A Guide to Compressible Flow CFD

with Eilmer4, (Guide Ver. 2.)

Toby J. van den Herik

September 2022

University of Queensland,
Centre for Hypersonics, XLabs

Context & Contributions

This digital PDF guide began as a summer research scholar project at the University of Queensland. The author's purpose was to create a stand-alone document to detail the revision of theory, installation processes, and CFD tutorials useful to PhD students in the XLabs - CFD verification should accompany experimental Hyperonic research and this guide aims to make this verification easier.

Thank you to Dr. Chris James for all of your advice and encouragements and to Dr. Kyle Damm for sharing your Eilmer4 expertise.

Thank you to Mragank Singh, Sneha Srinivasan, and Samy Ilaventhan for being initial users and stress testors of the guide.

Contents

Chapter 1

Introduction

1.1 Chapter Structure

This guide is designed to be a swift but informative walk through for using Eilmer4 to create realistic compressible flow CFD (CFCFD). The chapters are formatted beginning with two pre-simulation chapters (introductory and theory/revisionary). Following this, the chapters are aligned to the process of installing Eilmer4 and working through a simulation. Each chapter details relevant CFD theory before providing examples in Eilmer4.

References to external documentation are given as hyperlinks. This is because the guide is intended to be digital and used alongside building a CFD simulation - I believe navigating to the end of a document only to then have to go and search for a reference yourself is a waste of time and as such the hyperlinks are provided in-text. Also, please do not waste paper on my guide.

1.2 Introduction to Computational Fluid Dynamics

1.2.1 What is CFD

Computational fluid dynamics (CFD) is the application of the field of Mathematics known as Numerical Analysis to the engineering problem of solving fluid flows. The equations governing the flow of fluids (the Navier Stokes) are complex partial differential equations which remain analytically unsolved:

$$\nabla \cdot \overrightarrow{V} = 0 \tag{1.1}$$

$$\rho \frac{D\overrightarrow{V}}{Dt} = -\nabla p + \rho \overrightarrow{g} + \mu \nabla^2 \overrightarrow{V} \tag{1.2}$$

Demonstrating that any solution to these equations both exists and is unique would result in a 1 Million dollar prize - we will use them anyway. In attempting to solve them numerically, there are techniques which we can use to validate solutions.

1.2.2 The Important Steps to CFD

Performing or 'doing' CFD is characterised by three main steps:

1. **Discretisation of the Fluid Domain**

   The first step is to divide the fluid domain into cells in a meaningful way - which can be difficult. This is often a challenge when performing CFD - in fact, one must take care to effectively demonstrate that their fluid domain grid is sufficient.

2. **Discretisation of the Governing Equations**

   In this step, one must manipulate the Navier-Stokes equations into discrete form using finite difference or finite volume schemes. Fortunately, this is what CFD codes, such as Eilmer4, do for us!

3. **Solution of the set of Discretised Equations.**

   In this step, one needs to represent all the discretised Navier-Stokes approximations in linear-algebraic form and solve computationally. Again, this is handled by codes such as Eilmer4. This solving step is a topic of much importance in the field of mathematics as it can be thought of as requiring the inversion of phenomenally large and often sparse matrices - which cannot be done by elementary row operations but rather requires techniques from the field of 'Advanced Computational Linear Algebra' (ex: MATH3204 at the University of Queensland)

Additionally, one can consider there being a $\mathbf{4}^{\text{th}}$ overarching step - Iteration. One often needs to iterate their simulation through many versions with incrementally increasing complexity - gradually enabling more and more phenomena such as turbulence, chemistry, etc.

1.2.3 Eilmer4

Eilmer4 is a CFD code for compressible flows - it is optimised for this use case. This guide is centered around using Eilmer4 to build grids and perform CFD considering chemically reacting flows, catalytic surfaces, and thermo and chemical non-equilibrium. The following are some useful links for the features of Eilmer4 and the greater Gas Dynamics Toolkit (GDTk).

1. The main Eilmer4 website can be found **here** and contains a of list features.
2. The general website for the entire GDTk can be found **here** - Eilmer4 is the compressible flow CFD software within the GDTk.
3. The 'about' page for the GDTk can be found **here**.
4. The development team for the GDTk can be found **here**.

1.3 Plans for Further Additions to this Guide

This guide is always being extended and revised. The following sections are in the works.

1. Two-temperature Simulations
2. CFL and simulation stability
3. Grid independence studies
4. Eilmer4 Bunya supercomputer installation (when Bunya is made open to all UQ users)
5. Turbulence
6. y+ wall studies

Chapter 2

Revision and Theory

2.1 Hypersonic Flow and Compressibility

2.2 Continuum Flow and Altitude

The Navier-Stokes equations govern flow as a continuum - in reality fluids are comprised of molecules or atomic species. High up in the atmosphere it would not be accurate to describe fluid as a continuum instead rather it is referred to as 'free molecular flow' or rarefied gas. The Knudsen number is the dimensionless number used to determine which regime is present:

$$Kn = \frac{\lambda}{L} \tag{2.1}$$

Where λ is the mean free path of particles present and L is the characteristic length. A Knudsen number below 0.01 implies that flow is accurately represented as a continuum. The flows studied within this guide are regarded as continuum flows.

2.3 Perfect Gases

2.3.1 Calorically Perfect

A gas that is calorically perfect (ideal gas) has constant specific heats (C_v and C_p). That is, the specific heats are not a function of temperature and pressure. Generally, Earth's atmosphere can be considered ideal below altitudes of 100 km and for flows less than 1.5 kms^{-1}.

2.3.2 Thermally Perfect Gas

Definition

A gas can be treated as being thermally perfect when C_v and C_p can be accurately modelled by a function of a single temperature (one temperature common to all thermal modes - that is, thermal equilibrium - see section 2.4.4). A thermally perfect gas obeys the equation of state given as:

$$p = \rho RT \tag{2.2}$$

Limitations and Applicability Range

At high pressures (near to a gas' triple point) the Van der Waal's equation of state must be used as the gas must be treated as 'real' (see section 2.4.1):

$$p = \rho RT\left(\frac{1}{1-\beta\rho} - \frac{\alpha\rho}{RT}\right) \tag{2.3}$$

Where β and α are functions of a gas' critical temperature and pressure. Fortunately, by the nature of a critical point for gases such as O_2 and N_2, this only occurs in the negative Celsius range. Further, one must consider the other extreme of high temperatures and low pressures. If a gas dissociates or ionizes it is no longer thermally perfect. Fortunately, even if a flow mixture is reacting, the individual species in the mixture are still thermally perfect - this is a very useful fact.

A useful resource here is from the lecture series on fundamental gas dynamics at Virginia Polytechnic by Bernard Grossman - it can be found **here** - a recommended read.

2.4 High Temperature Gas Effects

2.4.1 Real vs Perfect Gas

Here, the differentiation between 'real' and 'high-temperature' gas effects is noted. Commonly, in the field of hypersonics they are mistaken as being the same notion. However, a 'real' gas complements the notion of a 'perfect' gas. That is, as a perfect gas assumes negligible intermolecular forces (Van der Waals forces) whereas a real gas does not!

2.4.2 Chemical Reactions

What is Chemistry

Chemical reactions occur when molecules collide. If molecules collide with the 'right' amount of energy, chemical bonds can be broken and new bonds can form. This is related to the activation energy, E_a. Should a mixture of gas be 'hot enough' (particles having sufficient enough energy to react), then some molecules are changed - either dissociated, recombined, or combined.

Finte Rate Reactions

Consider a high temperature mixture - the molecules and atoms are buzzing around colliding and rearranging into different molecules and atomic species. This is happening at a particular rate for each reaction that is occurring. Consider a generic reversible reaction:

$$\sum_{i=1}^{N} \alpha_i X_i \underset{h_b}{\overset{h_f}{\rightleftharpoons}} \sum_{i=1}^{N} \beta_i X_i \tag{2.4}$$

An example might be the dissociation of Nitrogen gas with some other random molecule/atom which we call a collision partner (M):

$$\mathrm{N_2} + \mathrm{M} \underset{h_b}{\overset{h_f}{\rightleftharpoons}} 2\,\mathrm{O} + \mathrm{M} \tag{2.5}$$

Note, reactions can occur forward and backward (reversible) and can do so at different rates. Much research has occurred and continues to occur in determining these rates. One form of expressing the rates is the Arrhenius form. It is suitable when reaction rate is only as a function of temperature. It is given as:

$$k = AT^n e^{-\frac{E_a}{k_B T}} \tag{2.6}$$

Where k_B is the Boltzmann constant. In Eilmer4, the form is:

$$k = AT^n e^{-C/T} \tag{2.7}$$

Where $C = E_a/k_B$. This means that to specify any reaction in Eilmer4 a user needs to know the values [A, n, C] for both the forward and backward reaction rates. Note, Eilmer4 does have pressure-dependent reaction rate models such as Troe and Lindemann-Hinshelword.

Reaction Schemes

A reaction scheme lists a set of chemical reactions (forward and backward) which are deemed to accurately encompass the significant reactions that may occur in a gas as it's temperature (and/or pressure) changes. Fundamentally, reaction schemes such as Arrhenius schemes provide $[A_f, n_f, C_f, A_b, n_b, C_b]$ for each reaction in the scheme.

Examples of reaction schemes include the Park Earth and Mars Atmosphere models (1993/1994), the Gupta air model, and the Bittker & Scullin as well as Rogers & Chinitz Hydrogen combustion models.

2.4.3 Chemical Nonequilibrium

When the reaction rates in a chemical mixture occur equally forward and backward, the net mass fractions of all the species remains constant despite the flow chemically reacting. This is called chemical equilibrium. Conversely, chemical nonequilibrium is defined as when these forward and backward reaction rates are not equal - that is, the mixture is in a state where it has only 'just' been excited and new species are being formed - here the mixture is yet to find its reacting equilibrium.

2.4.4 Thermal Nonequilibrium

Energy Storage Modes

Here, the notion of different energy modes become important. Molecules and lone atoms have the capacity to 'store' energy in different ways - each of which we refer to as a different energy mode. The modes are:

- 1. Translational - Classic notion of temperature, the kinetic energy stored by the translation (motion) of a molecule.
- 2. Rotational - Different molecules have different rotational degrees of freedom - each describing an axis about which a molecule can rotate and thus store energy (kinetic). This mode is another 'common' mode at lower temperatures.
- 3. Vibrational - Energy stored due to the oscillation of atoms in a molecule where the chemical bonds act as springs.
- 4. Electronic - One can think of electronic excitation as being the ability of electrons to 'move' to a higher (more energetic) state - in doing so they 'store' energy.

Note, lone atoms have only translational and electronic modes.

Thermal Nonequilibrium

Now, with the above notions revised, thermal non/equilibrium can be explained. First note that classical 'temperature' is the measure of the translational energy stored by a specie. Now, we simply extend this idea to the measure of temperature in other modes! That is, we state that each mode has a different temperature.

Now that we have a temperature for each mode, we can define thermal nonequilibrium as: **when the temperature is not the same for each energy mode**. Of course, in a given control volume of fluid, not every specie is at the same temperature per mode. In fact, it is either Boltzmann distributed or just chaotic - only if each mode is Boltzmann distributed can we say that it has a temperature.

2.5 Energy Transfer

As time passes, energy is exchanged between modes. Here, many important exchanges are discussed. An overarching worthwhile paper for further reading and for many constants useful to the following subsections can be found **here** - it is a NASA technical paper by Peter A. Gnoffo et. al.

2.5.1 Vibration-Translation (V-T)

The Landau-Teller model is available in Eilmer4. It is derived regarding jumps in vibrational quantum. The Anderson Jr. text on hypersonics and high-temperature gas dynamics is a good associated text for further information - find it **here**. The Landau-Teller models approximates the rate of change of vibrational energy of a particle as the mole fraction averaged sum of the linear approximation of the difference in a particles translational and vibrational energies over some relaxation time.

$$\frac{\partial e_{vp}}{\partial t} = \sum_{i=1}^{n} x_i \frac{e_{vp}^{tr} - e_{vp}^{vib}}{\tau_v^{p-i^{th}}} \tag{2.8}$$

The relaxation time, $\tau_v^{p-i^{th}}$, for the V-T mechanism is calculated by the Millikan-White empirical correlations. This correlation can be further explored **here**. Note, often in models such as Gupta these details are also presented.

2.5.2 Electronic-Translational (E-T)

For further detail beyond the two following subsections for the E-T mechanism, refer the Gnoffo et. al. technical paper given above.

Electron-Ion Collision

Collisions between ionised particles and electrons are governed differently to collisions between electrons and neutral particles. The energy exchange is approximated using Coulomb cross section.

Electron-Neutral Collision

For collisions between neutrals and electrons, Gnoffo presents a curve fit relating E-T energy exchange and electronic temperature:

$$\frac{\partial e_{ep}}{\partial t} = a_p + b_p T_e + c_p T_e^2 \tag{2.9}$$

Examples of such constants can be found in the above technical paper.

2.6 Fluid Domains

A fluid domain is a region occupied by a fluid. When building CFD simulations one cares about these domains first and foremost. In simulations where one wishes to study heat transfer from a fluid into a solid medium, solid domains are also required. To drive this point home, the fluid domain is the region where one expects fluid to flow (or where one wishes to simulate the flow of fluid through). The following is an image of the NASA X-15 in hypersonic flow - the location of the solid vs fluid domains are shown.

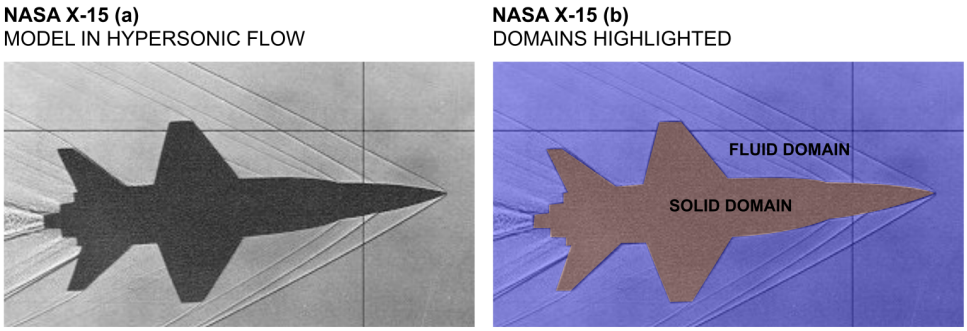

Figure 2.1: a) X-15 model in hypersonic flow. b) Solid vs fluid domain regions.

Another example is provided in the re-entry context with the Mercury space craft.

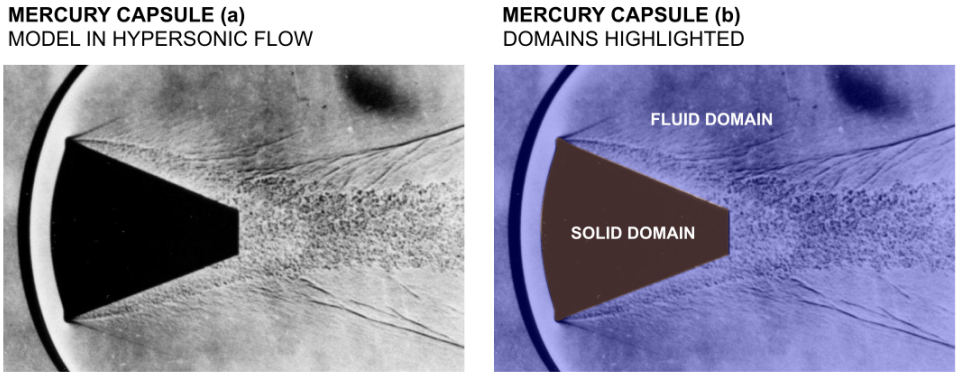

Figure 2.2: a) Mercury capsule model in hypersonic flow. b) Sold vs fluid domain regions.

Chapter 3

Installations and Setup

3.1 Ubuntu (A version of Linux)

3.1.1 The Options

The options for installing a version of ubuntu (Linux) are outlined in this section. The key recommended method (Virtual Box with ubuntu on Windows (section 3.1.4) is given in full detail. Reference material for the other methods are provided as hyperlinks.

The options are:

1. **Lone ubuntu OS Install** (section 3.1.2) where one machine has only ubuntu installed.
2. **Dual-boot with Windows** (section 3.1.3) where one machine allows the user to choose from ubuntu or windows 10 on startup.
3. **Windows Subsystem for Linux** (section 3.1.4) where a command line only (initially) version of Linux is installed in windows. This can be a challenge - particularly for new-comers.
4. **Virtual Machine - Virtual Box (RECOMMENDED)** (section 3.1.5) where the virtual box software is installed on windows and allows the user to operate a virtual ubuntu computer.
5. **Virtual Machine - Windows Hyper-V** (section 3.1.6) similar to Virtual Box but only accessible by users with professional and enterprise versions of windows.

3.1.2 Lone ubuntu OS Install

Installing ubuntu (20.04 +) on your machine can be done quite easily. The user is referred to Ubuntu's **website** for this process. Note: this method of Ubuntu install resets one's machine.

3.1.3 Dual-boot with Windows

Dual booting is a popular method for installing ubuntu as the user can choose which OS to run upon computer startup. However, one must note it is possible to make a mistake with windows BitLocker encryption which can result in losing all existing files on the computer.

The link **here** has proven to be a successful guide for dual booting ubuntu alongside windows - however does not discuss the problem of BitLocker encryption from above.

3.1.4 Windows Subsystem for Linux (WSL)

Installing WSL is quite straight forward. However, installing all the prerequisite software on WSL, allowing cross-OS permissions, and getting third-party software such as paraview (for flow field and grid viewing) to work is tricky. I will not cover it and recommend that a virtual box or a lone ubuntu install is used for new-comers - at least initially, this will allow you to get used to the command line.

3.1.5 Virtual Machine - Oracle Virtual Box (RECOMMENDED for New-comers)

Installing ubuntu on a virtual box is the recommended method for new-comers. This method (unlike the lone ubuntu method) will not require you to reformat your machine. Begin this install process on your windows machine.

Download and Install Virtual Box

Download virtual box from the official link **here**. Find the Virtual Box 6.1.26 option and select 'windows hosts'. Note: we are using an older version of Virtual Box - this is because, as of writing (December 2021) all new versions have a significant unresolved memory issue and will not work for your purposes.

Run the downloaded Oracle Virtual Box 6.1.26 executable file and follow the installation prompts. If asked, tick the 'Always trust software from Oracle Corporation' and select install. After the install is complete you must restart your system.

Ubuntu in Oracle Virtual Box

You must now download the latest ubuntu OS (an iso file) from **here** (it is roughly 3 Gigabytes in size) which will be used to install ubuntu into the virtual box. The Ubuntu 20.04 LTS version is recommended as it is well supported. Having downloaded the file, follow the following steps in the Virtual Box software.

1. Select 'New'. Give your OS a name such as "Ubuntu 20.04".
2. Specify a location to store all the files (machine folder) - best left as default.
3. Select the type to be 'Linux'.
4. Select the version 'Ubuntu 64bit'. Select 'Next'.
5. Set the memory which you will allow your ubuntu to use - be generous. Select 'Next'.
6. Select 'create a virtual hard disk now'. Select 'create' and choose VDI. Select 'Next'.
7. Choose dynamically allocated storage as this will allow you to expand it later if needed. Select 'Next'.
8. Set the amount of Gigabytes allocated to the storage (I would suggest 30 Gb initially). Select 'Create'.

We need to now install ubuntu in the virtual machine. Follow these instructions:

11. Select 'settings' and go to 'Storage'. Then, under storage devices, select 'empty'.
12. Under attributes click the blue disk and select 'choose a disk file...'.
13. Find the ubuntu iso file (approx. 3 Gigabytes), select the file and then select 'Open'.
14. Press 'okay' and then start your virtual machine with the large green arrow.

Wait for your machine to boot for the first time and then follow all the first ubuntu installation process.

3.1.6 Virtual Machine - Windows Hyper-V

Windows Hyper-V (Viridian) is a virtual machine tool provided by Microsoft. A tutorial for how to create an ubuntu 20.04 machine in Windows 10 is provided **here** and Windows 11 **here**. *Note, Hyper-V is only available on Windows Pro, Education, or Enterprise.*

3.2 Eilmer4 (on Ubuntu)

The Gas Dynamics Toolkit ***website*** does give documentation for the installation of Eilmer4 and associated code's prerequisite software - it can be found ***here***. The below is a rapid run through of the installation process on an ubuntu machine.

3.2.1 Prerequisite Software for GDTk

This section will walk through the installation of all relevant GDTk prerequisite software.

Updating the APT

The below installations will make use the Linux *advanced package tool* (apt). This tool has a list of authenticated sources from which it may retrieve packages and install them upon your computer. First, we will make sure the apt has an up-to-date list of available packages. Open a new termal and run:

```
sudo apt-get update
```

'sudo' Privileges

Here, 'sudo' means 'super used do'. It is a strange saying but essentially means that you wish for the subsequent command to be run by a superior user (by the admin with 'root' privileges). The first time it is used you will be required to input an admin passcode.

LDC2 Download

LDC2 is a 'linux D compiler' (number 2). D is the computer programming language that Eilmer4 was written in - as such, your machine will need this compiler if you wish to use these codes.

Go to the link ***here*** and download the file titled **ldc2-?.??.?-linux-x86_64.tar.xz** from the assets section (a stable build version). The following commands will install this compiler. For greater detail on the installation process visit the Eilmer4 website ***here***.

LDC2 Installation

Make an */opt* directory to install LDC2:

```
cd
mkdir -p opt/ldc2
```

Note: the */opt* directory usage is a legacy idea from an age when computers were installed by vendors and certain packages were 'optional'. The following command will unpack and move the downloaded file (presumed to be in your Downloads folder):

```
tar -Jxvf Downloads/ldc2-?.??.0-linux-x86_64.tar.xz -C opt/ldc2 --strip-components=1
```

Finally, open your computer's (ubuntu) *.bash_aliases* file:

```
cd
gedit .bash_aliases
```

In the file we need to add an alias (insert this line anywhere in the file):

```
1 export PATH=${PATH}:${HOME}/opt/ldc2/bin
```

Save and close. Now run these commands in the terminal:

```
cd
. .bash_aliases
```

This has registered the updated file. Now, any program on your machine now looking for the D compiler will know where to find it.

Ubuntu build-essentials

On an ubuntu machine there is a fantastic installation we can use to get many of the packages required for the GDTk all in one go - it is called 'build-essentials'. This includes tools such as:

1. makefile (or 'make')
2. gcc (which includes a C++ compiler)

Install it with:

```
sudo apt install -y build-essential
```

Fortran Compiler

To install this run both:

```
sudo apt-get install -y gfortran-multilib
```

and

```
sudo apt-get install -y gfortran
```

GIT

Git should exist by default on an ubuntu machine. If not,

```
sudo apt install -y git
```

readline-dev

This library is to do with interpreting and reading the command line, install it with:

```
sudo apt-get install -y libreadline-dev
```

libncurses5-dev

Install this with:

```
sudo apt-get install -y libncurses5-dev
```

OpenMPI

This is a tool that will allow you to use the Eilmer4 program 'e4mpi'. It is for parallel processing. Install it with,

```
sudo apt-get install -y libopenmpi-dev
```

libplot-dev

Install this with:

```
sudo apt-get install -y libplot-dev
```

Python3 and Python Foreign Functionality Integration

Ubuntu 20.04 (etc) will come with Python 3 pre-installed. Should you not have python,

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt-get update
sudo apt-get install python3.10
```

It is also useful to have **pip** for easy python module installation:

```
sudo apt install -y python3-pip
```

Now, install the foreign function functionality,

```
sudo apt-get install -y python3-cffi
```

With this, the prerequisite software are installed upon your ubuntu machine and we can move on to installing Eilmer4.

3.2.2 Eilmer4 Installation

Assuming now that you have a functioning Ubuntu operating system and all the prerequisite software installed (subsection above), we can install Eilmer4 and the other GDTk tools with it.

Clone the GDTk

We shall install Eilmer4 in the /home/ directory. In a terminal, run the following.

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

This command has created a new file called 'gdtk' in the /home/ directory. Here, it has placed the repository containing the source code and examples.

Making Eilmer4

We now need to compile Eilmer4. This will be done in a new file called 'gdtkinst'. It is here that the user can specify the 'flavour' and 'settings' of our Eilmer4 version. Starting simple it is recommended that the user install the default Eilmer4. To install Eilmer4 first navigate to the Eilmer source file:

```
cd gdtk/src/eilmer
```

Then, use the makefile functionality:

```
make install
```

This will install Eilmer4. However, we must now add the following to the .bash_aliases file so that the Eilmer4 software can be located on the computer:

```
1 export DGD=$HOME/gdtkinst
2 export DGD_REPO=$HOME/gdtk
3 export PATH=$PATH:$DGD/bin
4 export DGD_LUA_PATH=$DGD/lib/?.lua
5 export DGD_LUA_CPATH=$DGD/lib/?.so
```

Gas Packages in Python

To use the loadable python libraries (or RUBY) add the following to the .bash_aliases file also:

```
export PYTHONPATH=${PYTHONPATH}:${DGD}/lib
export RUBYPATH=${RUBYPATH}:${DGD}/lib
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${DGD}/lib
```

Installing Eilmer4 with Added Features

Note: this subsection is not required. Having finished the above subsections of 3.2.2, the user should now have a working Eilmer4. This section simply discuss alternative 'make install' methods.

Eilmer can be installed with three flavours using this command:

```
make FLAVOUR=? install
```

The default flavour is 'debug' and is recommended for new-comers.

Should you wish to install Eilmer4 on a different version of linux (not ubuntu) you will need to specify the platform with:

```
make PLATFORM=? install
```

For example, installing on a macintosh (apple) computer the command would be:

```
make PLATFORM=macosx install
```

Finally, to explore all the options available when making Eilmer, see the build options **here**

3.3 Paraview

Paraview is an easy to use flow field visualisation tool and, if you do not wish to write your own scripts, can allow you to plot or extract information over time or distance with ease. On an ubuntu machine installing paraview is easy - simply run:

```
sudo apt install paraview
```

3.4 Useful References

You now have the majority of the Eilmer4 tools which you will need. You can find many examples of Eilmer4 simulations in the '/gdtk/examples/eilmer/' file. I will quickly recommend a few useful go-to pages on the GTDk website for new Eilmer4 users.

1. User Guides can be found **here** for general Eilmer4, Geometry, and Reacting Gas Thermochemistry.
2. A quick-reference for simulation options available to the user can be found **here**.
3. Technical notes on interesting topics can be found **here**. And,
4. Frequently Asked Questions can be found **here**

3.5 Installation on Supercomputers/HPC

Previously, this section detailed the setup of Eilmer4 on the UQ Tinaroo supercomputer. As the University of Queensland is in the process of replacing all of it's supercomputers with a new supercomputer, dubbed 'Bunya', no details about HPC installation is currently available.

Chapter 4

Fundamental Hypersonic Geometries

4.1 Tutorial Geometries and Simple Simulations

This chapter offers examples on constructing geometries with Eilmer4. The examples given are to demonstrate how parametric and data-point based geometry can be built. All of these fundamental geometry simulations will use an ideal air gas model. Reacting gas models will be introduced in chapter 5. Therefore, it must be noted that any flow solutions presented in this chapter are not strictly considered sensible.

Note: the first example (wedge) is set out as a complete tutorial. The following examples simply detail the geometry and relevant .lua custom functions.

4.2 Wedge Geometry

The first geometry to practise is a simple wedge. The wedge is characterised by a base length, d, and angle, θ.

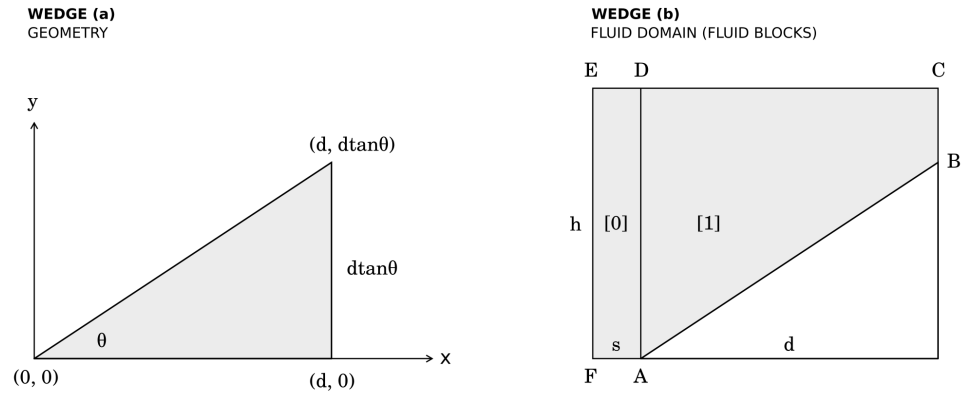

Figure 4.1: a) The geometry of a simple wedge parameterised by d and θ. b) The division of the fluid domain into two blocks labelled starting from [0]. Vertices are represented as capital letters. Lengths are represented in lowercase.

4.2.1 Making the Directory and a Gas File

Begin by creating a new directory (folder) on your machine and navigating to it. This is done with the following terminal commands:

```
mkdir "Wedge Project"
cd "Wedge Project"
```

These above two commands run separately within the terminal and will create a folder named "Wedge Project" and then navigate into said folder. Now, a simple ideal air gas model can be built. Create a new *ideal-gas-air.inp* file and open it. The file will need to include the following 2 lines.

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

Close this file and return to the terminal. Generate the gas file with the command,

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

This has built a gas file (from the Eilmer4 database) named *ideal-air.lua*. Within this gas file, the gas species name is defined as 'air' and has common properties such as $\gamma = 1.4$. Also, a Sutherland's law model is specified for the calculation of viscosity and thermal conductivity. The simulation will use this in run time.

4.2.2 The Main Project File (wedge.lua)

Now that a gas file is defined, the main project simulation file (including the geometry definition) can be created.

Preliminary Configuration

Create a file with a title of your choosing - here it will be named *wedge.lua*. We begin with simple Eilmer4 configuration settings:

```
-- Wedge Project Eilmer4
-- Ideal Air
-- Preliminary Configuration
config.title = "Flow over a Wedge"
config.dimensions = 2
nsp, nmodes, gmodel = setGasModel("ideal-air.lua")
```

Pausing here, we have defined the title, number of dimensions, and the gas model. Importantly, line 6 has done 4 different things:

1. Set the gas model to ideal air for use by the simulation at run time
2. Returned the value, nsp, specifying the number of chemical species
3. Returned the value, nmodes, specifying the number of non-equilibrium energy modes (zero for our ideal-air example)
4. Returned the lua reference to the gas model object (instance of a class)

Whilst point 1 is necessary, points 2-to-4 are done by convention as the user may wish to have this information when performing 'more complex' simulations.

Flow States

The flow in this example will be defined as having x-velocity of 1500 ms^{-1}, pressure 90 kPa, and temperature 1100 K. Note, this is a made-up arbitrary flow state just for demonstration. The code:

```
7  -- Flow Parameters for free stream (inf) and cell initialisation (init)
8  P_init = 5000 -- Pa
9  T_init = 300 -- K
10 P_inf = 90e3 --Pa
11 T_inf = 1100 -- K
12 Vel = 3000 -- m/s
13 initial = FlowState:new{p=P_init, T=T_init}
14 inflow = FlowState:new{p=P_inf, T=T_inf, velx=Vel}
```

Lines 8-to-12 have created lua variables and assigned them a value. Lines 13 and 14 have used the Eilmer4 FlowState class to build an initial and inlet flow condition for the simulation.

Building Geometry

Firstly, the parameters of the geometry and flow field must be defined. An arbitrary wedge of base length 170 mm and angle 20 degrees is chosen:

```
15 -- Geometry and Fluid Domain Parameters
16 s = 20/1000 -- 20 mm given as m
17 d = 170/1000 -- 170 mm given as m
18 theta = 20 -- degrees
19 h = 150/1000 -- 150 mm given as m
```

Next, the points A, B, ..., F will be specified as Vector3 objects (equivalent to point representation) based upon figure 4.4:

```
20 -- The points (vertices) of the fluid domain blocks
21 A = Vector3:new{x = 0.0, y = 0.0, z = 0.0}
22 B = Vector3:new{x = d, y = d*math.tan(math.rad(theta)), z = 0.0}
23 C = Vector3:new{x = d, y = h, z = 0.0}
24 D = Vector3:new{x = 0.0, y = h, z = 0.0}
25 E = Vector3:new{x = -s, y = h, z = 0.0}
26 F = Vector3:new{x = -s, y = 0.0, z = 0.0}
```

Next, the lines defining the boundary of the fluid blocks, [0] and [1], are required. The nature of the mathematics for parametric patching mandates that the north and south boundaries be defined left to right (along x) and that the east and west boundaries be defined bottom to top (along y).

```
27 --The lines (edges) of the fluid blocks
28 AB = Line:new{p0 = A, p1 = B}
29 BC = Line:new{p0 = B, p1 = C}
30 DC = Line:new{p0 = D, p1 = C}
31 AD = Line:new{p0 = A, p1 = D}
32 ED = Line:new{p0 = E, p1 = D}
33 FA = Line:new{p0 = F, p1 = A}
34 FE = Line:new{p0 = F, p1 = E}
```

The Surfaces, Grids, and Fluid Blocks

Next, the parametric surfaces are made (sometimes referred to as quads or patches). It is useful when making simulations containing many surfaces/grids/blocks to store them in a .lua table. As such, we will do this here to teach good habits. First, create an empty table called *surfs*:

```
35 -- Parametric Surfaces
36 surfs = {}
```

Now, we can populate the 0th and 1st entries in this table with the patches (see figure 4.4):

```
37 surfs[0] = makePatch{north = ED, east = AD, south = FA, west = FE}
38 surfs[1] = makePatch{north = DC, east = BC, south = AB, west = AD}
```

Two important points arise here:

1. The *makePatch* functionality does not require a *:new* - the function itself calls either *AOPatch:new* or *CoonsPatch:new* when called.
2. The default patch type is *CoonsPatch* - which is what is generated in our case.

Now the grids can be build. In this example no grid clustering (bunching) will be used. Importantly, how we wish to divide our patches up in x and y directions must be specified. We create some lua variables which define the number of cells we wish to have in each direction in each grid.

```
39 -- Making the grids
40 N_in_x0 = 8 -- Number of cells in x (grid 0)
41 N_in_x1 = 60 -- Number of cells in x (grid 1)
42 N_in_y = 60 -- Number of cells in y (shared by both grids)
43
44 grids = {}
45 grids[0] = StructuredGrid:new{psurface=surfs[0], niv = N_in_x0 + 1, njv = N_in_y + 1}
46 grids[1] = StructuredGrid:new{psurface=surfs[1], niv = N_in_x1 + 1, njv = N_in_y + 1}
```

Of note here is that niv, etc is set with a +1 term. This is because the *StructuredGrid:new* takes the number of nodes (one more then the number of cells). This is of importance as literature often refers to the number of cells a CFD simulation had rather than the nodes.

Next, the the fluid blocks are defined:

```
47 -- The fluid blocks
48 blocks = {}
49 blocks[0] = FluidBlock:new{grid=grids[0], initialState=initial}
50 blocks[1] = FluidBlock:new{grid=grids[1], initialState=initial}
```

Note: both fluid blocks have an initial fluid state defined as *initial* as per line 13.

Next, we must inform Eilmer4 that we wish for it to find the connections of our blocks and link them. We use:

```
51 identifyBlockConnections()
```

All this hard work leads to Eilmer4 having a complete specification of the fluid domain through which we wish to simulate flow. Discussion on how to view the grid will occur below.

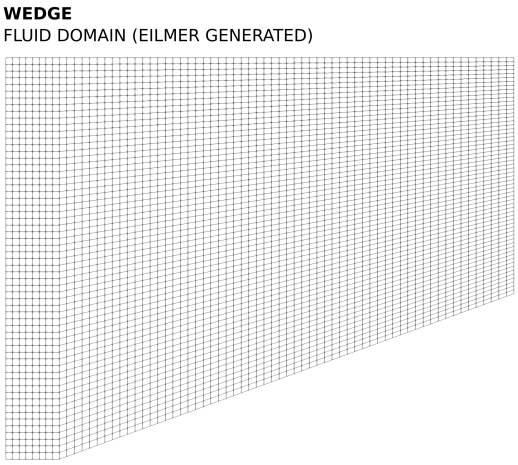

Figure 4.2: The generated simple wedge grid.

Boundary Conditions

For this simulation the most simple of boundary conditions are introduced. The code used is:

```
52 -- The Boundary Conditions
53 blocks[0].bcList[west] = InFlowBC_Supersonic:new{flowCondition=inflow}
54 blocks[1].bcList[east] = OutFlowBC_Simple:new{}
```

This sets the west face of block [0] to be a supersonic inflow boundary with the free stream conditions used in line 14. The east face of block [1] is specified as a simple flux condition ideal for having flow exit the domain with supersonic conditions. The remaining boundary conditions are left as the 'default condition' which Eilmer4 assumes to be the *WallBC_WithSlip* condition. This details a solid boundary which erases the normal component velocity.

Note: for future reference, this method of programming the B.C. will not be possible for Fluid Block Array types in Eilmer4, instead use the B.C. setting method detailed in the cylinder geometry example.

Finalising Simulation Configuration

Now the settings of the simulation are specified:

```
55 -- The Final Config
56 config.max_time = 5e-3
57 config.max_step = 3000
58 config.dt_init = 1e-6
59 config.dt_plot = 1.5e-3
```

The *max_time* specifies how much time the simulation should simulate for (not real world time) befor terminating. Likewise, the *max_step* specifies how many steps the simulation show take in total before terminating. Finally, *dt_plot* is the interval at which the simulation will save the flow fields for plotting.

4.2.3 Executing Simulations and Viewing Results

Now that all the code is written, the simulation can be prepped, ran, and visualied.

Prep and Run

The simulation can now be prepared and run with the following terminal commands. You do not need the .lua file name extension.

```
e4shared --job=wedge --prep
e4shared --job=wedge --run
```

Visualising the Results

To achieve this the post-processing functionality of Eilmer4 is used. Specifically, the post-processing function to write all the flow fields to paraview format. The terminal command is:

```
e4shared --job=wedge --post --vtk-xml --tindx-plot=all --add-vars="mach"
```

This specific command will create a new directory called *plot* inside which a 'wedge.pvd' file will be created to be viewed by paraview. By default the *–vtk-xml* functionality does not write Mach number that is why it has been added specifically. To view the solution in paraview:

```
paraview ./plot/wedge.pvd
```

The results for mach and pressure follow below.

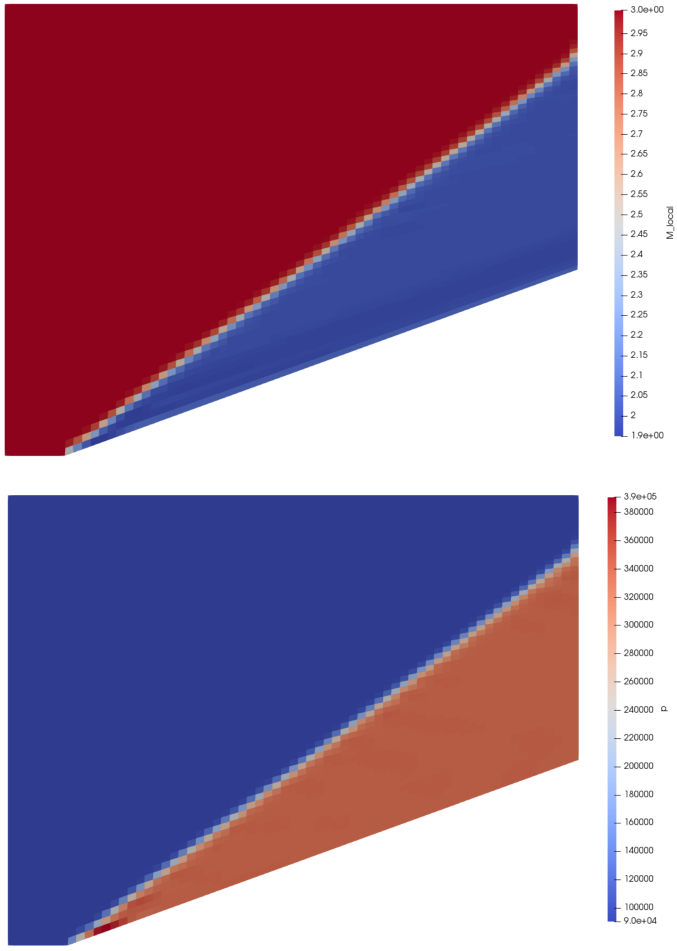

Figure 4.3: Mach and Pressure Fields for the 20 degree wedge.

Final Wedge Notes

This simulation took approximately 94 seconds to run on the full spec Windows Surface Book 3, 2021. Of course, wanting finer grids and higher quality will require greater computation time. Importantly, the grid used in this simulation has not been validated as 'correct' - a good CFD user will need to perform grid-independence studies - this will be discussed later on.

4.3 Cone

A cone is simply an axisymmetric wedge model - also parameterised by d and θ.

4.3.1 Preliminary Configuration

Simply insert the following into your code for the *wedge.lua* model to turn it into an axisymmetric model about $y = 0$ (a 20 degree cone).

```
6 config.axisymmetric = true
```

4.4 Double Wedge Airfoil

One possible parameterisation of a supersonic wing is by a chord length, c, and angle θ.

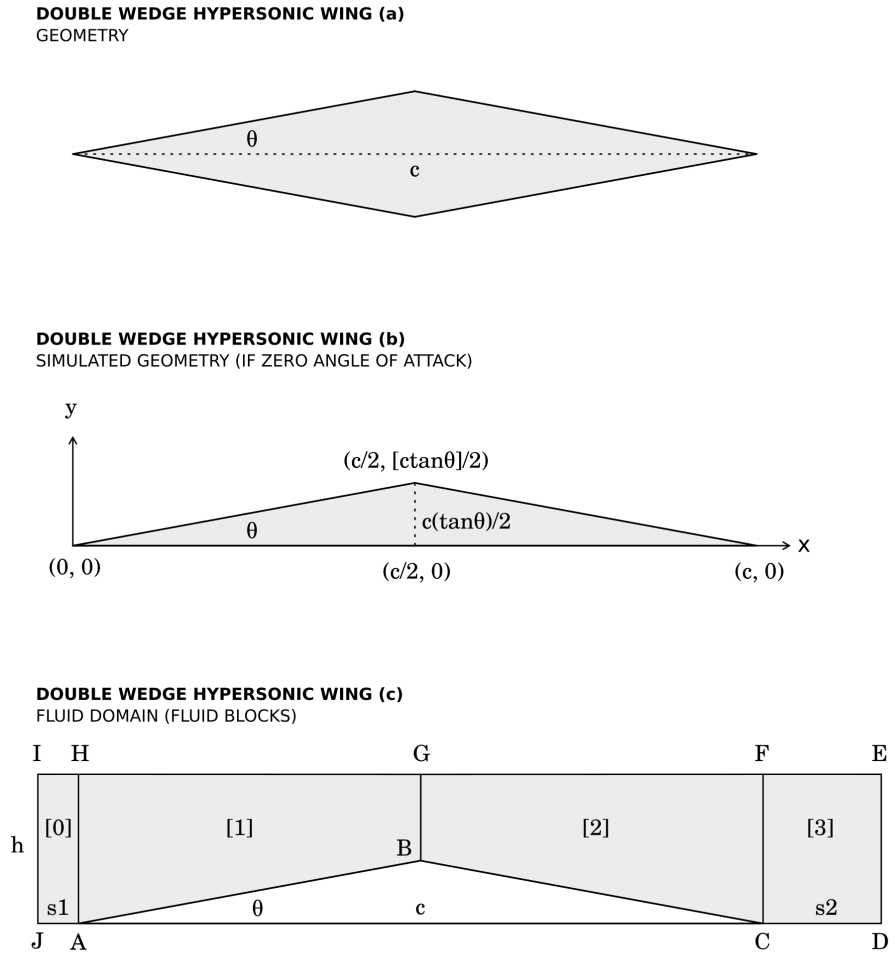

Figure 4.4: a) The geometry of a hyersonic wing parameterised by c and θ. b) The half-wing required for a zero angle of attack simulation. c) The division of the fluid domain into two blocks labelled starting at 0. Vertices represented as capital letters. Lengths represented as lowercase.

4.4.1 Building the Geometry

As should become typical of any Eilmer4 2D CFD simulation, begin by defining the parameters of the geometry and the fluid domain:

```
-- Geometry and Fluid Domain Parameters
h = 0.2 -- m
c = 1 -- m
s1 = 0.05 -- m
```

```
s2 = 4*s1 -- m
theta = 10 -- degrees
```

Next, the points are defined:

```
-- The points (vertices) of the fluid domain blocks
A = Vector3:new{x = 0.0, y = 0.0, z = 0.0}
B = Vector3:new{x = c/2, y = 0.5*math.tan(math.rad(theta)), z = 0.0}
C = Vector3:new{x = c, y = 0.0, z = 0.0}
D = vector3:new{x = c + s2, y = 0.0, z = 0.0}
E = Vector3:new{x = c + s2, y = h, z = 0.0}
F = Vector3:new{x = c, y = h, z = 0.0}
G = Vector3:new{x = c/2, y = h, z = 0.0}
H = Vector3:new{x = 0.0, y = y = h, z = 0.0}
I = Vector3:new{x = -s1, y = h, z = 0.0}
J = Vector3:new{x = -s1, y = 0.0, z = 0.0}
```

Now, the lines must be defined for the boundary of each fluid block:

```
-- Lines
-- The horizontal lines
JA = Line:new{p0 = J, p1 = A}
AB = Line:new{p0 = A, p1 = B}
BC = Line:new{p0 = B, p1 = C}
CD = Line:new{p0 = C, p1 = D}
IH = Line:new{p0 = I, p1 = H}
HG = Line:new{p0 = H, p1 = G}
GF = Line:new{p0 = G, p1 = F}
FE = Line:new{p0 = F, p1 = E}
-- The vertical lines
JI = Line:new{p0 = J, p1 = I}
AH = Line:new{p0 = A, p1 = H}
BG = Line:new{p0 = B, p1 = G}
CF = Line:new{p0 = C, p1 = F}
DE = Line:new{p0 = D, p1 = E}
```

Once again, we define surfaces, grids, and fluid blocks:

```
-- Defining the surfaces
surfs = {}
surfs[0] = makePatch{north = IH, south = JA, east = AH, west = JI}
surfs[1] = makePatch{north = HG, south = AB, east = BG, west = AH}
surfs[2] = makePatch{north = GF, south = BC, east = CF, west = BG}
surfs[3] = makePatch{north = FE, south = CD, east = DE, west = CF}
-- Defining the grids
N_x0 = 5; N_x1 = 30; N_x2 = N_x1; N_x3 = math.ceil(N_x0*s2/s1)
N_y = 20
grids = {}
```

```
grids[0] = StructuredGrid:new{psurface=surfs[0], niv = N_x0, njv = N_y}
grids[1] = StructuredGrid:new{psurface=surfs[1], niv = N_x1, njv = N_y}
grids[2] = StructuredGrid:new{psurface=surfs[2], niv = N_x2, njv = N_y}
grids[3] = StructuredGrid:new{psurface=surfs[3], niv = N_x3, njv = N_y}
-- Fluid Blocks
blocks = {}
blocks[0] = FluidBlock:new{grid=grids[0], initialState = inflow}
blocks[1] = FluidBlock:new{grid=grids[1], initialState = initial}
blocks[2] = FluidBlock:new{grid=grids[2], initialState = initial}
blocks[3] = FluidBlock:new{grid=grids[3], initialState = initial}
-- Connect the Blocks
identifyBlockConnections();
```

From here, boundary conditions can be built and the simulation run - simple B.C. could be used such as in the wedge example for preliminary study of ideal-gas flow. The Eilmer4 generated grid with the given grid divisions and parameters follows in figure 4.5

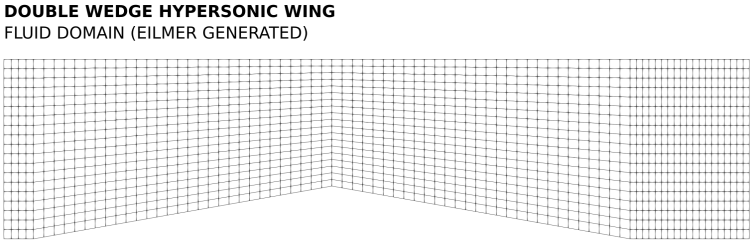

Figure 4.5: The generated supersonic wing grid.

4.4.2 Example Pressure Field

As a point of comparison, a 4000-iteration pressure field is provided for the following conditions:

```
-- Flow Parameters for free stream (inf) and cell initialisation (init)
P_init = 5000 -- Pa
T_init = 300 -- K
P_inf = 90e3 --Pa
T_inf = 1100 -- K
Vel = 1300 -- m/s
initial = FlowState:new{p=P_init, T=T_init}
inflow = FlowState:new{p=P_inf, T=T_inf, velx=Vel}
```

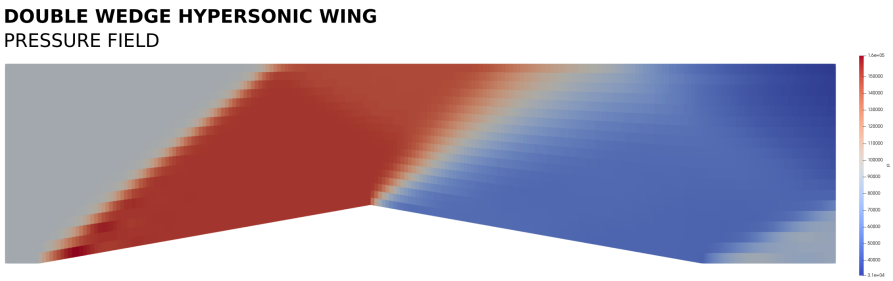

Figure 4.6: The pressure field of the supersonic wing.

4.5 Cylinder

A cylinder section is characterised by the radius, r:

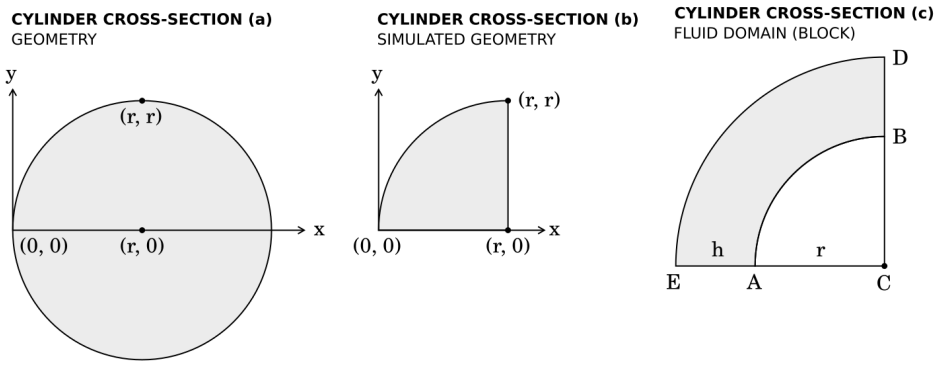

Figure 4.7: a) The geometry of a cylinder cross-section (a circle). b) The portion of the cylinder to be modelled. c) The fluid domain.

4.5.1 Building the Geometry

The fluid domain extents are defined by r and h (figure 4.7). This fluid domain geometry (when fixed) is a poor choice and will require shock fitting. Therefore this section will include and introduction to shock-fitting (changing the grid to include the shock in this case).

```
8  -- Geometry and Fluid Domain Parameters
9  r = 1 -- m
10 h = 0.75*r -- m
11 -- Grid Division
12 Nx = 2*15
13 Ny = 2*40
14 -- The points (vertices) of the fluid domain blocks
15 A = Vector3:new{x = 0.0, y = 0.0, z = 0.0}
16 B = Vector3:new{x = r, y = r, z = 0.0}
17 C = Vector3:new{x = r, y = 0.0, z = 0.0}
```

```
D = Vector3:new{x = r, y = r+h, z = 0.0}
E = Vector3:new{x = -h, y = 0.0, z = 0.0}
-- Fluid Block Edges
-- The Lines
DB = Line:new{p0 = D, p1 = B}
EA = Line:new{p0 = E, p1 = A}
-- The Arcs
AB = Arc:new{p0 = A, centre=C, p1 = B}
ED = Arc:new{p0 = E, centre=C, p1 = D}
-- The Surface, Grid, and Fluid Block
surf = makePatch{north = DB, south = EA, east = AB, west = ED}
grid = StructuredGrid:new{psurface = surf, niv = Nx, njv = Ny}
block = FluidBlock:new{grid = grid, initialState = initial}
block.bsList[west] = InFlowBC_Supersonic:new{flowState=inflow}
block.bcList[north] = OutFlowBC_Simple:new{}
```

4.5.2 Example Pressure Field

The generated grid and rough simulated pressure distribution are found for the following conditions:

```
-- Flow Parameters for free stream (inf) and cell initialisation (init)
P_inf = 101000 -- Pa
Vel = 2000 -- m/s
T_inf = 300 -- K
-- Conditions / States
initial = FlowState:new{p=P_inf/3, T=T_inf/2}
inflow = FlowState:new{p=P_inf, T=T_inf, velx=Vel}
```

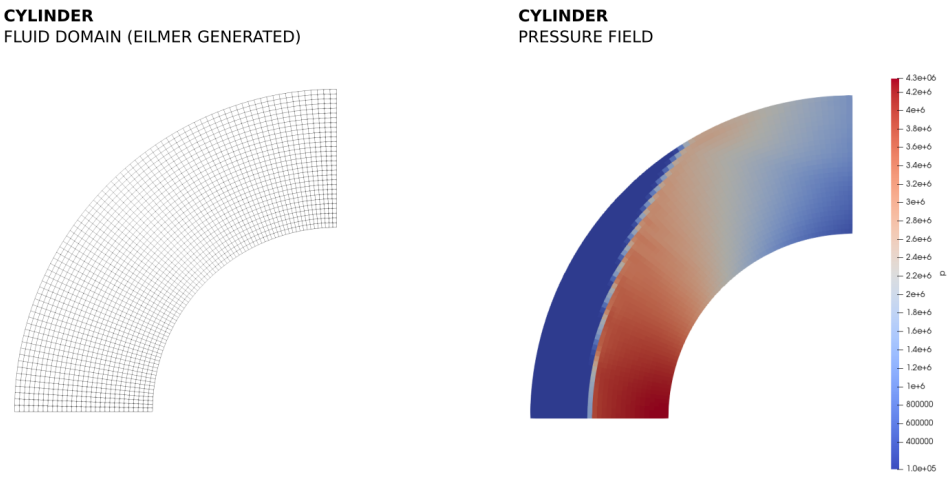

Figure 4.8: The grid generated by Eilmer4 and a rough pressure field.

Note: from the pressure field figure, it is observed that the west face does not contain the shock wave. Further, all cells pre-shock (dark blue) required unnecessary compute power. As such, we will attempt shock fitting - shock fitting will reshape the grid such that the edge of the grid becomes aligned to the location of the shock.

4.5.3 Introduction to Shock-fitting

To achieve shock fitting, we will require a 'Shock-fitting boundary condition'. This B.C. mandates that the fluid block instead be specified as a 'FBArray:new' (a.k.a a Fluid Block Array). Fundamentally, a fluid block array divides the grid into specified divisions when created:

```
BCs = {
      west = InFlowBC_ShockFitting:new{flowState=inflow},
      north = OutFlowBC_Simple:new{}
    }
block = FBArray:new{grid = grid, initialState = initial, nib = 3, njb = 2, bcList=BCs}
```

Here the fluid block array has divisions specified as 2×3.

Further, we must define some relevant settings for a moving grid:

1. A moving grid gas dynamics update scheme
2. Grid motion
3. A shock-fitting delay (a period of time to wait before attempting to fit the shock).

In Eilmer4, these manifest as:

```
Charac_Time = (r*2)/Vel
config.gasdynamic_update_scheme = "backward_euler"
config.grid_motion = "shock_fitting"
config.shock_fitting_delay = Charac_Time
```

Note, the characteristic time, *Charac. Time.* is a good estimation of the time required for the flow to pass-over the cylinder cross-section and for the shock to begin to form - after which Eilmer4 will engage its shock-fitting algorithm and begin to reshape the fluid domain.

Over the simulation time the grid reshapes as follows.

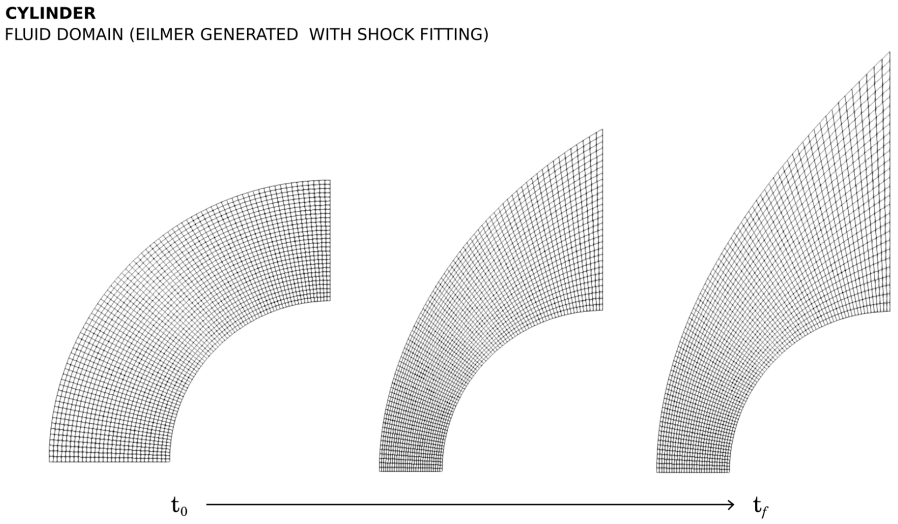

Figure 4.9: An adapting (shock-fitting) grid - converging as simulation progresses.

The final pressure field (rough grid as shown in 4.9) follows:

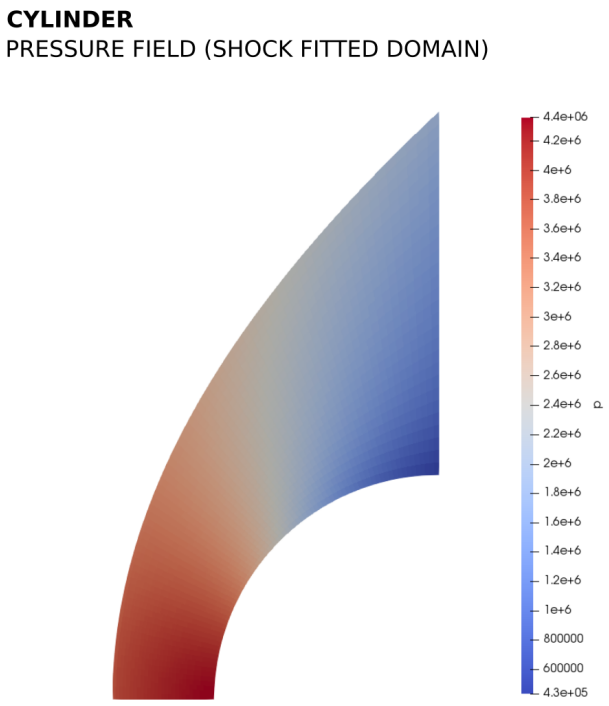

Figure 4.10: The pressure field post-shock. The west boundary has adapted to align with the shock location implying all observed flow is post-shock.

4.6 Data-point Based Geometries: Stardust Sample Return Capsule Shield

If one has a CAD drawing of the model they wish to use for CFD, coordinates can simply be taken from this CAD model and fluid domains built in the following way. Note, building parametric grids is more ideal (see the Hayabusa example in the following section). Now, consider the Stardust return capsule as defined by a set of points:

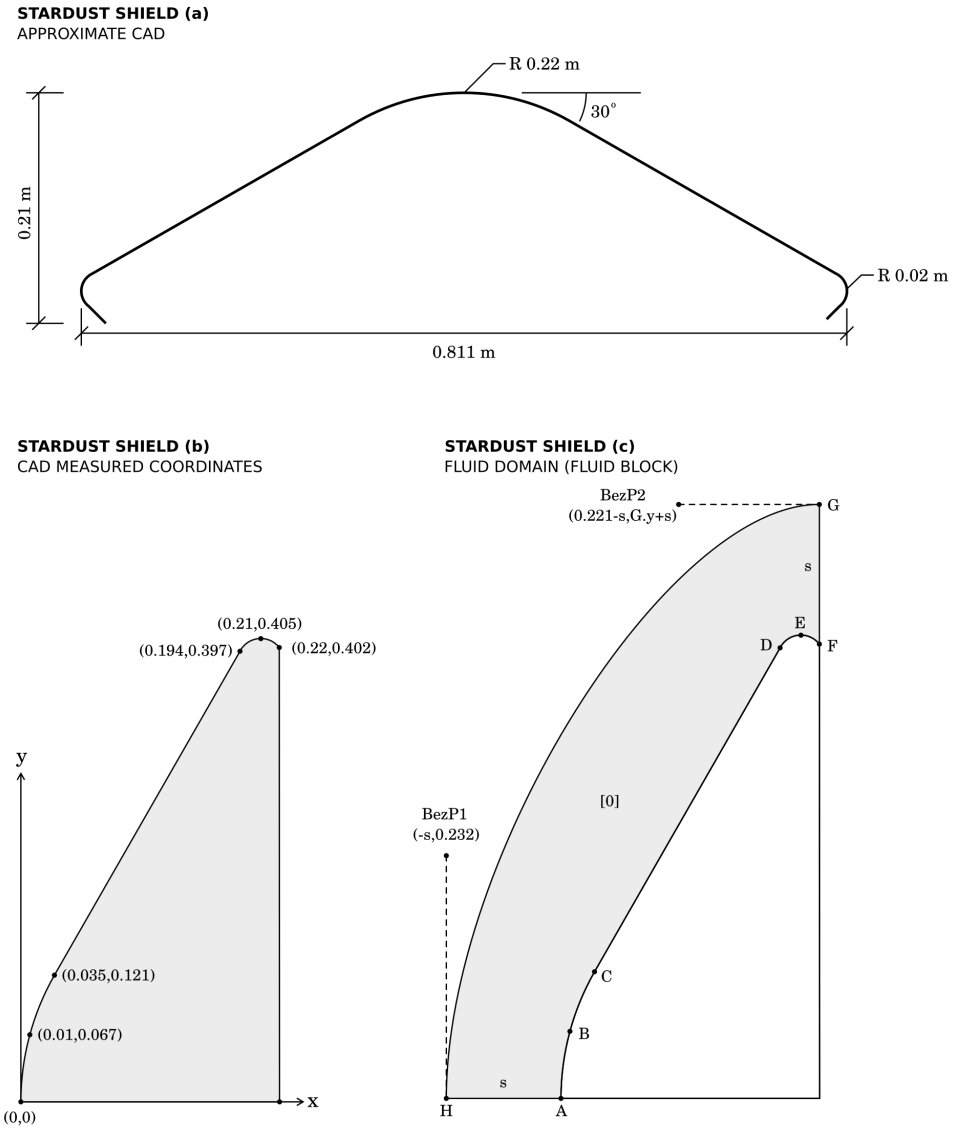

Figure 4.11: a) The approximate geometry of the stardust return capsule. b) The half-geometry for simulation and shape coordinates. c) The fluid domain.

Note: this fluid block will be built of more than four edges. This will be made possible by the *Polyline:new* which is capable of turning n lines into a single parametric line. Further, a Bezier

line will be used in this simulation for the west edge of the fluid block (the line HG).

4.6.1 The Geometry

The geometry and example B.C. are defined in lua as follows:

```
-- Fluid Domain Parameters
s = 0.06
-- Coordinates
A = Vector3:new{x = 0, y = 0}
B = Vector3:new{x = 0.01, y = 0.067}
C = Vector3:new{x = 0.035, y = 0.121}
D = Vector3:new{x = 0.194, y = 0.397}
E = Vector3:new{x = 0.21, y = 0.405}
F = Vector3:new{x = 0.22, y = 0.402}
G = Vector3:new{x = F.x, y = F.y+s}
H = Vector3:new{x = -s, y = 0}
BezP1 = Vector3:new{x = -s, y = 0.232}
BezP2 = Vector3:new{x = 0.221-s, y = G.y}
-- Building the Individual Lines, Arcs, Beziers
AC = Arc3:new{p0 = A, pmid = B, p1 = C}
CD = Line:new{p0 = C, p1 = D}
DF = Arc3:new{p0 = D, pmid = E, p1 = F}
GF = Line:new{p0 = G, p1 = F}
HG = Bezier:new{points={H,BezP1,BezP2,G}}; HG = ArcLengthParameterizedPath:new{underlying_path=HG}
HA = Line:new{p0 = H, p1 = A}
-- Shield Surface Polyline
AF = Polyline:new{segments={AC,CD,DF}}
-- The Parametric Surface, [0]
surf = makePatch{north = GF, south = HA, east = AF, west = HG}
-- The grid
Nx = 40
Ny = 70
grid = StructuredGrid:new{psurface = surf, niv = Nx, njv = Ny}
-- The boundary Conditions
BCs = {
      west = InFlowBC_ShockFitting:new{flowState=inflow},
      north = OutFlowBC_Simple:new{}
    }
-- The FluidBlockArray (split as 2 by 3 blocks)
block = FBArray:new{grid = grid, initialState = initial, nib = 3, njb = 2, bcList=BCs}
```

Important notes:

1. The *FBArray* is used rather than a *FluidBlock* (required for shock-fitting).
2. The west (bezier) edge will adjust to fit the shock - this is a four-point bezier and requires 'handle' points as shown in figure 4.11.
3. The *ArcLengthParameterizedPath:new{}* method is used. This is done to redefine the bezier as being evaluated with more equal spacing - allowing us to reduce the skew at the upper end of the grid (try it without and see the skewed cells!).

4.6.2 The Generated Fluid Domain

A flow state similar to that used in the cylinder geometry example is utilised in generating the below figures. A velocity of 7000 ms^{-1} and pressure 1.1 Pa are used for the free stream conditions. The Grid:

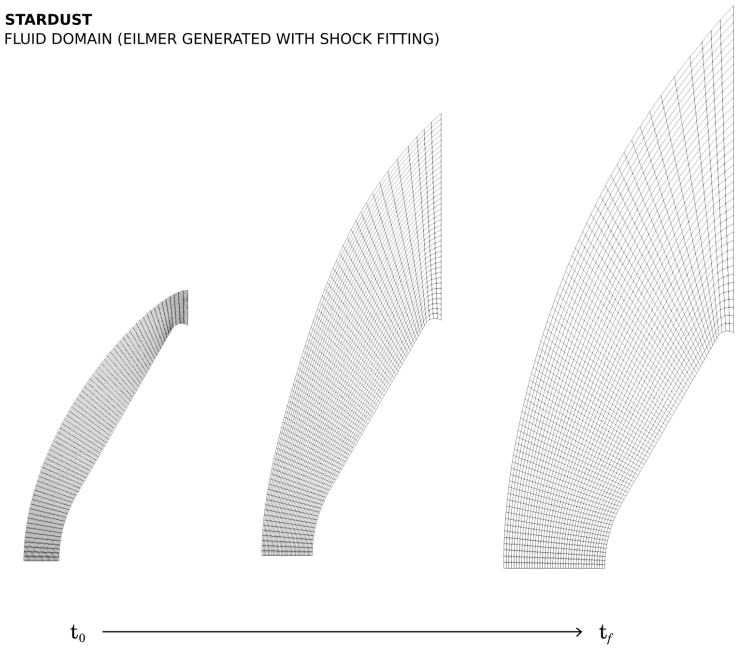

Figure 4.12: An adapting (shock-fitting) grid - converging as simulation progresses

Two important take-aways:

1. A shock-fitting grid may grow the volume of the fluid domain and as such the cell sizes increase and, in this case, skew.
2. CFD is iterative - one must often run a few 'smaller' (less computationally intensive) simulations to be happy knowing they have a reasonable grid.

4.7 Entirely Parametric Re-entry Geometries: Hayabusa Shield (Sphere Cone Template)

In the last example, stardust, the fluid domain was built entirely from coordinates - Whilst coordinates can be sufficiently precise in defining the domain (and often make the grid-building process faster) it does not allow for making quick changes to the geometry when needed. This example will demonstrate the mathematical mind set necessary when making entirely parameterised grids. Consider the Hayabusa shield geometry:

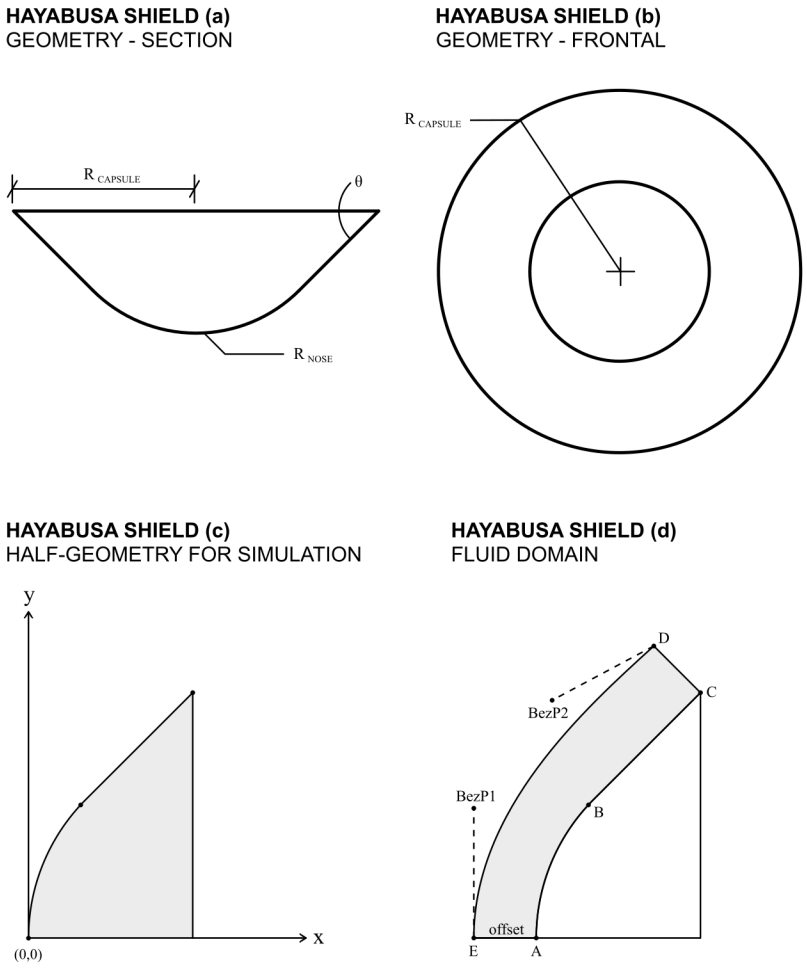

Figure 4.13: a) Hayabusa sphere-cone section. b) Frontal Hayabusa geometry. c) Geometry required for a fluid simulation. d) Fluid domain with arbitrary Bezier midpoints.

4.7.1 Mathematics of the Geometry

The sphere-cone geometry is comprised of a linear and circular component. The mathematics can be derived by equating a straight line ($y = mx + c$) with a circle and solving for the

intersection. In doing so, the quadratic form arises. By setting the discriminant to 0 (single intersection point) and solving, one can conclude the following:

$$m = \tan(\theta) \tag{4.1}$$

$$c = \left(\sqrt{1+m^2} - m\right) \cdot R_{\text{NOSE}} \tag{4.2}$$

$$B_x = \frac{(2R_{\text{NOSE}} - 2mc) + \sqrt{(2mc - 2R_{\text{NOSE}})^2 - 4(1+m^2)(c^2)}}{2(1+m^2)} \tag{4.3}$$

$$B_y = mB_x + c \tag{4.4}$$

4.7.2 Mathematically Driven Eilmer4 Geometry

The above mathematics can now be used to define the fluid domain geometry in a Hayabusa project *.lua* file. First, since our simulation is axisymetric the simulation must be configured as such. Note, the typical preliminary config is left out - refer to the wedge example for a reminder.

Make the simulation axisymetric:

```
config.axisymmetric = true
```

Now, define the parameters. These can be change to give any spherecone, the parameters given in the below code are for the Hayabusa 2.

```
NoseRadius = 0.2  -- [m]      (radius of the spherecone)
CapsuleRadius = 0.2  -- [m]      (radius (frontal) of the capsule)
ConeTheta = 45   -- [degrees] (spherecone angle)
Offset = 0.05 -- [m]      (initial offset from geometry of left inflow boundary)
```

Building a very basic custom LUA function to calculate linear and inverse linear relationships:

```
function linearPart(x, m, c)
   -- """Just a linear relation"""
   return m*x + c
end

function inverseLinearPart(y, m, c)
   -- """Just an inverse linear relation"""
   return (y - c)/m
end
```

Implementing the point-wise mathematics of the geometry:

```
-- The coefficients of the linear part
m = math.tan(ConeTheta * math.pi / 180) -- gradient of linear part
c = (math.sqrt(1+m^2)-m)*NoseRadius     -- y-intercept of linear part

--Coordinates
Bx = ( (2*NoseRadius - 2*m*c) + math.sqrt( (2*m*c - 2*NoseRadius)^2 - 4*(1+m^2)*(c^2) ) ) / (2 * (1+m^2))
By = linearPart(Bx, m, c)

Cy = 0.2
Cx = inverseLinearPart(Cy, m, c)

A = Vector3:new{x = 0.0, y = 0.0}
B = Vector3:new{x = Bx, y = By}  -- the intersection of the line and arc
C = Vector3:new{x = Cx, y  =Cy}
CircCentre = Vector3:new{x = NoseRadius, y = 0.0}

D = C + Vector3:new{x = 0.0, y = Offset}
E = A - Vector3:new{x = Offset, y = 0.0}

bez_p1 = E + Vector3:new{x = 0.0, y = 2*Offset}
bez_p2 = D - Vector3:new{x = 2*Offset, y = Offset}
```

Define the lines, arc, and northern Bezier:

```
--Lines, arcs
AB = Arc:new{p0 = A, centre = CircCentre, p1 = B}
BC = Line:new{p0 = B, p1 = C}
DC = Line:new{p0 = D, p1 = C}
ED = Bezier:new{points = {E, bez_p1, bez_p2, D}}
EA = Line:new{p0 = E, p1 = A}
```

Consolidate the lines AB and BC into AC. This is done so that the east boundary of the fluid block can be given as one line.

```
--Lines, arcs
AB = Arc:new{p0 = A, centre = CircCentre, p1 = B}
BC = Line:new{p0 = B, p1 = C}
DC = Line:new{p0 = D, p1 = C}
ED = Bezier:new{points = {E, bez_p1, bez_p2, D}}
EA = Line:new{p0 = E, p1 = A}
AC = Polyline:new{segments = {AB,BC}}
```

Finally, the surface, grid, fluid block array can be built. Here an example of clustering (toward the shock and stagnation line) is demonstrated. A shock-fitting boundary condition is set on the west edge of the fluid block so that the shock can be captured when the simulation is run.

```
niv =32+1; njv =128+1;
cluster_generic = RobertsFunction:new{end0=true, end1=false, beta = 1.01}
clusterlist = {north=none, south=none, east=cluster_generic, west=cluster_generic}
grid = StructuredGrid:new{psurface=surface, niv=niv, njv=njv, cfList=clusterlist}

blocks = FBArray:new{grid=grid,
                     initialState=initial,
                     bcList={
                          north=OutFlowBC_Simple:new{},
                          west=InFlowBC_ShockFitting:new{flowState=inflow}},
                     nib=6, njb=8}
```

The inflow and initial conditions are left to the user.

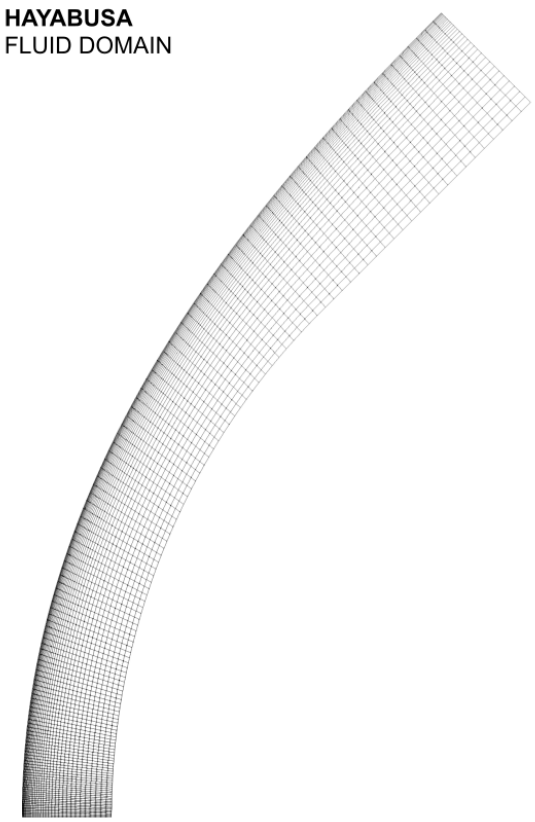

Figure 4.14: Hayabusa grid before shock-fitting.

As this chapter is focused upon geometry generation of the fluid domain, simulation results are not given for this example. To be accurate, this simulation would require a higher grid quality (refinement) and run times best suited for a super computer.

Chapter 5

Simulating High-temperature Gas Effects

5.1 What High Temperature does to the Flow

Importantly, when a shock processes a flow and a temperature increase is observed, the following effects can occur.

1. Reacting flows
2. Surface chemistry (catalysis, pyrolysis, etc.)
3. Thermal non-equilibrium
4. Chemical non-equilibrium
5. Reduction in shock stand off
6. Reduction in oblique shock angles
7. Radiating flows

Importantly, when compared to a calorically perfect gas, the post-shock chemistry results in a significant reduction in post-shock temperature. This occurs as energy (temperature) is instead dissociating or even ionising molecules or lone atoms - these reactions are endothermic and as such the flow temperature decreases.

5.2 Finite-rate Chemistry Simulations

Remembering back to the Rankine-Hugoniot equations and the normal shock relations, one notes that 'shock processing' increases the temperature and pressure of a fluid. Additionally, from the chemistry recap in chapter two, this increase in temperature is why we expect to see some chemistry (reactions) happen! After all, our reaction rates are a function of temperature.

5.2.1 When to use Finite-Rate Chemistry Effects

When simulating any given flow, one needs to conduct a review of the literature to understand when the chemical elements of the fluid begin to react. When considering air, information such as the graph produced by C. James in *Hypersonics*, Lecture 2, *"High Temperature Phenomena"* can be very useful:

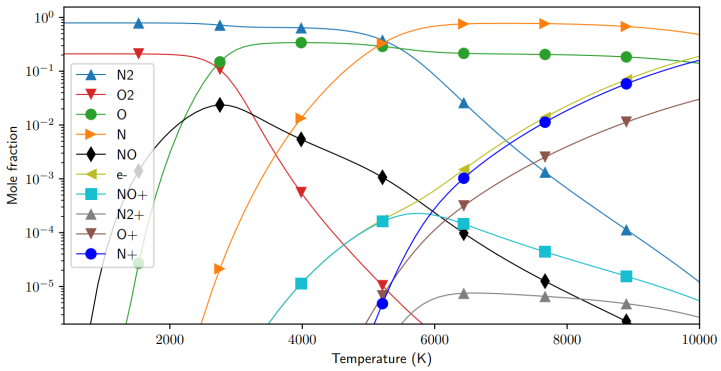

Figure 5.1: The mole fraction of 5-species air as a function of temperature (at 1 atm).

Evident from this figure is that, at 1 atm, air starts reacting above approximately 500 K, with NO (nitric oxide) forming 10 % of the composition (by mole fraction) at 2400 K before reducing as dissociation begins.

This additional figure by Gupta, 1990, (found **here**) shows when particular air chemistry models should be used - accounting for more and more species as velocity increases.

Regions with chemical and thermal nonequilibrium	
Region	Aerothermal phenomenon
Ⓐ	Chemical and thermal equilibrium
Ⓑ	Chemical nonequilibrium with thermal equilibrium
Ⓒ	Chemical and thermal nonequilibrium

Chemical species in high-temperature air		
Region	Air chemical model	Species present
Ⅰ	2 species	O_2, N_2
Ⅱ	5 species	O_2, N_2, O, N, NO
Ⅲ	7 species	$O_2, N_2, O, N, NO, NO^+, e^-$
Ⅳ	11 species	$O_2, N_2, O, N, NO, O_2^+, N_2^+, O^+, N^+, NO^+, e^-$

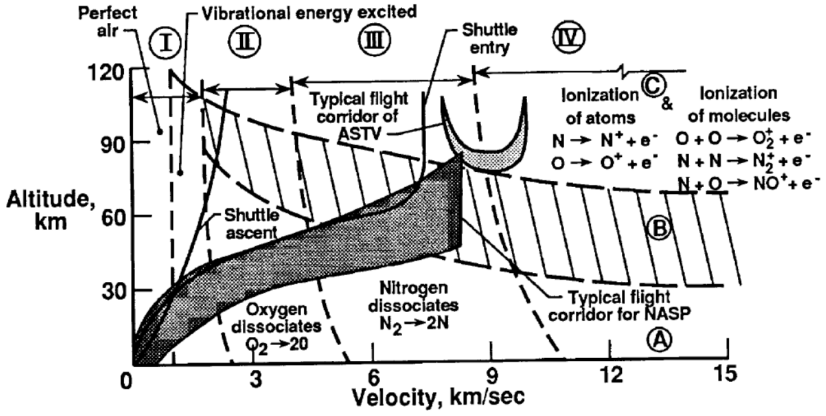

Figure 5.2: Gupta flight stagnation region air chemistry for a 30.5 cm radius sphere.

5.2.2 An Example of Dissociating Nitrogen in Eilmer4 (Wedge Entering Titan)

On the Eilmer4 Examples Repository

Assuming you have installed Eilmer4 from the gdtk as per chapter three, navigate to the 'gdtk' file (likely located in the home directory). Here, an examples file can be found within which one can find Eilmer4 examples in 2D and 3D which use many types of gas and chemistry models - including chemical models for Mars, Earth, and more - based on works such as Park, Gupta, Dunn, and Kim. It is okay to borrow these but it is important to verify that the constants do indeed align to the paper's from which they are taken.

A Real-sol Problem

To demonstrate the creation of a more simple chemical model, a simple atmospheric composition is selected: Titan. Titan, one of Saturn's moons, has an atmospheric composition of approximately 94 % nitrogen gas and 5.7 % methane - the rest are trace hydrocarbon species. In this example, the Titan atmosphere will be approximated as 100 % nitrogen gas (N_2) - the validity of this approximation will not be studied here.

Building a Gas Model File

Begin by creating a new project folder/directory, for the purposes of this tutorial, it shall be named 'titanwedge'.

```
mkdir titanwedge
cd titanwedge
```

Within this directory create a gas model file. Conventionally, this is going to be named 'nitrogen-2sp.inp'. This will be the input (.inp) for the Eilmer4 prep-gas command.

Within this 'nitrogen-2sp.inp' file, the model and species are to be specified as follows:

```
1 model = 'thermally perfect gas'
2 species = {'N2', 'N'}
```

The gas model file ('nitrogen-2sp.inp') can now be saved and closed.

Building a Reaction Rates File

Within the same directory, create a new file named 'nitrogen-2sp-2r.lua'. Here we will take the works of those such as Park, Kim, Gupta, etc. to define our reaction rates in the Arrhenius form (see chapter two for revision):

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

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

Full credit for the collection of these rates goes to R. J. Gollan and P. Jacobs - these constants were taken from examples in the Eilmer4 database.

Again, save this file and exit.

Preparing the Gas Model

The gas model will be prepared using the following terminal command:

```
prep-gas A B
```

Where A is the name of the gas file we created: 'nitrogen-2sp.inp' and B is the desired name of the finalised gas model file (.lua). In full the command is,

```
prep-gas nitrogen-2sp.inp gm-nitrogen-2sp.lua
```

A new file has now been created (named 'gm-nitrogen-2sp.lua') which details much about the gas model and properties such as viscosity and conductivity - this is the gas file that Eilmer4 will require later.

Preparing the Chemistry Model

To prepare the chemistry, we will need our new gas model file, 'gm-nitrogen-2sp.lua' as well as the reactions file created above.

The prep-chem command is of the form:

```
prep-chem A B C
```

Where A is the name of our recently generated gas model file, B is the title of our reaction rates lua, and C is the desired final name of our reactions model. In full the command is,

```
prep-chem gm-nitrogen-2sp.lua nitrogen-2sp-2r.lua rr-nitrogen-2sp-2r.lua
```

After running this command, a new file titled 'rr-nitrogen-2sp-2r.lua' will now exist in the directory. This is the final reactions file that Eilmer4 will require.

Selecting a Geometry

Now that our gas model and chemistry files are built, an arbitrary entry vehicle is selected for this example's geometry: a wedge. We shall take the wedge example given in chapter four and extend it by:

1. adding a shoulder of length w.
2. increasing the angle of the wedge.

The following figure represents the wedge object and the addition of the shoulder - including the new fluid domain block required (3 in total).

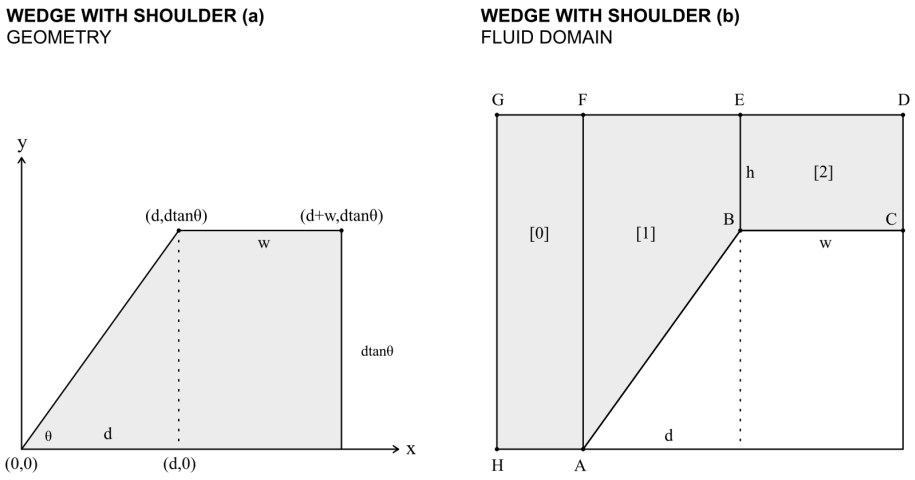

Figure 5.3: Geometry of a wedge with a shoulder. a) Wedge geometry. b) fluid domains of a wedge with shoulder.

Making the Project LUA file

Begin by creating a new 'titanwedge.lua' file. Here, the bulk of the work will be done. Firstly, set the preliminary configurations:

```
1 -- Wedge Project Eilmer4
2 -- Ideal Air
3 -- Author: Toby J. Herik
4
5 -- Preliminary Configuration
6 config.title = "Wedge Entering Titan - Dissociating Nitrogen"
7 config.dimensions = 2
```

Next, the parameters which define the geometry are set. *len_AH* is the length of the line AH. The other parameters are as per figure 5.3

```
-- Geometric Parameters
d      = 100/1000 -- 100 [mm]
theta  = 50       -- [degrees]
w      = 100/1000 -- 100 [mm]
h      = 70/1000  -- 70 [mm]
len_AH = 30/1000  -- 30 [mm]
```

Next, the gas and chemistry files (which were prepared before) are loaded into our simulation:

```
-- Setting the gas model and chemistry
nsp, nmodes, gmodel = setGasModel("gm-nitrogen-2sp.lua")
config.reacting = true
config.reactions_file = "rr-nitrogen-2sp-2r.lua"
```

Two new configurations are made here compared to prior simulation files: 1) the simulation is set to be reacting (line 16) and 2) the simulation is provided with a reactions file (line 17).

The free-strem conditions can now be set and the boundary and initial flow states defined. The conditions for this simulation were selected from the range studied within Brandis and Cruden in 2017. This study can be found **here**.

```
-- Free-stream conditions
T_inf = 1000  -- [K]
p_inf = 2000  -- [Pa]
V_inf = 5e3   -- [m/s]
mf = {N2=1.0} -- 100% nitrogen by mass fraction

-- Boundary and initial conditions
inflow  = FlowState:new{p=p_inf, T=T_inf, velx=V_inf, massf=mf}
initial = FlowState:new{p=p_inf/4, T=T_inf/4, massf=mf}
```

The geometric points (A, B, C, ..., H) are now defined as per figure 5.3:

```
-- Geometry
-- -- Points
A = Vector3:new{x = 0.0, y = 0.0}
B = Vector3:new{x = d, y = d*math.tan(math.rad(theta))}
C = Vector3:new{x = d + w, y = B.y}
D = Vector3:new{x = C.x, y = C.y + h}
E = Vector3:new{x = B.x, y = D.y}
F = Vector3:new{x = A.x, y = E.y}
G = Vector3:new{x = F.x - len_AH, y = F.y}
H = Vector3:new{x = G.x, y = A.y}
```

Further, the lines can now be drawn between points and the surfaces boundaries defined:

```
-- -- Lines
AB = Line:new{p0 = A, p1 = B}
BC = Line:new{p0 = B, p1 = C}
CD = Line:new{p0 = C, p1 = D}
HA = Line:new{p0 = H, p1 = A}
HG = Line:new{p0 = H, p1 = G}
AF = Line:new{p0 = A, p1 = F}
BE = Line:new{p0 = B, p1 = E}
ED = Line:new{p0 = E, p1 = D}
FE = Line:new{p0 = F, p1 = E}
GF = Line:new{p0 = G, p1 = F}

-- -- Surfaces
surfs = {}

surfs[0] = makePatch{north = GF, south = HA, east = AF, west = HG}
surfs[1] = makePatch{north = FE, south = AB, east = BE, west = AF, gridType = "ao"}
surfs[2] = makePatch{north = ED, south = BC, east = CD, west = BE}
```

Importantly, the middle surface, [1], is set as an “AO" patch. This instructs Eilmer4 to attempt to maintain orthogonality of grids at the walls / patch edges. This is well-suited for a wedge simulation as grid orthogonality is matched near the edges and .

Next, the grid divisions are set and the grids and fluid blocks defined:

```
-- -- Grid Settings
N_in_y = 40   -- Number of cells in y-direction
N_in_x0 = 10  -- Number of cells in patch 0 x-direction
N_in_x1 = 50  -- Number of cells in patch 1 x-direction
N_in_x2 = 50  -- Number of cells in patch 2 x-direction

-- -- Grids
grids = {}

grids[0] = StructuredGrid:new{psurface = surfs[0], niv = N_in_x0 + 1, njv = N_in_y + 1}
grids[1] = StructuredGrid:new{psurface = surfs[1], niv = N_in_x1 + 1, njv = N_in_y + 1}
grids[2] = StructuredGrid:new{psurface = surfs[2], niv = N_in_x2 + 1, njv = N_in_y + 1}

-- -- Fluid Blocks
blocks = {}

blocks[0] = FluidBlock:new{grid = grids[0], initialState = initial}
blocks[1] = FluidBlock:new{grid = grids[1], initialState = initial}
blocks[2] = FluidBlock:new{grid = grids[2], initialState = initial}
```

Do not forget to identify the fluid block connections:

```
-- Identify block connections
identifyBlockConnections()
```

The inflow and boundary conditions can now be defined. Importantly, the north edge of all fluid blocks are defined as an outflow.

```
-- Set inflow and outlet boudnary conditions
-- -- inflow
blocks[0].bcList[west] = InFlowBC_Supersonic:new{flowCondition=inflow}

-- -- outflow
blocks[2].bcList[east] = OutFlowBC_Simple:new{}

-- -- tops of all blocks must be outflow BC!
blocks[0].bcList[north] = OutFlowBC_Simple:new{}
blocks[1].bcList[north] = OutFlowBC_Simple:new{}
blocks[2].bcList[north] = OutFlowBC_Simple:new{}
```

Finally, the configuration of the simulation is set. The characteristic flow length and time is found and the flow is simulated for 5 characteristic flow times.

```
-- The Final Config
charac_length = w + d --aproximate flow length
charac_time   = charac_length / V_inf -- approximate time flow to travel one length

config.max_time = 5 * charac_time    -- stop if simulation runs for 6 'flows' over the wedge if
                                     -- flow were to stay supersonic (unprocessed by the shock)
config.max_step = 10000              -- stop if the simulation runs for 3000 steps
config.dt_init = 1e-6                -- initial time step size
config.dt_plot = config.max_time/10  -- plot ten times
```

Running and Post Processing the Simulation

The simulation can be prepared and run with the following commands:

```
e4shared --prep --job=titanwedge
e4shared --run --job=titanwedge
```

After the simulation has finished running, post-processing is done making sure to include whatever extra variables the user may want - in this case I will include Mach and Mole Fraction:

```
e4shared --post --job=titanwedge --vtk-xml --tindx-plot=all --add-vars="mach,molef"
```

ParaView can be used to view the solutions:

```
paraview ./plot/titanwedge.pvd
```

Results

The grid generated has three distinct fluid blocks - two are regular rectangular blocks, and the middle block uses the AO patch type to achieve orthogonal cells neaar-to the fluid block edges:

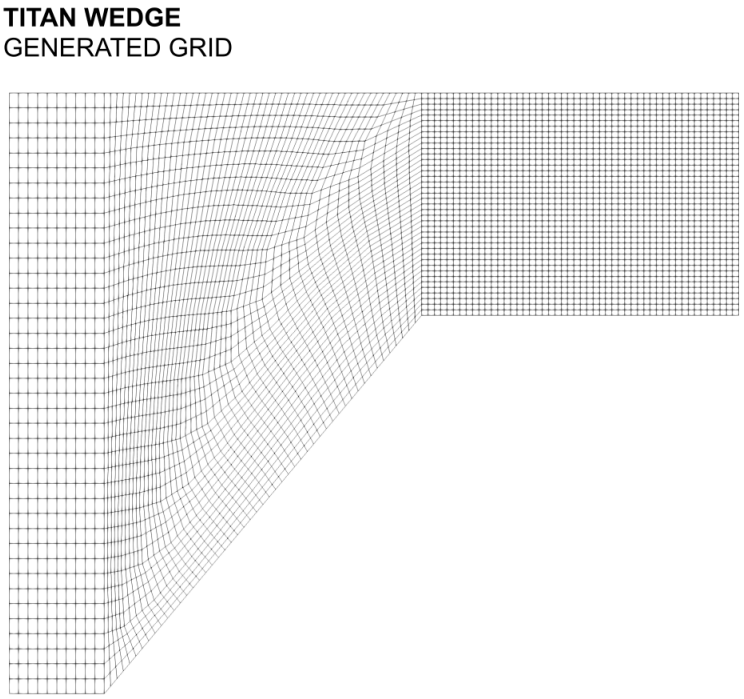

Figure 5.4: Titan wedge fluid domain grid.

The flow field depiction of Nitrogen (N) by mole fraction well-represents the dissociation in this instance - the increase in mole fraction of N is demonstrating the dissociation of N_2 into N.

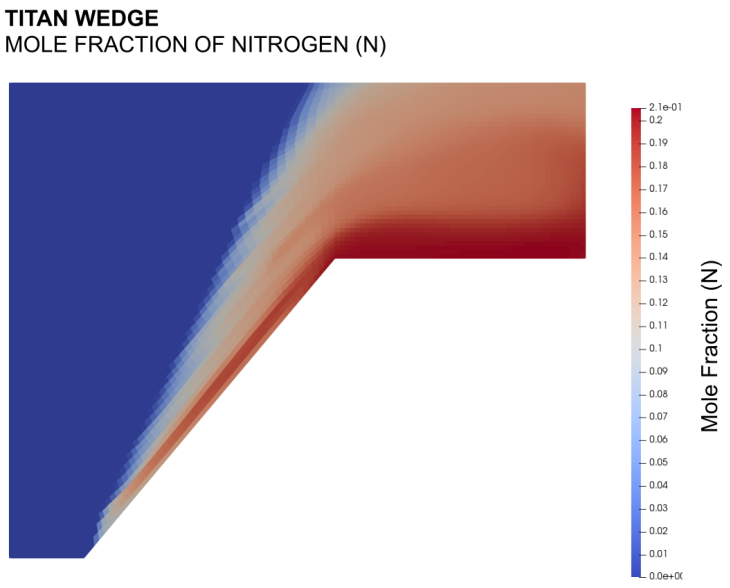

Figure 5.5: Flow Field Mole Fraction of N

Finally, the temperature field, for shock-shape comparison is the following:

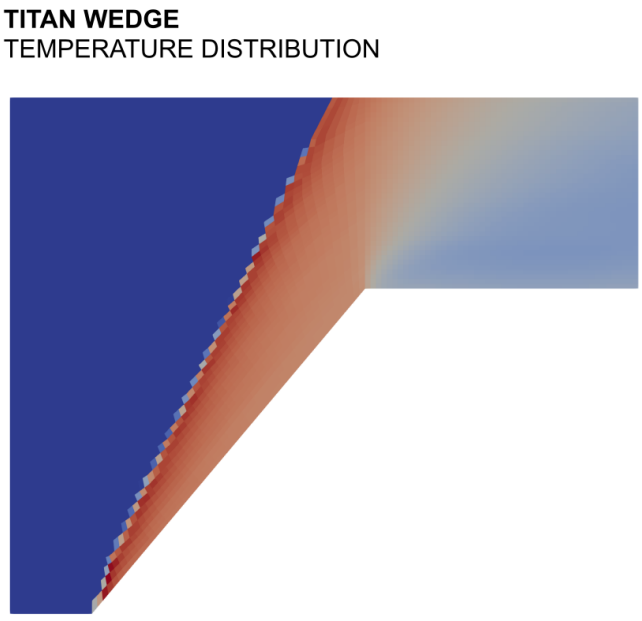

Figure 5.6: Temperature of the Fluid Domain - Depicting attached oblique shock shape

Importantly, the grid used for this simulation does not represent a 'GOOD' choice of grid. To know if one has used a sufficiently refined grid, a user MUST conduct a 'grid independence study' - the topic of future works in this guide.

~

This guide is constantly being improved.

For a description of the topics to be added next, please see section 1.3 of the introduction. If you would like certain topics to appear in this guide contact me! **t.vandenherik@uq.edu.au**