`Onedval` : a tool to extract one-dimensionalised properties from CFD data

Mechanical Engineering Report 2013/04

The University of Queensland
Rowan J. Gollan

26 February 2013

Abstract

Computational fluid dynamics can be used to provide rich datasets of three-dimensional data. Often however, for engineering analyses, a one-dimensionalised average of the three-dimensional flow properties are required. `Onedval` is a software program that computes one-dimensionalised averages from CFD data using a variety of techniques.

Contents

1 Introduction

Modern computational fluid dynamics (CFD) simulations can give an impressively detailed amount of information about three-dimensional flow fields. Despite all the detail available in 3D datasets, most engineering design, analysis and evaluation relies on "averaged" properties to give a one-dimensional representation of the flow field. There is not a 'one-size-fits-all' technique for averaging multi-dimensional data and so it is useful to have a variety of methods available to post-process CFD data to compute these one-d averaged properties. The program described in this report, `Onedval`, is a utility which can compute one-dimensionalised properties and integrated quantities from multi-dimensional CFD data.

`Onedval` is written in Python as a small program to do one job: compute averaged properties from CFD datasets. The program itself is small but it does leverage a lot of functionality from the code collection developed by the Compressible Flow CFD project [1], such as the geometry and gas libraries. As such, the user is required to download and install the CFCFD code collection in order to use this program. The program has been developed and tested on Linux platforms.

The report is organised to give the theory of the one-dimensionalisation methods followed by practical information on how to use `Onedval`. Section 2 presents the theory behind the various one-dimensionalisation methods which are available to the user. In Section 3, the installation and use of `Onedval` is discussed. Finally, some examples of use are given in Section 4.

2 One-dimensionalisation methods

There are three methods implemented in `Onedval` to compute one-dimensionalised average flow properties. These methods are:

1. area-weighted average;
2. mass-flux-weighted average; and
3. flux-conserved average.

These averages are useful for different applications. For example, mass-flux-weighted average properties are often used to compare directly to experimentally measured values. Flux-conserved average properties are required to maintain conservation of mass, momentum and energy when using the results of CFD calculations as input for zero- and one-dimensional analyses. The calculation method of each of these averages is described in the following paragraphs. The notation and calculation methods used here draw heavily from the work of Baurle and Gaffney [2]. The reader is referred to the paper by Baurle and Gaffney for a detailed discussion of the one-dimensionalisation techniques for CFD data and the trade-offs associated with various selections.

2.1 Area-weighted average

A general expression for the weighted average of some property ϕ is

$$\phi = \frac{\int \phi w \, dA}{\int w \, dA} \tag{1}$$

where w is a weighting function and A is the area of interest for extracting an averaged property. For an area-weighted average, $w = 1$.

2.2 Mass-flux-weighted average

To get a mass-flux-weighted average, $w = \rho(\vec{v} \cdot \vec{n})$, and so Eq. 1 becomes

$$\phi = \frac{\int \phi\rho(\vec{v} \cdot \vec{n}) \, dA}{\int \rho(\vec{v} \cdot \vec{n}) \, dA}. \tag{2}$$

2.3 Flux-conserved average

A flux conserved average, as the name implies, gives a consistent set of one-dimensionalised flow properties such that when combined give the same integral flux of mass, momentum and energy across the area of interest. The flux conserved average is also sometimes referred to as the stream thrust average. Mathematically, the definition of flux-conserved average properties are related to the fluxes as follows. The fluxes of mass, momentum, energy and species mass across an area, A, are

$$f_{\text{mass}} = \int_A \left[\rho(\vec{v} \cdot \vec{n})\right] \, dA \tag{3}$$

$$\vec{f}_{\text{mom.}} = \int_A \left[\rho(\vec{v} \cdot \vec{n}) + p\vec{n}\right] \, dA \tag{4}$$

$$f_{\text{energy}} = \int_A \left[\rho(\vec{v} \cdot \vec{n})h_0\right] \, dA \tag{5}$$

$$f_{\text{isp}} = \int_A \left[\rho(\vec{v} \cdot \vec{n})Y_{\text{isp}}\right] \, dA \tag{6}$$

The flux conserved flow properties, given in bold face, satisfy the integral flux values:

$$
\begin{aligned}
f_{\text{mass}} &= [\boldsymbol{\rho}(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})]\, \mathbf{A} && (7)\\
\vec{f}_{\text{mom.}} &= [\boldsymbol{\rho}(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})\vec{\mathbf{v}} + \mathbf{p}\vec{\mathbf{n}}]\, \mathbf{A} && (8)\\
f_{\text{energy}} &= [\boldsymbol{\rho}(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})\mathbf{h_0}]\, \mathbf{A} && (9)\\
f_{\text{isp}} &= \left[\boldsymbol{\rho}(\vec{\mathbf{v}} \cdot \vec{\mathbf{n}})\mathbf{Y}_{\text{isp}}\right] \mathbf{A} && (10)
\end{aligned}
$$

The flux-conserved mass fraction values, $\mathbf{Y}_{\text{isp}}$, can be computed directly as

$$\mathbf{Y}_{\text{isp}} = \frac{f_{\text{isp}}}{f_{\text{mass}}}. \quad (11)$$

The other flux-conserved flow properties are computed in an iterative manner, described next.

The method to compute the flux-conserved properties differs from the approach given by Baurle and Gaffney [2] and so is described here in some detail. In `Onedval`, the flux-conserved averages are computed by solving a minimisation problem[1]. The advantage of solving this minimisation problem compared to the technique suggested by Baurle and Gaffney is that no assumption about the gas model is required. In other words, the approach used here can treat general gas models which may not necessarily obey the perfect gas equation of state.

In this minimisation problem, a set of flow values are found which combined give the best match to the integrated flux quantities. There can be more than one solution to the minimisation problem. The search space is restricted by using the mass flux weighted average values as the initial guess for the minimisation problem. The objective function to minimise is given in terms of flux-conserved properties as

$$f_{\text{obj.}} = \text{err}_{\text{mass}} + \text{err}_{\text{mom.}} + \text{err}_{\text{energy}} \quad (12)$$

where the errors in mass, momentum and energy are computed as

$$
\begin{aligned}
\text{err}_{\text{mass}} &= \frac{|f_{\text{mass}} - \boldsymbol{\rho}\mathbf{u}A|}{|f_{\text{mass}}| + 1} && (13)\\
\text{err}_{\text{mom.}} &= \frac{|f_{\text{mom.}} - (\boldsymbol{\rho}\mathbf{u}^2 + \mathbf{p})A|}{|f_{\text{mom.}}| + 1} && (14)\\
\text{err}_{\text{energy}} &= \frac{\left|f_{\text{energy}} - \boldsymbol{\rho}\mathbf{u}A(\mathbf{h} + \frac{1}{2}\mathbf{u}^2)\right|}{|f_{\text{energy}}| + 1}. && (15)
\end{aligned}
$$

In the above, $f_{\text{mom.}}$ is the magnitude of the momentum flux vector and $\mathbf{u}$ is a scalar value of velocity. Specifically, $\mathbf{u}$ is the component of velocity in the averaged normal direction of the surface for averaging; it is equivalent to $\vec{\mathbf{v}} \cdot \vec{\mathbf{n}}$. Reducing the momentum and velocity to scalar quantities is not usually a problem because often the scalar values are all that is required for subsequent one-dimensional analyses. This multi-variate problem is solved in terms of three variables: $\boldsymbol{\rho}$, $\mathbf{T}$ and $\mathbf{u}$. This minimisation problem is solved using the Nelder-Mead algorithm [3]. Note that in the formulation of this minimisation problem, the vector components of momentum and velocity are lost.

After solving the minimisation problem, the flux-conserved values for density, temperature and velocity are known and the mass fractions were computed earlier. The remaining flow properties are computed using the equation of state.

[1]This approach was suggested by Dr Peter Jacobs.

Assumptions about the thermodynamics of the gas

When computing the flux-conserved average, `Onedval` needs to use a model for the thermodynamic behaviour of the gas. Presently, the gas is assumed to be a mixture of thermally perfect gases. This gas model is suitable for most applications in hypersonic flows where a single-temperature model is valid. There will be some modelling inconsistency if the CFD simulation was performed with a gas other than a mixture of thermally perfect gases. For example, there is some inconsistency if the CFD assumed a gas with constant specific heats. However, this inconsistency should only be small. The argument is that at low temperatures, the differences between a model that assumes constant specific heats and thermally perfect gases is only small. At high temperatures, the difference is larger. However, at these higher temperatures, where the difference is important, one should go back and question the validity of using a constant specific heat model for the gas in the CFD calculation. This assumption of gas model does not affect the computation of area-weighted or mass-flux-weighted averages.

3 Using `Onedval`

3.1 Prerequisite software

`Onedval` has been developed and tested on linux platforms. It uses code from the Compressible-Flow CFD project and is itself stored in the CFCFD code collection. The instructions for obtaining and installing that code are given in the next section. You will require an installation of Mercurial[2] in order to get a copy of the code. There is a list of packages required to use the code collection at:

`http://www.mech.uq.edu.au/cfcfd/getting-started.html#your-computational-environment`

`Onedval` requires specific versions of the Python packages `numpy` and `scipy` as listed here:

numpy: version 1.7.0 or later

scipy: version 0.11 or later

3.2 Installation and setup

Download

The CFCFD code collection is stored in a revision control system. The most up-to-date instructions for obtaining this source can be found at:

`http://www.mech.uq.edu.au/cfcfd/getting-started.html`

A reproduction of the instructions for obtaining a copy of the code is given in Listing 1.

```
The code repository
-------------------
The codes are available for download from a Mercurial repository.
To make a clone of the repository::

  $ cd $HOME
  $ hg clone https://cfcfdlocal@triton.pselab.uq.edu.au/cfcfd3-hg/cfcfd3-hg/ cfcfd3

This takes about 40 seconds on campus at UQ.
It may take much longer, depending on your internet connection.

To see what's changed::

  $ cd cfcfd3
  $ hg incoming https://cfcfdlocal@triton.pselab.uq.edu.au/cfcfd3-hg/cfcfd3-hg/
  ...
  $ hg pull -u https://cfcfdlocal@triton.pselab.uq.edu.au/cfcfd3-hg/cfcfd3-hg/

Notes

#. You will need a password for any access.  Please ask.
#. You can read but not write with the "cfcfdlocal" username.
#. Some usernames (by negotiation) may push changesets back to the repository.
```

Listing 1: Instructions for downloading the code from the Mercurial repository.

[2] Mercurial is a distributed version control system and is available for installation via packages for most linux distributions.

Installation

Onedval is installed by using the Make system. The commands to install Onedval are:

```
> cd $CFCFD_SRC/app/onedval/build
> make install
```

In the above, $CFCFD_SRC is the directory where you have the cloned the code collection. This compilation takes some time; it is building the gas library, the geometry library and a full compilation of Eilmer[3].

Environment setup

The last step to getting a running version of Onedval is to set up some environment variables which help the program locate certain Python and Lua modules. You need to correctly set three environment variables: $PYTHONPATH, $LUA_PATH and $LUA_CPATH. The additions needed in a typical .bashrc file are shown in Listing 2. This shows the correct settings assuming that the executables and modules were installed in the default installation directory $HOME/cfcfd3. Remember to re-'source' your .bashrc before trying to use Onedval, which can be done by logging out and logging back in to your session.

```
export PYTHONPATH=$HOME/e3bin
export LUA_PATH=$HOME/e3bin/?.lua
export LUA_CPATH=$HOME/e3bin/?.so
```

Listing 2: Additions required in a .bashrc file to allow Onedval to find the required Python and Lua modules.

Testing the installation

If all has gone well, you are ready to test the installation. Navigate over to the directory $CFCFD_SRC/app/onedv
Once in that directory, execute the run.sh script:

```
> ./run.sh
```

You should see some output to the screen indicating that the program is running and a new text file should be created in the working directory. This new file is exit-props.txt. The contents of this file should match those shown in Listing 4 which appears in Section 4.

3.3 Preparing data slices

Presently, Onedval only works on data slices that have been extracted using Tecplot [4]. The capability to handle other data formats will be implemented if and when the need arises. The discussion here focusses on using Tecplot to prepare data slices.

Some CFD programs give the user options regarding what flow properties are included in the output data files. In order to apply the averaging methods, there are some mandatory flow properties that must be included in the data slice when using Onedval. These required flow properties along with their units are listed in Table 1. It is often easiest to request these desired quantities from the CFD flow solver. However, if the quantities can be computed in Tecplot then that is fine also. What is important is that these required flow properties are available in the data slices handed to Onedval. Also, the dimensional values must be in SI units. It may be necessary to convert values in Tecplot

[3] In future versions of Onedval, it should be possible to remove the dependency on Eilmer.

(or by some other means) if they are not in SI units. There is one exception to the SI units rule: the physical coordinate values may be in any units and then a grid scaling factor can be set in the input to convert these values to metres.[4] Note that the mass fractions of *all* included species are required. This is so that the thermodynamics of the gas are correctly computed. In addition to the mandatory values, the user should have available any properties for which an area-weighted or mass-flux-weighted average is requested. For example, temperature and Mach number are not required properties for `Onedval` but they should be included if the averaged values of these quantities is desired.

Table 1: Required flow properties in data slices supplied to `Onedval`.

Flow property	Units	Symbol used in `Onedval`
x-ordinate	*may be scaled*	`x`
y-ordinate	*may be scaled*	`y`
z-ordinate	*may be scaled*	`z`
u-velocity	m/s	`u`
v-velocity	m/s	`v`
w-velocity	m/s	`w`
density	kg/m^3	`rho`
pressure	Pa	`p`
total enthalpy	J/kg	`h0`
species mass fractions	-	species name, eg. `H2O`

`Onedval` can process a single data slice or multiple slices. Each of those slices should be created in Tecplot by extracting a slice from a plane. This can be accessed from the Tecplot menu as `Data --> Extract --> Slice from Plane...`. When preparing many slices, it saves time and reduces the opportunity for human error if you use a Tecplot macro. An example of building a macro and using Tecplot in batch mode is shown in Section 4.2. When Tecplot extracts a slice from a plane, it will grab every bit of data that is on that plane. Often you might only be interested in a small portion of all the available data. For example, you might want to limit your focus to the internal duct section of a combustor and not have Tecplot grab data in the external flow field which happens to cross the plane of interest. In these cases, it is helpful to limit the active `Zones` in Tecplot to those of interest.

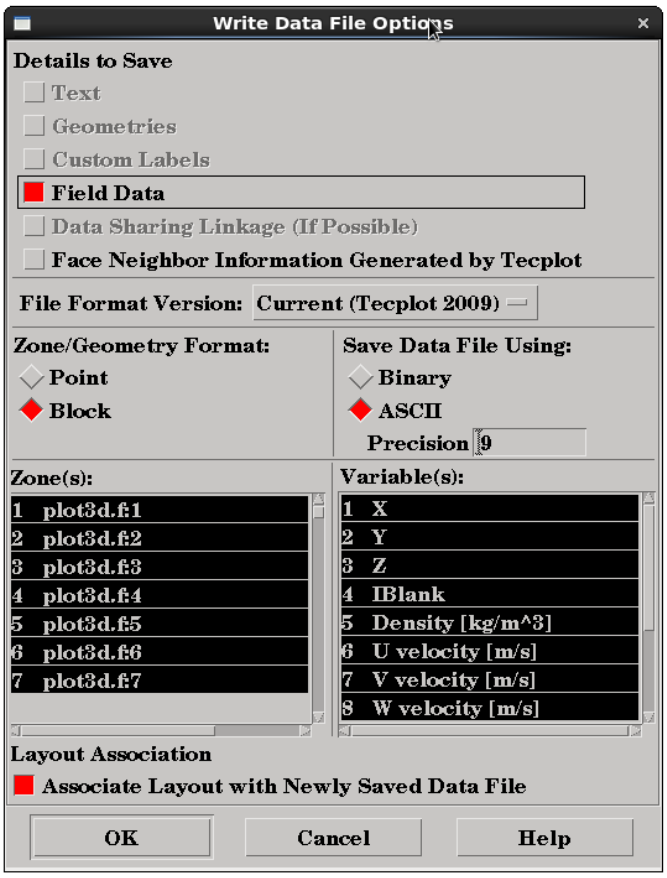
Having extracted the slice(s), the next step is write out the data so that `Onedval` may process it. The slice data needs to be written in `block` format as ASCII text. A screenshot of the Tecplot data writing dialogue box with the appropriate data format options is shown in Figure 1.

3.4 Preparing a configuration file

The next step is to prepare a configuration file which gives `Onedval` some information about what is contained in your data slices and what kinds of quantities you would like computed. A configuration file is a simple text file which contains *keyword-value* pairs of the form `keyword = value`. The actual syntax of the file conforms to the Python programming language syntax. This is because the configuration file is parsed by `Onedval` as an executable Python script. Since it is Python, the hash symbol (`#`) can be used to introduce comment text. The reader may jump forward to Listing 3 to see an example of what a configuration file looks like. Here, the keywords that should appear in the configuration file and their allowable values are described:

`species = [...]` A list of the species in the flow must be provided. The names of individual species should be species known by the `lib/gas` module. This list of species is used to con-

[4]The scaling of coordinates is made available because it is common for CFD grids that are built from CAD surfaces to be in units of millimetres.


Write Data File Options

Details to Save

- Text
- Geometries
- Custom Labels
- [✓ Field Data]
- Data Sharing Linkage (If Possible)
- Face Neighbor Information Generated by Tecplot

File Format Version: Current (Tecplot 2009)

Zone/Geometry Format: Point [✓ Block]

Save Data File Using: Binary [✓ ASCII]

Precision 9

Zone(s):

1 plot3d.f:1
2 plot3d.f:2
3 plot3d.f:3
4 plot3d.f:4
5 plot3d.f:5
6 plot3d.f:6
7 plot3d.f:7

Variable(s):

1 X
2 Y
3 Z
4 IBlank
5 Density [kg/m^3]
6 U velocity [m/s]
7 V velocity [m/s]
8 W velocity [m/s]

Layout Association

[✓ Associate Layout with Newly Saved Data File]

Figure 1: Screenshot of Tecplot writing dialogue box showing the selected options that are required for `Onedval` processing. The required options are: `block` and `ASCII`.

struct the thermodynamics model for a mixture of thermally perfect gases. This list should correspond to all species included in the data file.

`variable_map = { ... }` This is a map (or a dictionary, Python parlance) which maps the variable names known by `Onedval` to the variable names used in the data file. Every different flow solver gives different variations of the data field names. This map is a convenient way to access the relevant data in the data file without the hassle of renaming all of the data in Tecplot. At a minimum, the user *must* provide a mapping entry for each of the required flow properties listed in Table 1 except for the species. The `Onedval` symbols (which are the keys in the `variable_map`) are also listed in Table 1.

`grid_scale = val` The coordinates in the data slice will be scaled by `val` to convert them to metres if they are not already. If this keyword is not supplied, the scaling defaults to 1.0. In other words, no scaling is applied.

`filter_function = <Python-function>` The user may write a Python function in the configuration script which filters cells from the slice data based on user-defined criteria. The function accepts one argument, a single cell, and returns either `True` if the cell should be kept/included in the averaging or `False` if the cell should be removed/excluded from the averaging. This could be used, for example, to exclude the boundary layer cells from an average over the core flow. The use of this option is best demonstrated by an example. Section 4.2 provides such an example.

`one_d_averages = [ ... ]` This list specifies which of the available one-dimensalisation methods are requested. The list can contain more than one method; in which case, multiple one-d averages are computed and put in the output. The allowable method names are:

- `'area-weighted'`
- `'mass-flux-weighted'`
- `'flux-conserved'`

`one_d_outputs = [ ... ]` A list of the requested output flow properties to be averaged in a one-dimensionalised sense. The symbol names correspond to those used by `Onedval` (see Table 1).

`integrated_outputs = [ ... ]` A list of requested integrated quantities to be output. As part of the averaging, the integrated fluxes are computed. `Onedval` will also output these values if requested. The allowable names in this list are:

- `'mass flux'`
- `'momentum flux'`
- `'energy flux'`
- `'species mass flux'`

Note that when you list the single phrase `'species mass flux'`, the program will actually report all of the individual species mass fluxes in the output.

`output_file = 'filename.txt'` The desired name of the output file.

`output_format = 'format'` The `output_format` can be one of `'verbose'` or `'as_data_file'`. Typically, one would use `'verbose'` when using `Onedval` on a single data slice because it gives a lengthy output in a human-readable form. The `'as_data_file'` option is usually used when `Onedval` is processing a number of slices. In this format, the data for each slice becomes a separate row in the output data file. This is convenient to plot directly or read into another program.

3.5 Running the program

Having prepared the data slices and a configuration file, `Onedval` is invoked at the command line as follows:

```
> onedval config_file slice_file[s]
```

The first argument is always a configuration file. The second and following arguments list the data slice(s) to be processed. You could process a whole series of files by letting the shell expand the file names for you. For example, in a `bash` shell, you could process all data files of the form `slice-01.dat`, `slice-02.dat`, ..., `slice-20.dat`, by invoking this command:

```
> onedval config_file slice-??.dat
```

If everything is successful, `Onedval` will print some diagnostic messages to the screen and write the requested data to the output file. If something goes wrong, the program should end abruptly and noisily, and hopefully give you some hint about what went wrong. Most errors in user input will be reported to the screen, so read the screen output carefully.

4 Examples

There are typically two uses of `Onedval`: 1) to extract properties at a single plane; or 2) to extract properties at multiple planes to give a sense of the one-dimensional variation of properties. The two examples included here demonstrate the use of `Onedval` for both of these cases. The first example shows how the properties at the exit of a hypersonic inlet are extracted. These properties are useful on their own to give an assessment of inlet performance. They could also be used as input to a one-dimensional analysis of the flow through an attached combustor. In the second example, several slices through a CFD simulation of a scramjet combustor are collated to give a one-dimensionalised variation of flow properties. From these values, the performance measures of the combustor can be computed, such as combustion efficiency. Additionally, this second example shows how a filtering function can be used to focus the averaging on a subset of the cells in the supplied slice of data.

4.1 Exit flow properties of a hypersonic inlet

In this example, the one-dimensionalised flow properties at the exit of a hypersonic inlet are computed. The inlet is a 3D shape-transitioing inlet which is attached to a conical vehicle. A view of this inlet is shown in Figure 2. For this example, we are interested in determining the average flow properties at the exit of the inlet (the exit being just past the throat at the end of an isolator section).

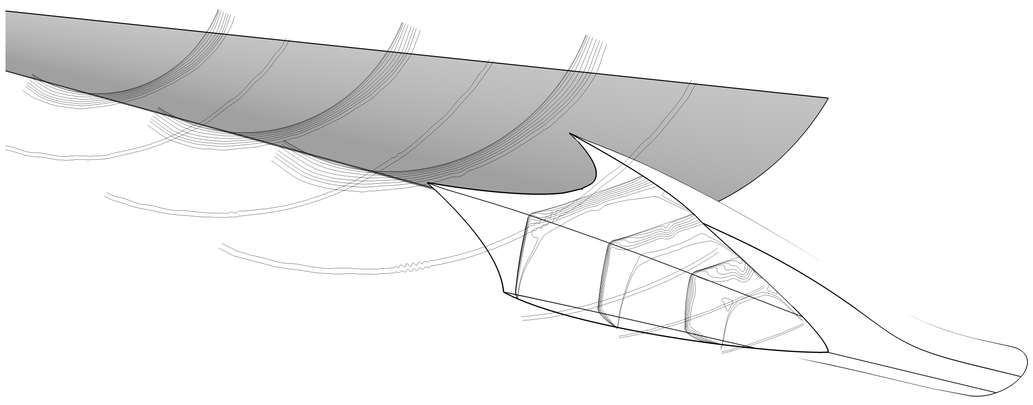


Figure 2: View of a 3D shape-transitioing inlet installed on a conical forebody. `Onedval` is applied to the exit plane to give one-dimensionalised flow properties across the outlet.

Before using `Onedval`, we need to extract a slice of flow data. A slice has been extracted at the exit plane and saved to a file called `exit-slice.dat`. When using Tecplot, the slice data should be written out in `block` form as ASCII text. The pressure, temperature and Mach number fields for the exit plane are shown in Figure 3. The slices only show half of the configuration because the CFD was performed assuming a symmetry condition.

Next, we prepare a configuration file. This configuration file is shown in Listing 3. It is important that the fields listed in `variable_map` correctly correspond to the field variables in the supplied Tecplot slice files. In this case, for example, version 6.2.1 of the VULCAN flow solver [5] was used to generate the flow field data. As such, the internal variable pressure `'p'` used in `Onedval` maps to the field variable listed as `'Pressure [Pa]'` in the Tecplot file. These mapping names need to be specified exactly: they are case sensitive and spacing sensitive.

The rest of the configuration file is fairly self-explanatory, and the details of various options were listed in Section 3.4. In this example, the output will be written to the file `exit-props.txt`. The requested `output_format` is 'verbose' which means that a descriptive text file is generated which

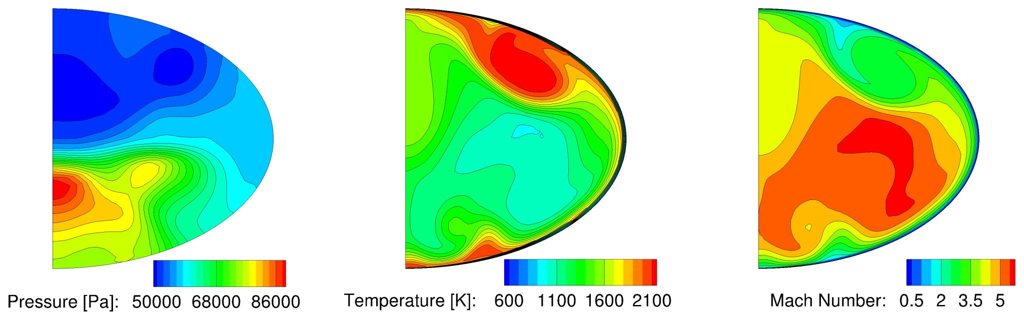


Figure 3: Selected property contours at the isolator exit of a 3D shape-transitioning inlet.

lists all of the requested properties.

To compute the requested averaged and integrated quantities, the `Onedval` program is run from the directory containing the data slice and configuration file, as follows:

```
> onedval exit.config exit-slice.dat
```

Some diagnostic output is printed to the screen to show the user the progress of the calculation. For this example, the output is written to a filed called `exit-props.txt`. This file is shown in Listing 4.

The one-dimensionalised properties at the exit of the inlet using the various averaging methods are listed in Table 2. Inspection of the values gives an indication of the differences one can expect based on choosing different one-dimensionalisation methods.

Table 2: One-dimensionalised flow properties at the exit of a hypersonic inlet.

Property	Area-weighted average	Mass-flux-weighted average	Flux-conserved average
pressure, kPa	61.20	64.74	64.09
temperature, K	1421.4	1299.8	1419.2
density, kg/m^3	0.1632	0.1833	0.1573
velocity, m/s	3056	3247	3242
Mach number	4.30	4.72	4.43

```
# list of species
species = ['air']

# A mapping of variable names for onedval to those
# found in the Tecplot file
variable_map = {'x':'X', 'y':'Y', 'z':'Z',
                'u':'U velocity [m/s]',
                'v':'V velocity [m/s]',
                'w':'W velocity [m/s]',
                'rho':'Density [kg/m^3]',
                'p':'Pressure [Pa]',
                'T':'Temperature [K]',
                'M':'Mach Number',
                'h0':'Total Enthalpy [J/kg]' }

# list of types of one-D averages to compute
one_d_averages = ['area-weighted', 'mass-flux-weighted', 'flux-conserved']

# A scaling factor if coordinates are NOT in metres
grid_scale = 0.23026347715684

# Output properties
one_d_outputs = ['p', 'T', 'rho', 'u', 'M']

# Integrated quantities
integrated_outputs = ['mass flux', 'momentum flux',
                      'energy flux', 'species mass flux']

# Output options
output_file = 'exit-props.txt'
output_format = 'verbose'
```

Listing 3: `exit.config`: example config file for use with a VULCAN-generated CFD solution.

```
------------------- onedval output ---------------------
number of cells in averaging:
ncells = 2418
cumulative area of cells (m^2):
area = 3.168593e-03
---------------------
Integrated quantities
---------------------
mass flux (kg/s)
m_dot = 1.615846e+00

momentum flux (kg.m/s^2)
mom_dot = Vector3(5440.93971431, 64.240864597, -26.0419054161)

energy flux (W)
e_dot = 1.048768e+07

mass flux of air (kg/s)
mair_dot = 1.615846e+00
------------------------------
One-dimensionalised quantities
------------------------------
-- area-weighted average --
pressure (Pa)
p = 6.119774e+04

temperature (K)
T = 1.421435e+03

density (kg/m^3)
rho = 1.631800e-01

U velocity (m/s)
u = 3.056031e+03

Mach number
M = 4.297213e+00

-- mass-flux-weighted average --
pressure (Pa)
p = 6.474022e+04

temperature (K)
T = 1.299762e+03

density (kg/m^3)
rho = 1.832700e-01

U velocity (m/s)
u = 3.247233e+03

Mach number
M = 4.724648e+00

-- flux-conserved average --
pressure (Pa)
p = 6.408712e+04

temperature (K)
T = 1.419203e+03

density (kg/m^3)
rho = 1.573070e-01

U velocity (m/s)
u = 3.241777e+03

Mach number
M = 4.431429e+00
```

Listing 4: `exit-props.txt`: example output file when using ‘verbose’ mode.

4.2 Core flow properties in a scramjet combustor

In this example, the flow through a scramjet combustor is considered.[5] To keep things simple, we will consider the case when there is no injection of fuel. In other words, this is a ducted supersonic flow. A visualisation of the flow field is depicted in Figure 4 which shows Mach number and temperature contours at various slices of constant x-ordinate. There is a ramp entry into the duct, and an oblique shock is formed. That shock reflects at the top wall and continues to reflect down the duct. At the end of the duct, the flow is expanded. The free stream conditions for the simulation are taken from estimates of a T4 shock tunnel condition. Those conditions are M_∞ = 6.44, p_∞ = 8.99 kPa, and T_∞ = 446 K.

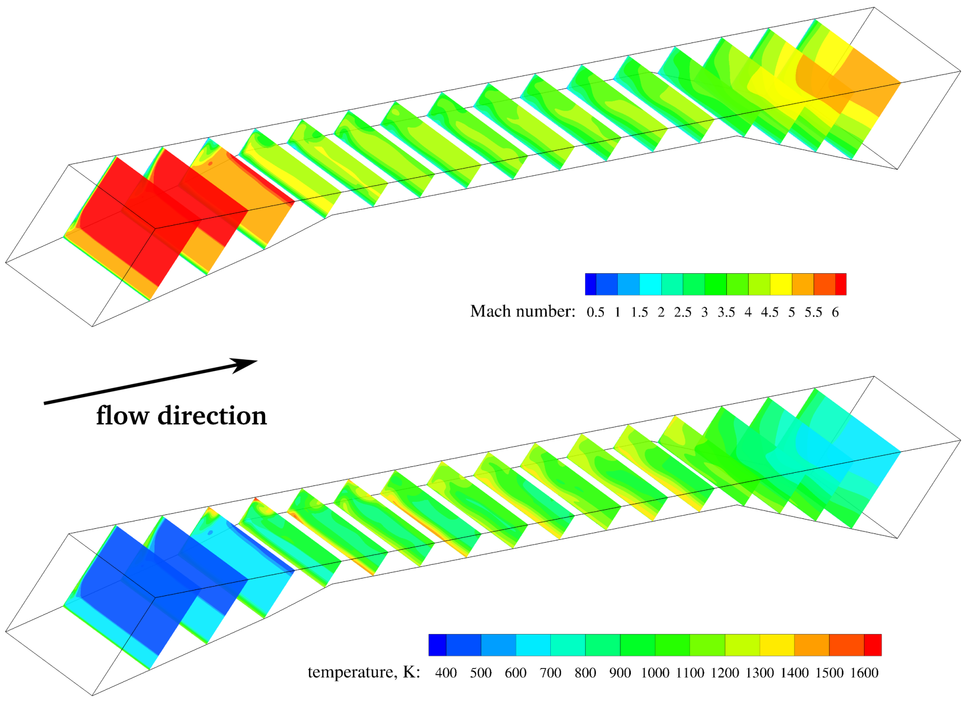

Mach number: 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6
flow direction
temperature, K: 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600

Figure 4: Slices of Mach number and temperature contours in a scramjet combustor.

We will use `Onedval` to compute the flux-conserved average of properties down the length of the combustor. We would also like to compare the properties of core flow (excluding the boundary layer) to the properties of a complete slice across the combustor. So, in this example, we will learn two aspects of using `Onedval`:

1. how to prepare a number of slices and use `Onedval` to process those slices; and
2. how to use a `filter_function` to exclude certain cells from the averaging.

Preparing multiple slices of data

The largest amount of work in this example is to prepare the data slices using Tecplot. For the averaging, we will focus our attention on just the combustor section which ranges from $x = 0.20$ m to $x = 0.56$ m. The particular slices of interest are shown in Figure 5. The process I describe here is not unique and can be achieved in a number of ways. The main steps in the process are:

[5]Thanks goes to Dr Bianca Capra for providing this example.

1. Record a macro in Tecplot for extracting a *single* slice of data.
2. Create a template macro based on the recorded macro.
3. Prepare a script to process all desired slices.
4. Run the script.

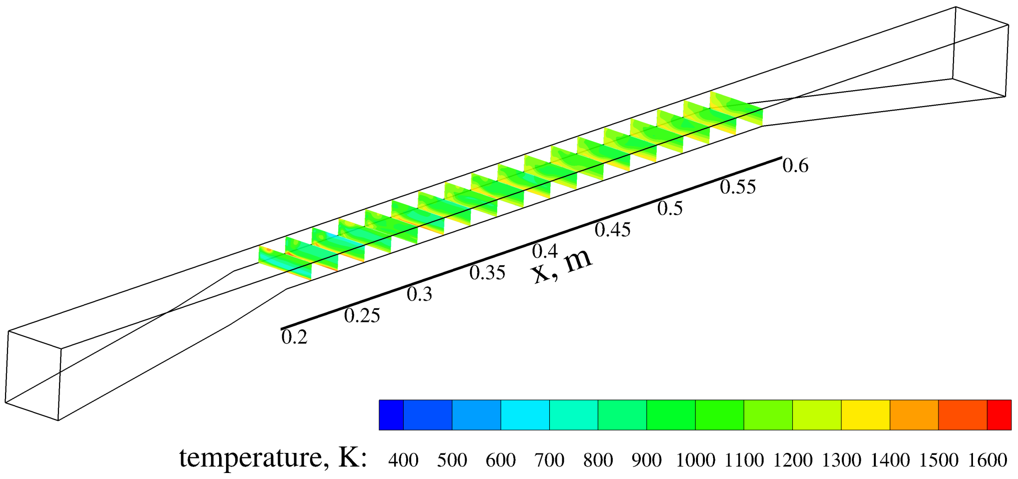


Figure 5: Slices of data in the combustor section only, ignoring the inlet and exit ramps. These are the slices of data passed to `Onedval`.

1. Record a macro We will record a macro in Tecplot which captures the process of extracting a single slice of data and writing that data to a file. It is best to restrict your actions to just those things when recording the macro so that you avoid unnecessary junk in the macro file. The reason we want to minimise the junk in the macro file is because we will need to inspect and edit this file to create a template macro for all of the slices. For this example, the steps to creating the macro file are:

1. Start Tecplot.
2. Start recording macro: `Scripting --> Record Macro`.
3. Load data file.
4. Extract a single slice: `Data --> Extract --> Slice from Plane`. In this particular example, I selected `Constant X` and entered a value of 0.2.
5. Write the data slice out: `File --> Write Data File`. Be sure to select just the slice which should be the last entry in the list of Zones. Also, be sure to select block, ASCII format.
6. Stop recording macro.
7. Quit Tecplot.

At the end of these steps I had a macro file called `slice.mcr` which is shown in Listing 5.

```
 1  #!MC 1300
 2  # Created by Tecplot 360 build 13.1.0.15185
 3  $!VarSet |MFBD| = '/work/examples/capra'
 4  $!READDATASET  '"|MFBD|/s10703_NoFuel.plt" '
 5    READDATAOPTION = NEW
 6    RESETSTYLE = YES
 7    INCLUDETEXT = NO
 8    INCLUDEGEOM = NO
 9    INCLUDECUSTOMLABELS = NO
10    VARLOADMODE = BYNAME
11    ASSIGNSTRANDIDS = YES
12    INITIALPLOTTYPE = CARTESIAN3D
13    VARNAMELIST = '"X" "Y" "Z" "P" "T" "U" "V" "W" "R" "M" <-- line truncated -->
14  $!GLOBALTHREED SLICE{NORMAL{X = 1}}
15  $!GLOBALTHREED SLICE{NORMAL{Z = 0}}
16  $!GLOBALTHREED SLICE{ORIGIN{X = 0.200000000000000011}}
17  $!CREATESLICEZONEFROMPLANE
18    SLICESOURCE = VOLUMEZONES
19    FORCEEXTRACTIONTOSINGLEZONE = YES
20    COPYCELLCENTEREDVALUES = NO
21  $!WRITEDATASET  "|MFBD|/slice.dat"
22    INCLUDETEXT = NO
23    INCLUDEGEOM = NO
24    INCLUDECUSTOMLABELS = NO
25    ASSOCIATELAYOUTWITHDATAFILE = NO
26    ZONELIST =  [2]
27    BINARY = NO
28    USEPOINTFORMAT = NO
29    PRECISION = 9
30    TECPLOTVERSIONTOWRITE = TECPLOTCURRENT
31  $!RemoveVar |MFBD|
```

Listing 5: `slice.mcr`: example macro file recorded using Tecplot.

2. Create a template for all macros Next, we will create a template based on the macro file we just created. We will insert special symbols into the template file so that it is easy to make a text substitution to generate a macro file for a particular slice. The steps for creating the template macro are:

1. Copy the single slice macro to a template file: `cp slice.mcr slice.mcr.template`

2. In the `slice.mcr.template`, edit line 16. Replace the specific x-ordinate with a symbol, `$XPOS`, to be used for substitution. Line 16 now becomes:

   ```
   $!GLOBALTHREED SLICE{ORIGIN{X = $XPOS}}
   ```

3. In the `slice.mcr.template`, edit line 21. Replace the specific output file name with a symbol, `$ONAME`, to be used for substitution. Line 21 no becomes:

   ```
   $!WRITEDATASET  "|MFBD|/$ONAME"
   ```

The final template file is shown in Listing 6.

```
#!MC 1300
# Created by Tecplot 360 build 13.1.0.15185
$!VarSet |MFBD| = '/work/examples/capra'
$!READDATASET  '"|MFBD|/s10703_NoFuel.plt" '
  READDATAOPTION = NEW
  RESETSTYLE = YES
  INCLUDETEXT = NO
  INCLUDEGEOM = NO
  INCLUDECUSTOMLABELS = NO
  VARLOADMODE = BYNAME
  ASSIGNSTRANDIDS = YES
  INITIALPLOTTYPE = CARTESIAN3D
  VARNAMELIST = '"X" "Y" "Z" "P" "T" "U" "V" "W" "R" "M" <-- line truncated -->
$!GLOBALTHREED SLICE{NORMAL{X = 1}}
$!GLOBALTHREED SLICE{NORMAL{Z = 0}}
$!GLOBALTHREED SLICE{ORIGIN{X = $XPOS}}
$!CREATESLICEZONEFROMPLANE
  SLICESOURCE = VOLUMEZONES
  FORCEEXTRACTIONTOSINGLEZONE = YES
  COPYCELLCENTEREDVALUES = NO
$!WRITEDATASET  "|MFBD|/$ONAME"
  INCLUDETEXT = NO
  INCLUDEGEOM = NO
  INCLUDECUSTOMLABELS = NO
  ASSOCIATELAYOUTWITHDATAFILE = NO
  ZONELIST =  [2]
  BINARY = NO
  USEPOINTFORMAT = NO
  PRECISION = 9
  TECPLOTVERSIONTOWRITE = TECPLOTCURRENT
$!RemoveVar |MFBD|
```

Listing 6: `slice.mcr.template`: example template macro file.

3. Build a script to process all slices We now need a script to call Tecplot in batch mode to extract data for all of the slices. This script would need to be tailored for the particular problem of interest. The script needs to do two things for each slice. First, the script should do a string substitution on the template macro file to create a macro file for the specific slice. Second, the script should call Tecplot in batch mode to process the macro file. An example Python script which coordinates this process is shown in Listing 7.

Most of this script should be straight-forward to understand. Some of the interesting features of the script are explained here. The Python `os` module is used to execute commands at the operating system level. In this example, the call to the system occurs twice in each loop using the `os.system()` function: first, to run `sed` over the template file to produce a macro for a particular slice; and, second, to run Tecplot in batch mode. This use of `os.system()` is not a particularly sophisticated way to interact with an operating system process, nor is there any error checking that the command executed properly. However, for this small script, using `os.system()` is simple and effective. If there are any errors, we will see it in the output then take appropriate action.

We mentioned the use of `sed` as one of the commands passed to the operating system. `sed` is a program which edit text streams in a programmatic way. If we expand out lines 30 and 31 of the Python script (Listing 7) for the first step when x = 0.2, the `sed` command would look like:

```
sed 's/$XPOS/0.200000/;s/$ONAME/slice-00.dat/' slice.mcr.template > slice-tmp.mcr
```

This command can be read as:

```
#!/usr/bin/env python
# Author: Rowan J. Gollan
# Date: 21-Feb-2013
# Place: University of Queensland, Brisbane, Australia
#

import os
from numpy import arange

# Unlikely to have to change these
TECPLOT_CMD = ’tec360’
MCR_TEMPLATE = ’slice.mcr.template’
TMP_MCR = ’slice-tmp.mcr’

# Set some parameters
xstart = 0.2     # m
xfinish = 0.56   # m
dx = 0.02        # m
bfn = ’slice’    # base file name

print "Creating slices from x= ", xstart
print "                  to x= ", xfinish
print "         at spacing dx= ", dx
# Begin creating slices
xs = arange(xstart, xfinish+0.5*dx, dx)
for i,x in enumerate(xs):
    print "slice at x= ", x
    # 1. Prepare the macro file using sed
    slc_fname = "%s-%02d.dat" % (bfn, i)
    cmd = "sed ’s/$XPOS/%.7f/;s/$ONAME/%s/’ %s > %s" % (x, slc_fname,
                                                        MCR_TEMPLATE, TMP_MCR)
    print cmd
    os.system(cmd)
    # 2. Call Tecplot to actually create the slice
    cmd = "%s -b %s" % (TECPLOT_CMD, TMP_MCR)
    print cmd
    os.system(cmd)
print "Done."
```

Listing 7: `create-slices.py`: example script to extract all slices.

Open the file `slice.mcr.template` and process it line-by-line according to the instructions enclosed in `’ ’`. Place the transformed output in `slice.mcr`. The particular instructions enclosed in `’ ’` say: replace any occurrences of `$XPOS` with `0.2000000` and repalce an occurrences of `$ONAME` with `slice-00.dat`

4. Run the script I run this script in my terminal using:

```
> python create-slices.py
```

That is the hard part done. The next step is to sit back and let `Onedval` work on the slices.

Running `Onedval` on the multiple slices

Before running `Onedval`, we need to prepare a configuration script. The configuration file for this example is shown in Listing 8. The CFD simulation for this example was done with CFD++ [6]; this is reflected in `variable_map`. To process the multiple data slices, the command is:

```
> onedval all-cells.config slice-??.dat
```

Note that `slice-??.dat` will be expanded out into a long list of file names by the shell. After running this command, the data for each of the slices is written in rows into the file `ac-1D-props.txt`.

```
# list of species
species = ["H2", "H", "O", "O2", "H2O", "H2O2", "HO2", "OH", "NO", "NO2",
           "HNO", "N", "N2"]

# A mapping of variable names for onedval to those
# found in the Tecplot file
variable_map = {'x':'X', 'y':'Y', 'z':'Z',
               'u':'U',
               'v':'V',
               'w':'W',
               'rho':'R',
               'p':'P',
               'T':'T',
               'M':'M',
               'h0':'Enthalpy_total' }

# list of types of one-D averages to compute
one_d_averages = ['flux-conserved']

# A scaling factor if coordinates are NOT in metres
grid_scale = 1.0

# Output properties
one_d_outputs = ['p', 'T', 'rho', 'u', 'M']

# Integrated quantities
integrated_outputs = ['mass flux', 'species mass flux']

# output options
output_file = 'ac-1D-props.dat'
output_format = 'as_data_file'
```

Listing 8: `all-cells.config`: example configuration file with the `variable_map` set to work with CFD++-generated data.

Using a filter function

For some analyses, it is useful to consider just certain cells in the averaging. For example, some analyses are based on the core flow properties. It would be desirable then to have a way to filter out the boundary layer cells. In `Onedval`, that facility is provided by using a filter function. The user writes a Python function into the configuration script. Each cell in the area is passed to the function, in turn. The job of the function is to apply some criteria to determine whether or not the cell should be included in the averaging. Thus the user writes the function to accept one argument, a cell, and return one Boolean value: `True` to include the cell, or `False` to exclude the cell.

Let's give an example to make that description more concrete. Here we would like to *exclude* the cells in the boundary layer. We will define the boundary layer as the edge where the total enthalpy of the flow is less than 99.5% of the free stream total enthalpy. So we will test all cells and exclude the ones with a local total enthalpy less than 99.5% of the free stream total enthalpy. The Python function to apply this criterion is:

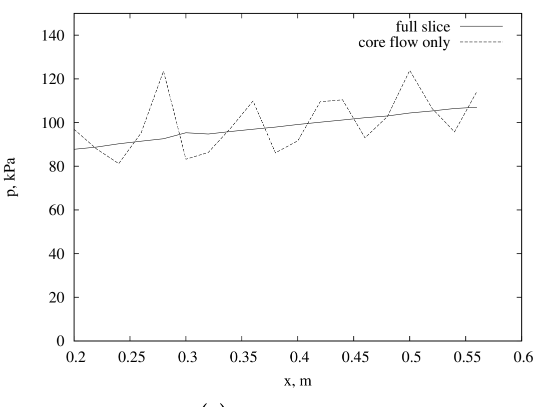
```
h0_inf = 3.88951e6 # J/kg
bl_criterion = 0.995*h0_inf # 99.5% of free stream total enthalpy
def core_flow(cell):
    global bl_criterion
    if ( cell.get('Enthalpy_total') < bl_criterion ) :
        # Consider cell to be in boundary layer so it's NOT included
        return False
    else:
        # Consider cell in core flow, so include it
        return True
```

Note that for this particular example, the free stream total enthalpy is 3.88951×10^6 J/kg. Also note that the function refers to the variable `bl_criterion` which was computed and set once *outside* of the function. Since the variable is outside of the function, the `global` keyword is needed inside the function to tell the Python interpreter where to look. This Python function along with the `filter_function` option is added to the configuration file. The new file, `core-flow.config`, is shown in Listing 9.

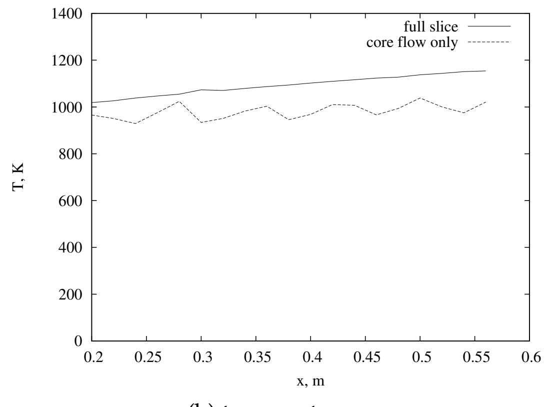
The one-dimensionalised values for full slices across the duct are compared to values just considering the core flow in Figure 6. The core flow has been defined as flow where the total enthalpy is 99.5% or greater of the free stream total enthalpy. The oscillations in the core flow properties at different x-locations reflect the location of the shock reflections in the combustor. The core flow properties vary compared to the complete slice data because the influence of the shock is not included when it coincides with the boundary layer near the wall.

```
# list of species
species = ["H2", "H", "O", "O2", "H2O", "H2O2", "HO2", "OH", "NO", "NO2",
           "HNO", "N", "N2"]

# A mapping of variable names for onedval to those
# found in the Tecplot file
variable_map = {'x':'X', 'y':'Y', 'z':'Z',
             'u':'U',
             'v':'V',
             'w':'W',
             'rho':'R',
             'p':'P',
             'T':'T',
             'M':'M',
             'h0':'Enthalpy_total' }

# list of types of one-D averages to compute
one_d_averages = ['flux-conserved']

# A scaling factor if coordinates are NOT in metres
grid_scale = 1.0

# Output properties
one_d_outputs = ['p', 'T', 'rho', 'u', 'M']

# Integrated quantities
integrated_outputs = ['mass flux', 'species mass flux']

# output options
output_file = 'cf-1D-props.dat'
output_format = 'as_data_file'

# a filter function
h0_inf = 3.88951e6 # J/kg
bl_criterion = 0.995*h0_inf # 99.5% of free stream total enthalpy
def core_flow(cell):
    global bl_criterion
    if ( cell.get('Enthalpy_total') < bl_criterion ) :
        # Consider cell to be in boundary layer so it's NOT included
        return False
    else:
        # Consider cell in core flow, so include it
        return True

filter_function = core_flow
```

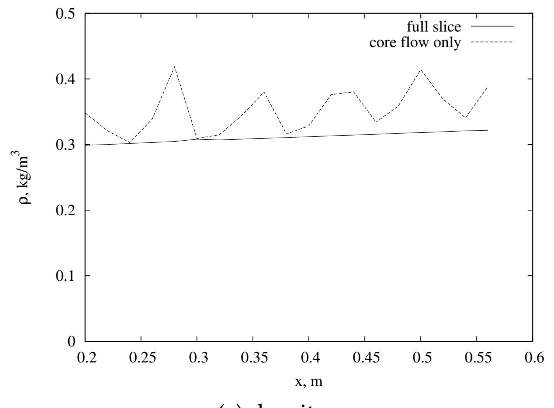
Listing 9: `core-flow.config`: the same configuration file as in Listing 8 except that now a filter function is declared.



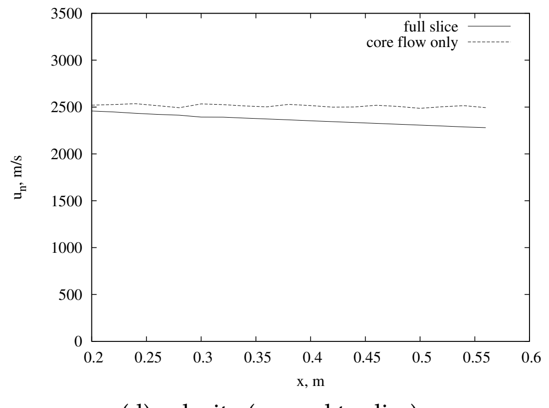
(a) pressure



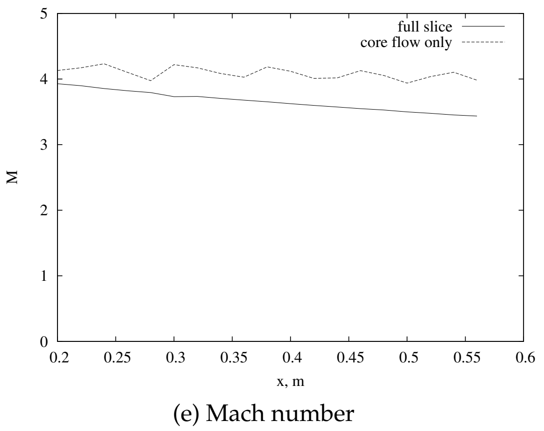
(b) temperature



(c) density



(d) velocity (normal to slice)



(e) Mach number

Figure 6: One-dimensionalised properties in a scramjet combustor. Properties for a full slice across the combustor and just considering the core flow are shown.

References

[1] The Compressible-Flow CFD Project. `http://www.mech.uq.edu.au/cfcfd`.

[2] R. A. Baurle and R. I. Gaffney. Extraction of one-dimensional flow properties from multidimensional data sets. *AIAA Journal of Propulsion and Power*, 24(4):704–714, 2008.

[3] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[4] The Tecplot Homepage. `http://www.tecplot.com`.

[5] The VULCAN Homepage. `http://vulcan-cfd.larc.nasa.gov`.

[6] CFD++ Homepage. `http://www.metacomptech.com`.

A Code listing

Rather than try to describe in detail how the calculations in `Onedval` are performed, the source code is included here. The source code for `Onedval` is in three files. The listing of the main coordinating program is in the file `onedval.py`. This file is largely housekeeping. The module `cell.py` houses the calculations related to tri and quad cell geometries. The finite-element slice data supplied to `Onedval` is interpolated onto a set of cells of either tri or quad type. The functions for calculating one-dimensionalised averages are placed in the `prop_avg.py` module.

onedval.py

```
#!/usr/bin/env python
"""
Python program to compute one-dimensionalized quantities from 2D data.

This program is supplied with two inputs by the user:
1) a previously prepared Tecplot slice of data in ASCII block format; and
2) a config file to coordinate the calculation.
The program will then compute one-dimensionalized quantities from the
the 2D surface based on the methods selected by the user.

.. Author: Rowan J. Gollan

.. Versions:
   12-Feb-2013  Initial coding.
"""

import sys, gc

from e3prep import select_gas_model
from libprep3 import get_gas_model_ptr, vabs
from cell import create_cells_from_slice, area
from prop_avg import *
from copy import copy

output_formats = [’verbose’, ’as_data_file’]

default_var_map = {’x’:’x’, ’y’:’y’, ’z’:’z’, ’u’:’u’, ’v’:’v’, ’w’:’w’,
                   ’rho’:’rho’, ’p’:’p’, ’T’:’T’, ’M’:’M’, ’h0’:’h0’}

def print_usage():
    print ""
    print "Usage: onedval CONFIG INPUT_FILE(S)"
    print ""
    print "where:"
    print "CONFIG -- name of config file to control calculation"
    print "INPUT_FILE(S)  -- a list of one or more Tecplot files with slice data"
    print ""

pretty_var_names = {’rho’:’density (kg/m^3)’,
                    ’p’:’pressure (Pa)’,
                    ’T’:’temperature (K)’,
                    ’M’:’Mach number’,
                    ’u’:’U velocity (m/s)’,
                    ’v’:’V velocity (m/s)’,
                    ’w’:’W velocity (m/s)’}

def pretty_print_props(f, props, species, outputs):
    for o in outputs:
        if o in pretty_var_names:
            f.write("%s\n" % pretty_var_names[o])
            f.write("%s = %.6e\n" % (o, props[o]))
        elif k in species:
            f.write("mass fraction of %s : %.6e\n" % (o, props[o]))
        else:
            f.write("%s : %.6e\n" % (o, props[o]))
        f.write("\n")
    return

short_one_d_av_names = {’area-weighted’:’a-w’,
                        ’mass-flux-weighted’:’m-w’,
                        ’flux-conserved’:’f-c’}
units = {’rho’:’kg/m^3’,
         ’p’:’Pa’,
         ’T’:’K’,
         ’M’:’-’,
         ’u’:’m/s’,
         ’v’:’m/s’,
         ’w’:’m/s’,
         ’mass flux’:’kg/s’,
         ’momentum flux’:’kg.m/s^2’,
         ’energy flux’:’W’}

column_names = {’mass flux’ : ’mass-flux’,
                ’momentum flux’ : ’mom.-flux’,
                ’energy flux’ : ’energy-flux’ }

def data_file_header(f, one_d_av, one_d_out, int_out, species):
    f.write("# x[m]     y[m]     z[m]    area[m^2]     ")

    for av in one_d_av:
        for o in one_d_out:
            f.write("%s[%s]:%s    " % (o, units[o], short_one_d_av_names[av]))

    for i_out in int_out:
        if i_out == ’species mass flux’:
            for sp in species:
                f.write("%s-flux[kg/s]    " % sp)
        else:
            f.write("%s[%s]    " % (column_names[i_out], units[i_out]))
    f.write("\n")
    return

def data_file_row(f, pos, area, phis, int_quants, one_d_av, one_d_out, int_out, species):
    f.write("%20.12e %20.12e %20.12e %20.12e " % (pos.x, pos.y, pos.z, area))
    for av in one_d_av:
        for o in one_d_out:
            f.write("%20.12e " % (phis[av][o]))

    for i_out in int_out:
        if i_out == ’species mass flux’:
            for sp in species:
                f.write("%20.12e " % int_quants[’species mass flux’][sp])
        else:
            f.write("%20.12e  " % (int_quants[i_out]))
    f.write("\n")


def main():
    """
    Top-level function for the onedval program.
    """
    print "onedval: A program to compute integrated and one-dimensionalised quantities."
    print "onedval: Beginning."
    # 0. Gather command-line info
    if len(sys.argv) < 3:
        print "At least two arguments are required."
        print_usage()
        sys.exit(1)

    config_file = sys.argv[1]

    # 1. Gather info from config file
    # Set some defaults.
    # If set to ’None’, we expect to find something from the user.
    cfg = {}
    try:
        execfile(config_file, globals(), cfg)
    except IOError:
        print "There was a problem reading the config file: ", config_file
        print "Check that is conforms to Python syntax."
        print "Bailing out!"
        sys.exit(1)
```

```
print "onedval: Setting up gas model"
# 1a. Look at species and setup gas model
if not 'species' in cfg:
    print "No 'species' list was found, so defaulting to ['air']."
    cfg['species'] = ['air']
select_gas_model("thermally perfect gas", cfg['species'])
gmodel = get_gas_model_ptr()
nsp = gmodel.get_number_of_species()

print "onedval: Checking over user inputs"
# 1b. Check for variable map
if not 'variable_map' in cfg:
    print "No 'variable_map' was set so the following default is used:"
    print default_var_map
    cfg['variable_map'] = default_var_map

# 1c. Look for one_d_averages methods
if not 'one_d_averages' in cfg:
    print "No list of 'one_d_averages' was set."
    print "The default method of 'flux-conserved' will be used."
    cfg['one_d_averages'] = ['flux-conserved']

# 1d. Look for grid_scale
if not 'grid_scale' in cfg:
    print "No 'grid_scale' was set."
    print "The default value of 1.0 will be used."
    cfg['grid_scale'] = 1.0

# 1e. Look for one_d_outpus
if not 'one_d_outputs' in cfg:
    cfg['one_d_outputs'] = []

# 1f. Look for integrated_outputs
if not 'integrated_outputs' in cfg:
    cfg['integrated_outputs'] = []

# 1g. Looking for output options
if not 'output_file' in cfg:
    print "No 'output_file' was set."
    print "An output file name must be set by the user."
    print "Bailing out!"
    sys.exit(1)

if not 'output_format' in cfg:
    print "No 'output_format' was set."
    print "An output format must be set by the user."
    print "Bailing out!"
    sys.exit(1)

if not cfg['output_format'] in output_formats:
    print "The selected output format: ", cfg['output_format']
    print "is not one of the available options. The available options are:"
    for o in output_formats:
        print "'%s'" % o
    print "Bailing out!"
    sys.exit(1)


# 2. Read data from slices and process
print "onedval: Reading in data from slice(s)"
f = open(cfg['output_file'], 'w')
phis = {}
int_quants = {}
if cfg['output_format'] == 'as_data_file':
    data_file_header(f, cfg['one_d_averages'], cfg['one_d_outputs'],
                     cfg['integrated_outputs'], cfg['species'])
```

```
for slice_file in sys.argv[2:]:
    print "onedval: Creating cells from slice: ", slice_file
    cells = create_cells_from_slice(slice_file, cfg['variable_map'], cfg['grid_scale'])
    print "Total number of cells created from slice: ", len(cells)
    # 2a. apply filtering if required
    if 'filter_function' in cfg:
        print "Using filter function to remove unwanted or unimportant cells."
        cells = filter(cfg['filter_function'], cells)
        print "Number of cells after filtering: ", len(cells)

    # 3. Do some work.
    print "onedval: Doing the requested calculations"
    if cfg['output_format'] == 'verbose':
        f.write("------------------- onedval output ---------------------\n\n")
        f.write("number of cells in averaging:\n")
        f.write("ncells = %d\n" % len(cells))
        f.write("cumulative area of cells (m^2):\n")
        f.write("area = %.6e\n" % area(cells))

    # 3a. Compute requested integrated quantities (if required)
    fluxes = compute_fluxes(cells, cfg['variable_map'], cfg['species'], gmodel)

    if cfg['output_format'] == 'verbose':
        if len(cfg['integrated_outputs']) > 0:
            print "onedval: Writing out integrated quantities"
            f.write("\n---------------------\n")
            f.write("Integrated quantities\n")
            f.write("---------------------\n")
            f.write("\n")

            for flux in cfg['integrated_outputs']:
                if flux == 'mass flux':
                    f.write("mass flux (kg/s)\n")
                    f.write("m_dot = %.6e\n\n" % fluxes['mass'])
                elif flux == 'momentum flux':
                    f.write("momentum flux (kg.m/s^2)\n")
                    f.write("mom_dot = %s\n\n" % fluxes['mom'])
                elif flux == 'energy flux':
                    f.write("energy flux (W)\n")
                    f.write("e_dot = %.6e\n\n" % fluxes['energy'])
                elif flux == 'species mass flux':
                    for sp in cfg['species']:
                        isp = gmodel.get_isp_from_species_name(sp)
                        f.write("mass flux of %s (kg/s)\n" % sp)
                        f.write("m%s_dot = %.6e\n\n" % (sp, fluxes['species'][isp]))
                else:
                    print "Requested integrated quantity: ", flux
                    print "is not part of the list of available integrated quantities."
                    print "Bailing out!"
                    sys.exit(1)

    if cfg['output_format'] == 'as_data_file':
        for flux in cfg['integrated_outputs']:
            if flux == 'mass flux':
                int_quants['mass flux'] = fluxes['mass']
            elif flux == 'momentum flux':
                int_quants['momentum flux'] = vabs(fluxes['mom'])
            elif flux == 'energy flux':
                int_quants['energy flux'] = fluxes['energy']
            elif flux == 'species mass flux':
                int_quants['species mass flux'] = {}
                for sp in cfg['species']:
                    isp = gmodel.get_isp_from_species_name(sp)
                    int_quants['species mass flux'][sp] = fluxes['species'][isp]
            else:
                print "Requested integrated quantity: ", flux
                print "is not part of the list of available integrated quantities."
                print "Bailing out!"
                sys.exit(1)

        # 3b. Compute requested one_d_properties
        if cfg['output_format'] == 'verbose':
            if len(cfg['one_d_averages']) > 0:
                print "onedval: Writing out one-dimensionalised quantities"
                f.write("\n------------------------------\n")
                f.write("One-dimensionalised quantities\n")
                f.write("------------------------------\n")

        phis_all = {}
        for avg in cfg['one_d_averages']:
            if avg == 'area-weighted':
                phis = area_weighted_avg(cells, cfg['one_d_outputs'], cfg['variable_map'])
                phis_all[avg] = copy(phis)
                if cfg['output_format'] == 'verbose':
                    f.write("-- area-weighted average --\n\n")
                    pretty_print_props(f, phis, cfg['species'], cfg['one_d_outputs'])
                    f.write("\n")
            elif avg == 'mass-flux-weighted':
                phis = mass_flux_weighted_avg(cells, cfg['one_d_outputs'], cfg['variable_map'])
                phis_all[avg] = copy(phis)
                if cfg['output_format'] == 'verbose':
                    f.write("-- mass-flux-weighted average --\n\n")
                    pretty_print_props(f, phis, cfg['species'], cfg['one_d_outputs'])
                    f.write("\n")
            elif avg == 'flux-conserved':
                phis = stream_thrust_avg(cells, cfg['one_d_outputs'], cfg['variable_map'], cfg['species'], gmodel)
                phis_all[avg] = copy(phis)
                if cfg['output_format'] == 'verbose':
                    f.write("-- flux-conserved average --\n\n")
                    pretty_print_props(f, phis, cfg['species'], cfg['one_d_outputs'])
                    f.write("\n")
            else:
                print "Requested one-D averaging method: ", avg
                print "is not known or not implemented."
                print "Bailing out!"
                sys.exit(1)

        if cfg['output_format'] == 'as_data_file':
            A = area(cells)
            pos = avg_pos(cells, cfg['variable_map'])
            data_file_row(f, pos, A, phis_all, int_quants, cfg['one_d_averages'], cfg['one_d_outputs'],
                          cfg['integrated_outputs'], cfg['species'])

    f.close()
    print "onedval: Done."

if __name__ == '__main__':
    main()
```

cell.py

```
# Author: Rowan J. Gollan
# Place: UQ, Brisbane, Queensland, Australia
# Date: 25-Jun-2012

from libprep3 import *

def tri_centroid(p0, p1, p2):
    """
    Compute centroid of triangle given 3 points

           p1
           +
          / \
         /   \
        /     \
       +-------+
       p0      p2
    """
    return (1.0/3.0)*(p0 + p1 + p2)

def tri_area(p0, p1, p2):
    """
    Compute the area of a triangle.
    """
    vector_area = 0.5*cross(p1-p0, p2-p1)
    return vabs(vector_area)

def tri_normal(p0, p1, p2):
    """
    Compute the normal direction of a triangle in a plane.
    """
    vector_area = 0.5*cross(p1-p0, p2-p1)
    return unit(vector_area)

class Cell(object):
    """
    Python base class to abstract the behaviour of
    all cells.
    """
    def __init__(self, data):
        self._data = data.copy()
        return

    def variables(self):
        return self._data.keys()

    def get(self, k, default=’no value’):
        return self._data.get(k, default)

    def area(self):
        return self._area

    def normal(self):
        return self._normal

    def centroid(self):
        return self._centroid

class QuadCell(Cell):
    """
    A quadrilateral cell in a 2D plane.
    """
    count = 0
    def __init__(self, data, pts):
        Cell.__init__(self, data)
        self._centroid = quad_centroid(pts[0], pts[1], pts[2], pts[3])
        self._normal = unit(quad_normal(pts[0], pts[1], pts[2], pts[3]))
        self._area = quad_area(pts[0], pts[1], pts[2], pts[3])
        QuadCell.count = QuadCell.count + 1
        return

class TriCell(Cell):
    """
    A triangular cell in a 2D plane.
    """
    count = 0
    def __init__(self, data, pts):
        Cell.__init__(self, data)
        self._centroid = tri_centroid(pts[0], pts[1], pts[2])
        self._normal = tri_normal(pts[0], pts[1], pts[2])
        self._area = tri_area(pts[0], pts[1], pts[2])
        TriCell.count = TriCell.count + 1
        return

def unique(lst):
    seen = set()
    unique = []
    for item in lst:
        if not item in seen:
            unique.append(item)
            seen.add(item)
    return unique

def create_cell(idx, data, cell_cnrs, var_map, scale):
    c_list = unique(cell_cnrs[idx])
    pts = []
    xlabel = var_map[’x’]
    ylabel = var_map[’y’]
    zlabel = var_map[’z’]
    for cnr in c_list:
        i = cnr-1
        pts.append(scale*Vector3(data[xlabel][i], data[ylabel][i], data[zlabel][i]))

    if len(pts) == 3:
        pC = tri_centroid(pts[0], pts[1], pts[2])
        area = tri_area(pts[0], pts[1], pts[2])
        a0 = tri_area(pts[1], pts[2], pC)
        a1 = tri_area(pts[0], pts[2], pC)
        a2 = tri_area(pts[0], pts[1], pC)
        d = {}
        for v in data.keys():
            d0 = data[v][c_list[0]-1]
            d1 = data[v][c_list[1]-1]
            d2 = data[v][c_list[2]-1]
            d[v] = (a0*d0 + a1*d1 + a2*d2)/area
        return TriCell(d, pts)
    elif len(pts) == 4:
        pC = quad_centroid(pts[0], pts[1], pts[2], pts[3])
        area = quad_area(pts[0], pts[1], pts[2], pts[3])
        p01 = 0.5*(pts[0]+pts[1])
        p12 = 0.5*(pts[1]+pts[2])
        p23 = 0.5*(pts[2]+pts[3])
        p30 = 0.5*(pts[3]+pts[0])
        a0 = quad_area(pC, p12, pts[2], p23)
        a1 = quad_area(pC, p23, pts[3], p30)
        a2 = quad_area(pC, p30, pts[0], p01)
        a3 = quad_area(pC, p01, pts[1], p12)
        d = {}
        for v in data.keys():
            d0 = data[v][c_list[0]-1]
            d1 = data[v][c_list[1]-1]
            d2 = data[v][c_list[2]-1]
            d3 = data[v][c_list[3]-1]
            d[v] = (a0*d0 + a1*d1 + a2*d2 + a3*d3)/area
        return QuadCell(d, pts)
    else:
        print "Unknown cell type with num points= ", len(pts)
        print "Bailing out!"
        import sys
        sys.exit(1)
    return

def create_cells_from_slice(fname, var_map, scale):
    f = open(fname, 'r')
    # Read Title line and discard
    f.readline()
    # Gather variables
    var_list = []
    while 1:
        line = f.readline()
        if line.startswith('ZONE'):
            break
        if line.startswith('DATASETAUXDATA'):
            continue
        if line.startswith('VARIABLES'):
            tks = line.split()
            var_list.append(tks[2].strip('"'))
        else:
            v = line.strip().strip('"')
            var_list.append(v)
    # Read Strand and Soltime info and discard
    f.readline()
    # Read Nodes, elements -- pick these up
    line = f.readline()
    tks = line.split()
    nnodes = int(tks[0].split("=")[1][:-1])
    nelems = int(tks[1].split("=")[1][:-1])
    # Read datapacking and discard
    f.readline()
    # Read datatype and discard
    f.readline()
    # Read all data and place in tks
    tks = f.read().split()
    f.close()
    # Now pick up some data
    data = {}
    pos = 0
    for v in var_list:
        data[v] = []
        for i in range(nnodes):
            data[v].append(float(tks[pos]))
            pos = pos + 1
    cell_cnrs = []
    for i in range(nelems):
        cell_cnrs.append([int(tks[pos]), int(tks[pos+1]), int(tks[pos+2]), int(tks[pos+3])])
        pos = pos + 4

    cells = []
    for i in range(nelems):
        cells.append(create_cell(i, data, cell_cnrs, var_map, scale))

    return cells

def area(cells):
    A = 0.0
    for c in cells:
        A = A + c.area()
    return A
```

prop_avg.py

```
# Author: Rowan J. Gollan
# Place: UQ, Brisbane, Queensland, Australia
# Date: 26-Jun-2012
#
# This module provides methods to give
# one-dimensionalised (averaged) flow properties
# based on a variety of techniques.
#

import sys

from math import sqrt
from libprep3 import Vector3, dot
from libprep3 import Gas_data, set_massf
from cfpylib.nm.zero_solvers import secant
from scipy.optimize import minimize

def area(cells):
    A = 0.0
    for c in cells:
        A = A + c.area()
    return A

def avg_pos(cells, var_map):
    x = 0.0; y = 0.0; z = 0.0
    xlabel = var_map[’x’]
    ylabel = var_map[’y’]
    zlabel = var_map[’z’]
    A = area(cells)
    for c in cells:
        dA = c.area()
        x += c.get(xlabel)*dA
        y += c.get(ylabel)*dA
        z += c.get(zlabel)*dA
    x /= A
    y /= A
    z /= A
    return Vector3(x, y, z)

def oriented_normal(n):
    """Orients the normal in a consistent direction.

    Tecplot does not list the corners of cells in a consistent
    manner, so the orientation (outward or inward facing)
    of the computed normals can differ between adjacent cells.
    Presently, we’ll enforce the assumption
    of a positive-x sense to all normals.
    """
    return Vector3(abs(n.x), n.y, n.z)


def compute_fluxes(cells, var_map, species, gmodel):
    f_mass = 0.0
    f_mom = Vector3(0.0, 0.0, 0.0)
    f_energy = 0.0
    f_sp = [0.0,]*len(species)
    nsp = gmodel.get_number_of_species()
    N = Vector3(0.0, 0.0, 0.0)
    A = 0.0
    rholabel = var_map[’rho’]
    plabel = var_map[’p’]
    ulabel = var_map[’u’]
    vlabel = var_map[’v’]
    wlabel = var_map[’w’]
    h0label = var_map[’h0’]
    for c in cells:
        dA = c.area()
        n = oriented_normal(c.normal())
        rho = c.get(rholabel)
        p = c.get(plabel)
        vel = Vector3(c.get(ulabel),
                      c.get(vlabel),
                      c.get(wlabel))
        h0 = c.get(h0label)
        # Add increments
        u_n = dot(vel, n)
        f_mass = f_mass + rho*u_n*dA
        f_mom = f_mom + (rho*u_n*vel + p*n)*dA
        f_energy = f_energy + (rho*u_n*h0)*dA
        for isp, sp in enumerate(species):
            # Try to grab mass fraction or set to 1.0 if not found
            massf = c.get(sp, 1.0)
            f_sp[isp] = f_sp[isp] + rho*massf*u_n*dA
    return {’mass’: f_mass, ’mom’: f_mom, ’energy’: f_energy, ’species’:f_sp}

def area_weighted_avg(cells, props, var_map):
    phis = dict.fromkeys(props, 0.0)
    area = 0.0
    for c in cells:
        dA = c.area()
        area = area + dA
        for p in props:
            label = p
            if p in var_map:
                label = var_map[p]
            phis[p] = phis[p] + c.get(label)*dA

    for p in props:
        phis[p] = phis[p]/area

    return phis

def mass_flux_weighted_avg(cells, props, var_map):
    phis = dict.fromkeys(props, 0.0)
    f_mass = 0.0
    rholabel = var_map[’rho’]
    ulabel = var_map[’u’]
    vlabel = var_map[’v’]
    wlabel = var_map[’w’]
    for c in cells:
        dA = c.area()
        rho = c.get(rholabel)
        vel = Vector3(c.get(ulabel),
                      c.get(vlabel),
                      c.get(wlabel))
        n = oriented_normal(c.normal())
        w = rho*dot(vel, n)
        f_mass = f_mass + w*dA
        for p in props:
            label = p
            if p in var_map:
                label = var_map[p]
            phis[p] = phis[p] + c.get(label)*w*dA

    for p in props:
        phis[p] = phis[p]/f_mass

    return phis


def stream_thrust_avg(cells, props, var_map, species, gmodel):
    f_mass = 0.0
    f_mom = Vector3(0.0, 0.0, 0.0)
```

```
f_energy = 0.0
f_sp = [0.0,]*len(species)
nsp = gmodel.get_number_of_species()
N = Vector3(0.0, 0.0, 0.0)
A = 0.0
rholabel = var_map['rho']
plabel = var_map['p']
ulabel = var_map['u']
vlabel = var_map['v']
wlabel = var_map['w']
h0label = var_map['h0']
for c in cells:
    dA = c.area()
    n = oriented_normal(c.normal())
    rho = c.get(rholabel)
    p = c.get(plabel)
    vel = Vector3(c.get(ulabel),
                  c.get(vlabel),
                  c.get(wlabel))
    h0 = c.get(h0label)
    # Add increments
    u_n = dot(vel, n)
    f_mass = f_mass + rho*u_n*dA
    f_mom = f_mom + (rho*u_n*vel + p*n)*dA
    f_energy = f_energy + (rho*u_n*h0)*dA
    if nsp > 1:
        for isp, sp in enumerate(species):
            massf = c.get(sp)
            f_sp[isp] = f_sp[isp] + rho*massf*u_n*dA
    A = A + dA
    N = N + n*dA

N = N / A
f_mom_s = dot(f_mom, N)
Q = Gas_data(gmodel)
if nsp > 1:
    massf = [ f_isp/f_mass for f_isp in f_sp ]
    set_massf(Q, gmodel, massf)
else:
    Q.massf[0] = 1.0

LARGE_PENALTY = 1.0e6

def f_to_minimize(x):
    rho, T, u = x
    if rho < 0.0 or T < 0.0:
        # Give a big penalty
        return LARGE_PENALTY
    # Use equation of state to compute other thermo quantities
    Q.rho = rho
    Q.T[0] = T
    flag = gmodel.eval_thermo_state_rhoT(Q)
    if flag != 0:
        # If there are problems, then these are NOT good values
        # so return a large error
        return LARGE_PENALTY
    h = gmodel.mixture_enthalpy(Q)
    p = Q.p
    # Compute errors
    fmass_err = abs(f_mass - rho*u*A)/(abs(f_mass) + 1.0)
    fmom_err = abs(f_mom_s - (rho*u*u + p)*A)/(abs(f_mom_s) + 1.0)
    fe_err = abs(f_energy - (rho*u*A*(h + 0.5*u*u)))/(abs(f_energy) + 1.0)
    # Total error is the sum
    return fmass_err + fmom_err + fe_err

# Compute an initial guess based on mass-flux weighted averages
mfw_props = mass_flux_weighted_avg(cells, ['rho', 'p', 'T', 'u', 'v', 'w', 'M'], var_map)
u = sqrt(mfw_props['u']**2 + mfw_props['v']**2 + mfw_props['w']**2)
guess = [mfw_props['rho'], mfw_props['T'], u]
result = minimize(f_to_minimize, guess, method='Nelder-Mead')
rho, T, u = result.x
# Check the results make sense
if rho < 0.0 or T < 0.0:
    result.success = False

if not result.success:
    # Try again but use area-weigthed average as starting point
    aw_props =  area_weighted_avg(cells, ['rho', 'p', 'T', 'u', 'v', 'w', 'M'], var_map)
    u = sqrt(aw_props['u']**2 + aw_props['v']**2 + aw_props['w']**2)
    guess = [aw_props['rho'], aw_props['T'], u]
    result = minimize(f_to_minimize, guess, method='Nelder-Mead')
    rho, T, u = result.x
    if not result.success:
        print "The minimizer had difficulty finding the best set of one-d values: rho, T, u"
        print "result [rho, T, u]= ", result.x

    if rho < 0.0 or T < 0.0:
        print "The minimizer claims a successful finish, but the gas state is non-physical."
        print "result [rho, T, u]= ", result.x

Q.rho = rho; Q.T[0] = T
gmodel.eval_thermo_state_rhoT(Q)

phis = dict.fromkeys(props, 0.0)

for k in phis:
    if k == 'rho':
        phis[k] = rho
    elif k == 'p':
        phis[k] = Q.p
    elif k == 'T':
        phis[k] = T
    elif k == 'M':
        phis[k] = u/Q.a
    elif k == 'u':
        phis[k] = u
    elif k in species:
        isp = gmodel.get_isp_from_species_name(k)
        phis[k] = massf[isp]
    else:
        print "Do not know what to do for flux-conserved average of:", k

return phis
```