ROWAN J. GOLLAN

PETER A. JACOBS

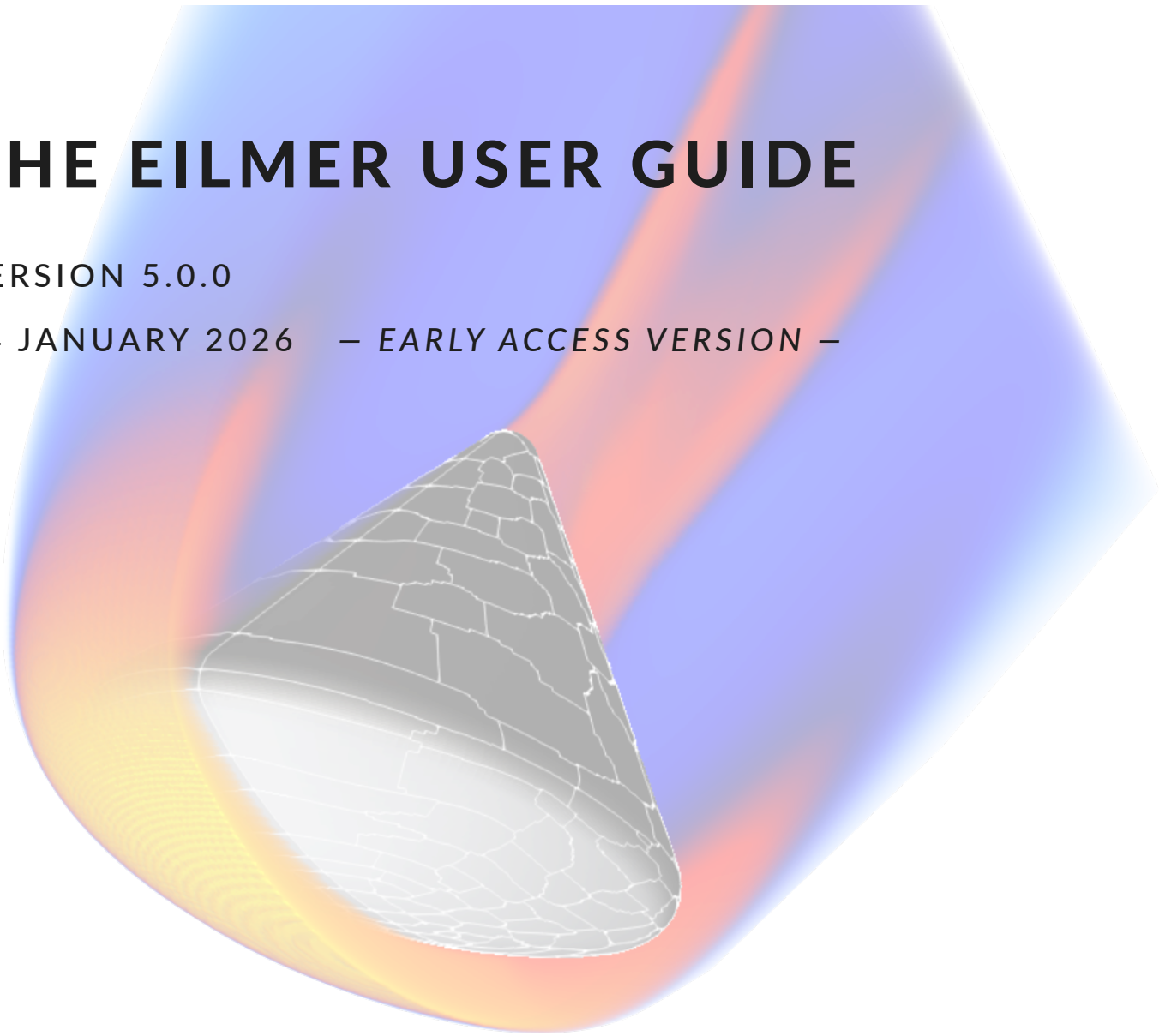
NICHOLAS N. GIBBONS

KYLE A. DAMM

THE EILMER USER GUIDE

VERSION 5.0.0

04 JANUARY 2026 — *EARLY ACCESS VERSION* —



CENTRE FOR HYPERSONICS
SCHOOL OF MECHANICAL & MINING ENGINEERING
THE UNIVERSITY OF QUEENSLAND

Note about this document
This document has been designed for print and binding at left. The font selections have been made with print in mind first, but should be reasonable on screen. The margins are set for binding this document at the left.

If viewing on screen, try 2-page mode in book style (or "show cover page"). This should have the facing pages displayed correctly, with odd-numbered pages on the right (for Arabic-numbered pages).

Cover image
Apollo capsule in Mach 20 flow prepared by Nicholas Gibbons and Kyle Damm.

Find the simulation in our examples at `gdtk/examples/lmr/3D/apollo`.

EARLY ACCESS VERSION

This is an Early Access version of the User Guide. The sections that are available are in a final draft state. These sections are indicated with red headings in the Contents. Other sections await to be written.

PREFACE

This User Guide for version 5.0 of `Eilmer` provides an introduction to its capabilities and use. Our goal is to provide new users with enough instruction to be productive in performing their own simulations of compressible flows. We have deliberately avoided discussing every feature and configuration available in `Eilmer`. That decision is primarily because it lowers the risk of overwhelming the new user with the complexity of configuration options, and it increases the chances of the developers completing the document. In that light, you may consider this User Guide like a tasting plate: we aim to give you a flavour of what you can do with `Eilmer` and how to do it. We also give pointers to the broader ecosystem of documentation so that the new user can pick up on any details relevant to their curiosity or simulation needs.

Let's talk about that broader ecosystem of documentation. We have a website, https://gdtk.uqcloud.net, that overviews all of the activities in the Gas Dynamics Toolkit (GDTk) project. That website also provides a gateway to all our documentation. Our set of documentation falls into different categories. We describe those categories here and the intended audiences.

User Guides, like the one you are reading now, are designed to give a prose-form introduction to specific tools and packages. As other examples, we also provide a geometry/grid user guide and a gas modelling user guide. User guides are designed with the new user in mind in the opening sections. However, they also contain information that the initiated or advanced user might like to refer to at times. For this reason, the structure of the document is designed so that the reader can drop in at a section relevant to their needs.

Tutorials are designed to be followed along by the user to achieve a certain goal. For the most part, our Tutorials are embedded in the **User Guides**. This document contains several tutorials, the first of which appears as Chapter 3. Tutorials are aimed at new users.

Examples are provided in the source code repository, found in the directory `gdtk/examples`. We also host a catalogue on the website to help users navigate the examples: https://gdtk.uqcloud.net/docs/eilmer/examples-catalogue/. It is very common for new and old users alike to find an example close to their simulation goals and adapt it for use. We encourage you to do the same.

Reference Manuals contain comprehensive yet terse information about input options, command usage and data formats. They are not written in prose form, but rather more like catalogue entries with short descriptions for each entry. Reference manuals are written with the familiar user in mind. We rarely try to describe *why* a certain option works in the way it does; we just state *how* to use it, *what* are the options. We rely on the user to be sufficiently experienced to make judgement about what they are looking for. Reference Manuals are hosted on the web (as HTML) because this seems to be the most convenient way to use them, that is, with a web browser open and hyperlink navigation. The Reference Manual for `Eilmer` v5.0 is at: https://gdtk.uqcloud.net/docs/eilmer/eilmer-reference-manual/. We intend this link to be evergreen: content should match the latest release version of `Eilmer`.

Command line help Eilmer is operated by a command line interface. As part of the documentation ecosystem, we provide built-in help for the Eilmer commands. You can

acces that by typing: `lmr help <command-name>`. We have also scraped the help messages for all commands and placed them in the Reference Manual: https://gdtk.uqcloud.net/docs/eilmer/eilmer-reference-manual/#_running_a_simulation.

Cheatsheets are 1–2 pages with a concise reminder of commonly used Eilmer commands. Stick these beside your monitor or paste into the front cover of your workbook.

Technical notes We have written several short technical notes that provide a deeper look at methods, implementation or applications related to `Eilmer`. You can view them at: https://gdtk.uqcloud.net/docs/eilmer/technical-notes.

Issue Tracker As part of the Github-hosted repository, we have enabled the issue tracker. Ideally, this forum would be strictly for implementation issues and bugs. However, in a complex code like Eilmer, it can be hard to distinguish issues in the code and issues with use and user expectations. The developers are happy to support users with queries on the issue tracker, and we can usually quickly distinguish between code issues and how one is using it. The queries and resolutions on the issue tracker are another form of documentation.

Source code `Eilmer` is open source. You can open and inspect the source code itself. This the final word on the implementation. You will also find notes from the developers in the source code including: references to original sources; rationale for design decisions (and alternatives that were considered); and descriptions of nuances and subtleties that arise in the algorithms. Remember: "Use the source, Luke."

This User Guide is organised in two parts. Part One is designed to get a new user going with using `Eilmer` to simulate compressible flows. It covers basic functionality. For example, Part One will cover some commonly used boundary conditions, but it won't discuss every single boundary condition in detail (because some are quite niche in terms of application).

Part Two is called Advanced Usage. In this part, we discuss some of the more advanced usage of `Eilmer` and some of the specialised functionality.

Readers might not read the User Guide from cover-to-cover like a good novel; and random access to sections is expected for familiar users of the code. However, we have tried to organise Part One of the User Guide so that the concepts of operation are developed and re-inforced as one reads the pages in a continuous order. We have chosen to intersperse tutorials in Part One. Each subsequent tutorial introduces new complexity and dimensions to the modelling and simulation toolkit.

We have employed a few typographic conventions to guide the reader. Commands to be typed in a terminal are designated with a dollar sign (`$`) prompt, like so:

```
$ cmd to type
```

Complete your command by hitting `Enter` when you're ready. `Filenames` and `Eilmer constructs` (for use in an input file) are indicated in a typewriter font.

CONTENTS

Chapter One

EILMER AND YOU

We begin with a conversation about you, your simulation goals, and how `Eilmer` can support them. `Eilmer` is a versatile tool for the simulation of compressible flows. It even includes capabilities to simulate the interaction of gases with structures, with both thermal and elastic analyses. However, it is not the right tool for every job, and numerical simulation is not the right choice for every gas dynamic analysis. That's why we begin with this conversation to figure out if `Eilmer` is right for you.

It is not our intent to scare away users in this section, although the questions might be challenging. Rather, our goal is for you to make informed decisions as you plan and execute a simulation.

1.1. *Why do you want to do a simulation?*

We should start with *why*: why do you want to do a simulation? We can ask this question in a few ways. What are you hoping to learn from a simulation? What are your goals? "I was told I should do a simulation" is not a good response. You'll sometimes hear these questions phrased rather bluntly when a user asks for help: "what is it you are trying to do, *exactly?*"

Why is important because it helps you decide if a simulation is the right thing to do and, if so, what kind of simulation and what outputs are needed.

Harlow and Fromm [1], in their 1965 article, give us one of the earliest reasons for *why*: to perform experiments in fluid dynamics. In this case, the computer simulation augments or replaces the wind tunnel for experiment. The use of computational fluid dynamics has expanded in many directions since then. Here is a non-exhaustive list of reasons why you might want to perform a simulation. They are loosely ordered from fundamental physics investigation to engineering design and decision making (and, admittedly, it is imperfect to try to order and separate the uses along strict boundaries).

- to perform experiments in fluid dynamics
- to supplement information gathered in physical experiments
- to plan and design physical experiments
- to test and validate a model of how a physical system behaves
- to simulate the operation of a system with gas as working fluid and assess its performance
- to estimate the aerodynamics of an object pushing through gas (or immersed in moving gas as befits the frame of reference)
- to estimate forces and heat loads from gases interacting with solid objects
- to use analysis to inform design (such as optimisation)
- to use analysis to inform decision making (such as safety assessment)

- to answer *what-if* questions when exploring a parameter space, such as in a design exercise.

Bossel [6] has attempted to categorise the motivations for simulation into two categories: scientific knowledge and technological knowledge. `Eilmer` has been used to advance knowledge in both of these categories, but predominantly its uses lean towards the technological side: the creation of applied knowledge; the creation of new physical systems or processes. Figure 1 displays a variety of ways in which `Eilmer` has been applied in simulation.

The main point is that you need to have a clear vision of why you are doing a simulation because the purpose has a large bearing on how you approach the simulation and a large bearing on the required effort. We are talking about both the effort of the researcher and the computational effort.

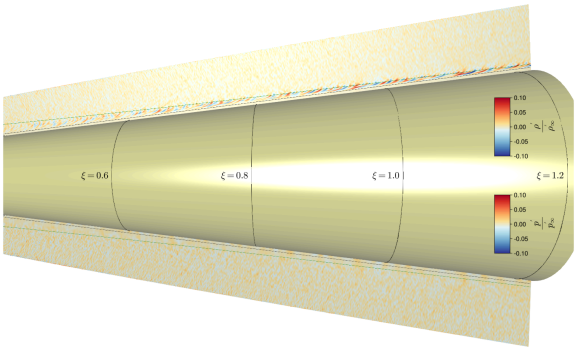
A related point is the credibility of the simulation and how much effort is invested. Typically, a high degree of credibility requires a high degree of effort to generate the evidence to build trust. That being said, the degree of credibility required often scales with how the high consequences are. A *what-if*-type simulation to satisfy curiosity may not require the same effort and care as an analysis of the aerothermodynamics surrounding a capsule bound for Jupiter when there is one shot at atmospheric entry at the end of a 7-year voyage. A full discussion on credibility in scientific computing is given in Section 1.2 of Oberkampf and Roy [7].

1.2. *Eilmer's role is to solve equations*

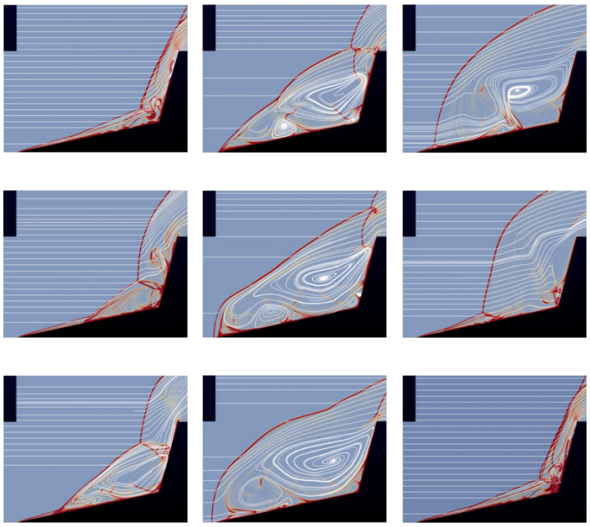
If you're still reading and decided that simulation is right for your needs, then we have good news. `Eilmer` is designed to simulate the dynamic motions of gases and their interactions with structures. Before we get into the modelling features, let's talk about what `Eilmer` does in a fundamental sense.

Fundamentally, `Eilmer` solves partial differential equations (PDEs), and specifically, the Euler equations, Navier-Stokes equations and Reynolds-Averaged Navier-Stokes equations. There are other governing PDEs available that move beyond gases and move us into the realm of multi-physics simulation. Those others include the heat equation in solid domains and magneto-hydrodynamics equations. The solution of a PDE requires inputs from the user; at a minimum those are: (a) specification of a computational domain; (b) specification of boundary conditions at the edge of the domain; and (c) specification of an initial condition. `Eilmer` uses a finite-volume formulation for discretisation of the domain. The user also builds the grid of cells to fill the domain. Other inputs are required depending on the complexity of the physics being modelled. The theory related to the numerical solution process and a discussion of the multi-physics capabilities are presented in the `Eilmer` journal paper [8].

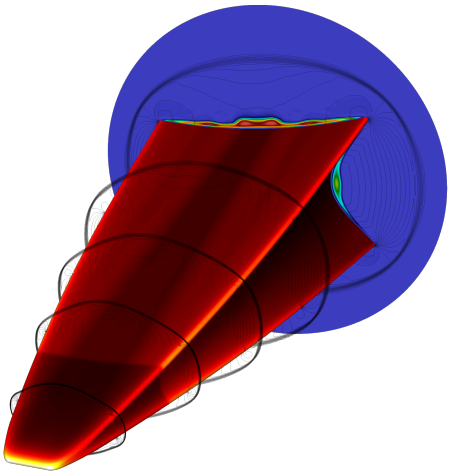
The features for `Eilmer` are shown in Table 1. Features are grouped by physical modelling. For the most part, this means the selections within a group are mutually exclusive. For example, you cannot select two different turbulence models simultaneously for a single simulation. The features have been categorised as **Production**, **Experimental**, or **Developmental**. These boundaries of division are not hard; they serve to indicate the maturity of the implementation.



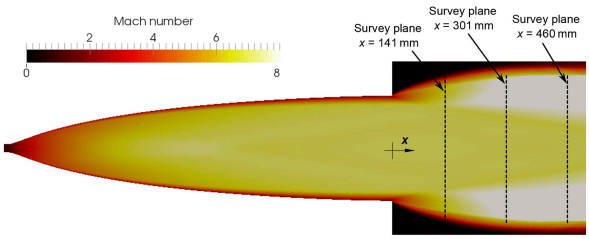
(a) Boundary layer response to far-field disturbances. Density - top; Pressure - bottom; *Source: Whyborn (2023)* [2]



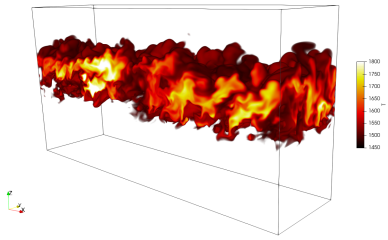
(b) Unsteady flow over a double-cone. Time sequence displayed down colum, then moves to next column. *Source: Hornung et al. (2021)* [3]



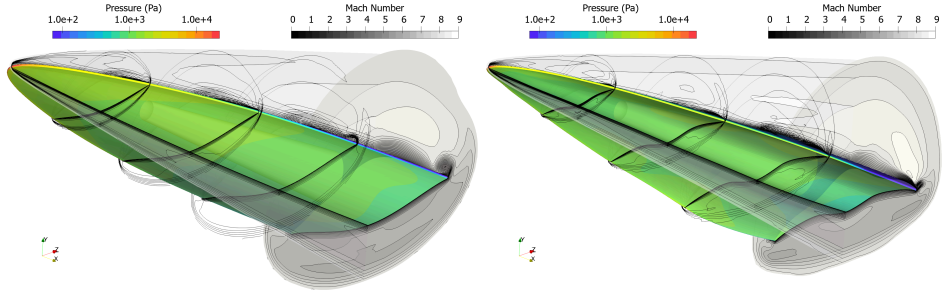
(c) Coupled fluid-thermal analysis of the BoLT-II flight experiment. *Source: Damm et al. (2024)* [4]



(d) Design of Mach 7 nozzle for T4 shock tunnel. *Source: Chan et al. (2018)* [5]



(e) Scale-resolving simulation of mixing layer with air and diluted ethylene.



(f) Lifting body optimised for lift-to-drag ratio using adjoint method. *left:* baseline *right:* optimised.
Figure 1: A gallery of Eilmer simulations capturing some of the diversity of use cases.

Table 1: Feature list for `Eilmer` v 5.0

Feature	Production	Experimental	Developmental
Equation sets	Euler	heat equation in solid domains	magneto-hydrodynamics
	Navier-Stokes	electric fields	
	Reynolds-Averaged Navier-Stokes	rotating frame	
	multiple species		
Solver mode	transient (time-marching)		block-marching
	steady-state		
Grid types	structured		
	unstructured		
Spatial approximation	piecewise parabolic (structured)	3rd order reconstruction (structured)	
	van Albada limiter (structured)	blended 4th/2nd order scheme	
	least-squares (unstructured grids)		
	selection of limiters for unstructured		
Gas models	ideal (calorically perfect)		state-specific
	thermally perfect, mixtures		
	multi-temperature, mixtures		
Gas-phase Kinetics	equilibrium chemistry		
	finite-rate chemistry		
	finite-rate energy exchange (for multi-T models)		
Conjugate heat transfer		2D/3D structured domains	

Feature	Production	Experimental	Developmental
Moving boundaries	shock-fitting		
	coupled motion via run-time loads and user-defined motion		
Turbulence models	Spalart-Allmaras (S-A)		S-A for IDDES
	S-A, BCM variant		
	S-A, Modified Edwards		
	$k-\omega$ (Wilcox, 2006)	$k-\omega$, vorticity-based source term	
User-defined customisation	boundary conditions		
	source terms		
	grid motion		
	supervisory functions		

Eilmer is typically used in rolling release mode. Users update the source code from the master branch of the github repository if and when updates are required for their work. However, we also provide tagged releases with a version number. The point of version numbers is to signal which features we feel are mature and we are able to support as a development team. So that leads to our use of these categories:

Production features that are mature and well-tested; development team commit to support and bug fixes

Experimental features that are relatively new and have had limited use (perhaps only in a one-off project)

Developmental features that are under active development; inputs and implementation may change; no guarantees on stability

A list of supported features for each release is maintained at https://gdtk.uqcloud.net/docs/eilmer/releases/.

So, Eilmer solves equations. It is up to you to decide what outputs are required. This gets back to what is the purpose of your simulation. Do you require time history at select locations (like a virtual transducer)? Do you require loads at surfaces? Do you need data recorded at a certain outflow plane? Eilmer can give you these outputs but only if instructed. This User Guide will help you learn how to instruct Eilmer.

1.3. Preliminary planning

Let's close this conversation with some words of advice on preliminary planning, the kind of planning to do with pen and paper before opening a text editor. What we'd really like to emphasise is the distinction between transient (time-marching) simulations, and those that are accelerated to steady state. We don't just mean the technical distinction of how they operate; we mean understanding the differences in use cases for these two solver modes. This is why we laboured the earlier point "why do you want to do a simulation". Here then is our advice.

The time-marching solver, when configured for time-accurate simulations, gives a time-dependent physics-based simulation result. If you are looking to perform investigation of fluid physics, then you should preference using the time-marching (transient) solver. All real-world flows are unsteady at some scale.

The steady-state solver should be used when you are reasonably confident there is a steady flow solution at the physical scales of interest. A typical use is engineering calculations of aerodynamics at modest angles-of-attack.

When starting a new simulation with limited experience of the time and space scales involved, we recommend using a coarse-resolution time-marching simulation. At coarse resolution, the time-marching solver is the most efficient way to get feedback about your simulation. You can use it to check the appropriateness of your initial condition, your boundary conditions, and the extent of your computational domain.

You should also consider the spatial dimensionality of the problem. All real-world flows are three dimensional, but some are excellently approximated with 2D or axisymmetric domains. If your problem permits, begin with a 2D

simulation (planar/axisymmetric). You will get feedback much quicker as compared to a 3D equivalent.

Part One

BASIC USAGE

Chapter Two

GETTING STARTED WITH EILMER

The goal of this section is to get you to the point of a working `Eilmer` installation, ready for simulation. You can exercise the installation in the follow-on tutorial, Chapter 3.

2.1. *Prerequisites: operating system and operator*

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.

Second, let's talk about you, the operator. More specifically, let's talk about your assumed background. 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 `Eilmer`, we try to make the analysis of compressible, reacting flow accessible and reliable; we cannot make it trivial.

2.2. *Preparing your compute 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 2 for linux and Table 3 for macOS. In Table 2, we have given the package names for some of the more commonly encountered systems. On linux, we recommend using the package manner appropriate to your distribution to install the packages. On macOS, the package manager to use is `Homebrew`.

Table 2: Prerequisite software for `Eilmer` on linux

Software	Debian family	RedHat family
basic build environment	`build-essential`	"C Development Tools and Libraries"[1]
LLVM D compiler	`ldc`	`ldc`
Fortran compiler	`gfortran`, `gfortran-multilib`	`gcc-gfortran`
git	`git`	`git`
readline	`libreadline-dev`	`readline-devel`
ncurses	`libncurses-dev`	`ncurses-devel`
OpenMPI	`libopenmpi-dev`	`openmpi-devel`
Paraview	*— download latest is recommended —*	
gnuplot	`gnuplot`	`gnuplot`
Pandas	`python-pandas`	`python-pandas`
matplotlib	`python-matplotlib`	`python-matplotlib`

[1] Install this collection using `dnf group install`

Table 3: Prerequisite software for `Eilmer` on macOS

Software	Package name
basic build environment	xcode[1]
LLVM D compiler	`ldc`
Fortran compiler	`gcc`
git	`git`
readline	`readline`
ncurses	`ncurses`
OpenMPI	`open-mpi`
Paraview	`paraview`
gnuplot	`gnuplot`
Python	`python`
Pandas	`pandas`[2]
matplotlib	`matplotlib`[2]
sed	`gnu-sed`[3]

[1] Install xcode as a macOS package, not using Homebrew.

[2] Install using `pip3`.

[3] After install, type `brew info gnu-sed` to get hints about setting your `PATH`.

2.3. *Dowloading, 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:

console

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

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

console

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

2.3.1. A note on `FLAVOUR=debug` and `FLAVOUR=fast`

In the `make` command just shown, we gave only the target `install`. This sets the recipe in action to build using default options and then install to a default area. There are a number of options available to control the build and install of `Eilmer`. We discuss these later as part of advanced usage in Chapter 12.

There is one option important to discuss now because it has tripped up users in the past who have not read all of the documentation. `Eilmer` has `FLAVOUR` options for building the executable. The default `FLAVOUR` is `debug`. Executables built with `debug` include certain run-time checking and are able to print more diagnostic information in the case of an error. This is why we recommend it for those new to the code.

The second `FLAVOUR` option is `fast`. We recommend this for production calculations when speed matters such as when being charged for computer hours on a time-shared cluster. The `fast` build turns on compiler optimisations and disables most checking at run-time. Note `fast` refers to the performance of the executable, not the build time. It actually takes *much longer* to build the `fast` executable because the compiler needs to do more work to find and implement optimisations. There is a compromise for speedy executable: we cannot capture full diagnostic information in the event of a program error. To build with `fast`, try:

console

```
$ cd gdtk/src/lmr
$ make clean
$ make FLAVOUR=fast install
```

2.4. *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:

bash

```
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
```

Chapter Three

TUTORIAL: FLOW OVER A CONE

This tutorial is designed to get you acquainted with using `Eilmer`. We do not attempt to explain everything here; that comes later in the guide. We do however show you the steps to get a simulation result: pre-processing, running a simulation, and post-processing. For this tutorial, you will copy from pre-existing files from the repository.

Let's start with a simple-to-imagine flow of ideal air over a sharp-nose of a supersonic projectile. Figure 2 is a reproduction of Fig. 3 from Maccoll's 1937 paper [9]. and shows a shadowgraph image of a two-pounder projectile, in flight at Mach 1.576. We'll restrict our simulation to just the gas flow coming onto and moving up the conical surface of the projectile and work in a frame of reference attached to the projectile. Further, we will assume that all of the interesting features of the three-dimensional flow can be characterized in a two-dimensional plane. The red lines mark out the region of our gas flow simulation, assuming axial symmetry about the centreline of the projectile.

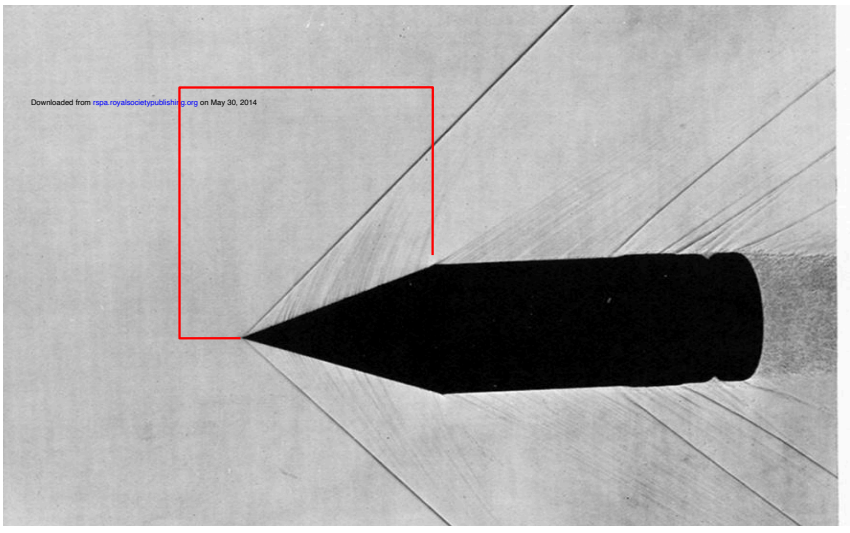


Figure 2: A two-pound projectile in flight. A conical shock is attached to the sharp nose of the projectile. This photograph was published by Maccoll in 1937 [9]. The red lines have been added to demark the region of gas flow for which we will set up our simulation.

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 set-up

To build our simulation, we abstract the boxed region from Figure 2 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. In Figure 2, we have suggested a computational domain shown in red. Figure 3 shows the two-block computational domain that corresponds to the red bounded region in Figure 2.

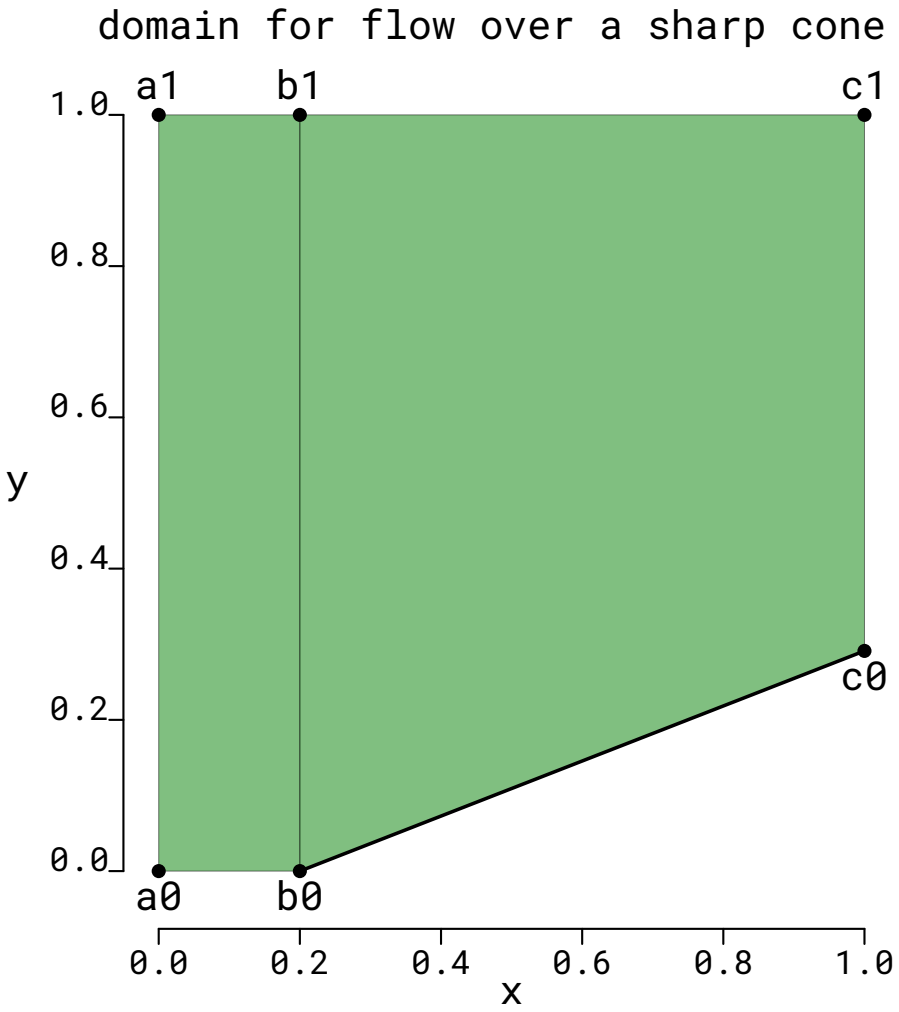


Figure 3: Schematic diagram of the geometry for a cone with 20 degree half-angle. The thick dark line represents the cone surface and the green coloured region represents the gas domain. Boundary conditions will be added such that gas flows into the domain on the left (west) boundary and out on the right (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 2 being a good motivator for this simulation, the free-stream conditions of $p_\infty = 95.84$ kPa, $T_\infty = 1103$ K and $V_\infty = 1000$ m/s are actually related to the shock-over-ramp test problem in the original ICASE Report [10] and are set to give a Mach number of 1.5. It is left as an exercise for the reader to run a simulation at Maccoll's value of Mach number and check that the simulation closely matches the shadowgraph image.

3.2. *Preparing 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 ~/gdtk/examples/lmr/2D/sharp-cone-20-degrees/
sg-minimal/ .
```

to set up a work space that is separate to your copy of the source code tree. 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.

3.2.1. Preparing a gas model

The first task in starting our simulation is to prepare an input file for the gas model. Our gas model is very simple. It is ideal air. The gas model file is correspondingly simple. Create a file, using a text editor, called `ideal-air.lua` and place in it:

lua

```lua
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
```

Hopefully 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.

3.2.2. Preparing grids

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 3.

lua

```lua
-- grid.lua
print("Set up geometry and grid for a Mach 1.5 flow
over a 20 degree cone.")
--
-- 1. Geometry
```

```
 5 a0 = {x=0.0, y=0.0};     a1 = {x=0.0, y=1.0}
 6 b0 = {x=0.2, y=0.0};     b1 = {x=0.2, y=1.0}
 7 c0 = {x=1.0, y=0.29118}; c1 = {x=1.0, y=1.0}
 8 --
 9 quad0 = CoonsPatch:new{p00=a0, p10=b0, p11=b1, p01=a1}
10 quad1 = AOPatch:new{p00=b0, p10=c0, p11=c1, p01=b1}
11 --
12 -- 2. Grids
13 grid0 = registerFluidGrid{
14    grid=StructuredGrid:new{psurface=quad0, niv=11,
   njv=41},
15    fsTag="inflow",
16    bcTags={west="inflow"}
17 }
18 grid1 = registerFluidGrid{
19    grid=StructuredGrid:new{psurface=quad1, niv=31,
   njv=41},
20    fsTag="initial",
21    bcTags={east="outflow"}
22 }
23 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 texttt{i} and texttt{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` [=*flow state tag*] 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` [=*boundary condition tags*], 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`[1] command.

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

3.2.3. Preparing the flow domain and configuring settings

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.

lua

```lua
-- transient.lua
print("Set up transient solve of Mach 1.5 flow over a
  20 degree cone.")
--
-- 0. Assume that a previous processing has step set up
  the grids.
--
-- 1. Domain type, gas model and flow states
config.solver_mode = "transient"
config.axisymmetric = true
setGasModel('ideal-air.gas')
initial = FlowState:new{p=5955.0, T=304.0} -- Pa,
  degrees K
inflow = FlowState:new{p=95.84e3, T=1103.0,
  velx=1000.0}
flowDict = {initial=initial, inflow=inflow}
--
```

[1] This was not installed by default on my Mac. Try: `brew install tree`

```
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!

3.3. Running the 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`.

3.4. *Post-processing the simulation*

3.4.1. Producing VTK files for visualisation

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

```
$ lmr snapshot2vtk
```

since the default snapshot to process is the final one in a sequence.

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 4 shows surface plots coloured by pressure (on *left*) and velocity in x-direction (on *right* with grid overlayed).

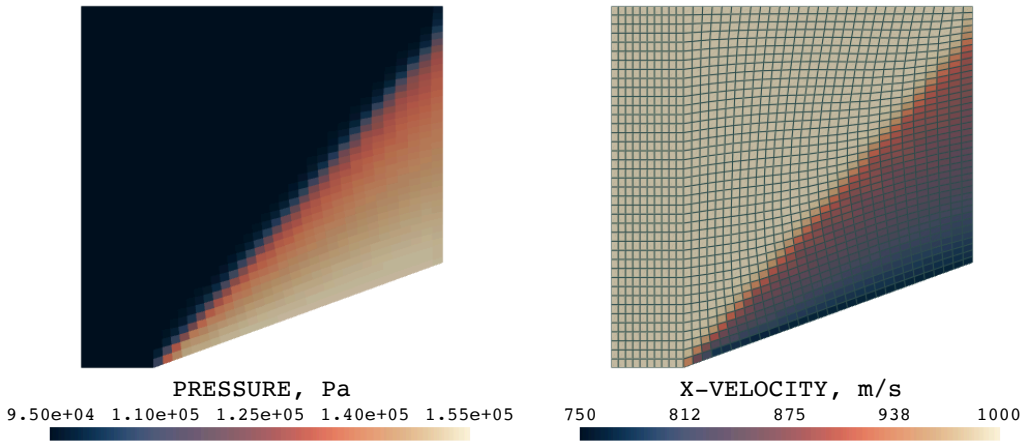


Figure 4: Paraview visualisation of supersonic flow over a sharp cone at $t = 5.0$ ms. VTK file produced using the `snapshot2vtk` command.

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.[2] 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 5. The high-resolution solution is looking clean but the computational cost, in terms of calculation time, has gone up from less than a second to more than 13 minutes.[3]

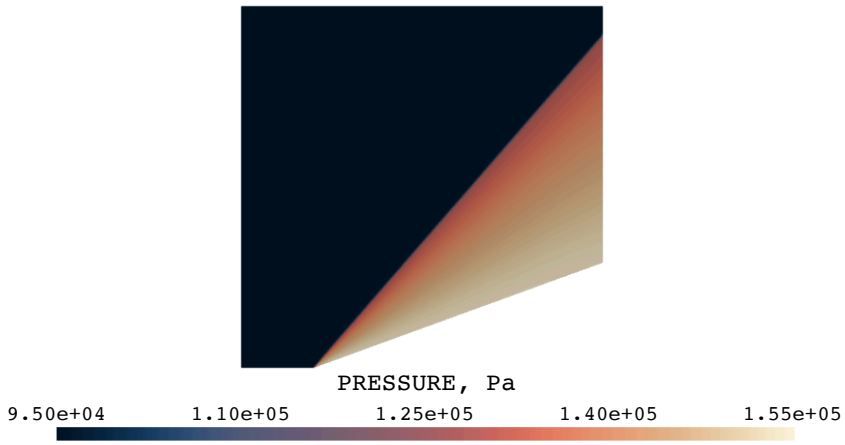


Figure 5: Pressure field for a mesh with 8 times more resolution in each direction compared to the original simulation shown in Figure 4.

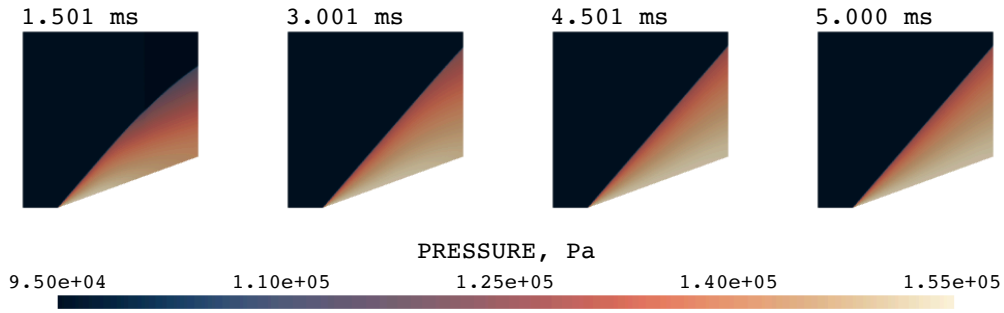


Figure 6: Evolution with time of supersonic flow over cone.

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

[3] These wall-clock numbers resulted when using 2 cores (one per block) on an Apple M4 chip with the code compiled as `FLAVOUR=fast`.

Chapter Four

WORKING WITH EILMER

This chapter is rather brief. It is designed to give you an overview of how you use `Eilmer`. The details are deferred to individual chapters that follow.

4.1. Overview of simulation workflow and user interface

At a top-level view, `Eilmer`'s workflow is much like most other CFD programs. It involves, in order, a **preparation stage** (or pre-processing), a **simulation stage**, and a **post-processing stage** to extract or generate desired outputs. Those stages might require one or more substeps.

Typical steps at **preparation stage** are:

- preparing a gas model
- preparing a computational domain and grid
- configuring the computational domain with boundary conditions and initial conditions
- configuring the flow solver such as numerics selections, physical models and run-time outputs

Some complex simulations might require more steps at preparation stage.

At the **simulation stage**, typical actions are:

- start a shared-memory simulation on a local machine
- start a distributed-memory simulation on a local machine (using MPI)
- configure and launch a job script to a queue on a shared-resource high-performance computer

The **post-processing stage** can vary widely based on the simulation goals, and can involve multiple steps for a single simulation. Some typical steps are:

- generate VTK files for visualisation from flow field data
- plot history data recored during simulation
- extract lines or planes of data from the flow field
- extract data from boundary surfaces
- plot surface loads
- compute integral quantities such as mass flow across boundaries or aerodynamic loads on bodies

`Eilmer`'s **user interface** is text based. Users prepare input files as plain text using a text editor. These files are used as input to `Eilmer` commands that are executed in a terminal. We call this a command-line interface and explain it in the next section. The preparation of the input files themselves appears in subsequent chapters. During user interaction, `Eilmer` generates folders and files to hold the output from executing commands. A brief description of the inputs and outputs is given in Section 4.3. You saw this use of the command-line interface and the generation of `Eilmer` outputs earlier in the cone flow tutorial (Chapter 3).

4.2. *The command-line interface explained*

`Eilmer` is a command-line driven program: commands are used to execute the workflow just described. The command-line interface (CLI) in `Eilmer` is of the form:

`lmr` *action* *options/arguments*

`lmr`[4] is used to invoke the `Eilmer` program. Not all commands require options or arguments. 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:

```
1 $ 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:

```
1 $ 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.

```
1 $ lmr help
```

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

```
1 $ lmr help -a
```

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

```
1 $ 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.

4.3. *Inputs, outputs and the `lmrsim` directory*

Users create input files to instruct and configure `Eilmer` operation. Some common input files, shown on left in Figure 7 as user-supplied files, are the description of the thermochemistry model (`gas-model.lua` and

[4] Why `lmr` and not `eilmer`? It's shorter to type at the command line, and dropping vowels is not without precedent; we've borrowed from the Phoenecians from more than 3000 years back.

reactions.lua), construction of a domain and grid (grid.lua), and configuration of the flow domain and simulation settings (transient.lua).

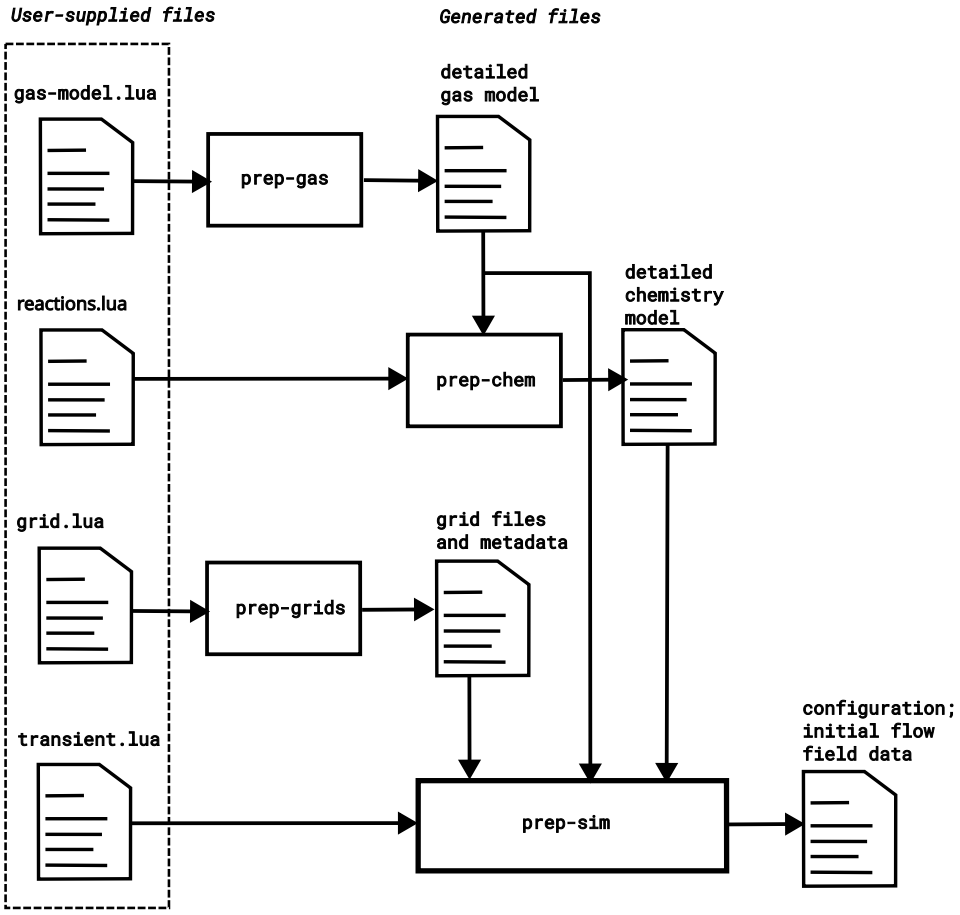


Figure 7: An overview of the Eilmer simulation process showing inputs and outputs. Note that outputs are created in the lmrsim area, except for the gas-related files. *Source: A version of this figure appears in Gollan & Jacobs [11] as Figure 4.*

You issue Eilmer commands to process your user-supplied inputs. Such commands are shown in Figure 7: prep-gas, prep-chem, prep-grids and prep-sim.

The outputs from these commands are generated files. These generated files go in the lmrsim directory. (Eilmer will automatically create an lmrsim directory when first required.) We like this single top-level directory arrangement for storing outputs because it is convenient for copying, synchronising (from workstation to remote cluster), archiving and cleaning out (when you want to start afresh).

There are exceptions to the "everything generated goes into lmrsim" rule, but those exceptions are few. The detailed files for the thermochemistry are placed in the current working directory. This is because the thermochemistry files are used in many other places and tools in the GDTk collection. We cannot enforce the lmrsim location rule on those other tools. So we live with this contained set of exceptions.[5]

[5] Besides, keeping a record of your inputs is not a big deal. You are keeping all of your own inputs in your own repository, right?

Chapter Five

PREPARING A SIMULATION

In this chapter, we get into some detail about preparing a simulation. We focus mostly on the preparation of input scripts since that is where the bulk of a user's effort will go. Recall that our goal for Part 1 of this user guide is to equip users with the basics for a tip-to-tail simulation workflow using `Eilmer`. So even though we present some detail here, we do not present *all* details. Those can be found in Part 2 of this user guide and in the online reference manual: `https://gdtk.uqcloud.net/docs/eilmer/eilmer-reference-manual/`.

5.1. *Input scripts overview*

Because your input scripts become a part of the program when run, 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", which is available online, is a good read and has everything that you need to successfully write good Lua scripts.

For the simulation of simple non-reacting gas flows, you will usually have three Lua input files: one to specify the gas model; one to build the grid; and one to configure the simulation domain and related settings. The various preparation commands convert the descriptions and instructions in your Lua input files into a form that is read by the main simulation program. 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.

The remaining sections in this chapter describe those three basic input files individually. We will refer to these files as `gas-model.lua`, `grid.lua` and `flow.lua`. The particular choice of names is not important so you may name these in a descriptive manner as best fits your needs. The `.lua` extension is not required by `Eilmer`, though it is helpful if your editor gives you Lua-specific syntax highlighting. It is important to note that a complete description of a simulation's inputs comprises all three files. The advantage of separate files is for re-use and consistency across families of simulations: you can re-use gas and grid definitions by copying those files where needed.[6]

5.2. *Specifying a thermochemical model*

The thermochemical models are provided by the `gas` module [12]. This is a D-language module with a Lua interface so that its objects and methods can be

[6] The advanced user might even place those common files in a `common` folder and link to them as needed.

accessed from the user's input script. For the moment, we'll just remind you how to set the gas model for ideal air, which we saw earlier in the cone flow example in Chapter 3. Start by placing the following text:

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

into a file called `gas-model.lua` and run the following command:

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

to produce the file `ideal-air.gas` which contains the fully-specified gas model.

Later, you will want to make use of this model in your `flow.lua` input file. To do so, add the line (to `flow.lua`):

```lua
setGasModel('ideal-air.gas')
```

This will initialize the gas model within the simulation preparation program.

For even more sophisticated gas models, the line shown above is all that is needed within your job script to initialize the gas model. You will adjust the name to match that of the detailed gas file you produced using `prep-gas`. 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 [12] but there are a couple of the examples to study later in the present user guide.

5.2.1. 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 [12].

5.3. Grid preparation

We mentioned that three separate input files are needed for an Eilmer simulation. The second of those is for the grid specification. Commonly, users name this file `grid.lua`.

In `Eilmer`, grid preparation has its own stage in the overall process to prepare a simulation. The grid preparation stage precedes flow domain specification. The role of the grid preparation stage is to write a grid, in `Eilmer` format, into the `lmrsim/grid` area and to declare some metadata about the grids, which is mostly about labels on grid boundaries so that boundary conditions can be attached at a later stage. The `Eilmer` command for grid preparation is `prep-grid`.

There are two broad categories for preparing grids in `Eilmer`: natively-built grids using `Eilmer` tools; and importing grids built using 3rd-party grid generators. For both categories, we use `lmr prep-grid` to prepare grid

information in a form ready for an Eilmer simulation.[7] We will discuss the two categories of grid type separately, but before getting to those, let's discuss some core functions that are the workhorse functions of every grid input script.

5.3.1. Core grid preparation functions

The primary purpose of a grid input script is to declare some information about multiple block grids including their grid points, connections and boundary labels. We call this declaration process "registering a grid" and provide functions so named for that purpose. Here we describe the basic registerFluidGrid.

```lua
registerFluidGrid{
  grid, tag, fsTag, bcTags, gridArrayId
}
```

The parameters in registerFluidGrid are:

grid	a StructuredGrid or UnstructuredGrid object that has been generated or imported
tag	a *string* to identify the grid later in the user's script
fsTag	a *string* that will be used to select the initial flow condition from a dictionary when the FluidBlock is later constructed
bcTags	a *table of strings* that will be used to attach boundary conditions from a dictionary when the FluidBlock is later constructed
gridArrayId	an *integer* that needs to be supplied only if the grid is part of a larger array

Note that we have not yet mentioned in this section what a FluidBlock is but you did see it earlier in the tutorial chapter (Chapter 3) in the transient.lua script. FluidBlocks are created in the stage *after* grid preparation but we need to set up some information now during grid preparation that maps from grid to the associated block. Those items are the initial condition (fsTag) and the boundary conditions (bcTags).

In the grid.lua example script in Chapter 3, you saw registerFluidGrid called twice: once to declare a grid in the block upstream of the cone tip, and a second time to declare the grid that lies along the cone surface.

There are some other grid registration functions to round out the family. Those are: registerFluidGridArray, registerSolidGrid, and registerSolidGridArray. These have some specialised purposes. The GridArray variants are used to help decouple parallel load balancing from grid topology concerns, and certain features, like shock-fitting, require grids arranged as a grid array. The Solid variants are used to declare grids that are part of the solid domain in coupled fluid-thermal analyses. We discuss these variants as they arise in Part II on Advanced Usage.

[7] For advanced uses cases, one could bypass lmr prep-grid and write appropriate information to lmrsim/grid directly. We provide a description of the grid data formats in the appendix to this guide for those attempting to build such advanced preparation workflows.

The next function to mention is not essential for registering grid, but it is extremely convenient: `identifyGridConnections`. This function can be called to connect a number of independently registered grids. It searches for connections on grid faces with matching corner positions. When it finds a match, it connects those faces of the associated grids. That information is propagated into the grid metadata. For most uses, `identifyGridConnections()` is called without parameters since the defaults are perfectly adequate. For the rarer special cases, you can read about the parameters to `identifyGridConnections` in the reference manual.

The registration functions require a `Grid` object, and specifically, either a `StructuredGrid` or `UnsutructuredGrid` object. We discuss the creation of those next.

5.3.2. Natively-built `Eilmer` grids

`Eilmer` has access to the GDTk geometry engine for building grids. The typical way to build grids using the geometry engine is to embed the constructs directly in the `grid.lua` file. In that way, the grids are built just-in-time before registration in a call to `prep-sim`. You can see that approach in `grid.lua` in Chapter 3.

Natively-built `Eilmer` grids are a good choice for simple domains in 2D and 3D, and predominantly for structured grids. With care and perseverance, users have built quite geometrically-sophisticated domains using the text-based interface. There are several advantages to script-style natively-built grids:
- batteries-included `Eilmer` install — no need for 3rd-party grid packages;
- good reproducibility afforded by text-based record of grid-building process;
- good reusability of grid pieces across multiple simulations; and
- easy parameterisation of geometry and grids.

So how do you go about using the built-in geometry engine? Let's talk about structured grids first. Grids are usually built up from primitive pieces. First is construction points (`Vector3` objects). From construction points, you can build `Path`s. You can connect `Path`s as edges on a 4-sided `ParametricSurface`. A `ParametricSurface` is a mathematical description of $(x, y[, z])$ co-ordinates based on two independent parameters ($0 \leq r, s \leq 1$). In 2D, you can build a `StructuredGrid` from a `ParametricSurface`. To do that, you need to provide information about numbers of grid points on the edges and, optionally, how those grid points are clustered. In 3D, the process is similar, only now you require a `ParametricVolume` as the mathematical description of the 6-sided gridding domain. This description of process is for a single grid. You can build up multiple-block grids by repeating this process.

Let's look at an example that builds a flow domain about a hemi-spherical geometry. We will use an axisymmetric assumption, so we only construct geometry in $x - y$ plane. This example is adapted from the Eilmer example: `2D/sphere-lehr/m355`.

lua
```
R = 7.5e-3 -- nose radius, metres
a = {x=0.0, y=0.0}
b = {x=-R, y=0.0}
```

```
 4 c = {x=0.0, y=R}
 5 d = {{x=-1.5*R,y=0.0}, {x=-1.5*R,y=R}, {x=-R,y=2*R},
   {x=0.0,y=3*R}}
 6
 7 psurf = CoonsPatch:new{
 8    north=Line:new{p0=d[#d], p1=c}, east=Arc:new{p0=b,
   p1=c, centre=a},
 9    south=Line:new{p0=d[1], p1=b},
   west=Bezier:new{points=d}
10 }
11 ni = 16; nj = 32
12 -- Shock-fitting is coordinated across four blocks.
13 registerFluidGridArray{
14    grid=StructuredGrid:new{psurface=psurf, niv=ni+1,
   njv=nj+1},
15    nib=1, njb=4,
16    fsTag="initial",
17    shock_fitting=true,
18    bcTags={west="inflow_sf", north="outflow"}
19 }
```

Recall that the `Eilmer` command for grid preparation is called `prep-grid`. We could call that with this example input file as so:

```
$ lmr prep-grid -j grid.lua
```

In this example file, we see the idea mentioned earlier of building up a grid from primitives. Points are defined on lines 2–5. Paths — here: lines, an arc and a Bézier curve — are created as needed on lines 8–9. Those paths form edges on a `ParametricSurface`; in this case, we chose a `CoonsPatch`. A `StructuredGrid` is created from the `ParametricSurface` on line 14, with numbers of vertices in the i and j logical directions chose. (Here i runs in the radial direction from shock towards body, and j runs in the tangential direction from axis towards shoulder.) This single-piece grid is shown in Figure 8. Lastly, the grids are registered for later use when specifying the flow domain. Here we use a `registerFluidGridArray` function because we want to use the shock-fitting mode. We have also chosen to split the single `StructuredGrid` into four pieces, with splits along $j =$ const lines.

There are some additional notes worth mentioning about this example. We did not use `Vector3` objects explicitly to specify points; here we preferred Lua tables as a lightweight alternative when specifying points. The reason is that for these construction points we had no intention of performing any vector operations with them. The table form is adequate and the geometry functions will convert them internally to `Vector3` objects as needed. We have also used some other pieces from the geomtry engine such as `Paths` and `ParametricSurfaces`. We do not intend to document all of those pieces in this User Guide. We refer the reader to the Geometry User Guide [13] or the `online reference manual.`

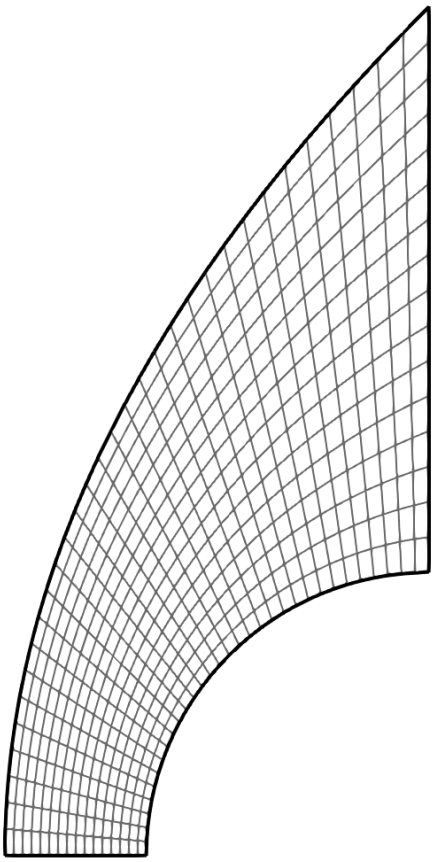


Figure 8: Single-piece structured grid for flow over a hemi-spherical geometry. This grid is built in the $x - y$ plane for later use in an axisymmetric flow simulation.

Our final comment on this example relates to our earlier point about easy parameterisation of geometry and grids. We have a simple but useful parameterisation of geometry here: we made the radius of the sphere a variable, R. This means we can re-use this grid input file for any sphere radius of interest (although one might need to adjust the inflow Bézier curve based on free-stream Mach number and the nature of the gas model). We can also change the discretisation of this grid by altering `ni` and `nj` variables which are numbers of cells in $i-$ and $j-$directions respectively. This is useful when performing a systematic grid refinement study for your flow simulation. We show more complicated examples of parametric input files in the later examples in this User Guide.

Let's now talk about building unstructured grids in an `Eilmer` input script. By comparison to structured grids, we have very few GDTk-built tools for generating unstructured grids. You could look at a program like `Gmsh` [14] for generating unstructured grids. In an `Eilmer` input file, you can convert any structured grid into an unstructured grid by using the `UnstructuredGrid` constructor. Reasons you might want to do this include: joining structured and unstructured grids; and for passing the grid off to a partitioner that works on unstructured grids. The example here shows an `UnstructuredGrid` constructor that takes a previously built structured grid with variable name `my_sgrid`.

```lua
my_ugrid = UnstructuredGrid:new{sgrid=my_sgrid}
```

For some special use cases, there is an Eilmer command-line tool that can convert multi-block structured grids into unstructured grids. You can learn more details by looking up the help for Eilmer command structured2unstructured.

```
$ lmr help structured2unstructured
```

5.3.3. Importing grids from 3rd-party tools

The second category of grids for use in Eilmer are imported grids, usually built with 3rd-party grid generators. We find that for complex 3D geometries most users prefer a GUI-style grid generator for building grids. Another reason for using a 3rd-party grid generator is when dealing directly with CAD surfaces as geometry description. Eilmer does not support reading in common CAD formats directly.

As the main route, you import grids into an Eilmer grid input file by calling an import/read function. There are many grid formats in the big world of CFD. Eilmer provides import functions for a few select grid formats. For structured grids, there are importers for GridPro and Plot3D; for unstructured grids, SU^2 format. For other grid formats, we recommend converting to one of these types if your gridding tool allows. Alternatively, you can write a converter from your gridding tool's output to Eilmer grid format. Let's look at examples of those import function calls.

Importing GridPro grids

This example file shows the import and registration of grid built with GridPro. You can find this example in the source code at: examples/lmr/3D/gridpro-import/. The import of a multi-block GridPro grid occurs in three steps:

1. import GridPro grids as a table of StructuredGrids into Eilmer;
2. a loop to register each of those structured grids; and
3. function calls to configure block connections and boundary condition labels.

lua

```
gproGrid = "blk.tmp"
gproConn = "blk.tmp.conn"
gproPty = "blk.tmp.pty"

config.dimensions = 3
scale = 1.0
grids = importGridproGrid(gproGrid, scale)

for i,g in ipairs(grids) do
   registerFluidGrid{grid=g, fsTag="initial"}
end

importGridproConnectivity(gproConn)
importGridproBCs(gproPty)
```

This script contains examples of the import functions mentioned earlier. Here we need three import functions to completely initialise multi-block GridPro grids for

use in `Eilmer`. Also note `Eilmer` only takes full-face-matching multi-block structured grids. GridPro can stitch together elemental blocks into what it calls superblocks. These superblocks are connected with cuts and often have faces that match up with multiple other blocks. These will not work in `Eilmer`.

Import Plot3D grids

The next example shows the import of a Plot3D grid. The grid is a single-piece structured grid for a flat plate, which comes from the NASA Langley turbulence model validation test cases. The interesting part related to this discussion is the `importPlot3DGrid` function call. However, the whole example script is included because it shows some `Eilmer` capability to split a grid into subgrids and then place grids in an array for parallel processing. You can find this example at: `examples/lmr/2D/flat-plate-larc-test-case/larc-grids/lmr-coarsest-grid.lua`.

lua

```lua
config.dimensions = 2
gridFile = "flatplate_clust2_4levelsdown_35x25.p2dfmt"
nptsInX = 35
nptsInY = 25
nptsOnSolidPlate = 29

-- 1. Read in grid from Plot3D format
singleGrid = importPlot3DGrid(gridFile, config.dimensions)

-- 2. Grid is a single piece. Now split at x = 0
--    We can use the information about number of points on the
--    solid plate to determine the break point in the grid.
--    Then we form two subgrids from the single grid.
nptsOnSymm = nptsInX - nptsOnSolidPlate + 1
subgrid0 = singleGrid[1]:subgrid(0, nptsOnSymm, 0, nptsInY)
subgrid1 = singleGrid[1]:subgrid(nptsOnSymm-1, nptsOnSolidPlate, 0, nptsInY)

-- 3. Use a GridArray to chop these up for parallel processing
--    on 10 processors
registerFluidGridArray{grid=subgrid0, nib=1, njb=2, fsTag='initial',
   bcTags={north='supersonic', west='supersonic', south='symm'}}
registerFluidGridArray{grid=subgrid1, nib=4, njb=2, fsTag='initial',
   bcTags={north='supersonic', east='outflow', south='fixedT'}}
identifyGridConnections()
```

Importing SU^2 grids

The SU^2 format is for unstructured grids. We do not require an import function for SU^2 format. Instead, we can read in these grids directly as part of the call to an `UnstructuredGrid` constructor. We simply have to let the constructor know that we want SU^2 format, `fmt='su2text'`. Here is an example of creating an unstructured grid based on a file called `cone20.su2`.

lua

```lua
ugrid = UnstructuredGrid:new{filename='cone20.lua',
fmt='su2text'}
```

5.4. Flow domain preparation

5.5. Post-processing at the preparation stage

Chapter Six

FLOW OVER A CONE, RELOADED

Chapter Seven

RUNNING A SIMULATION

Chapter Eight

POST-PROCESSING A SIMULATION

Chapter Nine

TUTORIAL: FLOW OVER A CONVEX RAMP

Chapter Ten

DEBUGGING A SIMULATION

Chapter Eleven

TUTORIAL: FLOW OVER A SPHERE

Part Two

ADVANCED USAGE

Chapter Twelve

ADVANCED COMPILATION

Chapter Thirteen

PARALLEL PROCESSING

Chapter Fourteen

USING HIGH-TEMPERATURE GAS MODELS

Chapter Fifteen

USING TURBULENCE MODELS

Chapter Sixteen

USER-DEFINED CUSTOMISATION

References

1. Harlow FH, Fromm JE. 1965. Computer Experiments in Fluids. *Scientific American*. 212(3):104–10
2. Whyborn L. 2023. *Direct numerical simulations of instabilities in the entropy layer of a hypersonic blunted slender cone*. PhD thesis. School of Mechanical & Mining Engineering, The University of Queensland
3. Hornung HG, Gollan RJ, Jacobs PA. 2021. Unsteadiness boundaries in supersonic flow over double cones. *Journal of Fluid Mechanics*. 916:A5
4. Damm KA, Curran DA, Gollan RJ, Veeraragavan A. 2024. Accelerating Transient Aerothermal Simulations of the BoLT-II Flight Experiment. *Journal of Spacecraft and Rockets*. 61(6):1577–91
5. Chan W, Jacobs P, Smart MK, Grieve S, Craddock CS, Doherty LJ. 2018. Aerodynamic design of nozzles with uniform outflow for hypervelocity ground-test facilities. *Journal of Propulsion and Power*. 34(6):1467–78
6. Bossel H. 1994. *Modeling and Simulation*. Wellesley, MA. 1st ed.
7. Oberkampf WL, Roy CJ. 2010. *Verification and Validation in Scientific Computing*. Cambridge University Press
8. Gibbons NN, Damm KA, Jacobs PA, Gollan RJ. 2023. Eilmer: an Open-Source Multi-Physics Hypersonic Flow Solver. *Computer Physics Communications*. 282:108551
9. Maccoll JW. 1937. The conical shock wave formed by a cone moving at high speed. *Proceedings of the Royal Society of London*. 159(898):459–72
10. Jacobs PA. 1991. Single-block Navier-Stokes Integrator. *ICASE Interim Report 18*, NASA Langley Research Center
11. Gollan R, Jacobs P. 2025. Lua and lunar re-entry: Experiences with Lua for modelling and simulation of high-speed aerodynamics. *Journal of Computer Languages*. 85:101356
12. Gollan RJ, Jacobs PA. 2017. Guide to the basic gas models package: including gas-calc and the API. *Technical Report 2017/17*, School of Mechanical & Mining Engineering, The University of Queensland
13. Jacobs P, Gollan R, Jahn I. 2025. Guide to the geometry package: for construction of flow paths. *Technical Report 2017/25*, School of Mechanical & Mining Engineering, The University of Queensland
14. Geuzaine C, Remacle J-F. 2009. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal of Numerical Methods in Engineering*. 79(11):

Index