A Tutorial Guide to the Simulation of Hypersonic Flows with `Eilmer`

Rowan J. Gollan, Kyle A. Damm, Nicholas N. Gibbons,
Lachlan S. Whyborn, Robert G. Watt and Peter A. Jacobs.
The University of Queensland

20 March 2024

Contents

1 Introduction

`Eilmer` is a multi-physics simulation tool that has been developed with a special emphasis on the modelling of hypersonic flows and associated physics. The core of the code is built around solving the governing equations for compressible flows using a finite-volume discretisation. The simulation tool provides a complete simulation work flow — grid generation and pre-processing, flow simulation, and post-processing — that we find useful when working with relatively simple geometric configurations. `Eilmer` also has interoperability with 3rd-party tools for grid generation, mesh partitioning and visualisation, which we make use of in larger scale and more complex geometric configurations. This tutorial has a focus on the "batteries-included" tip-to-tail workflow provided in `Eilmer` to simulate hypersonic flows.

`Eilmer` can be used to solve the governing equations for Euler (inviscid), Navier-Stokes (viscous), Reynolds-Averaged Navier-Stokes (RANS), and magneto-hydrodynamic flows. These can be solved in two dimensions (planar and axisymmetric) and three dimensions on multiple-block body-fitted grids, structured or unstructured. There are two major iteration modes that can be selected depending on the dynamics of the flow field: a transient mode for time-accurate simulations; and a steady mode for acceleration of the flow solution towards steady state. A range of thermochemical modelling options are available that span ideal gases, chemical nonequilibrium, multiple-temperature thermal nonequilibrium and an emerging capability to simulate state-specific gas behaviour. In `Eilmer`, turbulence can be modelled with the k-ω model (Wilcox (2002)), the Spalart-Allmaras model (Allmaras et al. (2012)) and a recently-implemented Delayed Detached Eddy Simulation technique (Shur et al. (2008)). `Eilmer` provides a simulation capability for the coupling between the interaction of gases and structures: specifically, fluid-thermal coupling (as shown in Figure 1) and, separately, fluid-structural coupling. The latter is enabled with a moving grid technology. In a separate use of moving grids, shock-fitting can be applied to external flows with a detached bow shock. This is a sample of the capabilities that have been developed in `Eilmer` and a fuller discussion is available in Gibbons et al. (2023). The open-source nature of the code has led to involvement in very specialised *ad hoc* modelling as the need arises.

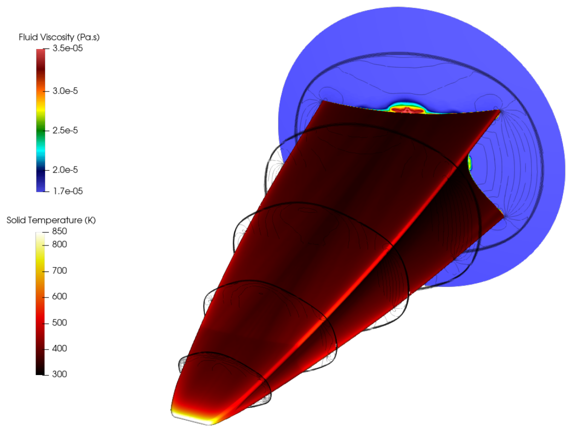


Figure 1: BoLT-2 fluid-thermal solution. Source: Damm et al. (2023)

With these features and capabilities, `Eilmer` has been used in a large variety of simulation work. As research engineers, we tend to categorise the simulation work along a continuum from applied engineering calculations to fundamental physics investigations. At the engineering end, a small selection of example applications include: the simulation of impulse facilities for the production of hypersonic test flows (Jacobs (1994), Goozée et al. (2006), Gildfind et al. (2018)); flight regime assessment of electron transpiration cooling for hypersonic leading edges (Gibbons et al. (2021)); and optimised aerodynamic design of hypersonic devices (Damm (2020) and Damm et al. (2020)). For the investigation of fundamental flow physics, `Eilmer` has been used to characterise steady and unsteady flow behaviours in the supersonic/hypersonic flow over double cones (Hornung et al. (2021)), and to investigate flow instabilities in hypersonic flow over blunted cones (Whyborn (2023) and Whyborn et al. (2023)).

We refer the reader to our paper (Gibbons et al. (2023)) for a fuller discussion on the code, our development process, capabilities of the solver and examples of application. Also, the website `https://gdtk.uqcloud.net` has more extensive documentation, including user guides in PDF and online reference manuals, for those occasions when you have forgotten some detail.

1.1 A brief history

Today `Eilmer` is at a version 5 release. That does not necessarily imply that earlier versions were named as formally or started counting from 1. Instead `Eilmer` has grown and evolved over the past 30+ years as development tools, development practices and computer architectures have evolved. We provide a brief history here to provide context for where we are today with the project, and why we have made certain decisions about its direction.

The origins of `Eilmer` can be traced to its earliest ancestor, `cns4u` (Jacobs (1991)), which was a compressible Navier-Stokes time integrator written in the C language. The choice of C at that time, in 1990, was considered risky by those in the numerical computation world; Fortran dominated as implementation language in the successful projects (and not so successful projects — it just dominated that world). `cns4u` could handle a single structured-grid block in 2D. A multi-block capability would be required to exploit domain decomposition on parallel architectures and to allow for more complicated body-fitted grids requiring more than a single block.[1] `mb_cns` (multi-block compressible Navier-Stokes) was developed by the mid 1990s to address that need (Jacobs (2004)). In essence, it was a multiple-block version of `cns4u`, written in C and used the message passing interface (MPI) for distributed memory parallelism on the high-performance computers of the day. In the 2000s, the codebase branched into parallel lines of development before ultimately merging as `Eilmer`: `mbcns2` was mostly written in C++, motivated by using object-oriented technology to address the complexity in gas modelling for hypersonic flows, and the variety of geometric entities in our geometry engine; meanwhile, `Elmer` was our first code for the 3D geometries, primarily motivated to do large eddy simulation work where the simulation of turbulent structures required the freedom of three dimensions to

[1]Today, we still have multi-block capability for the same reasons, but we have decoupled the concerns of block topology for geometry representation and domain decomposition for parallel performance.

develop in a physically realistic manner. `Elmer` was the first generation of code that was not named via acronym. We also added a silent 'i' to its spelling later on to avoid collision with a Finnish code of the same name (`Elmer`) based on the finite-element method.

In September 2007, the project focus was to bring `mbcns2` and `Elmer2` into a single codebase. By November 2008, `Eilmer3` had arrived: a 2D/3D multiple-block on structured grids flow solver with a C++ core and user interface in Python. We developed that code for at least a half decade (Gollan and Jacobs (2013)) and our users got use for more than a decade. However, as mechanical and aerospace engineers first and software developers second, we were not happy with the complexity of the C++ language at that time. For example, we were finding that the apprenticeship for grad students to write safe and performant code was becoming a significant chunk of the 3.5 year PhD thesis in Australia. We had gone through the C++0x years where the release of the standard was given an 'x' because they were unsure when they would agree.[2] Our dissatisfaction with C++ motivated our experiments with the D programming language.

In 2014, we began some preliminary experiments with D: could we write the geometry and gas models in D and maintain our productivity as developers and have code performance similar to our C++ codes? We did have to have our cake and eat it too, otherwise the development cost to change to D would not be worth it. Our early experiments seemed promising so we poured more effort into the D language rewrite of `Eilmer`. The new code, written in D with a Lua interface, became known as `Eilmer4`. By the end of 2015, all development focus was on `Eilmer4` and we had retired the development on `Eilmer3` to maintenance only, no new feature additions. We presented an experience report of the D language for numerical simulation tool building at the :w 2nd Australasian Conference on Computational Mechanics in November 2015 (Jacobs and Gollan (2016)).

Today, we are at version 5 of `Eilmer`. This comes about a decade after the beginnings of `Eilmer4`. There are no new surprises in the core code of version 5 compared to version 4; the number-crunching code is in D, the user interface remains in Lua. The biggest change in version 5 is to user-facing configuration, a workflow that separates grid preparation from flow description, and some layout of folders and files generated by `Eilmer`. The biggest motivation for building a version 5 of `Eilmer` was to formalise the distinctions between the transient solver mode and the steady-state mode. This has been formalised at both the user interaction and internally in the code.

1.2 Overview for this tutorial

The goal of this tutorial is to introduce how to use `Eilmer` so that you are ready to get started using `Eilmer` for your own simulations of hypersonic flow. Our approach towards that goal is to provide a set of examples that show different aspects of `Eilmer` usage.

In the interest of conciseness, we have made some assumptions about the reader of this tutorial. We have assumed a background in fluid mechanics, familiarity with computational fluid dynamics (CFD), and some exposure to hypersonics. That explains what we do cover and what we do not cover in this document. We aim to cover how to use `Eilmer`

[2]C++0x later became C++11, once settled.

for relatively simple hypersonic configurations. We do not intend to cover the methods of best practice CFD nor give detailed discussion about physical modelling choices in hypersonic flows. That latter topic is not settled and opinions vary. If the explanations are too brief or you find them hard to follow, remember that we also have gentler longer-form documentation available on the web and as PDF.

Here is what you will find in this tutorial document. We use Section 2 to help the reader install `Eilmer` so they are ready to simulate! Section 3 is the core of the tutorial. A series of examples are provided. They have been sequenced to go from basic to more advanced in terms of complexity, with the dimensions to complexity being geometric, physical modelling, numerical settings and data extraction/post-processing.

2 Getting started with `Eilmer`

First, let's talk about operating systems. `Eilmer` is principally developed on linux for linux. All the high-performance computers we have access to run a linux operating system, so we need to develop code for that environment. At the laptop/desktop scale, `Eilmer` has been successfully installed on modern versions of linux (obviously), macOS and Windows (via version 2 of windows-subsystem-for-linux, WSL2). We will give instructions for setup and install for linux and macOS. For Windows, first install a linux system into WSL2 and then follow the linux install instructions. `Eilmer` use on Windows is within the WSL2 environment.

2.1 Preparing your environment

Several pieces of supporting software are required to build and install `Eilmer`. We have tried to keep the number of dependencies small. It helps us maintain a nomadic existence with installing and running the code in many places. The required supporting software is listed in Table 1. In Table 1, we have given the package names for some of the more commonly encountered systems. On linux, we recommend using your package manner to install the packages (but there are one or two exceptions noted). On macOS, the package manager to use is Homebrew.

2.1.1 Installing the LLVM D compiler on linux

One of these exceptions is the LLVM D compiler, which should be installed using the latest stable release from the LLVM website, rather than using a packaged-maintained version. Our experience is these versions are often quite old, and frequently cause failures and problems when trying to build our new-ish code. Installing the D compiler can be a tricky step for some who have not installed something manually before. The steps are: download, unpack, and point your `PATH` variable to find the compiler executables.

Here are some steps that you can use at the command line to install the D compiler in an `opt/` area under your `HOME` directory. If you find this too terse, please look at our guide at `https://gdtk.uqcloud.net/docs/getting-started/install-d-compiler/`.

Table 1: Prerequisite software for `Eilmer` and this tutorial

Software	linux, Debian-family	linux, RedHat-family[1]	macOS
build environment	`build-essential`	"C Development Tools and Libraries"	`xcode`
LLVM D compiler	*— on linux, download latest is recommended —*		`ldc` `dub`
Fortran compiler	`gfortran` `gfortran-multilib`	`gcc-gfortran`	`gcc`
git	`git`	`git`	`git`
readline	`libreadline-dev`	`readline-devel`	`readline`
ncurses	`libncurses5-dev`	`ncurses-devel`	`ncurses`
OpenMPI	`libopenmpi-dev`	`openmpi-devel`	`open-mpi`
plotutils	`libplot-dev`	`plotutils-devel`	`plotutils`
Paraview	*— on linux, download latest is recommended —*		`paraview` (as cask)
gnuplot	`gnuplot`	`gnuplot`	`gnuplot`
Python	*— on linux, system default is fine —*		`python`
Pandas	`python-pandas`	`python-pandas`	`pandas` (via pip3)
matplotlib	`python-matplotlib`	`python-matplotlib`	`matplotlib` (via pip3)
sed	*— on linux, system default is fine —*		`gnu-sed`[2]

[1] Quoted packages are via `dnf group install`.
[2] After install, type `brew info gnu-sed` to get hints about setting your `PATH`.

```
$ cd
$ mkdir opt
$ cd opt
$ wget https://github.com/ldc-developers/ldc/releases/download/v1.37.0/ldc2-1.37.0-linux-x86_64.tar.xz
$ tar -Jxvf ldc2-1.37.0-linux-x86_64.tar.xz
```

After that, you should add `$HOME/opt/ldc2-1.37.0-linux-x86_64/bin` to your `PATH`. We explain more about setting the `PATH` variable in Section 2.3. For the moment, you could set your `PATH` in your current session with:

```
$ export PATH=$HOME/opt/ldc2-1.37.0-linux-x86_64/bin:$PATH
```

2.2 Downloading, building and installing

Assuming you have the prerequisite software ready to go, we can proceed with download and install of `Eilmer`. Let's download the complete source code from github:

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

Now, we change to `Eilmer` source, then build and install:

```
$ cd gdtk/src/lmr
$ make install
```

2.3 Setting environment variables

There are certain environment variables that require setting for running `Eilmer`. So we do not need to type these at the start of every session, it is convenient to place these in a file that is read at the start of a login session. Typical files to place these in are `.bash_aliases` on Ubuntu, `.bashrc` on Fedora, and `.zshrc` on macOS. It really just depends on what shell you are using. The required environment variables with typical settings are:

```
export DGD_REPO=${HOME}/gdtk
export DGD=${HOME}/gdtkinst
export PATH=${PATH}:${DGD}/bin:${HOME}/opt/ldc2-1.37.0-linux-x86_64/bin
export DGD_LUA_PATH=${DGD}/lib/?.lua
export DGD_LUA_CPATH=${DGD}/lib/?.so
```

3 Simulation examples

In this tutorial, we present three examples as a means to teach the use of `Eilmer`. The examples are chosen to demonstrate various aspects of the modelling available in `Eilmer` and they are sequenced to increase in complexity. The first example shows how to simulate supersonic flow over a cone. It introduces the simulation workflow, the basics of geometry and grid generation, how to run a simulation, and how to post-process to generate VTK files for visualisation in Paraview. In the second example, we look at the viscous flow over a convex ramp. Here the geometry is more complicated than the cone, the air is modelled with two temperatures, and we show how to use MPI-type parallelism in steady solver mode. For post-processing, we discuss how the loads on the ramp can be extracted and plotted. The third and final example demonstrates the high-temperature gas modelling available in `Eilmer` for aeroshell simulations. This example simulates multi-temperature reacting flow over a sphere fired into air. We import a grid generated by an external tool (`GridPro`) and show the use of shock-fitting, so that we get excellent grid tailoring of the structured grid and the shock.

3.1 Workflow and the command-line interface

In broad steps, the workflow for an `Eilmer` simulation is:

1. Pre-processing which includes gas model preparation, grid preparation, and configuring a flow field description (boundary conditions and initial conditions);
2. Running the simulation; and
3. Post-processing which may include visualisation, extraction of slice or streamline data, extraction of history data or extraction of loads data.

Input for `Eilmer` is via text files. Those text files are in the Lua programming language. This gives us a programmable environment for our input for free. [3]

[3] Free for the user. There is backend developer cost to maintain the user input interface, but at least we aren't trying to maintain a programming language as well!

`Eilmer` is a command-line driven program: commands are used to enact the workflow just described. The command-line interface (CLI) in `Eilmer` 5 is of the form "*noun␣action␣options/arguments*". Not all commands require options or arguments. The *noun* is `lmr` to invoke the `Eilmer` 5 program. You may be familiar with this type of CLI from other tools such as `git`, `mercurial` and the package managers `apt`, `dnf` and `brew`.

Let's look at an example for the `Eilmer prep-grid` command. In its simplest form, we can invoke that command using:

```
$ lmr prep-grid
```

With no options provided, this command will work with the default input file `job.lua`. Had we prepared a different grid input file called `grid.lua`, then the command would be:

```
$ lmr prep-grid --job=grid.lua
```

In this case, we were explicit with the job script name because we used a non-default.

Help is available using the `help` command. It will display a list of commonly used commands.

```
$ lmr help
```

You can also get a list of all available commands by passing the "`-a`" option:

```
$ lmr help -a
```

Help for a specific command can be requested by passing the command name[4] as argument:

```
$ lmr help prep-grid
```

In the interests of reproducible research, we provide `revision-id` and `version` commands so that users have a convenient way to record which repository revision of `Eilmer` they are using for their production simulations, and which compiled version options they had enabled. For this tutorial, we used:

```
$ lmr revision-id
7975e97c
$ lmr version
Eilmer 5.0 compressible-flow simulation code.
Revision-id: 7975e97c
Revision-date: Wed Mar 20 20:16:10 2024 +1000
Compiler-name: ldc2
Build-date: Wed 20 Mar 2024 20:18:09 AEST
Build-flavour: fast
Profiling: omitted
```

We will show more usage of the commands in the tutorial examples.

[4]If you were to type `lmr help help` the whole system might descend into a recursive nightmare.

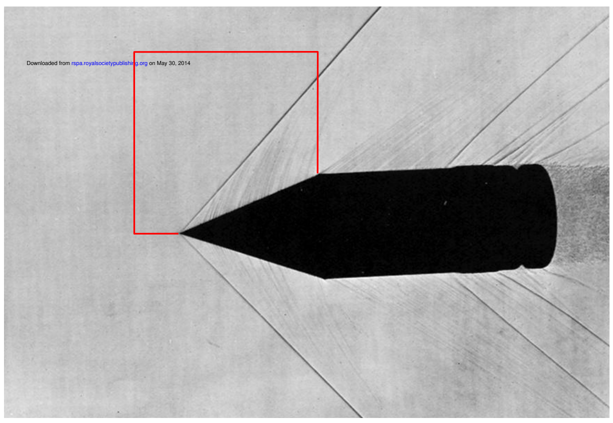
3.2 Supersonic inviscid flow over a sharp cone

Files for this example are located in
`gdtk/examples/lmr/2D/sharp-cone-20-degrees/sg-minimal`.

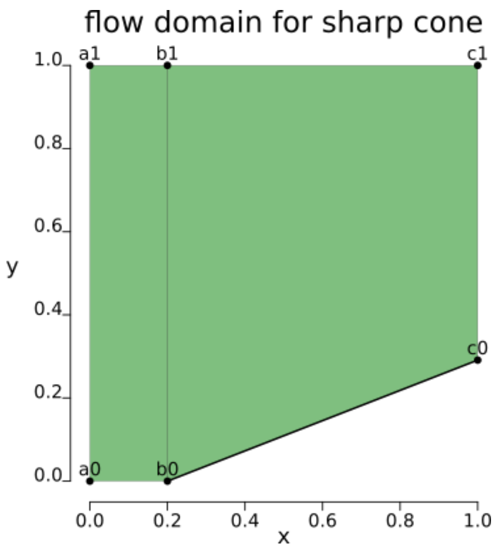
We will teach you how to simulate the Mach 1.576 flow of ideal air over a sharp cone with a half-angle of 20° using `Eilmer`. This example has long been used as a starting point for new users. In this example, you will learn how to:

- prepare a simple gas model for ideal air;
- generate a two-block domain with structured grids;
- configure flow solver settings to run the transient solver;
- run a simulation; and
- post-process the results to generate VTK files for viewing.

Our simulation goal is shown in Figure 2(a). We wish to simulate the conditions from Maccoll (1937) in which a shadowgraph was taken of a 2-pound projectile in supersonic flight. In Figure 2(a), we have suggested a computational domain shown in red. Figure 2(b) shows the two-block computational domain that corresponds to the red bounded region in Figure 2(a).


Downloaded from rspa.royalsocietypublishing.org on May 30, 2014

(a) Fig. 3 from Maccoll (1937)



(b) Simulation domain for `Eilmer`.

Figure 2: (a): 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. (b): Schematic diagram of the simulation domain corresponding to red-bounded region.

3.2.1 Pre-processing

Create a new directory where you would like to work for this example. In this directory, we will prepare a gas model, grid and flow solver description. For each of those steps, we will create a text file using your favourite editor and process that file with the appropriate `lmr` command.

WARNING: *When creating files, it will be tempting to copy and paste from this document. Be aware that PDF characters may not translate into a plain text editor as expected. Classic examples for poor translation are quote characters and ligatures. This will cause errors when the `lmr` commands try to process your files. It is recommended you type these by hand to get a feel for the syntax, or copy the source directly from the `examples/` area.*

Our gas model is very simple. It is ideal air. The gas model file is correspondingly simple. Create a filed called `ideal-air.lua` and place in it:

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

Once created, we are ready to process that file for use by `Eilmer`. That command is:

```
$ lmr prep-gas -i ideal-air.lua -o ideal-air.gas
```

Unsurprisingly, `-i` indicates the input file; `-o` indicates the output file. That output file, `ideal-air.gas`, is what is used by `Eilmer`. You may inspect it; it is just plain text. It contains full information about air properties which has been pulled from our species database. You may ask why this gas model preparation is a separate process. It seems like something trivial that could be captured elsewhere in the input. `Eilmer` provides more complicated gas models for hypersonic flow modelling that are served best by a stand-alone preparation process. We do that process too on the simplest of models to provide a consistent pattern of user interaction.

Next, we prepare the grids. The following is the text from a file we prepared called `grid.lua`. As you inspect the file, try to line the constructs up with the schematic in Figure 2(b).

```
-- grid.lua
print("Set up geometry and grid for a Mach 1.5 flow over a 20 degree cone.")
--
-- 1. Geometry
a0 = {x=0.0, y=0.0};     a1 = {x=0.0, y=1.0}
b0 = {x=0.2, y=0.0};     b1 = {x=0.2, y=1.0}
c0 = {x=1.0, y=0.29118}; c1 = {x=1.0, y=1.0}
--
quad0 = CoonsPatch:new{p00=a0, p10=b0, p11=b1, p01=a1}
quad1 = AOPatch:new{p00=b0, p10=c0, p11=c1, p01=b1}
--
-- 2. Grids
grid0 = registerFluidGrid{
   grid=StructuredGrid:new{psurface=quad0, niv=11, njv=41},
   fsTag="inflow",
   bcTags={west="inflow"}
}
grid1 = registerFluidGrid{
   grid=StructuredGrid:new{psurface=quad1, niv=31, njv=41},
   fsTag="initial",
   bcTags={east="outflow"}
}
identifyGridConnections()
```

The key steps are:

1. Define some construction points.
2. Use the construction points to define patches.
3. Assemble grids from patches, giving the discretisation in `i` and `j` directions as numbers of vertices.

In `Eilmer`, we treat a grid file like most other CFD programs: it is just a series of points that can be interpreted, in our case, as the corners of finite-volume cells. This means the grid itself has no information of its relation to the flow domain. To bridge this disconnect, we require that the user set some information about the grid that can be used later on when preparing the flow field. In this example, you will see that we set an `fsTag` to a string label that will later define the initial flow state in that grid. We actually use a different initial condition in the two grids: `'inflow'` in `grid0`; and `'initial'` in `grid1`. We define what those labels mean in terms of flow state later on when preparing the flow field description. When registering the grids, we also set boundary information on the west boundary of `grid0` as an `'inflow'` via the `bcTags`, and `grid1` gets an `'outflow'` set on its east boundary. The remaining unset boundaries will receive a default boundary condition (`WallBC_WithSlip`) when we prepare the simulation in a subsequent step.

We are ready to prepare the grid. Issue the following at the command line:

```
$ lmr prep-grid --job=grid.lua
```

On your screen, you should see output like:

```
Set up geometry and grid for a Mach 1.5 flow over a 20 degree cone.
  #connections: 1
  #grids 2
  #gridArrays 0
```

On successful completion, `Eilmer` has created a subdirectory called `lmrsim`. This directory will contain (almost) all files generated by an `lmr` command or process as we move through the workflow. Let's take a look at what folders and files are produced by `prep-grid` with the `tree` command.

```
$ tree lmrsim
lmrsim
`-- grid
    |-- grid-0000.gz
    |-- grid-0000.metadata
    |-- grid-0001.gz
    |-- grid-0001.metadata
    `-- grid.metadata
2 directories, 5 files
```

The final step in the pre-processing stage is to prepare a flow field description and define the numerical settings for the simulation. We will focus on a simulation using the transient solver in this example. The following is the text we prepared in a file called `transient.lua`. The file itself is commented (`--` in Lua), so hopefully that provides some explanation.

```
 1 -- transient.lua
 2 print("Set up transient solve of Mach 1.5 flow over a 20 degree cone.")
 3 --
 4 -- 0. Assume that a previous processing has step set up the grids.
 5 --
 6 -- 1. Domain type, gas model and flow states
 7 config.solver_mode = "transient"
 8 config.axisymmetric = true
 9 setGasModel('ideal-air.gas')
10 initial = FlowState:new{p=5955.0, T=304.0} -- Pa, degrees K
11 inflow = FlowState:new{p=95.84e3, T=1103.0, velx=1000.0}
12 flowDict = {initial=initial, inflow=inflow}
13 --
14 -- 2. Fluid blocks, with initial flow states and boundary conditions.
15 -- Block boundaries that are not otherwise assigned a boundary condition
16 -- are initialized as WallBC_WithSlip.
17 bcDict = {
18    inflow=InFlowBC_Supersonic:new{flowState=inflow},
19    outflow=OutFlowBC_Simple:new{}
20 }
21 --
22 makeFluidBlocks(bcDict, flowDict)
23 --
24 -- 3. Simulation parameters.
25 config.max_time = 5.0e-3  -- seconds
26 config.max_step = 3000
27 config.dt_plot = 1.5e-3
28 config.extrema_clipping = false
```

Here are some things to note about the `transient.lua` file. The gas model file we prepared earlier `ideal-air.gas` now makes an appearance when setting the gas model on line 9. The `fsTags` introduced as strings in the `grid.lua` file are now defined as `FlowStates` on lines 10 and 11. These are packed into a table called `flowDict` for later use. We also create a `bcDict` table on lines 17–20. This maps out `bcTags` in `grid.lua` to specific boundary condition objects. There is an important and powerful function call on line 22: the `makeFluidBlocks()` function is used to create blocks on our grids and define boundary conditions and initial conditions. The last part of the `transient.lua` file is used to configure some simulation parameters.

We use the `prep-sim` command as the final step in the pre-processing stage:

```
$ lmr prep-sim --job=transient.lua
```

Here is what should appear on your screen:

```
Read Grid Metadata.
  #connections: 1
  #grids: 2
Set up transient solve of Mach 1.5 flow over a 20 degree cone.
Build runtime config files.
Build fluid files.
```

Finally, let's look at the state of folders and files on disk at the end of a successful pre-processing stage:

```
$ tree lmrsim
lmrsim
|-- blocks.list
|-- config
|-- control
|-- fluidBlockArrays
|-- grid
|   |-- grid-0000.gz
|   |-- grid-0000.metadata
|   |-- grid-0001.gz
|   |-- grid-0001.metadata
|   `-- grid.metadata
|-- mpimap
`-- snapshots
    |-- 0000
    |   |-- fluid-0000.gz
    |   |-- fluid-0001.gz
    |   |-- grid-0000.gz
    |   `-- grid-0001.gz
    `-- fluid.metadata
4 directories, 15 files
```

We are ready to run our first simulation!

3.2.2 Running a simulation

Let's just do it and then talk about it.

```
$ lmr run
```

An abbreviated version of what appears on screen is:

```
Eilmer simulation code.
Revision-id: 7975e97c
Revision-date: Wed Mar 20 20:16:10 2024 +1000
Compiler-name: ldc2
Parallel-flavour: shared
Number-type: real
Build-flavour: debug
Build-date: Wed 20 Mar 2024 20:18:09 AEST
Heap memory used: 13 MB, unused: 9 MB, total: 22 MB (22-22 MB per task)
Step=     20 t= 1.201e-04 dt= 6.003e-06 cfl=0.50 WC=0.1 WCtFT=6.0 WCtMS=22.1
Step=     40 t= 2.401e-04 dt= 6.003e-06 cfl=0.50 WC=0.3 WCtFT=5.1 WCtMS=19.2
  ...
  ...
Step=    720 t= 4.322e-03 dt= 6.003e-06 cfl=0.50 WC=5.6 WCtFT=0.9 WCtMS=17.8
Step=    740 t= 4.442e-03 dt= 6.003e-06 cfl=0.50 WC=5.7 WCtFT=0.7 WCtMS=17.5
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++
+   Writing snapshot at step =    750; t = 4.502e-03 s +
++++++++++++++++++++++++++++++++++++++++++++++++++++
Step=    760 t= 4.562e-03 dt= 6.003e-06 cfl=0.50 WC=5.9 WCtFT=0.6 WCtMS=17.3
Step=    780 t= 4.682e-03 dt= 6.003e-06 cfl=0.50 WC=6.0 WCtFT=0.4 WCtMS=17.0
Step=    800 t= 4.802e-03 dt= 6.003e-06 cfl=0.50 WC=6.1 WCtFT=0.3 WCtMS=16.8
Step=    820 t= 4.922e-03 dt= 6.003e-06 cfl=0.50 WC=6.2 WCtFT=0.1 WCtMS=16.5
STOP-REASON: Reached target simulation time of 0.005 seconds.
FINAL-STEP: 833
FINAL-TIME: 0.00500048
++++++++++++++++++++++++++++++++++++++++++++++++++++
+   Writing snapshot at step =    833; t = 5.000e-03 s +
++++++++++++++++++++++++++++++++++++++++++++++++++++
```

What do we notice in the output? The simulation finished by taking 833 steps and got to a simulated time of 5 ms. That end time corresponds to our request `config.max_time = 5.0e-3`. What happened to our request for 3000 steps (`config.max_step = 3000`)? Seems it was ignored. Well, the stopping criteria will look for maximum steps or maximum time and stop on whichever comes first.

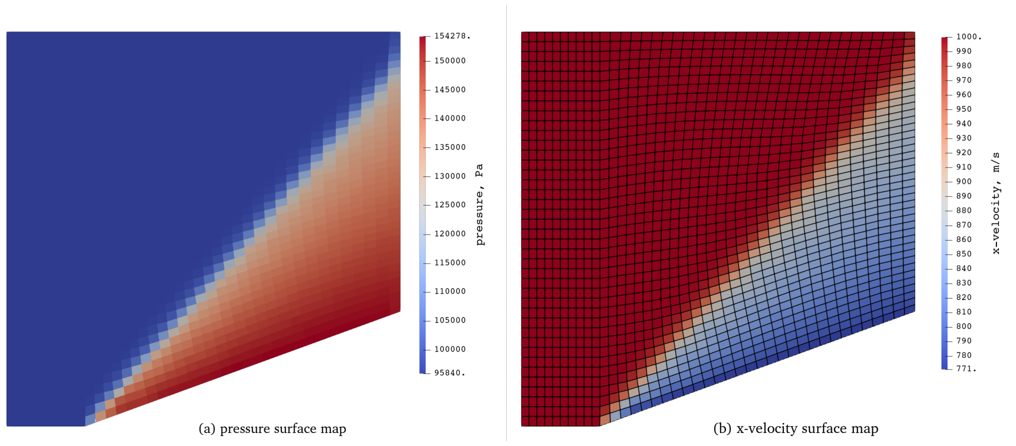
Another thing to note is that a new snapshot of the flow field was produced at each 1.5 ms (approximately). This corresponds to our request for `config.dt_plot = 1.5e-3`.

3.2.3 Post-processing to produce VTK files

If our simulation completed successfully, there should be five snapshots in the `lmrsim/snapshots` area. There is the initial condition (`0000`) plus four more snapshots produced during the simulation. We can convert all of these snapshots into VTK files with a simple command:

```
$ lmr snapshot2vtk --all
```

When that command concludes, there is a new folder: `lmrsim/vtk`. In that folder, you can pick up the `fluid.pvd` file in Paraview to do some visualisation. Figure 3 shows surface plots coloured by (a) pressure and (b) velocity in x-direction (with grid overlayed).



(a) pressure surface map (b) x-velocity surface map

Figure 3: Paraview visualisation of supersonic flow over sharp cone at $t = 5.0$ ms. VTK files produced using `lmr snapshot2vtk` command.

3.3 Hypersonic viscous flow over a convex ramp

Files for this example are located in `gdtk/examples/lmr/2D/convex-ramp`.

This example considers external viscous flow at Mach 12.25 over a convex ramp. It is a simulation of experiments performed by Mohammadian (1972) in the Imperial College gun tunnel. New aspects of `Eilmer` use introduced in this example include how to:

- select a thermochemical nonequilibrium gas model
- use Lua to define a custom path to represent the convex ramp
- request surface loads as output during simulation
- run the solver in steady mode using MPI
- gather loads data and plot results against experiment

It's time to start a new working directory for this example, a suggestion is to call this `convex-ramp`.

3.3.1 Selecting a thermochemical nonequilibrium model for air

The flow produced in the gun tunnel is cold; like 42 Kelvin cold! But, there is also an elevated vibrational temperature due to freezing in the nozzle. So it would seem we should account for this thermal nonequilibrium in the free stream in our simulation. It turns out that the thermochemical activity over the convex ramp is not large. Nonetheless, we will select a gas model that accommodatates two-temperature thermodynamics. To model this, we take a copy of a two-temperature air gas model from the `examples/` area. We are going to talk about this in more detail in the next example. For now: take a copy, run `prep-gas`, and set the model in the `Eilmer` input file.

```
$ cp ~/gdtk/examples/air-chemistry-2T/air-5sp-gas-model.lua .
$ lmr prep-gas -i air-5sp-gas-model.lua -o air-5sp-2T.gas
```

NOTE: There is a trailing dot (.) on the first command above indicating to copy to the current directory.

The output file needs to appear in the `Eilmer` job file when setting the gas model. Take a look in `steady.lua` at lines 13–16.

```
nsp, nmodes = setGasModel('air-5sp-2T.gas')
print('5-species, 2T air model: nsp= ', nsp, ' nmodes= ', nmodes)
inflow = FlowState:new{p=p_inf, T=T_inf, T_modes={T_vib,}, velx=u_inf,
                       massf={N2=0.767,O2=0.233}}
```

We set the gas model and hang on to the counts of species and energy modes as variables. The next line prints them to screen as a sense-check for us during the preparation. Note also the `FlowState` object. The vibrational temperature is set in a table for `T_modes`. There is one entry here, but if we hade more nonequilibrium modes, we would set their temperatures in that table. The gas composition is set by mass fractions in the table `massf`. We did not specify values for `N`, `O` and `NO`; they will be set to 0.0 automatically because they were unspecified.

3.3.2 Customized geometry to represent the ramp

Figure 4 shows the geometry, domain and topology to be created in this example. Mostly, there are straight-line segments. However, the bottom surface representing the ramp is interesting. Let's discuss how this is specified in the `grid.lua` file for this example.

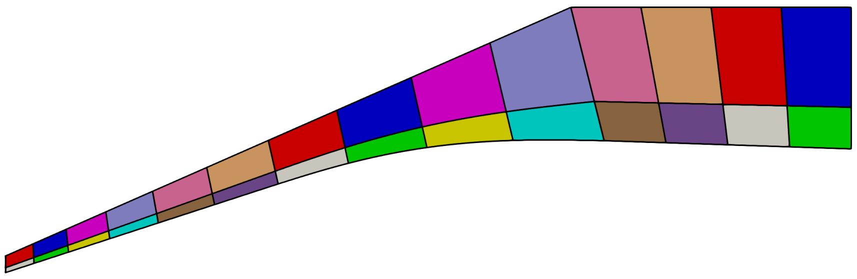


Figure 4: Convex ramp in hypersonic flow. Ramp geometry, domain and block topology.

Mohammadian's description for the ramp is:

- initially straight at 18° until $x = 3$ inches;
- then a faired section defined by

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

  where s and g are the local coordinates, in inches, rotated 18° to the x, y coordinates;
- the fairing second derivative is zero at both joining points: $s = 0$ and $s = 4.058$; and
- the final straight section is at -1.9° in the (x, y) plane, away from the free-stream flow direction.

Inspect the `grid.lua` file. The ramp geometry is specified in a parametric form (from $t = 0$ to $t = 1$) in the user-supplied function `ramp(t)`. `Path` objects in `Eilmer` are parameterised from $0 \rightarrow 1$. To help make sense of the `ramp()` function, there are two observations worth noting: 1) the `t2` parameter converts t to a parameter x in inches; and 2) `Eilmer` requires dimensions in S.I. units, so we convert coordinates to metres at the end of the function.

You can do the grid preparation at this point: `lmr prep-grid --job=grid.lua`

3.3.3 Requesting loads output on surfaces

`Eilmer` does not output loads during a simulation unless requested. For this example, the loads on the ramp are of interest. On line 24 in `steady.lua`, we added a group tag `'loads'` to the no-slip boundary condition object. This tag name is the default for writing loads during a simulation. The tag is not all that is required. We need to tell `Eilmer` to write loads and how often. For this steady mode simulation, we configure that on lines 54 and 55:

```
write_loads = true,
steps_between_loads_update = 20,
```

The result of these flags is that every 20 steps the surface loads on the no-slip ramp will be written into the `lmrsim/loads` folder.

Now, we are ready to do the simulation preparation: `lmr prep-sim --job=steady.lua`

3.3.4 Running the solver with MPI

This example used `FluidBlockArrays` to carve the domain into 28 blocks as shown coloured in Figure 4. If we have a machine with 28 cores or more, we could run one block per core in a parallel manner. If we have fewer cores than blocks, then we can distribute groups of blocks onto each core when running in parallel. The laptop we used for testing had 8 cores available (but some as hardware threads). In line 28 of `steady.lua`, we requested a distribution of the 28 blocks across 8 tasks: `mpiDistributeBlocks{ntasks=8}`. You may think of each task as occupying a core. This simulation is ready to run on 8 MPI tasks. Using OpenMPI, we launch with the `mpirun` command as follows:

```
$ mpirun --use-hwthread-cpus -np 8 lmrZ-mpi-run
```

Note this command breaks the pattern we are used to with the `Eilmer` CLI. This is because we need to invoke the OpenMPI apparatus for launching the executables for parallel processing.

For this example, the `solver_mode` is steady. This means the output to screen is quite different from the transient mode. The transient mode gives progress updates in terms of simulated time. The steady mode gives progress updates in terms of convergence as measured by drop of the residuals.

3.3.5 Post-processing loads

For this configuration, the loads are written every 20 steps. They appear in enumerated directories under `lmrsim/loads/` as `0000/`, `0001/`, etc. The loads at the end of a converged simulation will be in the final numbered directory. In each of those directories, loads are split by block and boundary. To plot these, it is convenient to gather the data in a single big array and then use a plotting tool. We find the Python packages `pandas` and `matplotlib` are useful for manipulating data and plotting. We have provided an example script that plots loads and properties at the ramp surface called `plot_surface_conditions.py`. This script also includes Mohammadian's experimental results on the plot. The resulting plot is shown in Figure 5.

```
$ python3 plot_surface_conditions.py [--save-figure]
```

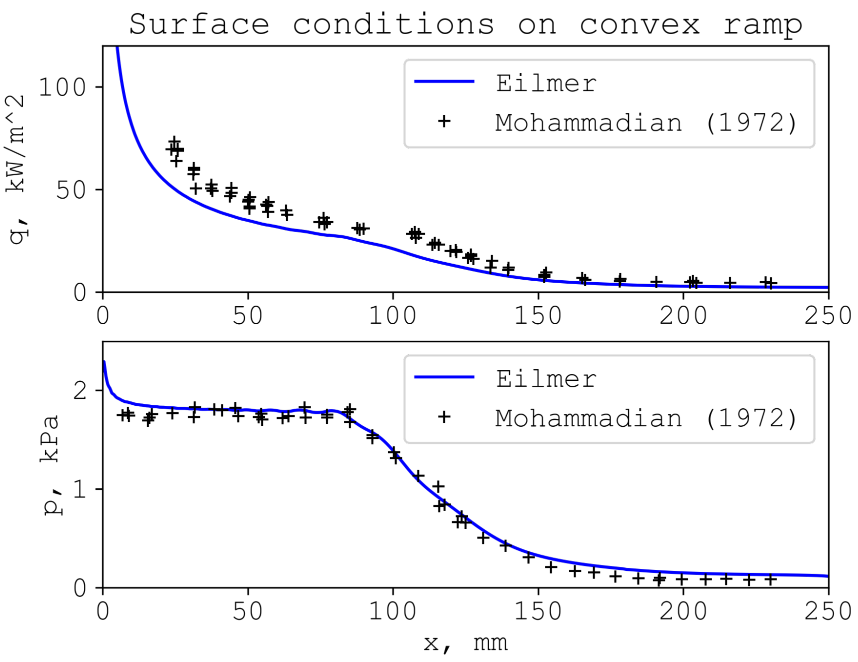


Figure 5: Comparison of `Eilmer` numerical simulation on a relatively coarse grid with experimental results of Mohammadian (1972) for surface heating and pressure load.

3.4 Multi-temperature reacting air flow over a sphere

Files for this example are located in `gdtk/examples/lmr/2D/sphere-nonaka`***.***

Nonaka et al. (2000) performed a series of experiments to measure the shock standoff on spheres fired into air in a ballistic range, with an aim to study flight in the intermediate hypersonic regime (Mach 8–15). This final example simulates one of those experiments and shows a comparison to the measurements of shock shape. In this example, you will learn how to:

- read in a grid prepared in an external tool, GridPro;
- configure chemical reactions and energy exchange mechanisms;
- configure the simulation to use a shock-fitting boundary;
- use custom post-processing to extract the shock shape; and
- use the slicing tool to extract data along the stagnation streamline.

3.4.1 Reading an externally prepared grid

Figure 6 shows a body-fitted structured grid for an axisymmetric simulation that we prepared in GridPro. This grid is a starting point for a simulation using shock-fitting mode, so the grid will move as the shock boundary moves. The import of the grid is a simple function call (shown below) that takes the name of the grid file as a string and a scale. Here we use scale 1.0 because the grid dimensions were prepared in metres.

```
gproGrid = "gridpro/sphere-orig.grd"
grids = importGridproGrid(gproGrid, 1.0)
```

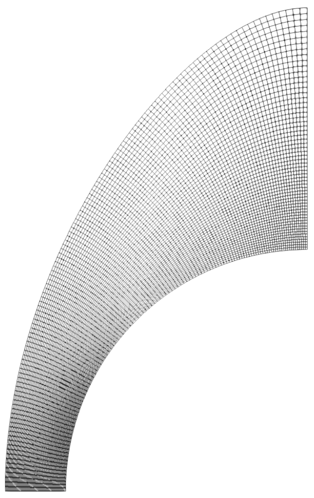


Figure 6: Starting grid for sphere simulations prepared with GridPro.

3.4.2 Configuring thermochemistry

In the convex ramp example, we used a two-temperature gas model for air, but we did not permit any thermochemical activity because we made the modelling judgement that things were rather cold. Not so for this blunt body example at 3.5 km/s. We want to model chemical reactions and thermal nonequilibrium via energy exchange.

We generally have a headstart for doing thermochemical nonequilibrium modelling with `Eilmer`: we can look in the `gdtk/examples/kinetics/` area to see if there are input files that suit our situation. We are in luck for this case. We can select from the files in `gdtk/examples/kinetics/air-chemistry-2T`. Let's take a copy of the following files and place them in our working directory:

```
$ cp ~/gdtk/examples/air-chemistry-2T/air-5sp-gas-model.lua .
$ cp ~/gdtk/examples/air-chemistry-2T/GuptaEtAl-air-reactions-2T.lua .
$ cp ~/gdtk/examples/air-chemistry-2T/air-energy-exchange.lua .
```

The Gupta air chemistry scheme can be used for 5-species chemistry, 7-species and 11-species. We will not need more than 5-species here in this speed/density regime, so let's select that. Edit line 20 in `GuptaEtAl-air-reactions-2T.lua` and change the `11` to a `5`. There are notes in that file about the options available and how it works.

Now we are ready to prepare the thermochemistry files. We have seen the `prep-gas` command before. Additionally, we will use `prep-reactions` and `prep-energy-exchange`. The following commands need to be issued in the order shown because outputs from one are used in subsequent commands:

```
$ lmr prep-gas -i air-5sp-gas-model.lua -o air-5sp-2T.gas
$ lmr prep-reactions -g air-5sp-2T.gas -i GuptaEtAl-air-reactions-2T.lua -o air-5sp-6r-2T.chem
$ lmr prep-energy-exchange -g air-5sp-2T.gas -r air-5sp-6r-2T.chem -i air-energy-exchange.lua -o air-VT.exch
```

The output files are needed as settings in the Eilmer job file. We showed how to set the gas model and flow state in the convex ramp case. The extra configuration required for setting to active the chemical reactions and energy exchanges are lines 21–23 in `job.lua`.

```
config.reacting = true
config.reactions_file = 'air-5sp-6r-2T.chem'
config.energy_exchange_file = 'air-VT.exch'
```

3.4.3 Configuring the solver for shock-fitting mode

In the input script `job.lua` directly beneath the grid import, we register a `FluidGridArray`. In general, a `FluidGridArray` is used to carve the domain of a structured grid into small conforming blocks. We saw that in the previous example with the convex-ramp. Here there is an additional piece of information set when registering the `FluidGridArray`: we also set `shock_fitting=true`. We took care to set a shock-fitting inflow on the `west` boundary. It is a restriction when using Eilmer for shock-fitting that the `west` boundary is the shock-fitting inflow boundary. You can see those settings in the snippet here:

```
registerFluidGridArray{
   grid=grids[1],
   nib=1, njb=njb,
   fsTag='initial',
   shock_fitting=true,
   bcTags={west='inflow_sf', north='outflow', east='wall'}
}
```

The settings just described take care of grid configuration and boundary conditions for using shock-fitting mode. We set three more configuration settings to enable shock-fitting. These relate to the time integration:

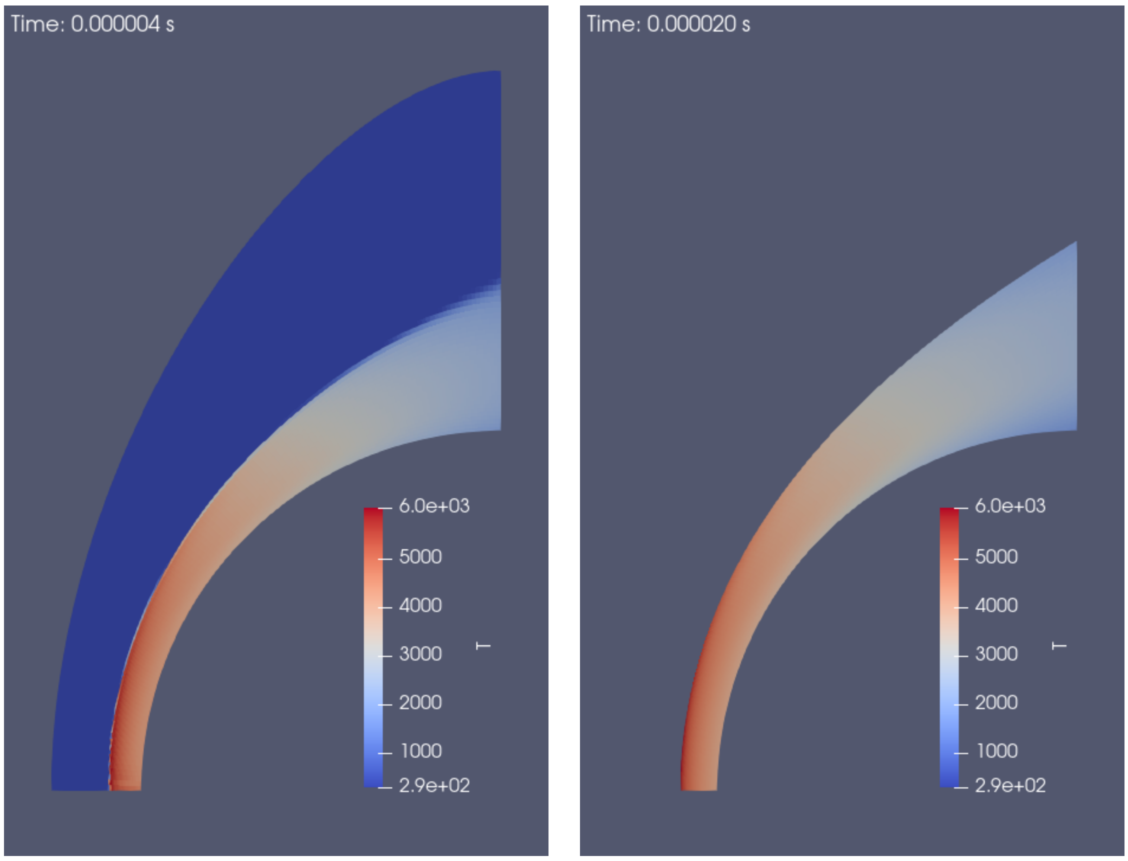
```
config.grid_motion = "shock_fitting"
config.gasdynamic_update_scheme = "moving_grid_2_stage"
config.shock_fitting_delay = body_flow_time
```

The output from this shock-fitting simulation is shown in Figure 7 which displays the (transrotational) temperature field. The "before" image is from early in the simulation: flow has developed, no grid motion driven by shock-fitting has taken place yet. The "after" image is the final state for the simulation after boundary and grid motion have occurred and when the shock and flow field have steadied.

3.4.4 Extracting the shock shape

Figure 8 shows the shock shape extracted from the Eilmer at the final steady state and compares to the measurements by Nonaka et al. (2000). In this shock-fitting mode, the shock shape can be determined by extracting the grid vertex locations directly from the shock-fitting boundary. We have done that using a custom script called `shock-shape.lua`, which you are encouraged to inspect. We can process that custom script as follows:

```
$ lmr custom-script --job=shock-shape.lua
```



(a) (b)

Figure 7: Temperature field at times (a) at 4 μs, prior to any grid motion; (b) at 20 μs, after grid motion settles.

3.4.5 Slicing the flow field

When people perform blunt body calculations, they often like to look at the flow properties along the stagnation streamline. Let's close out this example showing how that is done. `Eilmer` provides a `slice-flow` command for extracting slices or lines of data. It does require some knowledge of your grid indices and directions to use it correctly. Here we call the `slice-flow` command to produce a data file with temperatures along the stagnation streamline.

```
$ lmr slice-flow --final --slice-list="0,:,0,0" --names="T,T-vibroelectronic" \
      --output=stagnation-profile-T.dat
```

The `--slice-list` option is where grid/blocking knowledge is required. This particular example with string `"0,:,0,0"` reads: for block 0, for all `i` indices, at `j=0` and at `k=0`. We needed the knowledge that block 0 was against the symmetry plane, that `i` index ran from shock to body so we wanted all cells along that index direction, and that `j=0` was the common j-index for cells along the symmetry plane. (`k=0` has no effect here, but is required for the generality of 3D grids.) You can find that information by poking around in Paraview and inspecting cells. The `--names` option allows the user to select from the flow field only those quantities of interest. In the command above, we requested the two

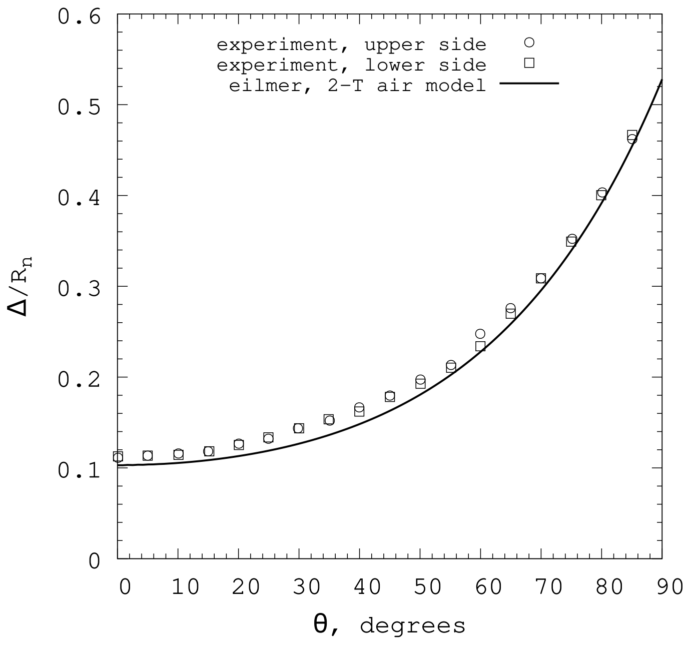
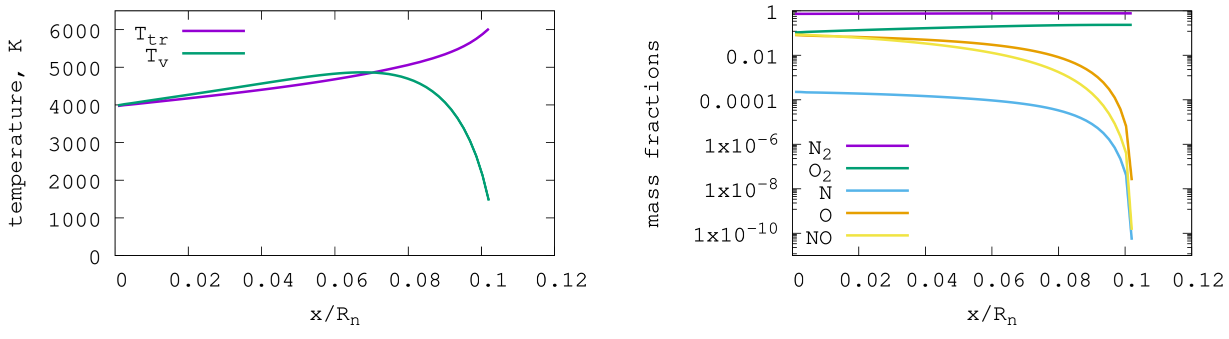


Figure 8: **Eilmer**-computed shock shape extracted using `shock-shape.lua` compared to Nonaka et al. (2000) measurements.
Conditions: $R_n = 7.0\,\mathrm{mm}$, $u_\infty = 3.49$ km/s, $p_\infty = 4850\,\mathrm{Pa}$, $\rho_\infty R_n = 4.0e-4\,\mathrm{kg/m^2}$

temperatures. We ran this `slice-flow` command a second time to extract species mass fractions also. The stagnation streamline properties are plotted in Figure 9, with the body at $x/R_n = 0$ and the shock at $x/R_n \approx 0.1$.



(a) (b)

Figure 9: Properties along stagnation streamline for a sphere fired into air, taken at final simulation time. (a) temperatures; (b) mass fractions of species.

References

Allmaras, S. R., Johnson, F. T., and Spalart, P. (2012). Modifications and clarifications for the implementation of the Spalart-Allmaras turbulence model. In *Seventh International Conference on Computational Fluid Dynamics*, Big Island, Hawaii.

Damm, K. A. (2020). *Adjoint-Based Aerodynamic Design Optimisation in Hypersonic Flow*. PhD thesis, The University of Queensland.

Damm, K. A., Gollan, R. J., Jacobs, P. A., Smart, M. K., Lee, S., Kim, E., and Kim, C. (2020). Discrete adjoint optimization of a hypersonic inlet. *AIAA Journal*, 58(6).

Damm, K. A., Gollan, R. J., and Veeraragavan, A. (2023). Trajectory-based conjugate heat transfer simulation of the BoLT-II flight experiment. AIAA paper 2023-0292, AIAA SCITECH 2023 Forum.

Gibbons, N. N., Damm, K. A., and Gollan, R. (2021). Flight regime limits of a hypersonic vehicle using electron transpiration cooling. AIAA paper 2021-4141, ASCEND 2021.

Gibbons, N. N., Damm, K. A., Jacobs, P. A., and Gollan, R. J. (2023). Eilmer: an open-source multi-physics hypersonic flow solver. *Computer Physics Communications*, 282(108551).

Gildfind, D. E., Jacobs, P. A., Morgan, R. G., Chan, W. Y. K., and Gollan, R. J. (2018). Scramjet test flow reconstruction for a large-scale expansion tube, part 2: axisymmetric CFD analysis. *Shock Waves*, 28:899–918.

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

Goozée, R. J., Jacobs, P. A., and Buttsworth, D. R. (2006). Simulation of a complete reflected shock tunnel showing a vortex mechanism for flow contamination. *Shock Waves*, 15:165–176.

Hornung, H. G., Gollan, R. J., and Jacobs, P. A. (2021). Unsteadiness boundaries in supersonic flow over double cones. *Journal of Fluid Mechanics*, 916(A5):1–23.

Jacobs, P. A. (1991). Single-block Navier-Stokes integrator. ICASE Interim Report 18, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

Jacobs, P. A. (1994). Numerical simulation of transient hypervelocity flow in an expansion tube. *Computers and Fluids*, 23(1):77–101.

Jacobs, P. A. (2004). MB_CNS example book. Mechanical Engineering Report 2004/10, Centre for Hypersonics, The University of Queensland.

Jacobs, P. A. and Gollan, R. J. (2016). Implementation of a compressible-flow simulation code in the D programming language. *Applied Mechanics and Materials*, 846:54–60.

Maccoll, J. W. (1937). The conical shock wave formed by a cone moving at high speed. *Proceedings of the Royal Society of London*, 159(898):459–472.

Mohammadian, S. (1972). Viscous interaction over concave and convex surfaces at hypersonics speeds. *Journal of Fluid Mechanics*, 55(1):163–175.

Nonaka, S., Mizuno, H., Takayama, K., and Park, C. (2000). Measurement of shock standoff distance for sphere in ballistic range. *Journal of Thermophysics & Heat Transfer*, 14(2).

Shur, M., Spalart, P., Strelets, M., and Travin, A. (2008). A hybrid RANS-LES approach with delayed-DES and wall modelled LES capabilities. *International Journal of Heat and Fluid Flow*, 29:1638–1649.

Whyborn, L. S. (2023). *Direct Numerical Simulations of Instabilities in the Entropy Layer of a Hypersonic Blunted Slender Cone.* PhD thesis, The University of Queensland.

Whyborn, L. S., Gollan, R. J., and Jacobs, P. A. (2023). Simulation of multiple instabilities in the entropy layer over a hypersonic blunt cone. AIAA paper 2023-0474, AIAA SCITECH 2023 Forum.

Wilcox, D. C. (2002). *Turbulence Modelling for CFD.* DCW Industries, Inc., second edition.