Rowan J. Gollan and Peter A. Jacobs

The User's Guide to the

Dlang `gas` Package

including `gas-calc` *and the API*

December 2016

Contents

Introduction

The Dlang `gas` package was written to support the `Eilmer` compressible flow simulation code. The package provides calculation services related to the thermodynamic and diffusive properties of a gas mixture and is a useful tool in its own right. This report documents the use of the `gas` package as a stand-alone tool for doing calculations related to gas properties. Think of it as a small laboratory for investigating gas properties. Some examples of the package's use include:

- computing the viscosity for a mixture of gases;

- plotting the enthalpy of a gas over a range of temperatures;

- evaluating equation-of-state expressions to find thermodynamic state; and

- building a thermodynamic cycle analysis tool.

The `gas` package provides two means of use: a) as a library for inclusion in D language programs; or b) through small scripts (or programs) written in Lua and executed by the `gas-calc` program that accompanies this library. In this report, we document the use of the gas package in Lua scripts that are fed to `gas-calc` for processing. The same functions are available to your Lua scripts in the pre- and post-processing stages of your flow simulations with `Eilmer`.

To give a feel for the interaction with the package, we will start by showing a small Lua script that prints the density of air at typical room temperature and pressure. This is shown in Listing 1.

```
gm = GasModel:new{"ideal-air-gas-model.lua"}
Q = GasState:new{gm}
Q.p = 1.0e5
Q.T = 300.0
gm:updateThermoFromPT(Q)
print("Density of air= ", Q.rho)
```

Listing 1: A Lua script for `gas-calc` that computes the density of air.

The output of this script prints directly to the terminal. This ouput is:

```
Density of air= 1.1610225176629
```

We won't describe the details of this script just now. For the impatient, hang in just a little longer. Right after we describe the installation process in Section 2.1, we'll walk you through your first `gas-calc` program in Section 2.2. We promise to give you a blow-by-blow description of the lines in a `gas-calc` script at that point.

1.1   What's in the box

Given its roots as a support package for `Eilmer`, the gas package features a range of gas models that are used in the simulation of compressible flows. These gas models include:

- ideal gas, with adjustable parameters
- mixture of thermally perfect gases, with species selected from a database
- CO_2 as an ideal fluid, for use in the analysis of supercritical thermodynamic cycles

This list of models is growing as the needs of `Eilmer` users evolve.

For each of these models, the gas package provides services related to the calculation (or look-up) of:

- thermodynamics
- diffusion coefficients
- gas species properties

1.2   How to use this document

This document has a focus on how to *use* the package and is rather light on the theory behind the various gas models. The early sections of this document focus on how to get started building small Lua scripts for use with `gas-calc`. The next chapter, *Getting Started*, walks through the installation process and gives a tutorial introduction to writing a `gas-calc` script. Separate to your script, all of the configurable details of a gas model are read from another Lua file, the so-called gas model file. Chapter 3 describes the use of a utility program for setting up this file with a minimum of fuss. Chapter 4 provides an overview of the available functions provided in the Lua Application Programming Interface (API). Don't be put off by the phrase Application Programming Interface: the tone of this overview is rather conversational and is aimed at giving an idea of *what* can be done rather than the details of *how*. Following that, Chapter 5 is a collection of examples of `gas-calc` in action and shows *how* to do some particular calculations. That rounds out the high-level guide to the `gas` package. Finally, Chapter 6 sets aside the tutorial conversation and documents all of the available methods in the Lua API.

Getting Started

This section has been written as a tutorial. The intent is to walk you through the installation and writing of your first program for use with `gas-calc`.

2.1 Installation

The Dlang gas package has been developed on a linux box, and the following instructions assume that you too are working on a linux system. That being said, our choice of development tools and languages are largely platform independent. If you choose to install on other operating systems your mileage may vary.[1]

2.1.1 Prerequisite software

The following is a list of packages and tools needed to build and install the gas packge:

- mercurial: to get a copy of the source repository from bitbucket.

- a D compiler: the package has been tested with the Digital Mars D compiler

- a C compiler and a Fortran compiler: the system GNU compilers on most linux systems will be adequate

- make: to automate the build and install process

- readline development package

- ncurses development package

2.1.2 Getting the source code

The `gas` package is included as part of a larger collection of gas dynamic simulation tools included the `dgd` project on **bitbucket**[2]. Even if you are only interested in the `gas` package, you will need to grab the entire repository because the `gas` package requires some of the supporting packages included in the rest of the repository. Using mercurial, download a copy of the repository:

---

[1]Please let us know if you do have success running this package on operating systems other than linux.

[2]`https://bitbucket.org/cfcfd/dgd`

```
$ hg clone https://bitbucket.org/cfcfd/dgd dgd
```

2.1.3   First-time setup

We need to create a directory to house the installation of the executable `gas-calc` and the supporting libraries. In the example here, we choose an installation directory called `dgdinst` and place it under the `$HOME` directory:

```
$ mkdir $HOME/dgdinst
```

We then need to configure several environment variables so that `gas-calc` knows where the installation directory is. Since our shell of choice is `bash`, we show here some lines to place in your `$HOME/.bashrc` file:

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

Remember to refresh your current shell (or log out and log in again) so that your newly configured environment is ready to go.

2.1.4   Build and install

The `gas-calc` package can be built and installed using `make`. Navigate to the gas source code area and issue a 'make install' command to built and install the `gas-calc` program and the supporting library modules.

```
$ cd $HOME/dgd/src/gas
$ make install
```

The `makefile` assumes that you want to place the installation files in the default area `$HOME/dgdinst`. If you had set your environment variables to point to a non-standard install location, then you also need to give `make` this hint by setting the `INSTALL_DIR` variable at install time. In that case, you replace the simple 'make install' command above with the one shown here that makes no assumptions about your chosen install area:

```
$ make INSTALL_DIR=/path/to/my/install_dir install
```

2.2 Tutorial: Isentropic expansion

In this tutorial, we will look at using `gas-calc` to compute the isentropic expansion of ideal air. We are given the stagnation conditions of the air and asked to determine the conditions when the flow is expanded to a speed of Mach 1 (sonic conditions). The stagnation conditions are[3]:

$$p_0 = 500.0\,\text{kPa}; \quad T_0 = 300.0\,\text{K}. \tag{2.1}$$

Let's first describe our analysis process in words, and then show the `gas-calc` program that achieves the same effect. We will use an equation for the conservation of energy stated in terms of enthalpy as our starting point. During the expansion process, there is no heat lost or gained (adiabatic flow). So, taking state 0 as the stagnation condition, and state 1 as the expanded condition, we write

$$h_0 + \frac{v_0^2}{2} = h_1 + \frac{v_1^2}{2}. \tag{2.2}$$

Also, at stagnation, the flow speed is zero, so $v_0 = 0$ giving

$$h_0 = h_1 + \frac{v_1^2}{2}. \tag{2.3}$$

We can compute h_0 from the given stagnation conditions. We'll also compute the entropy, s_0, associated with the stagnation conditions since we'll use this as a constraint on the expansion process. We expect the pressure to be lower when gas has accelerated to Mach 1. Using that idea, we can then start an iterative process of stepping down the pressure value and testing what speed the gas is for a given pressure. For each decrement in pressure, we'll apply the thermodynamic constraint that the entropy is constant. We'll stop the stepping process when the computed flow speed is at Mach 1.

2.2.1 Program description

Let's now look at a `gas-calc` program that performs the analysis procedure just described. The program is shown in Listing 2. In the Introduction (Chapter 1), we promised a blow-by-blow (line-by-line) description of a `gas-calc` program. Well, here it is.

The first five lines of the program are some general set up. These lines, or versions of it, will be common to nearly all `gas-calc` programs. On **line 1**, we select a gas model. In this case, it is a model for ideal air and the details are given in the gas model file called `ideal-air-gas-model.lua`. We will discuss how to create such a gas model file in the next section. For now, just know that all gas models are initialised based on an input file. The next line, **line 2**, creates a new variable named `Q` to hold a `GasState` object. A `GasState` object is simple a data structure that holds a number of property values of the gas. Using the `GasState:new{}` constructor takes care of allocating the storage inside the object. The `GasState` object doesn't hold every property of the gas, but it does hold those values that are most convenient to

---

[3]If these values are familiar, it might be because you have worked through some of the examples for the `Eilmer` simulation program. These values are the stagnation conditions of the Back nozzle.

```lua
1   gmodel = GasModel:new{'ideal-air-gas-model.lua'}
2   Q = GasState:new{gmodel}
3   Q.p = 500e3 -- Pa
4   Q.T = 300.0 -- K
5   gmodel:updateThermoFromPT(Q)
6   -- Compute enthalpy and entropy at stagnation conditions
7   h0 = gmodel:enthalpy(Q)
8   s0 = gmodel:entropy(Q)
9   -- Set up for stepping process
10  dp = 1.0 -- Pa, use 1 Pa as pressure step size
11  Q.p = Q.p - dp
12  M_tgt = 1.0
13  -- Begin stepping until M = M_tgt
14  while true do
15     gmodel:updateThermoFromPS(Q, s0)
16     h1 = gmodel:enthalpy(Q)
17     v1 = math.sqrt(2*(h0 - h1))
18     gmodel:updateSoundSpeed(Q)
19     M1 = v1/Q.a
20     if M1 >= M_tgt then
21        print("Stopping at M= ", M1)
22        break
23     end
24     Q.p = Q.p - dp
25  end
26
27  print("Gas state at sonic conditions are:")
28  print("p= ", Q.p)
29  print("T= ", Q.T)
```

Listing 2: A `gas-calc` program to compute the isentropic expansion of ideal air.

store in a CFD simulation. Two of those values stored in `GasState` are pressure and temperature. On **line 3**, we access the pressure value in the `GasState` (using the notation `Q.p`) and set it equal to 500.0 Pa. Similarly, on **line 4**, we set the temperature to 300.0 K. Finally, on **line 5** we compute the rest of the thermodynamic state by making a call the `GasModel` method `updateThermoFromPT()`. When we say the rest of the thermodynamic state, in this case the density and the internal energy are computed based on the values of pressure and temperature. You might be thinking, "But hey, I don't see any reference to density and internal energy on line 5. Where did those values go?" The values of density and internal energy actually get updated in the `Q` data structure. In other words, the values of `Q.rho` and `Q.u` are different before and after line 5. After line 5, the values are the correct state corresponding to the pressure and temperature we set earlier in the `Q` data structure. This is a common theme with a lot of the `GasModel` methods: they assume the caller sets the correct values for certain properties in the `GasState`, and from that the rest of the thermodynamic state is set. This idea is discussed in detail in Chapter 4.

In the Lua programming language, a comment begins with two dashes (`--`) and extends until the end of the line. **Line 6** is an example of a comment line. Lines 4 and 5 end with comments that remind us of the assumed units for pressure and temperature.

Lines 7 and 8 are used to compute the enthalpy and entropy at the stagnation

conditions. We know that enthalpy and entropy will be computed at the stagnation conditions because those are the state values in the `Q` variable which is passed to `enthalpy()` and `entropy()` methods. On line 5, we *updated* density and internal energy directly in the `Q` variable. Now on line 7 and 8, we *return* the values for enthalpy and entropy, and assign them to variables. Why the difference in method calls and return values? We said earlier that not all gas properties are stored in the `GasState` object. Enthalpy and entropy are two examples of gas properties that are not stored as part of the `GasState`. As such we can't *update* them in the variable `Q`. Instead, we *return* them to the caller to hang on to and do with as they please. [4]

Lines 9–12 don't show us any new aspects of the Lua `gas` module. These few lines of Lua code are use to set up the size of the decrement in pressure (`dp`), set initial pressure value for our iterative stepping process, and set a target Mach number value. We set the target Mach number as a variable so that we could easily reuse this program if we were interested in expansion to a different Mach number value.

Lines 14–25 are the code to perform the iterative stepping procedure. The `while` loop begins on line 14 and is set to loop always since the loop condition is `true`. To break out of this potentially infinite loop, we rely on the `break` statement on line 22 to be triggered at some point. At the start of each loop entry we have a new pressure value to test. On **line 15**, we update the thermodynamic state in `Q` based on pressure (which is found internally in `Q` as `Q.p`) and entropy (`s0`). Entropy is supplied as a direct argument to `updateThermoFromPS()` since entropy is not stored in `Q`. We call this function with `s0` always because we are interested in an isentropic expansion: the entropy remains constant as the pressure is changed. After the call to that function, temperature, density and internal energy are updated in `Q`. We are most interested in having temperature up-to-date because that will be used on **line 16** to compute the new enthalpy. So what we have at this point is:

1. pressure is adjusted
2. new temperature, density and internal energy are computed
3. new enthalpy is computed based on the new temperature

On **line 17**, we use a rearrangement of Equation 2.3 to find the speed of the gas, `v1`. Note that the `sqrt()` function is part of the Lua math module and so is accessed as `math.sqrt`. **Line 18** is used to update the sound speed value. Again, this function doesn't seem to return any value. The sound speed method is in the family of *update* methods. That gives us a clue that some value has been altered in `Q`. In this case, the value of `a`, the sound speed, is updated and available as `Q.a`. In fact, we use `Q.a` in the very next line, **line 19**, to compute the Mach number. We have deliberately avoided stating which values are stored in `Q`. We aren't trying to keep this a mystery: it's just we'd rather focus on the broader aspects of the program at this point. The full list of the values stored in the `GasState` is given in Section 4.2.

On **line 20**, we make a decision as to whether to continue decrementing the pressure or continuing the looping process. Since we are approaching the Mach number

---

[4]Now, if you're thinking why are some values stored in the `GasState` and others not, the answer relates back to the `gas` package supporting the `Eilmer` flow solver. In large CFD simulations, we need to take care with how much memory we use during simulation. If we included *all* of the thermodynamic state, we would have a larger memory overhead and carry around values that we didn't use very often, or at all. As such, we only store those values that we make frequent use of in a CFD program.

target from below, we test if we have stepped up to or exceeded the value of $M = 1$. If we have exceeded the target, we print out the current Mach number on **line 21** (which will hopefully be only small amount past the target). The `break` statement on **line 22** terminates the `while` loop and the execution of code jumps to **line 27**. The `end` statement on **line 23** closes out the compound **if**-statement that began on line 20. If we have not yet reached the target Mach number, then **line 24** is executed to decrement the pressure value by 1 Pa which prepares us for the next iteration of the loop.

  Lines 27–29 are print statements to write the results of our calculation to the terminal output. In particular, the pressure and temperature conditions when flow speed is at Mach 1.005 are printed and the program exits (but there is no explicit statement needed to end the program).

2.2.2 Running the program

Before we can run our `isentropic-air-expansion.lua` program, we need to prepare the gas-model file, `ideal-air-gas-model.lua`. This file is a plain-text file of data in Lua format. While it is possible to construct a gas model file by hand, it gets a little tedious and error-prone to do so. For that reason, the `prep-gas` program is supplied with the `gas` package to help prepare the gas model files. This program, in turn, requires a very small input file to select some options associated with the desired gas model. The input file for the `prep-gas` program used to generate the `ideal-air-gas-model.lua` file is shown in Listing 3.

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

Listing 3: Input file for the `prep-gas` program to set up the model of ideal air.

  The `prep-gas` program is run from the command line with two arguments. The first argument is the name of the input file and second is the name of the output file. The output file is the complete gas model file used by `gas-calc`: the file we said that was too tedious to deal with by hand. These gas model files are used by more than just `gas-calc`. They are the same gas model files used by `Eilmer` when it configures its gas model. To generate the gas model file for this example, do:

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

We will discuss the preparation of gas models in full in Chapter 3.

  The next step is to run `gas-calc` itself. It only takes one argument on the command line: the name of program file. You will need to have your gas model file in the same working directory as your `gas-calc` program. To run the program, type:

```
$ gas-calc isentropic-air-expansion.lua
```

The program takes a couple of seconds to run. The output to the screen should look like:

```
Stopping at M=  1.0000029005924
Gas state at sonic conditions are:
p=  264140
T=  249.99975828385
```

These two steps to execute the program are captured in a run script file called `run.sh`. You can find this complete example along with the run script in `dgd/examples/gas-calc/isentropic-expansion/`.

2.2.3 Improvements to the program

This tutorial example was designed to introduce you to the elements of a `gas-calc` program. As such, some effort was made to keep the example simple. There are several improvements we could make to this program, and we'll discuss those here. We will present improvements in terms of both accuracy and efficiency.

You might notice we stopped at a Mach number which was a little larger than 1.0, and took that value as an estimate of our sonic conditions. Without too much extra work, we could have interpolated the sonic conditions based on the final two steps, and, so, improved the accuracy.

A more elegant iteration procedure could have been used to improve efficiency and achieve a desired accuracy. For example, a secant iteration method would be much more efficient and would stop based on a user-supplied tolerance on the error.

Finally, the ultimate in improving accuracy and efficiency — and for those who were tearing their hair out waiting for us to mention the elephant in the room — just use the analytical expression. Good students of gas dynamics will remember (or be able to derive) the one-dimensional isentropic flow relations. We could have used those relations directly to compute the sonic conditions:

$$\frac{T^*}{T_0} = \frac{2}{\gamma + 1} \tag{2.4}$$

$$\frac{p^*}{p_0} = \left(\frac{2}{\gamma + 1}\right)^{\gamma/(\gamma-1)} \tag{2.5}$$

So, let's do that. Using our given stagnation conditions (Eq. 2.1) and $\gamma = 1.4$ gives

$$p^* = 26.414\,\text{kPa}; \quad T^* = 250.0\,\text{K}.$$

This is a nice confirmation of the results in our program.

Of course, we built this program as a demonstration rather than as an example of an expedient means to compute the sonic conditions of an isentropically expanding gas. The benefit of this program is that the calculation approach can be applied more generally to gases with complex behaviour. The analytical expressions are only valid for a calorically perfect gas, that is, a gas where the specific heats are constant.

2.2.4   Recap of what we've demonstrated

Before ending this section, let's briefly recap what was demonstrated in the `gas-calc` program used to compute the isentropic expansion of air. We were introduced to the initialisation methods that are part of every `gas-calc` program. These are the routines to initialise a `GasModel` and a `GasState`. We also mentioned that you require a separate gas model file to initialise a `GasModel`. The `prep-gas` program is provided to help create a gas model file. The next Chapter describes the available gas models and how to use `prep-gas` in more detail.

In describing the example program, we touched on the idea that the state of the gas is held in a `GasState` data structure, and that data is operated on by the methods in the `GasModel`. We were also introduced to two distinct classes of methods: a) those that *update* values inside the `GasState`; and b) those that *return* a value to the caller. For example, `updateThermoFromPT()` is a method in the *update* family. It updates the values of density and internal energy in the `GasState` data structure assuming that the pressure and temperature are present and correct. On the other hand, the `enthalpy` method is part of the *return* family of methods. It also relies on the thermodyanmic state inside the `GasState` object being correct when called, but it *returns* the result of its calculation as a value to the caller. We learnt that the reason for this distinction in behaviour is because some thermodynamic values are stored in the `GasState` data structure and, so, can be updated in place. There are other values that aren't stored in the `GasState`. These values need to be returned to the outside world when their calculation is requested.

Gas Model Preparation

This is a short chapter to describe a small but powerful tool: `prep-gas`, which is a program designed to take the pain out of preparing a gas model file. A gas model file contains a lot of information about the properties of the gas. When dealing with gas mixtures (or a multi-component gas), the gas model file needs to contain details for each of the species. In simulations of reacting hydrocarbon flows, it is common to have on the order of 50 species present in the gas mixture. You can see that, at this point, it becomes tedious to gather and prepare the data for that many species. Instead, we provide the `prep-gas` tool to do the chore of building a gas model file from a database of species and their properties.

3.1 Available gas models

The list of gas models supported by the `gas` package is shown in Table 3.1 along with a brief description of the model behaviour. The detail of the thermodynamic relations and transport property calculations are given in the D-language source code for the package. If you need to implement a new class of gas model, you will need to study this source code.

Table 3.1: Available gas models for use with the `gas` package.

Model	Description
IdealGas	a single component ideal gas: this gas has perfect collisional behaviour and constant specific heats
ThermallyPerfectGas	a multi-component gas: each component is treated as thermally perfect meaning that perfect collisional behaviour is assumed. Also, the internal energy of the components varies with temperature only.
CO2Gas	The Bender gas model [1] for dense gases with model parameters set for carbon dioxide.
CO2GasSW	The Span-Wagner gas model [2] for dense gases with model parameters set for carbon dioxide.

3.2   Available species

A list of available species can be generated by `prep-gas` by running it with the `--list-available-species` option.

```
$ prep-gas --list-available-species
```

The species names are case sensitive and use the conventional naming for chemical compounds. The exception to this naming rule is the pseudo-species `air`. This lower-case symbolic name is used to select the model for air as used in the NASA Chemical Equilibrium Analysis (CEA) program [3].

3.3   How to use `prep-gas`

In normal operation, `prep-gas` takes two arguments on the command line: 1) an input file; and 2) an output file. The input file is a small text file (in Lua) format that specifies the gas model and the species in the mixture. The output file is created, also as a Lua file, and given the name of the second argument. This output file is the gas model file, proper, containing all of the configuration values needed to specify the gas model. It is used when the `GasModel` constructor is called. An example input file to select ideal air is shown in Listing 4. The input file declares a `model` type and a list of `species`. The `model` may be any one of the supported models listed in Table 3.1. The `species` list can include any of the species that are listed when `prep-gas` is run with the `--list-available-species` option.

```
model = "IdealGas"
species = {'air'}
```
<div align="center">

Listing 4: `ideal-air.inp`: an input file for `prep-gas`.

</div>

An example of running `prep-gas` at the command line is:

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

Upon successful completion, the file `ideal-air-gas-model.lua` will be created and sitting in the working directory.

3.4   How `prep-gas` works

How `prep-gas` works will be of little interest to most, so we'll try to keep this brief. In the installation area in the `data` area, there is a large text file called `species-database.lua`. This database contains *all* of the parameters needed in the various gas models on a species-by-species basis. As we mentioned, this is *all* the possible data that could be called upon. For any given gas model, we only need a subset of that data. That's where `prep-gas` comes in. Its job is to load all of the species data, select only those species of interest and select only those parameters needed by the chosen gas model. Then, `prep-gas` writes that subset data to the gas model file.

Using this approach achieves two things: 1) it cuts down on the size of the gas model file; and 2) it creates a human-readable local record of the gas models parameters. That second point is very useful from a "reproducible research" point of view. With the locally-created gas model file, you have a complete record of the parameters that were used when performing your calculation. This is very handy when you try to convey your results to others such that they could reproduce your work.

A Tour of the Lua API

In this section, we will point out the functionality for gas modelling that is available in the `gas-calc` program. The `gas-calc` program operates as a standard Lua interpreter with some extra goodies added. Those goodies are special functions and objects related to the gas model. This discussion will focus on those goodies. To learn more about Lua programming in general, we recommend the excellent book "Programming in Lua" by Ierusalimschy [4]. This discussion about the specific extras provided by the `gas` package is presented as a tour of what's available. The "legalese" of function arguments, return types, the caller's pre-conditions and all of that other fun stuff is deferred to Chapter 6.

4.1 The relationship between `GasModel` and `GasState`

There are two objects that are central to every `gas-calc` program: the `GasModel` and the `GasState`. Indeed, the first two lines of the program in the tutorial example are dedicated to initialising a `GasModel` and a `GasState`. What then is the relationship between these objects?

Simply put, a `GasModel` object operates with/on the data in a `GasState` object. The `GasModel` object, as the name implies, is responsible for modelling the behaviour of the gas. It provides services that the user can call on. The `GasState` object is a data structure for storing the current state of the gas. Usually, we would have only one or a small number of `GasModel` objects in our program. (We might have more than one if we wanted to do some direct comparisons of behaviour of different gases over a temperature range, for example.) On the other hand, we could end up with many `GasState` objects if we wanted to keep record of the gas state in different regions of a simulation.

We always initialise a `GasModel` by passing the name of a gas model file:

```
gmodel = GasModel:new{'my-gas-model-file.lua'}
```

There are a few ways to initialise a `GasState` object, but the easiest is with a call to the constructor that takes a `GasModel` object as the argument:

```
Q = GasState:new{gmodel}
```

The reason we pass a `GasModel` object is so that the `GasState` constructor can get information about the number of species and number of nonequilibrium energy modes

in the gas model. These numbers are used to pre-size certain arrays in the `GasState` object. In the Lua language domain, the `GasState` object is nothing more than a Lua table. It is possible to construct a `GasState` table directly without calling the `new{}` constructor, though it's not recommended. The important to thing to know is that methods provided in the `gas` package will expect certain fields to be present in a `GasState` table. If you remove certain fields (either accidentally or deliberately), the behaviour of the `gas` package methods is undefined. [1]

4.2 Working with the `GasState` data structure

We mentioned that the `GasState` is a Lua table. The fields in a `GasState` table are described in Table 4.1.

Table 4.1: Data stored in a `GasState` table.

Field	Type	Description
rho	number	density, kg/m^3
p	number	pressure, Pa
p_e	number	electron pressure, Pa (Only non-zero when the gas model uses a separate electron temperature, otherwise the partial pressure of electrons is included with all other components in the regular pressure variable `p`.)
a	number	sound speed, m/s
u	number	specific internal energy (in transrotational mode), J/kg
e_modes	array of numbers	specific internal energies in nonequilibrium modes (typically, vibration), J/kg
T	number	(transrotational) temperature, K
T_modes	array of numbers	temperatures of nonequilibrium modes (corresponding to `e_modes`), K
mu	number	dynamic viscosity, Pa.s
k	number	thermal conductivity for transrotational mode, W/(m.K)
k_modes	array of numbers	thermal conductivities in nonequilibrium modes, W/(m.K)
sigma	number	electrical conductivity, S/m
massf	table of key-value pairs key : string – 'species-name' value : number – mass fraction	mass fractions of components in mix species *not* listed in the table are assigned 0.0
quality	number	vapour quality

---

[1]In fact, the `gas-calc` program will probably fail noisily with a cryptic message about a 'nil' value found where a legitimate value was expected.

Looking at the fields in the `GasState`, you might see some notable absences. For example, where is enthalpy, or the specific heats, C_p and C_v? We have chosen to store only a few of the many variables that could be used to describe the `GasState`. Our choice has been motivated by what is most frequently used in a CFD calculation.

There are scalar and array fields for temperature, energies and thermal conductivities. The scalar fields are all that are required for a single-temperature gas model, and temperature, internal energy and thermal conductivity take on their usual sense for equilibrium thermodynamics. For gases that have a nonequilibrium distribution of energy in internal modes, there are various multi-temperature gas models available. In these models, the energies, temperatures and thermal conductivites for the nonequilibrium modes are stored in arrays. They are arrays because the `gas` package is designed to handle gas models with an arbitrary partitioning of internal energy modes. Note that the energy, temperature and thermal conductivity of the transrotational mode is always stored in the scalar entries. The array entries are used for the *additional* nonequilibrium modes. What does that mean exactly? Take for example the two-temperature model popular in the simulation of high-speed Earth re-entry flows. In this model, the translational and rotational energies in the gas mixture are governed by one temperature. This is the first mode, stored in the scalar entries in the table. A second mode for the internal energy lumps together the vibrational and electronic energy of the internal states. These are described by a second temperature and the values are associated with this second temperature are stored as the first entry in the arrays. As such, we end up with single temperature value in the `T_modes` array, a single energy value in the `e_modes` array, and a single thermal conductivity value in the `k_modes` array. As an example, the following code would be used to print the transrotational and vibroelectronic temperatures in the previously described two-temperature model:

```
print("transrotational temperature= ", Q.T)
print("vibroelectronic temperature= ", Q.T_modes[1])
```

The mass fraction table is accessed by species names, not by indices as we do for the temperature array. The reason for this is that we will typically have very few modes and they are easy to keep track of with numbered indices. However, there are potentially many species in the mixture and it becomes far easier to deal with the few we are interested in by name. The mass fraction values are set using a key-value pair of the form `speciesName=massFraction`. For example, we could set the mass fractions of oxygen and nitrogen to approximate the composition of air like this:

```
Q.massf = {O2=0.22, N2=0.78}
```

An alternate means of setting those values would be:

```
Q.massf['O2'] = 0.22; Q.massf['N2'] = 0.78
```

or, using Lua's syntactic sugar of converting string keys to member access, we could even do this to achieve the same effect:

```
Q.massf.O2 = 0.22; Q.massf.N2 = 0.78
```

In all of these examples, the case of the species name is important. Trying to set the oxygen mass fraction with a dodgy name like `Q.massf['o2'] = 0.22` will cause an error. It won't be an error at the time of assignment but it will be triggered later on when you try to hand that to one of the gas model methods. You can retrieve a mass fraction value by name also using the regular Lua table access mechanisms:

```
mfO2 = Q.massf['O2']
mfN2 = Q.massf.N2
```

Since we've provided convenient access to the mass fractions by name, we need to tell you a few of the rules to the game. If you are only dealing with a single species gas, you can leave the mass fractions table untouched; the mass fractions table will default to a single entry with a value of 1.0 for a single species gas. If you are interested in working with a multiple species gas, then read on. The mass fractions table is a simple Lua table, and as such you are free to put whatever you like in there.... but don't do that! The table will only be checked when you actually use it in a `GasModel` method. When the mass fraction table is checked, it will first look for any unknown or invalid keys. These will cause an error and the program will terminate. Next, if all the keys are valid species names, a decision needs to be made as to what value to assign for mass fraction to all those species *not* listed in the table.[2] We have decided that all the missing species will be assigned a value of 0.0. This is typically what most users want. We then need to check that the supplied values sum to a total of 1.0. If the sum is slightly out (say less than 1 part in a million or whatever the internal tolerance is set to), then the mass fractions will be silently scaled so that the sum is 1.0. If the sum is out more than an acceptable amount, an error message is printed and the program terminates.

4.3 Update methods

Having introduced the `GasState` object, we are now in a position to discuss what we can do with it. There are several *update* methods provided by the `GasModel` object that will operate with the data in a `GasState` to update the state of some other variables. Each of these update methods is a thermodynamic equation of state. Let's look at the first *update* method provided by the `GasModel`: `updateThermoFromPT()`. If this method were a movie, then we've given away the plot in the title. [3] As the name of this method suggests, we are going to update the thermodynamic quantities in the `GasState` given the values of pressure and temperature. Let's see this method in action (assuming one has previously initialise a `gmodel` object):

```
Q = GasState:new{gmodel}
Q.p = 1.0e5; Q.T = 300.0
```

---

[2] There's a subtlety at play here. If you used the `GasState:new{}` constructor to initialise your `GasState` (which we encourage), then it will have set all species to a mass fraction value of 0.0. In other words, those species are set in the background even if you haven't explicitly set values yourself. It's not a big concern because the default value of 0.0 is usually what you want. It's then the users job to set just those species that have interesting (non-zero) values for mass fraction. In the case of a gas with a single species, a default value of 1.0 is placed in the mass fraction table.

[3] Unfortunately, our `GasModel` methods don't get nominated for Academy Awards, so we can dispense with creative titles and use something much more pedestrian but clear in its intent.

```
    print("rho= ", Q.rho, " u= ", Q.u)
    gmodel:updateThermoFromPT(Q)
    print("rho= ", Q.rho, " u= ", Q.u)
```

The output from running these lines of code is:

```
rho=    nan   u=    nan
rho=    1.1610225176629   u=    215327.43439227
```

In the first print statement, the values for density and internal energy are `nan`. We expect this because the values have been default initialised by the underlying dlang module. In the D programming language, floating point values are default initialised to `nan`. After the call to `updateThermoFromPT()`, the values of density and internal energy are updated. The update method was passed a `GasState` data structure. From that, it pulled out the values of pressure and temperature and then proceeded with the calculations to update the values of density and internal energy.

There are three other *update* methods that follow the same pattern:

- `updateThermoFromRHOE()`
- `updateThermoFromRHOT()`
- `updateThermoFromRHOP()`

Again, the name indicates which variables are assumed to be valid in the `GasState` object. The methods then compute the other primitive thermodynamic variables. We consider density (`rho`), internal energy (`u`), pressure (`p`) and temperature (`T`) as the primitive variables. There is nothing special about this particular set of values in terms of representing the thermodynamic state. We just consider them primitive in this implementation because they are directly stored in the `GasState` object.

The `updateSoundSpeed()` method is special. It will update the sound speed value (`a`) in the `GasState` assuming the other primitive variables in the `GasState` are up-to-date. It is special because it updates the sound speed *only*, whereas all the other *update* methods change the primitive thermodynamic variables. So, usually, one would call one of the previously mentioned *update* methods first before calling `updateSoundSpeed()`.

There are two other *update* methods that are slightly different. One of these, the `updateThermoFromPS()` method, is used to update the thermodynamic state given pressure and entropy as inputs. This method accepts two arguments: a `GasState` object and an entropy value. The reason this method breaks the pattern from earlier is quite simple: entropy is *not* available as a data member in the `GasState` and so we have to pass it in directly. However, pressure is available in the `GasState`, so for consistency with the other methods, we will pull the pressure value from the `GasState` object. We've previously seen this method used on line 15 of Listing 2 in Section 2.2. The last method in the *update* family is `updateThermoFromHS()`. This method accepts three arguments: a `GasState` object, an enthalpy value and an entropy value. This method uses enthalpy and entropy to start doing its cacluations of the thermodynamic state. Why then do we need to pass a `GasState` object? Well, this is still an update method. Even though we don't use any of the values directly in the `GasState` object, we do in fact change the state of the values as part of the update. After calling this method, the values of density, internal energy, pressure and temperature are updated in the `GasState` object.

That is the full set of `update` methods. They are all equations of state: given two thermodynamic variables, the remainder of the state can be computed. That's not quite the full story for multi-component (multiple species) gases. For multi-component gases, the gas composition is also required in order to compute the state. That's another reason the `GasState` object is always passed in as an agument. For multi-component gases, all of the *update* methods will use the `massf` values as well in order to perform the requested thermodynamic equation of state calculation. So, these values should also be set correctly before calling an *update* method for multi-component gases.

4.4   Return methods

There are a number of methods provided by the `GasModel` that form a family of *return* methods. All of these methods take a single argument, a `GasState` object. When calling any of the *return* type methods, it is assumed that the primitive variables are up-to-date. Usually, one would call one of the *update* methods before calling a *return* method. These methods all return a single floating point value when called. Again, the names of the methods leave no suprise as to what you get. For example, the method `dhdTConstP()` will compute the thermodynamic derivative $\left(\frac{\partial h}{\partial T}\right)_p$. Other *return* methods include `gasConstant()` to compute the mass-averaged gas constant and `molMass()` to compute the mixture molecular mass.

Some of the *return* methods have aliases. For example, the `Cv()` method gives the same result as calling `dedTConstV()`, and `Cp()` is an alias for `dhdTConstP`. A call to the method `R()` will delegate its work to `gasConstant()`.

As an example of using a *return* method, let's see how we could compute the effective ratio specific heats, γ, for a mixture of nitrogen and oxygen at 4000 K by calling the `gamma()` method:

```
gmodel = GasModel:new{'therm-perf-N2-O2.lua'}
Q = GasState:new{gmodel}
Q.p = 1.0e5; Q.T = 4000.0
Q.massf = {N2=0.78, O2=0.22}
gmodel:updateThermoFromPT(Q)
gamma = gmodel:gamma(Q)
print("gamma= ", gamma)
```

4.5   Converting between composition representations

To complete the tour of the Lua API, the final methods to be introduced are the conversion methods. These methods are used to convert between various representations of the composition. Let's see how we could use the `massf2conc()` method to get the concentration of N_2 and O_2 based on the mass fractions we used previously when computing the ratio of specific heats:

```
conc = gmodel:massf2conc(Q)
print("c[N2]= ", conc.N2, " c[O2]= ", conc["O2"])
```

There are four conversion functions available: `massf2molef()`, `molef2massf()`, `massf2conc()`, `conc2massf()`. All of these conversion methods return a table as a result. (In the example above, we saw that a table of concentrations was returned.) These tables work like the mass fraction tables: the values for various species are accessed by the species name. When using the conversion methods that convert *to* mass fractions, you are in fact getting an *update* method and a *return* method in one. During the call, the `massf` field in the passed in `GasState` object gets updated. That same `massf` field also gets returned to the caller. By returning the `massf` table, it keeps the behaviour of the method calls consistent with other conversion methods. If you don't want to deal with the returned table, just don't catch it (that is, bind a name to it with the assignment operator). It is perfectly valid to ignore a returned value in the Lua programming language. To demonstrate this, we'll ignore the returned `massf` table when computing the mass fraction of hydrogen in a stoichiometric mixture of nitrogen, oxygen and hydrogen:

```
gmodel = GasModel:new{'therm-perf-N2-O2-H2.lua'}
Q = GasState:new{gmodel}
mole_total = 1.0 + 2.0 + 3.76
molef = {H2=1.0/mole_total,
         O2=2.0/mole_total,
         N2=3.76/mole_total}
gmodel:molef2massf(molef, Q)
print("f[H2]= ", Q.massf.H2)
```

Examples

This section presents some short examples that demonstrate some of the ways in which the `gas-calc` program can be used. The idea behind building `gas-calc` is so that users of the CFD code, `Eilmer`, can access the same gas model functions used within the simulation code in a convenient scripting interface. This means that one can do calculations of gas properties that are consistent with the calculations performed internally in `Eilmer` during a simulation. This could be useful for doing some auxiliary calculations as part of pre- or post-simulation activities.

The first two examples show how to compute some specific gas properties. These are computed over a temperature range so that plots can be produced of the property variations with temperature. The final example shows how to construct a small analysis program. This particular program computes the states at various points in a thermodynamic cycle and reports on the cycle efficiency.

5.1   The Cp and enthalpy curves for molecular oxygen

In this example, we will compute the specific heat at constant pressure, C_p, and the enthalpy, h, of molecular oxygen over the temperature range of 200 – 20,000 K. The `gas-calc` program to perform this calculation is shown in Listing 5.

In this example, we selected the thermally perfect gas model for O_2 because we expect some variation in C_p over such a large temperature range. To make this selection, the input file to `prep-gas` looks like:

```
model = 'thermally perfect gas'
species = {'O2'}
```

In lines 11 and 12, the gas model is initialised. Setting up the gas model is the typical starting point for *every* `gas-calc` program since we cannot do much in the way of calculation without defining the gas behaviour. Lines 14–16 are used to set up a `GasState` table and fill it in with some values that will remain constant for the entire program. Note that the particular value of pressure is not important because the properties of C_p and h are not functions of temperature in a thermally perfect gas. However, it is good practice to set a pressure value since some of the thermodynamic routines will rely on having a meaningful value for pressure present (even though it is not used in these particular calculations of C_p and h).

```lua
1   -- A script to output Cp and h for O2 over temperature range 200--20000 K.
2   --
3   -- Author: Rowan J. Gollan
4   -- Date: 2016-12-23
5   --
6   -- To run this script:
7   -- $ prep-gas O2.inp O2-gas-model.lua
8   -- $ gas-calc thermo-curves-for-O2.lua
9   --
10
11  gasModelFile = 'O2-gas-model.lua'
12  gmodel = GasModel:new{gasModelFile}
13
14  Q = GasState:new{gmodel}
15  Q.p = 1.0e5 -- Pa
16  Q.massf = {O2=1.0}
17
18  outputFile = 'O2-thermo.dat'
19  print("Opening file for writing: ", outputFile)
20  f = assert(io.open(outputFile, "w"))
21  f:write("#  1:T[K]       2:Cp[J/kg/K]      3:h[J/kg]\n")
22
23  Tlow = 200.0
24  Thigh = 20000.0
25  dT = 100.0
26
27  for T=Tlow,Thigh,dT do
28      Q.T = T
29      gmodel:updateThermoFromPT(Q)
30      Cp = gmodel:Cp(Q)
31      h = gmodel:enthalpy(Q)
32      f:write(string.format(" %12.6e %12.6e %12.6e\n", T, Cp, h))
33  end
34
35  f:close()
36  print("File closed. Done.")
```

Listing 5: A `gas-calc` program to compute the C_p and h values for O_2.

Lines 18–21 are used to prepare an output file so that the calculations can be recorded in text form. We are going to write the data out in rows for each temperature value. The column headers are written at the top of the file (in line 21) for our convenience when inspecting the file at a later stage.

In lines 23-25, we set up the temperature range and the incremental step in temperature. We have selected to increment in steps of 100 K.

The main looping occurs in lines 27–33. On each step of the loop, we use a new temperature value. We place this new temperature value in the `GasState` table `Q` so that it is available and up-to-date for the thermodynamic calculation methods. We then update the thermodynamic state by calling the update based on p and T. Strictly speaking, this call is not necessary for this particular gas model but, for general cases, it is good practice to have a completely up-to-date thermodynamic state before calling the C_p and h evaluations. The final method calls are to the specific services to return C_p and enthalpy. These values are written to the file on line 32.

The program terminates by closing the file and printing a message to the screen to let us know that all is well.

The resulting data for C_p and h is shown in Figure 5.1. The variation of C_p with temperature for molecular oxygen is due to the excitation of the internal energy modes of vibration and the electronic structure. The value for enthalpy at T = 200 K is very slightly below zero. This is because the reference enthalpy value for molecular oxygen is set at 0.0 at T = 298.15 K. This reference enthalpy is quite common; it is used in the JANNAF tables and by the CEA program.
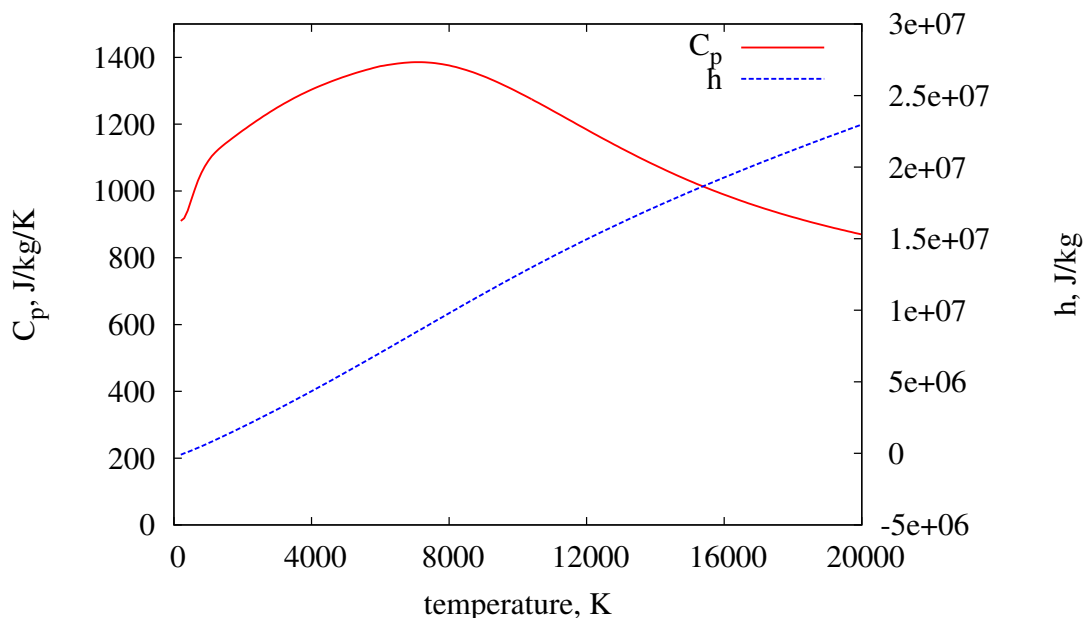

Figure 5.1: The variation of specific heat at constant pressure, C_p, and enthalpy, h, with temperature for molecular oxygen, O_2.

5.2 The viscosity and thermal conductivity of an N2-O2 mixture

In this example, we will build a `gas-calc` program to compute the viscosity and thermal conductivity of a gas mixture. The gas mixture is comprised of 78% nitrogen and 22% oxygen by mass. This mixture approximates the composition of air very well at low temperatures. At higher temperatures, we would expect some dissociation and, if hot enough, ionisation. At those higher temperatures, the composition of the gas is not as simple as this two-component mixture. In any case, as a demonstration, we will compute the transport coefficients for this mixture over the temperature range from 200 – 20,000 K.

The `gas-calc` program for this example is shown in Listing 6. It is very similar to the previous example that was used to compute C_p and h. To avoid repetition, only the differences in this script as compared to the previous example will be discussed.

In this example, the gas mixture has two components. This introduces two differences. Firstly, the gas model input file now looks like:

```
model = 'thermally perfect gas'
```

```lua
1  -- A script to compute the viscosity and thermal conductivity
2  -- of air (as a mixture of N2 and O2) from 200 -- 20000 K.
3  --
4  -- Author: Rowan J. Gollan
5  -- Date: 2016-12-23
6  --
7  -- To run this calculation:
8  -- $ prep-gas thermally-perfect-N2-O2.inp thermally-perfect-N2-O2.lua
9  -- $ gas-calc transport-properties-for-air.lua
10 --
11
12 gasModelFile = 'thermally-perfect-N2-O2.lua'
13 gmodel = GasModel:new{gasModelFile}
14
15 Q = GasState:new{gmodel}
16 Q.p = 1.0e5 -- Pa
17 Q.massf = {N2=0.78, O2=0.22} -- a good approximation for the composition of air
18
19 outputFile = 'trans-props-air.dat'
20 print("Opening file for writing: ", outputFile)
21 f = assert(io.open(outputFile, "w"))
22 f:write("#  1:T[K]       2:mu[Pa.s]       3:k[W/(m.K)]\n")
23
24 Tlow = 200.0
25 Thigh = 20000.0
26 dT = 100.0
27
28 for T=Tlow,Thigh,dT do
29     Q.T = T
30     gmodel:updateThermoFromPT(Q)
31     gmodel:updateTransCoeffs(Q)
32     f:write(string.format(" %12.6e %12.6e %12.6e\n", T, Q.mu, Q.k))
33 end
34
35 f:close()
36 print("File closed. Done.")
```

Listing 6: A `gas-calc` program to compute the μ and k values for a mixture of N_2-O_2.

```
species = {'N2', 'O2'}
```

Secondly, we need to take care to set the mass fractions for each of the components. This is done on line 17.

The only other differences occur in the looping construct. Again, we are fastidious about updating the thermodynamic state after each increment in temperature by calling the p and T update method. As mentioned earlier, this is a best practice to follow as it will ensure correctness for the most general of gas models. To update the transport coefficients, we call the corresponding method `updateTransCoeffs()`. Since μ and k are updated in-place in the `Q GasState` table we do not need to store them in temporary variables. Instead, we can directly access those up-to-date values when we write them to the file on line 32.

The viscosity and thermal conductivity curves for this mixture of nitrogen and oxygen are shown in Figure 5.2. Note the values above 15,000 K; they are constant.

This reveals something about the implementation in the gas module. The transport coefficient curves for nitrogen and oxygen are valid up to 15,000 K. Beyond that, we cannot continue to evaluate the polynomial with any confidence in its reliability. The choice we have made in the implementation is to treat the values as constant beyond that point. In terms of flow modelling, this usually introduces very little error because molecular nitrogen and molecular oxygen are only present in very small amounts at 15,000 K. Instead, we would expect atoms and their ions to be more prevalent.
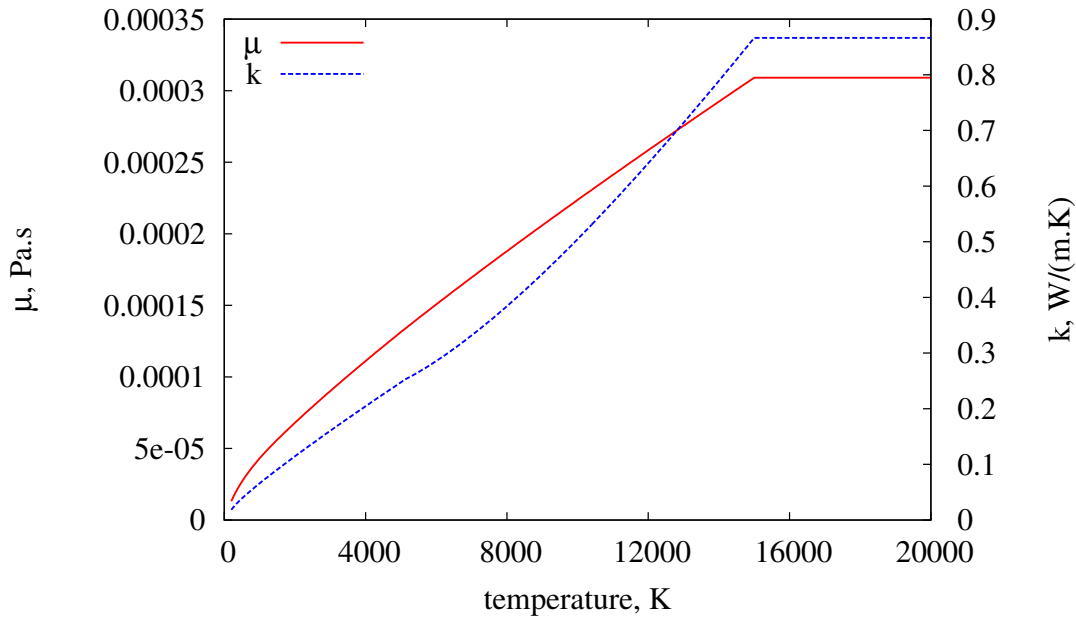

Figure 5.2: The variation of viscosity, μ, and thermal conductivity, k, with temperature for a mixture of N_2-O_2.

5.3 An ideal Brayton cycle

This example is taken from Çengel and Boles' thermodynamics text [5], example 9-5, and computes the performance of an ideal closed Brayton cycle, as shown in Figure 5.3. The flow of gas between the cycle components is shown in blue and the states of the gas at key points in the cycle are numbered 1 through 4. Heat flows into and out of the air working fluid are shown as red arrows at the heat exchangers.

The original problem description reads: "A gas-turbine power plant operating on an ideal Brayton cycle has a pressure ratio of 8. The gas temperature is 300 K at the compressor inlet and 1300 K at the turbine inlet. Utilizing the air-standard assumptions, determine (a) the gas temperature at the exits of the compressor and turbine, (b) the back work ratio, and (c) the thermal efficiency."

Identifying states 1 and 2 as the inlet and the outlet of the compressor, respectively, and states 3 and 4 as the inlet and outlet of the turbine, the Lua script shown below computes the data requested in the problem description, with the results shown later in Listing 7. We have tried to make the variable names self-explanatory.
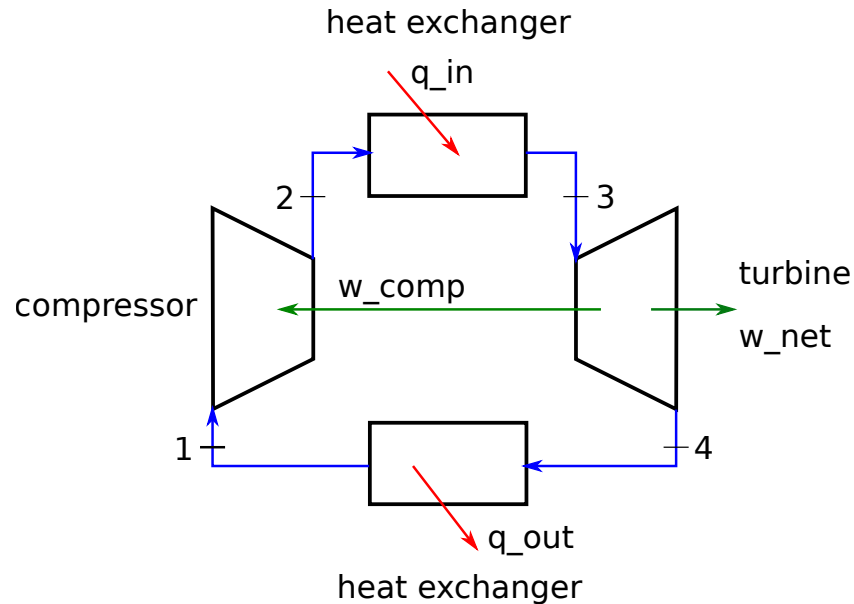
Figure 5.3: Schematic diagram of the closed Brayton cycle. Part of the work from the turbine goes directly into the compressor and the remainder is labelled as w_net.

```lua
1 -- brayton.lua
2 -- Simple Ideal Brayton Cycle using air-standard assumptions.
3 -- Corresponds to Example 9-5 in the 5th Edition of
4 -- Cengal and Boles' thermodynamics text.
5 --
6 -- To run the script:
7 -- $ prep-gas ideal-air.inp ideal-air-gas-model.lua
8 -- $ prep-gas thermal-air.inp thermal-air-gas-model.lua
9 -- $ gas-calc brayton.lua
10 --
11 -- Peter J and Rowan G. 2016-12-19
12
13 gasModelFile = "thermal-air-gas-model.lua"
14 -- gasModelFile = "ideal-air-gas-model.lua" -- Alternative
15 gmodel = GasModel:new{gasModelFile}
16 if gmodel:nSpecies() == 1 then
17     print("Ideal air gas model.")
18     air_massf = {air=1.0}
19 else
20     print("Thermally-perfect air model")
21     air_massf = {N2=0.78, O2=0.22}
22 end
23
24 print("Compute cycle states:")
25 Q = {} -- We will build up a table of gas states
26 h = {} -- and enthalpies.
27 for i=1,4 do
28     Q[i] = GasState:new{gmodel}
29     Q[i].massf = air_massf
30     h[i] = 0.0
31 end
```

```
32
33 print("   Start with ambient air")
34 Q[1].p = 100.0e3; Q[1].T = 300.0
35 gmodel:updateThermoFromPT(Q[1])
36 s12 = gmodel:entropy(Q[1])
37 h[1] = gmodel:enthalpy(Q[1])
38
39 print("   Isentropic compression with a pressure ratio of 8")
40 Q[2].p = 8 * Q[1].p
41 gmodel:updateThermoFromPS(Q[2], s12)
42 h[2] = gmodel:enthalpy(Q[2])
43
44 print("   Constant pressure heat addition to T=1300K")
45 Q[3].p = Q[2].p; Q[3].T = 1300.0
46 gmodel:updateThermoFromPT(Q[3])
47 h[3] = gmodel:enthalpy(Q[3])
48 s34 = gmodel:entropy(Q[3])
49
50 print("   Isentropic expansion to ambient pressure")
51 Q[4].p = Q[1].p
52 gmodel:updateThermoFromPS(Q[4], s34)
53 h[4] = gmodel:enthalpy(Q[4])
54
55 print("")
56 print("State    Pressure Temperature    Enthalpy")
57 print("             kPa              K       kJ/kg")
58 print("------------------------------------------")
59 for i=1,4 do
60    print(string.format(" %4d %10.2f  %10.2f %10.2f",
61                         i, Q[i].p/1000, Q[i].T, h[i]/1000))
62 end
63 print("------------------------------------------")
64 print("")
65 print("Cycle performance:")
66 work_comp_in = h[2] - h[1]
67 work_turb_out = h[3] - h[4]
68 heat_in = h[3] - h[2]
69 rbw = work_comp_in / work_turb_out
70 eff = (work_turb_out-work_comp_in) / heat_in
71 print(string.format("   turbine work out = %.2f kJ/kg", work_turb_out/1000))
72 print(string.format("   back work ratio = %.3f", rbw))
73 print(string.format("   thermal_efficiency = %.3f", eff))
```

In the script, we start by constructing a suitable gas model. The thermally-perfect air model is based on a mix of nitrogen and oxygen with an equilibrium thermal model dependent on temperature. The gas model data were selected from the database of species by the `prep-gas` program with the following input.

```
model = 'thermally perfect gas'
species = {'N2', 'O2'}
```

Depending on which gas model was constructed, lines 16 through 22 set up a suitable list of mass fractions for use in constructing the GasState objects. Lines 25 through 31 construct those GasState objects and hold them in as array called Q, a somewhat

unfortunate choice for this example, because `q` is often used to denote heat flow in the thermodynamics texts. No matter; we can work with this notation.

Lines 33 through 53 are the core of this example. The gas states are computed, one at a time, until the full cycle is defined. Because the Brayton cycle is a steady state flow machine, it is convenient to use differences enthalpies to compute the heat and work flows, as done in lines 66 through 68. Listing 7 shows the results, with the computed values being pleasingly close to the values found in Çengel and Boles' text.

```
Thermally-perfect air model
Compute cycle states:
   Start with ambient air
   Isentropic compression with a pressure ratio of 8
   Constant pressure heat addition to T=1300K
   Isentropic expansion to ambient pressure

State    Pressure Temperature   Enthalpy
            kPa              K      kJ/kg
--------------------------------------
   1     100.00       300.00       1.87
   2     800.00       539.08     247.11
   3     800.00      1300.00    1106.49
   4     100.00       771.40     496.25
--------------------------------------

Cycle performance:
   turbine work out = 610.23 kJ/kg
   back work ratio = 0.402
   thermal_efficiency = 0.425
```

Listing 7: Computed states and performance of a closed Brayton cycle.

Reference Guide to the Lua API

6.1 Global constants

When using a program written for `gas-calc`, there are some global constants that are set for the user's convenience. These are physical constants and are as follows.

`R_universal`
The universal gas constant with value 8.31451 J/(mol.K).

`P_atm`
Atmospheric pressure given in Pascals. $P_{atm} = 101.325 \times 10^3$ Pa.

6.2 GasState

The `GasState` is a Lua table with the following specific entries.

```
GasState = {
   rho = <density, float>, -- kg/m^3
   p = <pressure, float>, -- Pa
   T = <(transrotational) temperature, float>, -- K
   p_e = <electron pressure, float>, -- Pa
   a = <sound speed, float>, -- m/s
   u = <internal energy (in transrotational mode), float>, -- J/kg
   e_modes = <internal energies in other modes, array of floats>, J/kg
   T_modes <temperatures in other modes, array of floats>, K
   mu = <dynamic viscosity, float>, -- Pa.s
   k = <thermal conductivity (in transrotational mode), float>,
           -- W/(m.K)
   k_modes <thermal conductivities in other modes, array of floats>,
           -- W/(m.K)
   sigma = <electrical conductivity, float>, -- S/m
   massf = <mass fractions, array of float>, -- no units
   quality = <vapour quality, float> -- no units
}
```

6.3 GasModel

6.3.1 Constructor

GasModel:new{**filename**}
Initialises a gas model by reading instructions from `filename`. Returns a `GasModel` object.

6.3.2 General service methods

GasModel:nSpecies()
Returns the number of species in the gas mixture.

GasModel:nModes()
Returns the number of nonequilibrium energy modes in the gas mixture. For a single-temperature gas model, this number should be zero.

GasModel:molMasses()
Returns a table in array form (numbered from 1) with the molecular masses of the component species in kg/mole.

GasModel:speciesName(isp)
Returns the name of species `isp`. Species are indexed starting from 1 in the Lua API.

GasModel:createGasState()
Returns a GasState table with fields default initialised.

6.3.3 Equation of state service methods

GasModel:updateThermoFromPT(Q)
Computes the thermodynamic state assuming that pressure, temperatures and mass fractions are up-to-date in the GasState `Q`. Updates the density and internal energy values in `Q`.

GasModel:updateThermoFromRHOE(Q)
Computes the thermodynamic state assuming that density, internal energies and mass fractions are up-to-date in the GasState `Q`. Updates the pressure and the temperature values in `Q`.

GasModel:updateThermoFromRHOT(Q)
Computes the thermodynamic state assuming that density, temperatures and mass fractions are up-to-date in the GasState `Q`. Updates the pressure and internal energy values in `Q`.

GasModel:updateThermoFromRHOP(Q)
Computes the therodynamic state assuming that density, pressure and mass fractions are up-to-date in the the GasState `Q`. Updates the temperature and internal energy values in `Q`.

6.3.4 Transport coefficient service method

GasModel:updateTransCoeffs(Q)
Computes the viscosity and thermal conductivity based on an up-to-date thermodynamic state in the GasState `Q`. Updates the values of `mu` and `k` in `Q`.

6.3.5 General service methods that *return* values

All of the following methods assume that the thermodynamic state in `Q` is up-to-date. One should usually make a call to one of the thermodynamic update methods before using these service methods. All of the following methods return a value or array of values to the caller.

GasModel:dpdrhoConstT(Q)
Compute and return the derivative value $\left(\frac{\partial p}{\partial \rho}\right)_T$.

GasModel:intEnergy(Q)
Compute and return the internal energy of the gas mixture, u.

GasModel:enthalpy(Q)
Compute and return the enthalpy of the gas mixture, h.

GasModel:enthalpy(Q, i)
Compute and return the enthalpy of species `i`, h_i.

GasModel:entropy(Q)
Compute and return the entropy of the gas mixture, s.

GasModel:entropy(Q, i)
Compute and return the entropy of species `i`, s_i.

GasModel:Cv(Q)
Compute and return the specific heat at constant volume of the gas mixture, C_v.

GasModel:dudTConstV(Q)
This method is an alias for `GasModel:Cv` since the definition of C_v is $\left(\frac{\partial u}{\partial T}\right)_v$. It is provided in this form as it is thermodynamic derivative that appears commonly in expressions.

GasModel:Cp(Q)
Compute and return the specific heat at constant pressure of the gas mixture, C_p.

GasModel:dhdTConstP(Q)
This method is an alias for `GasModel:Cp` since the definition of C_p is $\left(\frac{\partial h}{\partial T}\right)_p$. It is provided in this form as it is thermodynamic derivative that appears commonly in expressions.

GasModel:R(Q)
Compute and return the specific gas constant of the gas mixture, R.

GasModel:gasConstant(Q)
An alias for `GasModel:R()`.

GasModel:gamma(Q)
Compute and return the ratio of specific heats (C_p/C_v) of the gas mixture, γ.

GasModel:molMass(Q)
Compute and return the molecular mass of the gas mixture.

6.3.6   Composition conversion methods

GasModel:massf2molef(Q)
Using the table of mass fractions in the GasState, `Q.massf`, compute and return a table of mole fractions.

GasModel:molef2massf(molef, Q) Using the supplied table of mole fractions and the gas state in `Q`, compute and return a table of mass fractions. As a side-effect, the table `Q.massf` is also updated. It may be convenient to ignore the returned table and simply work with the updated table in `Q.massf`.

GasModel:massf2conc(Q)
Using the table of mass fractions in the GasState, `Q.massf`, compute and return a table of species concentrations.

GasModel:conc2massf(molef, Q) Using the supplied table of concentrations and the gas state in `Q`, compute and return a table of mass fractions. As a side-effect, the table `Q.massf` is also updated. It may be convenient to ignore the returned table and simply work with the updated table in `Q.massf`.

References

[1] E. Bender. Equations of state exactly representing the phase behavior of pure substances. In *Proceedings of the Fifth Symposium on Thermophys. Prop., ASME, New York*, 1970.

[2] R. Span and W. Wagner. A new equation of state for carbon dioxide covering the fluid region from the triple-point temperature to 1100 k at pressures up to 800 mpa. *Journal of physical and chemical reference data*, 25(6):1509–1596, 1996.

[3] B. J. McBride and S. Gordon. Computer program for calculation of complex chemical equilibrium compositions and applications. Part 2: Users manual and program description. Reference Publication 1311, NASA, 1996.

[4] R. Ierusalimschy. *Programming in Lua*. Lua.org, PUC-Rio, Brazil, 2nd edition, 2006.

[5] Y. A. Çengel and M. A. Boles. *Thermodynamics: An Engineering Approach*. McGraw-Hill, 5th edition, 2006.