcast to all other Lua interpreters which may be used for setting source terms or boundary conditions. In an MPI simulation, the master task (rank 0) dominates and broadcasts its values to all other MPI tasks. On return from any other Lua function call, say for a boundary condition, the `userPad` data is not copied back into the D-language domain. On next occasion to call that Lua function, the master `userPad` is pushed again into the Lua interpreter's copy of `userPad`. This allows for a one-way synchronization of the data, which may change during the simulation.

If you want to have common data for many Lua interpreters and that data is to be unchanged during a simulation, it may be convenient to write a small Lua file containing some assignment statements and then use the Lua `dofile()` function. This will evaluate content of that file and bring those assignments into your interpreter.

There are a number of other globally-defined variables that contain configuration data for the simulation. These include:

blkId 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.

gmodel gas model object associated with current simulation
: This object provides complete access to the gas model services. You might need to get some gas properties or compute an equation of state. This object helps with that, and by using this object, you ensure consistency with the gas model that is internal to the simulation. A fuller description of the services provided by a gas model object is in the accompanying *Gas Model User's Guide* [5].

n_species number of species

n_modes number of energy storage modes (and temperatures)

Plus data describing the block that owns the current Lua interpreter. These data are listed in Section E.5.3

In the context of the Lua interpreter for a boundary condition, there are also the following variables:

boundaryId number indicating to which boundary the user-defined function is applied

boundaryLabel label given by user to the boundary (or set as default)

boundaryType type of boundary (as a string)
This is optionally set by user in input script or given default settings.

boundaryGroup group to which boundary belongs (also a string)

Note that the Lua interpreter that is started for your boundary condition is specific to that boundary attached to a particular block. Thus, each user-defined boundary condition within the simulation will have its own interpreter state, independent of all of the others. You may initialize several boundary conditions with the same Lua script so that they have common behaviour. To assist with the coordination of calculations across interpreters, the parameters `boundaryId`, `boundaryLabel`, `boundaryType` and `boundaryGroup` are potentially useful.

E.5.2 Functions

As well as the data available in the named variables, there are several functions that can be called to get more information about the flow at specific locations:

infoFluidBlock(blkId) returns a table of information related to the block with index `blkId`.

sampleFluidFace(faceType,blkId,i,j,k) returns a table of information about the geometric and flow properties at an interface in the grid. The argument `faceType` is a string. In structured grids, `faceType` is one of the strings `'i'`, `'j'`, or `'k'`. In unstructured grids, use `'u'` for `faceType`.

sampleFluidCell(blkId,i,j,k) returns a table of the flow state for a particular cell. The data supplied to the user is described below in Section E.5.5.

getRunTimeLoads(loadsGroup) returns two tables, one for the force components and one for the moment components for the specified `loadsGroup` name. The `loadsGroup` name is a string which matches one of the groups set up when the boundary conditions are constructed in the input script.

`blkId` refers to the block of interest. You should restrict your queries to blocks that are in the same memory space as the current block for your user-defined boundary condition. `i`, `j`, and `k` are the indices of the interface of interest. Although all values should be supplied, not all have meaning. For unstructured grids, only the `i` value

has meaning. You may supply a value of `0` for `j` and `k` because those indices are ignored. In 2D structured grids, the `k` has no meaning. Again, just supply the value `0`. In 3D structured grids, valid values are required for all of `i,j,k`. In an MPI-parallel simulation, be sure to sample only from blocks that are owned by the current MPI task. Sampling from other blocks will likely result in a segmentation fault.

E.5.3 Data returned by infoFluidBlock

dimensions number of spatial dimensions for this simulation

label string label, possibly identifying the block

grid_type string 'structured' or 'unstructured'

ncells count of interior cells in the block

nfaces count of interfaces defining cells in the block

nvertices count of vertices defining cells in the block

For structured-grid blocks, there are some additional item made available because they make sense only in the context of the structured-grid index directions. These are:

nicell number of cells in i-direction

njcell number of cells in j-direction

nkcell number of cells in k-direction

imin, imax range for i-index in storage space

jmin, jmax range for j-index in storage space

kmin, kmax range for k-index in storage space

north boundary index for the North boundary

east boundary index for the East boundary

south boundary index for the South boundary

west boundary index for the West boundary

top boundary index for the Top boundary (3D only)

bottom boundary index for the Bottom boundary (3D only)

E.5.4 Data returned by sampleFluidFace

The table returned by `sampleFluidCell` contains:

x,y,z coordinates of the centre of the cell, in m

area surface area of the face, in m^2. For an axsisymetric 2D simulation, this will be area per radian.

nx, ny, nz cosines for the unit normal to the face

t1x, t1y, t1z cosines for the first tangent to the face

t2x, t2y, t2z cosines for the second tangent to the face

Ybar y-coordinate of the centroid of face area for an axisymmetric 2D simulation

gvelx, gvely, gvelz velocity components of the face midpoint. Should be zero for a non-moving-grid simulation.

plus a description of the FlowState associated with the face. See the following section E.5.5.

E.5.5 Data returned by sampleFluidCell

The table returned by `sampleFluidCell` contains:

x,y,z coordinates of the centre of the cell, in m

vol volume of the cell, in m^3

plus a description of the FlowState associated with the cell:

p pressure, in Pa

T trans-rotational temperature, in K

T_modes array of other internal-mode temperatures, if `nmodes`> 0

u specific internal energy, J/kg

u_modes array of other internal energies

quality fraction of vapour for a multiphase gas model

massf table of named mass fractions

a sound speed, m/s

rho density, kg/m^3

mu dynamic viscosity, Pa.s

k thermal conductivity

k_modes array of conductivities of other energy modes

tke turbulent flow kinetic energy within the cell, J/kg

omega for the k-ω turbulence model, 1/s

mu_t turbulence viscosity

k_t turbulence conductivity

vel velocity components as a table with entries [x, y, z]

B magnetic field strength vector with components [x, y, z]

psi

divB divergence of magnetic field

Note that not all of these items will be relevant in all simulations. For example, `T_modes` and `u_modes` are not populated for simulations with a single-temperature gas.

E.5.6 Modules

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 documented in the gas model API [5].

Your Lua scripts may also use `Vector3` and `Matrix` objects to provide some 3D geometric calculation capabilities and some basic linear-algebra operations, respectively. For documentation on `Vector3` objects, see the companion report [8]. For the `Matrix` operations, see the source code module `bbla.d`.

E.6 Notes on Lua interpreters and global variables

For each boundary condition that uses a `UserDefinedBC` boundary condition, an independent Lua interpreter is started. The global state in each of these interpreters (read boundary conditions) is kept between time steps (*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, so do not make changes in one script and expect them to be reflected in another.

Further, the sampling functions that give information about blocks, cells and interfaces only work for the blocks in the memory of the computational process. The implication for MPI simulations is that you cannot get information about blocks and cells that are resident in a different computational process. You might get a segmentation fault if you try.

There is a run-time cost associated with the user-defined functions. There may be much data that need to be packed and unpacked on either-side of the Lua/D interface. If you find that implementing your customisation via user-defined functions make your simulation very slow, it might be time to think about an implementation in the core solver, in the D language. Repeated calculations and reading and writing of files from Lua may be other causes of a slow simulation. In such cases, consider computing or reading once and storing the values in a Lua table that will persist between calls of your customisation function.

E.6.1 Good Practice in User-Defined Lua Scripts

One can take advantage of the fact that the Lua state persists between time steps to minimise the number of computations required to return the required information. Additionally, naive use of the Lua interpreter can lead to problems, as the brief example below will demonstrate.

Here we have a simple user-defined ghost cell boundary condition. It fills the ghost cells at the specified boundary with constant velocity and temperature, with the pressure varying sinusoidally in time. The first version (page 133) shows an example of bad practice, while the second (page 134) shows how the same result can be achieved in a more efficient manner. The difference in these examples is not that great. It amounts to careful selection of where some reused objects are created.

In the first example, all the work is done within the `ghostCells()` function, with some delegated to `getFlowTable`, but all internal to `ghostCells()`. This is called multiple times per timestep (depending on number of update stages) at every interface on a user-defined boundary to fill in the adjacent ghost cells. So, in this first example, the unit conversions, gas calculations and other manipulations to attain the velocity and temperature are performed at every update stage for every interface along the boundary. If this were the only problem with this approach, it would not be a big issue: your calculation would simply take longer to run.

The second problem is more subtle, and relates to the way the Lua interpreter interacts with the D program underneath. The problem in this case is specifically with the `GasState` and `GasModel` objects. Whenever a `GasState` or `GasModel` object is created in the Lua interpreter, an accompanying object is created in the D environment beneath.[1] Unfortunately, these objects are deliberately retained in the D run-time environment because we cannot easily communicate the lifetime of a Lua object to the matching D object. As such, these D objects can and will accumulate over time. This will lead to increasing memory usage over the lifetime of the simulation. If you are particularly unlucky, this can lead to locking up of your local machine as your RAM fills up or the dumping of jobs on HPC facilities due to exceeding memory limits.

To address these issues, all the calculations which do not depend on the local simulation status or ghost cell information can be placed in the beginnings of the Lua file. These calculations will be performed on the initialisation of the Lua state, with the results being reused each time the `ghostCells` function is called. The only bit

[1] This is also true for other objects defined in the D language, such as `Vector3` objects or `FlowState` objects.

of work being done in `ghostCells` is calculating the pressure, which depends on the current simulation time. This is a value that cannot be calculated beforehand at initialisation.

User-Defined BC Script- Bad Practice (.lua)

```
-- An example of a script that will fill in the
-- ghost cells with a user-defined flow condition.
-- The flow condition will have constant velocity
-- and temperature, and a sinusoidally varying pressure.
function ghostCells(args)
    -- Use a secondary function to define the flowstate
    flowTable = {}
    getFlowTable(flowTable)

    -- Initialize the tables that will contain the flow data
    ghostCell1 = {}; ghostCell2 = {};

    -- Grab the base flow data from the flowTable
    ghostCell1.velx = flowTable.vel
    ghostCell1.vely = 0
    ghostCell1.T = flowTable.T
    ghostCell2.velx = flowTable.vel
    ghostCell2.vely = 0
    ghostCell2.T = flowTable.T

    -- Define the sinusoidally varying pressure-
    -- a constant pressure plus a disturbance
    -- with frequency of 200kHz and amplitude
    -- 1 millionth of the freestream pressure
    freq = 200e3 * 2 * math.pi

    ghostCell1.p = flowTable.p * (1 + 1e-6 * math.cos(freq * args.t))
    ghostCell2.p = flowTable.p * (1 + 1e-6 * math.cos(freq * args.t))

    return ghostCell1, ghostCell2
end

-- A function that will return the desired
-- freestream values as a table
function getFlowTable(tab)
    -- Define the gas state used to calculate the sound speed
    gamma = 1.4
    gmodel = GasModel:new{'ideal-air.lua'}
    gstate = GasState:new{gmodel}

    -- Define the freestream
    tab.M = 7.99
    tab.p = 0.060 * 6894.76
    tab.T = (1250 * 0.55556) / (1 + (gamma - 1) * tab.M^2 / 2)
    gstate.p = tab.p; gstate.T = tab.T

    -- Update the gas state and use the
    -- sound speed to get the velocity.
    gmodel:updateThermoFromPT(gstate)
    tab.vel = tab.M * gstate.a

    return tab
end
```

User-Defined BC Script- Good Practice (.lua)

```
1
2 -- An example of a script that will fill in the
3 -- ghost cells with a user-defined flow condition.
4 -- The flow condition will have constant velocity
5 -- and temperature, and a sinusoidally varying pressure.
6
7 -- The freestream only needs to be defined once in the Lua state
8 gamma = 1.4
9 gmodel = GasModel:new{'ideal-air.lua'}
10 gstate = GasState:new{gmodel}
11
12 -- Define the freestream
13 flowTable = {}
14 flowTable.M = 7.99
15 flowTable.p = 0.060 * 6894.76
16 flowTable.T = (1250 * 0.55556) / (1 + (gamma - 1) * flowTable.M^2 / 2)
17 gstate.p = flowTable.p; gstate.T = flowTable.T
18
19 -- Update the gas state and use the sound speed to get the velocity.
20 gmodel:updateThermoFromPT(gstate)
21 flowTable.vel = flowTable.M * gstate.a
22
23 -- Initialize the tables that will contain the flow data
24 ghostCell1 = {}; ghostCell2 = {};
25
26 -- Grab the base flow data from the flowTable
27 ghostCell1.velx = flowTable.vel
28 ghostCell1.vely = 0
29 ghostCell1.T = flowTable.T
30 ghostCell2.velx = flowTable.vel
31 ghostCell2.vely = 0
32 ghostCell2.T = flowTable.T
33
34 -- Define the frequency of the perturbation (in radians)
35 freq = 200e3 * 2 * math.pi
36
37 -- Only calculations that rely on the cell-local
38 -- properties (i.e. things in the args table) should
39 -- go inside the function called by the boundary condition
40
41 function ghostCells(args)
42
43     -- Define the sinusoidally varying pressure-
44     -- a constant pressure plus a small disturbance of
45     -- frequency 200kHz and amplitude 1 millionth of the freestream pressure
46     ghostCell1.p = flowTable.p * (1 + 1e-6 * math.cos(freq * args.t))
47     ghostCell2.p = flowTable.p * (1 + 1e-6 * math.cos(freq * args.t))
48
49     return ghostCell1, ghostCell2
50 end
```

Index