Estimation of high-enthalpy flow conditions for simple shock and expansion processes using the ESTCj program and library.

Mechanical Engineering Report 2011/02
P. A. Jacobs, R. J. Gollan, D. F. Potter, F. Zander,
D. E. Gildfind, P. Blyton, W. Y. K. Chan and L. Doherty
Centre for Hypersonics
The University of Queensland

January 25, 2014

Abstract

This report presents the software tools that we have built to do simple flow process calculations for ideal gases and gases in chemical equilibrium. The software comes in the form of a library for the most fundamental processes and an application program, ESTCj, for convenient calculation of the combined flow processes relevant to shock- and expansion-tube operation.

Contents

1 Introduction **3**

2 Operation of the ESTCj program **4**

2.1 Example of use for T4 condition 5
2.2 Subset calculations 9
2.3 Pitot pressure calculation 10
2.4 Cone surface pressure calculation 10
2.5 Total condition calculation 11

3 Building custom application with the supporting libraries **12**

3.1 Oblique shock for air in chemical equilibrium 12
3.2 Classic shock tube 15
3.3 Idealized expansion tube 18

A Source code for gas models **22**

A.1 ideal_gas.py 22
A.2 libgas_gas.py 28
A.3 cea2_gas.py 33

B Source code for flow process calculations **53**

B.1 ideal_gas_flow.py 53
B.2 gas_flow.py 69

C Source code for ESTCj application **90**

D Notes on conical flow **102**

1 Introduction

ESTCj [1] began as a re-implementation of the ideas in the ESTC code [1] written by Malcolm McIntosh in the late 1960s and the shock-tube-plus-nozzle (STN) code [2] written in the early 1990s. The new program [3] was started while PJ was on study leave at DLR Goettingen, with a decision to delegate the equilibrium thermochemistry issues to the Gordon and McBride's Chemical Equilibrium Analysis (CEA) code [4, 5]. With the thermochemistry provided by CEA, the ESTCj program had to be concerned only with the smaller task of computing the flow changes across shocks and through the steady nozzle expansion.

Implementation was done in the Python programming language [2] which was easy for end users to customize so the program tended to grow in an ad-hoc fashion. This report describes the current generation of the program, which has been refactored into three layers:

1. Thermochemical gas models for perfect gases and gases in thermochemical equilibrium. Appendix A.

2. A library of functions for simple flow processes such as normal shocks, oblique shocks, and steady and unsteady expansions. Appendix B.

3. A top-level code (actually called estcj.py) that coordinates the calling of the flow-process functions using information provided by the user on the command line. Appendix C.

One of the advantages of moving the flow-process calculations to a library is that they can be conveniently reused, as has been done for the NENZFr code [6], for example. Three simple examples of building specific programs with the library are shown in Section 3.

The following sections provide an overview of the use of the new functions and their capabilities. This is done mostly by way of examples. The bulk of the detail is in the source code which we've tried to make modular and very readable. Despite the code being central to this report, we have put it in the Appendix so that there is a reasonable chance that the reader might at least get the overview before being overwhelmed by detail and giving up.

[1] Equilibrium Shock Tube Conditions, junior

[2] http://www.python.org

2 Operation of the ESTCj program

The application-level code is essentially a command-line interpreter that writes the results of the requested calculation to the standard-output stream by default. It's easiest to get a reminder of the available settings by asking for "help" on the command line.

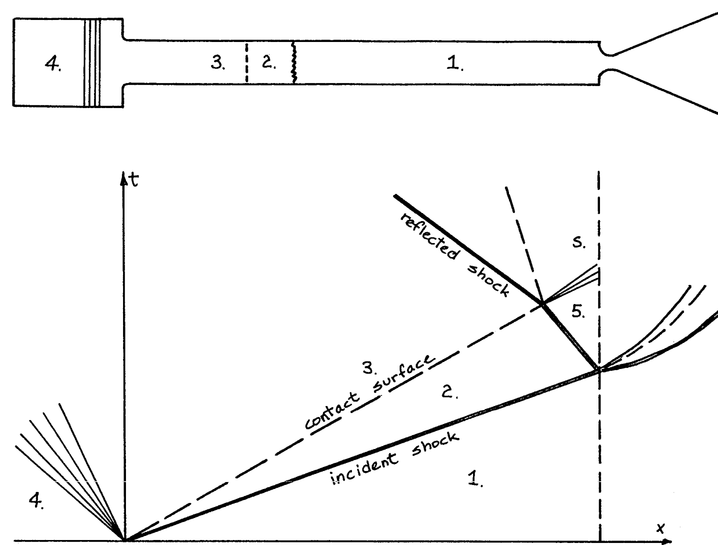
```
peterj@helmholtz ~/work/estcj-test $ estcj.py --help
Usage: estcj.py [options]

Options:
  --version              show program's version number and exit
  -h, --help             show this help message and exit
  --task=TASK            particular calculation to make: st = reflected shock
                         tube; stn = reflected shock tube with nozzle; stnp =
                         reflected shock tube with nozzle expanded to pitot;
                         ishock = incident shock only; total = free-stream to
                         total condition; pitot = free-stream to Pitot
                         condition; cone = free-stream to Taylor-Maccoll cone
                         flow
  --model=GASMODELNAME   type of gas model: cea2: equilibrium thermochemistry
                         provided by NASA CEA2 code; libgas: thermochemistry
                         provided by Rowan's libgas module; ideal: fixed
                         species with fixed thermodynamic coefficients.
  --gas=GASNAME          name of specific gas; To see the available gases, use
                         the option --list-gas-names
  --list-gas-names       list the gas names available for the current gas model
  --p1=P1                shock tube fill pressure or static pressure, in Pa
  --T1=T1                shock tube fill temperature, in degrees K
  --V1=V1                initial speed of gas in lab frame [default: none], in
                         m/s
  --Vs=VS                incident shock speed, in m/s
  --pe=PE                equilibrium pressure (after shock reflection), in Pa
  --pp_on_pe=PP_ON_PE    nozzle supply to exit pitot pressure ratio
  --ar=AREA_RATIO        exit-to-throat area ratio of the nozzle
  --sigma-deg=CONE_HALF_ANGLE_DEG
                         half-angle of the cone, in degrees
  --ofn=OUTFILENAME      name of file in which to accumulate output. file name
                         will be: outFileName-estcj.dat (Note that output
                         defaults to stdout.)
peterj@helmholtz ~/work/estcj-test $
```

The default supporting gas model library (Appendix A.3) calls upon the NASA Glenn CEA2 program for evaluation of the equilibrium thermochemical properties of gas mixtures. The list of available gases in `ESTCj` can be seen by using the `--list-gas-names` option on the command line. The list reflects the typical needs of the UQ shock and expansion tunnel operation but it is easy to add new gases to the `make_gas_from_name()` function in the gas model code.

2.1 Example of use for T4 condition

Built into ESTCj is an idealized model of a reflected shock tube. This model is composed of quasi-one-dimensional wave processes as shown in the following figure that has been taken from Ref. [2]. The numbers denote states of the gases between processes.



A typical low-enthalpy flow condition for the T4 shock tunnel may start with a test gas (air) at room temperature ($T_1 = 300\,\mathrm{K}$) and a little above atmospheric pressure ($p_1 = 125\,\mathrm{kPa}$). The observed shock speed, V_s, was 2414 m/s and the observed nozzle-supply pressure relaxed to 34.37 MPa. With the Mach 4 nozzle having an area ratio of 27, the flow conditions in the facility may be computed using the command:

```
$ estcj.py --task=stn --gas=air --T1=300 --p1=125.0e3 --Vs=2414 --pe=34.37e6 --ar=27.0
```

The full output is included below, where you should see that this condition has an enthalpy of $(H_{5a} - H_1) = 5.43\,\text{MJ/kg}$ and the nozzle-exit condition has a pressure of $P_7 = 93.6\,\text{kPa}$ and a static temperature of $T_7 = 1284\,\text{K}$, with a flow speed of $V_7 = 2.95\,\text{km/s}$. Note that we have selected to stop the expansion at a particular nozzle area ratio. Alternatively, we may stop the expansion at a particular Pitot pressure by specifying `--task=stnp` and a suitable ratio for the option `--pp_on_pe`. If you don't want to specify a relaxation pressure with option `--pe`, the reflected-shock conditions (5) will be used directly as the nozzle supply conditions.

```
$ estcj.py --task=stn --model=cea2 --gas=air --T1=300 --p1=125.0e3 --Vs=2414 --pe=34.37e6 --ar=27.0
estcj: Equilibrium Shock Tube Conditions
Version: 31-Dec-2013
Input parameters:
    gasModel is cea2, Gas is air, p1: 125000 Pa, T1: 300 K, Vs: 2414 m/s
Write pre-shock condition.
Start incident-shock calculation.
Start reflected-shock calculation.
Start calculation of isentropic relaxation.
Start isentropic relaxation to throat (Mach 1)
Start isentropic relaxation to nozzle exit.
Done with reflected shock tube calculation.
State 1: pre-shock condition
    p: 125000 Pa, T: 300 K, rho: 1.4515 kg/m**3, e: -88591 J/kg, h: -2475 J/kg, a: 347.2 m/s, s: 6806.3 J/(
       kg.K)
    R: 287.036 J/(kg.K), gam: 1.3999, Cp: 1004.8 J/(kg.K), mu: 1.8746e-05 Pa.s, k: 0.02639 W/(m.K)
    species massf: {'N2': 0.75518, 'Ar': 0.012916, 'CO2': 0.00048469, 'O2': 0.23142}
State 2: post-shock condition.
    p: 7.3158e+06 Pa, T: 2629.98 K, rho: 9.6848 kg/m**3, e: 2.09038e+06 J/kg, h: 2.84577e+06 J/kg, a: 971 m/
       s, s: 8128.4 J/(kg.K)
    R: 287.205 J/(kg.K), gam: 1.2482, Cp: 1280.8 J/(kg.K), mu: 8.4069e-05 Pa.s, k: 0.16977 W/(m.K)
    species massf: {'CO2': 0.00047578, 'NO': 0.02814, 'O': 0.0007597, 'Ar': 0.012916, 'N2': 0.74195, 'O2':
       0.21545, 'NO2': 0.00028032}
    V2: 361.796 m/s, Vg: 2052.2 m/s
State 5: reflected-shock condition.
    p: 5.95e+07 Pa, T: 4551.12 K, rho: 44.33 kg/m**3, e: 4.78627e+06 J/kg, h: 6.12847e+06 J/kg, a: 1277.7 m/
       s, s: 8446.5 J/(kg.K)
    R: 294.896 J/(kg.K), gam: 1.2163, Cp: 1326.1 J/(kg.K), mu: 0.00012602 Pa.s, k: 0.41556 W/(m.K)
    species massf: {'CO2': 0.00018873, 'CO': 0.00018836, 'NO': 0.12328, 'O': 0.030423, 'N': 0.00017756, 'Ar
       ': 0.012916, 'N2O': 0.00026787, 'N2': 0.69698, 'O2': 0.13453, 'NO2': 0.001022}
    Vr: 573.53 m/s
State 5s: equilibrium condition (relaxation to pe)
    p: 3.437e+07 Pa, T: 4161.9 K, rho: 28.201 kg/m**3, e: 4.20992e+06 J/kg, h: 5.42867e+06 J/kg, a: 1215.5 m
       /s, s: 8447 J/(kg.K)
```

```
    R: 292.819 J/(kg.K), gam: 1.2123, Cp: 1319.6 J/(kg.K), mu: 0.00011762 Pa.s, k: 0.37811 W/(m.K)
    species massf: {'CO2': 0.00024217, 'CO': 0.00015435, 'NO': 0.1058, 'O': 0.022439, 'Ar': 0.012916, 'N2O':
        0.00017027, 'N2': 0.70537, 'O2': 0.15202, 'NO2': 0.00079754}
Enthalpy difference (H5s - H1): 5.43114e+06 J/kg
State 6: Nozzle-throat condition (relaxation to M=1)
    p: 1.9291e+07 Pa, T: 3788.25 K, rho: 17.503 kg/m**3, e: 3.65885e+06 J/kg, h: 4.76102e+06 J/kg, a: 1155.5
        m/s, s: 8447.5 J/(kg.K)
    R: 290.923 J/(kg.K), gam: 1.2114, Cp: 1312.8 J/(kg.K), mu: 0.00010952 Pa.s, k: 0.333 W/(m.K)
    species massf: {'CO2': 0.00030528, 'CO': 0.00011418, 'NO': 0.087056, 'O': 0.015137, 'Ar': 0.012916, 'N2O
        ': 0.00010259, 'N2': 0.71427, 'O2': 0.16946, 'NO2': 0.00059814}
    V6: 1155.57 m/s, M6: 1.00006, mflux6: 20225.9 kg/s/m**2
State 7: Nozzle-exit condition (relaxation to correct mass flux)
    p: 93566 Pa, T: 1283.91 K, rho: 0.25388 kg/m**3, e: 706900 J/kg, h: 1.07545e+06 J/kg, a: 696.6 m/s, s:
        8447.5 J/(kg.K)
    R: 287.036 J/(kg.K), gam: 1.3166, Cp: 1186.2 J/(kg.K), mu: 5.1628e-05 Pa.s, k: 0.08122 W/(m.K)
    species massf: {'N2': 0.75501, 'Ar': 0.012916, 'O2': 0.23122, 'CO2': 0.00048469, 'NO': 0.00036646}
    V7: 2950.67 m/s, M7: 4.23582, mflux7: 20226.1 kg/s/m**2, area_ratio: 27, pitot: 2.1426e+06 Pa
    pitot7_on_p5s: 0.0623392
```

By default, the `cea2_gas` model is used, however, the CEA2 program can occasionally fail to provide data at conditions of interest so the `libgas_gas` module is provided as an alternative gas model. This `libgas` module is that used in the `Eilmer3` code and may model the gas thermochemistry via look-up table data that has previously been generated by the CEA2 program. Here is the same T4 condition computed with a look-up table describing the air gas model.

```
$ estcj.py --task=stn --model=libgas --gas=cea-lut-air-ions.lua.gz --T1=300 --p1=125.0e3 --Vs=2414 --pe
    =34.37e6 --ar=27.0
estcj: Equilibrium Shock Tube Conditions
Version: 31-Dec-2013
Input parameters:
    gasModel is libgas, Gas is cea-lut-air-ions.lua.gz, p1: 125000 Pa, T1: 300 K, Vs: 2414 m/s
Write pre-shock condition.
Start incident-shock calculation.
Start reflected-shock calculation.
Start calculation of isentropic relaxation.
Start isentropic relaxation to throat (Mach 1)
Start isentropic relaxation to nozzle exit.
Done with reflected shock tube calculation.
State 1: pre-shock condition
    p: 125000 Pa, T: 300 K, rho: 1.45151 kg/m**3, e: 215909 J/kg, h: 302026 J/kg, a: 346.041 m/s, s: 6796.22
        J/(kg.K)
```

```
15      R: 287.057 J/(kg.K), gam: 1.39083, Cp: 1021.54 J/(kg.K), mu: 2.5873e-05 Pa.s, k: 0.036838 W/(m.K)
16      filename: cea-lut-air-ions.lua.gz
17  State 2: post-shock condition.
18      p: 7.31558e+06 Pa, T: 2630.38 K, rho: 9.68296 kg/m**3, e: 2.39474e+06 J/kg, h: 3.15025e+06 J/kg, a:
           971.059 m/s, s: 8128.69 J/(kg.K)
19      R: 287.224 J/(kg.K), gam: 1.28907, Cp: 1280.86 J/(kg.K), mu: 8.40757e-05 Pa.s, k: 0.169875 W/(m.K)
20      filename: cea-lut-air-ions.lua.gz
21      V2: 361.868 m/s, Vg: 2052.13 m/s
22  State 5: reflected-shock condition.
23      p: 5.94881e+07 Pa, T: 4551.14 K, rho: 44.3195 kg/m**3, e: 5.09063e+06 J/kg, h: 6.43289e+06 J/kg, a:
           1277.77 m/s, s: 8446.67 J/(kg.K)
24      R: 294.927 J/(kg.K), gam: 1.28601, Cp: 1326.11 J/(kg.K), mu: 0.000126023 Pa.s, k: 0.415794 W/(m.K)
25      filename: cea-lut-air-ions.lua.gz
26      Vr: 573.581 m/s
27  State 5s: equilibrium condition (relaxation to pe)
28      p: 3.437e+07 Pa, T: 4160.98 K, rho: 28.2068 kg/m**3, e: 4.51268e+06 J/kg, h: 5.73119e+06 J/kg, a:
           1215.39 m/s, s: 8446.67 J/(kg.K)
29      R: 292.841 J/(kg.K), gam: 1.2852, Cp: 1319.62 J/(kg.K), mu: 0.000117598 Pa.s, k: 0.378141 W/(m.K)
30      filename: cea-lut-air-ions.lua.gz
31  Enthalpy difference (H5s - H1): 5.42916e+06 J/kg
32  State 6: Nozzle-throat condition (relaxation to M=1)
33      p: 1.93263e+07 Pa, T: 3787.55 K, rho: 17.5378 kg/m**3, e: 3.96169e+06 J/kg, h: 5.06367e+06 J/kg, a:
           1155.43 m/s, s: 8446.67 J/(kg.K)
34      R: 290.948 J/(kg.K), gam: 1.28474, Cp: 1312.75 J/(kg.K), mu: 0.000109502 Pa.s, k: 0.332947 W/(m.K)
35      filename: cea-lut-air-ions.lua.gz
36      V6: 1155.43 m/s, M6: 1, mflux6: 20263.7 kg/s/m**2
37  State 7: Nozzle-exit condition (relaxation to correct mass flux)
38      p: 93727.5 Pa, T: 1283.64 K, rho: 0.254376 kg/m**3, e: 1.01048e+06 J/kg, h: 1.37894e+06 J/kg, a: 696.536
            m/s, s: 8446.67 J/(kg.K)
39      R: 287.044 J/(kg.K), gam: 1.31937, Cp: 1185.81 J/(kg.K), mu: 5.15973e-05 Pa.s, k: 0.0811911 W/(m.K)
40      filename: cea-lut-air-ions.lua.gz
41      V7: 2950.34 m/s, M7: 4.23573, mflux7: 20263.3 kg/s/m**2, area_ratio: 27, pitot: 2.15045e+06 Pa
42      pitot7_on_p5s: 0.0625677
```

Note that there are some differences in computed detail. The look-up table was generated for temperatures higher than the initial shock tube fill temperature and the thermochemical model is using extrapolated parameter values for some of the calculation.

2.2 Subset calculations

Subset calculations of the shock-tube flow processing can be done by selecting a different task. For example, just the incident shock processing can be computed with the `--task=ishock`, specifying only the gas, initial pressure, temperature and incident shock speed. Here is an example from Huber's [7] Table IV for a speed of 37.06 ft/s at a geopotential altitude of 173500 feet. The expected pressure (from Table IV) is 86.5 kPa and the temperature is 12000 K, quite close to the values computed by `ESTCj` and shown below.

```
$ estcj.py --model=libgas --gas=cea-lut-air-ions.lua.gz --task=ishock --Vs=11296 --p1=59 --T1=283
estcj: Equilibrium Shock Tube Conditions
Version: 31-Dec-2013
Input parameters:
    gasModel is libgas, Gas is cea-lut-air-ions.lua.gz, p1: 59 Pa, T1: 283 K, Vs: 11296 m/s
Write pre-shock condition.
Start incident-shock calculation.
State 1: pre-shock condition
    p: 59 Pa, T: 283 K, rho: 0.000726319 kg/m**3, e: 203674 J/kg, h: 284905 J/kg, a: 336.094 m/s, s: 8935.09
        J/(kg.K)
    R: 287.037 J/(kg.K), gam: 1.3908, Cp: 1021.52 J/(kg.K), mu: 2.5873e-05 Pa.s, k: 0.036838 W/(m.K)
    filename: cea-lut-air-ions.lua.gz
State 2: post-shock condition.
    p: 86692.4 Pa, T: 12203.5 K, rho: 0.0111361 kg/m**3, e: 5.60285e+07 J/kg, h: 6.38133e+07 J/kg, a:
       3021.85 m/s, s: 18007.1 J/(kg.K)
    R: 637.915 J/(kg.K), gam: 1.4297, Cp: 2122.46 J/(kg.K), mu: 0.000253088 Pa.s, k: 5.34565 W/(m.K)
    filename: cea-lut-air-ions.lua.gz
    V2: 736.748 m/s, Vg: 10559.3 m/s
```

This equilibrium-chemistry result can be compared with the ideal gas calculation for the same speed and free-stream condition.

```
peterj@helmholtz ~/work/estcj-test $ estcj.py --model=ideal --gas=air --task=ishock --Vs=11296 --p1=59 --T1
   =283
estcj: Equilibrium Shock Tube Conditions
Version: 14-Jan-2014
Input parameters:
    gasModel is ideal, Gas is air, p1: 59 Pa, T1: 283 K, Vs: 11296 m/s
Write pre-shock condition.
Start incident-shock calculation.
State 1: pre-shock condition
    p: 59 Pa, T: 283 K, rho: 0.000726196 kg/m**3, e: 203113 J/kg, h: 284358 J/kg, a: 337.259 m/s, s: 2085.97
        J/(kg.K)
    R: 287.086 J/(kg.K), gam: 1.4, Cp: 1004.8 J/(kg.K), mu: 1.76518e-05 Pa.s, k: 0.024981 W/(m.K)
    name: air
```

```
State 2: post-shock condition.
    p: 77208.8 Pa, T: 61998.5 K, rho: 0.00433784 kg/m**3, e: 4.44972e+07 J/kg, h: 6.22961e+07 J/kg, a:
      4991.84 m/s, s: 5440.92 J/(kg.K)
    R: 287.086 J/(kg.K), gam: 1.4, Cp: 1004.8 J/(kg.K), mu: 0.000363093 Pa.s, k: 0.513854 W/(m.K)
    name: air
    V2: 1891.06 m/s, Vg: 9404.94 m/s
```

The following sections show the subproblems that can be exercised from the command line. These calculations can also be done inside other programs by calling the relevant gas_flow.py functions.

2.3 Pitot pressure calculation

Using the test flow conditions from the exit of the Mach 4 nozzle, we can then compute the expected Pitot pressure to be 2.14 MPa.

```
$ estcj.py --gas=air --task=pitot --p1=93.6e3 --T1=1284 --V1=2.95e3
estcj: Equilibrium Shock Tube Conditions
Version: 31-Dec-2013
Input parameters:
    gasModel is cea2, Gas is air, p1: 93600 Pa, T1: 1284 K, V1: 2950 m/s
Pitot condition:
    p: 2.1421e+06 Pa, T: 3875.52 K, rho: 1.8421 kg/m**3, e: 4.26382e+06 J/kg, h: 5.42667e+06 J/kg, a: 1176.1
      m/s, s: 9268.7 J/(kg.K)
    R: 300.036 J/(kg.K), gam: 1.1896, Cp: 1315.8 J/(kg.K), mu: 0.00011293 Pa.s, k: 0.52084 W/(m.K)
    species massf: {'CO2': 0.00014198, 'CO': 0.00021812, 'NO': 0.08357, 'O': 0.049853, 'N': 0.00010203, 'Ar
      ': 0.012916, 'N2': 0.716, 'O2': 0.137, 'NO2': 0.00016353}
```

2.4 Cone surface pressure calculation

Alternatively, the conditions on the surface of a conical pressure probe (with 15° half-angle) can be computed as:

```
$ estcj.py --gas=air --task=cone --sigma-deg=15 --p1=93.6e3 --T1=1284 --V1=2.95e3
estcj: Equilibrium Shock Tube Conditions
Version: 31-Dec-2013
Input parameters:
    gasModel is cea2, Gas is air, p1: 93600 Pa, T1: 1284 K, V1: 2950 m/s, sigma: 15 degrees
Free-stream condition:
```

```
7      p: 93600 Pa, T: 1284 K, rho: 0.25395 kg/m**3, e: 706980 J/kg, h: 1.07556e+06 J/kg, a: 696.6 m/s, s:
          8447.5 J/(kg.K)
8      R: 287.036 J/(kg.K), gam: 1.3166, Cp: 1186.2 J/(kg.K), mu: 5.1631e-05 Pa.s, k: 0.08122 W/(m.K)
9      species massf: {'N2': 0.75501, 'Ar': 0.012916, 'O2': 0.23122, 'CO2': 0.00048469, 'NO': 0.00036668}
10 Shock angle: 0.366546 (rad), 21.0015 (deg)
11 Cone-surface velocity: 2784.57 m/s
12 Cone-surface condition:
13     p: 271070 Pa, T: 1680.41 K, rho: 0.56197 kg/m**3, e: 1.07961e+06 J/kg, h: 1.56197e+06 J/kg, a: 790.3 m/s
          , s: 8472 J/(kg.K)
14     R: 287.036 J/(kg.K), gam: 1.2947, Cp: 1227.5 J/(kg.K), mu: 6.1822e-05 Pa.s, k: 0.10326 W/(m.K)
15     species massf: {'N2': 0.75389, 'Ar': 0.012916, 'O2': 0.22992, 'CO2': 0.00048465, 'NO': 0.0027591}
```

2.5 Total condition calculation

The hypothetical stagnation conditions for a specified free stream can be computed as:

```
1 $ estcj.py --gas=air --task=total --p1=93.6e3 --T1=1284 --V1=2.95e3
2 estcj: Equilibrium Shock Tube Conditions
3 Version: 31-Dec-2013
4 Input parameters:
5     gasModel is cea2, Gas is air, p1: 93600 Pa, T1: 1284 K, V1: 2950 m/s
6 Total condition:
7     p: 3.42e+07 Pa, T: 4160.57 K, rho: 28.07 kg/m**3, e: 4.2084e+06 J/kg, h: 5.42677e+06 J/kg, a: 1215.3 m/s
          , s: 8448 J/(kg.K)
8     R: 292.819 J/(kg.K), gam: 1.2123, Cp: 1319.6 J/(kg.K), mu: 0.00011759 Pa.s, k: 0.37814 W/(m.K)
9     species massf: {'CO2': 0.00024218, 'CO': 0.00015434, 'NO': 0.10572, 'O': 0.022444, 'Ar': 0.012916, 'N2O
          ': 0.00016972, 'N2': 0.70541, 'O2': 0.15205, 'NO2': 0.00079546}
```

3 Building custom application with the supporting libraries

Although the ESTCj is built specifically to do the calculations needed for flow conditions typical of the T4 reflected shock tunnel, the supporting libraries are more general. There are gas modules for:

- a perfect gas, with user specified properties (Appendix A.1).
- a mixture of gases in thermochemical equilibrium (Appendix A.3). This module delegates calculation to the CEA2 code.
- the same gas models that are used in the L1d3 and Eilmer3 codes, but with chemical reactions omitted (Appendix A.2).

The flow process modules cover simple processes associated with:

- normal shocks for one-dimensional flow.
- finite (isentropic) waves for one-dimensional flow.
- steady quasi-one-dimensional flow with area change.
- oblique shocks for planar and conical flow.

There is a module (Appendix B.1) that assumes an ideal gas model and is very much an implementation of the classic textbook gas-dynamic equations. When using this module, the user needs to specify the relevant gas properties (such as the ratio of specific heats). The second flow process module (Appendix B.2) uses `Gas` objects created by the cea2, libgas and ideal gas modules of Appendix A and so can compute flow processes for a gas with the species in chemical equilibrium or frozen.

There are many ways these functions can be combined, however, this section is deliberately terse because the codes in the appendices are well documented and follow the standard texts on gas dynamics. The only unusual formulation is that for the Taylor-Maccoll flow with the general gas model. For that formulation, the notes from PJ's workbook are included in Appendix D.

3.1 Oblique shock for air in chemical equilibrium

Hunt and Sounders [8] provide tabulated data for the processing of air in chemical equilibrium by oblique shocks. Here are a couple of examples of calling up the `gas_flow` functions to do the same job with a `cea2_gas` object.

```
#!/usr/bin/env python
"""
oblique_shock_example.py

Demonstration of using the library functions to compute flow conditions
across an oblique shock in equilibrium air.
Data are chosen to match examples from Hunt and Souders NASA-SP-3093.

PJ, 01-Jan-2014
"""
from math import pi
import sys, os
sys.path.append(os.path.expandvars("$HOME/e3bin"))
from cfpylib.gasdyn.cea2_gas import Gas
from cfpylib.gasdyn.gas_flow import theta_oblique, beta_oblique

print "Example 1: Hunt and Souders Table VIII, sub-table (j)"
print "Given shock angle, compute post-shock conditions."
s1 = Gas({'Air':1.0})
s1.set_pT(52.671, 268.858)
print "Initial gas state:"
s1.write_state(sys.stdout)
beta = 45.0 * pi/180 # shock angle
V1 = 7.9248e3
theta, V2, s2 = theta_oblique(s1, V1, beta)
print("Following oblique shock, beta=%g degrees, theta=%g degrees (Hunt&Souders 45 40.638)"
      % (beta*180/pi, theta*180/pi))
s2.write_state(sys.stdout)
print "Across shock:"
print "p2/p1=%g, T2/T1=%g (Hunt&Souders: 376.84 21.206)" % (s2.p/s1.p, s2.T/s1.T)

print "\nExample 2: Hunt and Souders Table VI, sub-table (a)"
print "Given deflection angle, compute shock angle and then post-shock conditions."
s1.set_pT(3542.7, 219.428)
print "Initial gas state:"
s1.write_state(sys.stdout)
theta = 33.671 * pi/180 # deflection angle
V1 = 1.8288e3
beta = beta_oblique(s1, V1, theta)
```

```
print("Following oblique shock, beta=%g degrees, theta=%g degrees (Hunt&Souders 45 33.671)"
      % (beta*180/pi, theta*180/pi))
theta2, V2, s2 = theta_oblique(s1, V1, beta)
s2.write_state(sys.stdout)
print "Across shock:"
print "p2/p1=%g, T2/T1=%g (Hunt&Souders: 22.23 4.4472)" % (s2.p/s1.p, s2.T/s1.T)

print "Done."
```

```
$ ./oblique_shock_example.py
Example 1: Hunt and Souders Table VIII, sub-table (j)
Given shock angle, compute post-shock conditions.
Initial gas state:
    p: 52.6687 Pa, T: 268.86 K, rho: 0.00068248 kg/m**3, e: -110920 J/kg, h: -33746 J/kg, a: 328.8 m/s, s:
       8927.3 J/(kg.K)
    R: 287.036 J/(kg.K), gam: 1.4006, Cp: 1003.6 J/(kg.K), mu: 1.7247e-05 Pa.s, k: 0.02421 W/(m.K)
    species massf: {'N2': 0.75518, 'Ar': 0.012916, 'CO2': 0.00048469, 'O2': 0.23142}
Following oblique shock, beta=45 degrees, theta=40.6318 degrees (Hunt&Souders 45 40.638)
    p: 19845 Pa, T: 5708.46 K, rho: 0.0089339 kg/m**3, e: 1.3354e+07 J/kg, h: 1.55753e+07 J/kg, a: 1583.5 m/
       s, s: 12925.2 J/(kg.K)
    R: 389.105 J/(kg.K), gam: 1.1288, Cp: 1417.7 J/(kg.K), mu: 0.00016594 Pa.s, k: 2.6222 W/(m.K)
    species massf: {'C': 5.9906e-06, 'CO': 0.00029308, 'CN': 1.3116e-06, 'NO': 0.0055341, 'O': 0.22853, 'N':
        0.14368, 'Ar': 0.012916, 'N2': 0.60891, 'O2': 0.00012124}
Across shock:
p2/p1=376.811, T2/T1=21.2321 (Hunt&Souders: 376.84 21.206)

Example 2: Hunt and Souders Table VI, sub-table (a)
Given deflection angle, compute shock angle and then post-shock conditions.
Initial gas state:
    p: 3543 Pa, T: 219.43 K, rho: 0.056245 kg/m**3, e: -146310 J/kg, h: -83325 J/kg, a: 297.1 m/s, s: 7515.4
        J/(kg.K)
    R: 287.036 J/(kg.K), gam: 1.4012, Cp: 1002.6 J/(kg.K), mu: 1.4699e-05 Pa.s, k: 0.02045 W/(m.K)
    species massf: {'N2': 0.75518, 'Ar': 0.012916, 'CO2': 0.00048469, 'O2': 0.23142}
Following oblique shock, beta=45.0253 degrees, theta=33.671 degrees (Hunt&Souders 45 33.671)
    p: 78800 Pa, T: 979.13 K, rho: 0.28037 kg/m**3, e: 438790 J/kg, h: 719850 J/kg, a: 613.1 m/s, s: 8181 J
       /(kg.K)
    R: 287.036 J/(kg.K), gam: 1.3374, Cp: 1137 J/(kg.K), mu: 4.3071e-05 Pa.s, k: 0.06511 W/(m.K)
    species massf: {'CO2': 0.00048469, 'NO': 2.5483e-05, 'Ar': 0.012916, 'N2': 0.75517, 'O2': 0.2314, 'NO2':
        2.1728e-06}
Across shock:
```

```
p2/p1=22.241, T2/T1=4.46215 (Hunt&Souders: 22.23 4.4472)
Done.
```

3.2 Classic shock tube

As an example of building a custom application, consider the idealized shock tube with equal area sections separated by a diaphragm. State 1 is air at low pressure on the downstream-side of a diaphragm and state 4 is high pressure helium initially on the upstream side of the diaphragm. When the diaphragm is removed (ideally), the helium expands into the part of the tube occupied initially by the air and drives a shock through the quiescent air. State 2 is the shock-compressed air and state 3 is the expanded helium driving the air. At the moving contact surface between the air and helium, the pressures and velocities of the air and helium have to match. See for example, Section 7.8 (Shock tube relations) in Anderson's text [9] for a discussion based on perfect gas behaviour.

The example code sets up a function that, given the pressure at the contact surface, returns the difference in velocities of the gases at the contact surface. This function is passed to a nonlinear equation solver to determine the pressure ratio at which the velocity difference is zero. All of the interesting calculation, along with the printing of the computed states, is done by line 60. The next 40 lines (approximately) of the script write out the flow data in small steps, so that they may be used for comparison with data from a CFD calculation.

```
#!/usr/bin/env python
"""
classic_shock_tube.py

Moderately high-performance shock tube with helium driving air.
Done as an example of using gas_flow functions but can be
compared the Eilmer3 sod shock tube example.

PJ, 22-Mar-2012
"""

import sys, os
sys.path.append(os.path.expandvars("$HOME/e3bin"))

from cfpylib.gasdyn.cea2_gas import Gas
from cfpylib.gasdyn.gas_flow import normal_shock, finite_wave_dp, normal_shock_p2p1
from cfpylib.nm.zero_solvers import secant

def main():
    print "Helium driver gas"
```

```
21    state4 = Gas({'He':1.0})
22    state4.set_pT(30.0e6, 3000.0)
23    print "state4:"
24    state4.write_state(sys.stdout)
25    #
26    print "Air driven gas"
27    state1 = Gas({'Air':1.0})
28    state1.set_pT(30.0e3, 300.0)
29    print "state1:"
30    state1.write_state(sys.stdout)
31    #
32    print "\nNow do the classic shock tube solution..."
33    # For the unsteady expansion of the driver gas, regulation of the amount
34    # of expansion is determined by the shock-processed test gas.
35    # Across the contact surface between these gases, the pressure and velocity
36    # have to match so we set up some trials of various pressures and check
37    # that velocities match.
38    def error_in_velocity(p3p4, state4=state4, state1=state1):
39        "Compute the velocity mismatch for a given pressure ratio across the expansion."
40        # Across the expansion, we get a test-gas velocity, V3g.
41        p3 = p3p4*state4.p
42        V3g, state3 = finite_wave_dp('cplus', 0.0, state4, p3)
43        # Across the contact surface.
44        p2 = p3
45        print "current guess for p3 and p2=", p2
46        V1s, V2, V2g, state2 = normal_shock_p2p1(state1, p2/state1.p)
47        return (V3g - V2g)/V3g
48    p3p4 = secant(error_in_velocity, 0.1, 0.11, tol=1.0e-3)
49    print "From secant solve: p3/p4=", p3p4
50    print "Expanded driver gas:"
51    p3 = p3p4*state4.p
52    V3g, state3 = finite_wave_dp('cplus', 0.0, state4, p3)
53    print "V3g=", V3g
54    print "state3:"
55    state3.write_state(sys.stdout)
56    print "Shock-processed test gas:"
57    V1s, V2, V2g, state2 = normal_shock_p2p1(state1, p3/state1.p)
58    print "V1s=", V1s, "V2g=", V2g
59    print "state2:"
60    state2.write_state(sys.stdout)
```

```
61      assert abs(V2g - V3g)/V3g < 1.0e-3
62      #
63      # Make a record for plotting against the Eilmer3 simulation data.
64      # We reconstruct the expected data along a tube 0.0 <= x <= 1.0
65      # at t=100us, where the diaphragm is at x=0.5.
66      x_centre = 0.5 # metres
67      t = 100.0e-6 # seconds
68      fp = open('exact.data', 'w')
69      fp.write('# 1:x(m)  2:rho(kg/m**3) 3:p(Pa) 4:T(K) 5:V(m/s)\n')
70      print 'Left end'
71      x = 0.0
72      fp.write('%g %g %g %g %g\n' % (x, state4.rho, state4.p, state4.T, 0.0))
73      print 'Upstream head of the unsteady expansion.'
74      x = x_centre - state4.a * t
75      fp.write('%g %g %g %g %g\n' % (x, state4.rho, state4.p, state4.T, 0.0))
76      print 'The unsteady expansion in n steps.'
77      n = 100
78      dp = (state3.p - state4.p) / n
79      state = state4.clone()
80      V = 0.0
81      p = state4.p
82      for i in range(n):
83          rhoa = state.rho * state.a
84          dV = -dp / rhoa
85          V += dV
86          p += dp
87          state.set_ps(p, state4.s)
88          x = x_centre + t * (V - state.a)
89          fp.write('%g %g %g %g %g\n' % (x, state.rho, state.p, state.T, V))
90      print 'Downstream tail of expansion.'
91      x = x_centre + t * (V3g - state3.a)
92      fp.write('%g %g %g %g %g\n' % (x, state3.rho, state3.p, state3.T, V3g))
93      print 'Contact surface.'
94      x = x_centre + t * V3g
95      fp.write('%g %g %g %g %g\n' % (x, state3.rho, state3.p, state3.T, V3g))
96      x = x_centre + t * V2g  # should not have moved
97      fp.write('%g %g %g %g %g\n' % (x, state2.rho, state2.p, state2.T, V2g))
98      print 'Shock front'
99      x = x_centre + t * V1s  # should not have moved
100     fp.write('%g %g %g %g %g\n' % (x, state2.rho, state2.p, state2.T, V2g))
```

```
    fp.write('%g %g %g %g %g\n' % (x, state1.rho, state1.p, state1.T, 0.0))
    print 'Right end'
    x = 1.0
    fp.write('%g %g %g %g %g\n' % (x, state1.rho, state1.p, state1.T, 0.0))
    fp.close()
    return

if __name__ == '__main__':
    main()
    print "Done."
```

3.3 Idealized expansion tube

As a second example of building a custom application, consider the idealized expansion of the test gas in an expansion tube [10]. We will include just the processing of the test gas by the incident shock, followed by the unsteady expansion to test-section conditions. The states in the calculation are:

1. Initial (quiescent) test gas, filling the shock tube.

2. Shock-processed test gas.

5. Expanded test gas, as would be expected to emerge from the downstream-end of the acceleration tube.

10. Initial accelerator gas, filling the acceleration tube, downstream of the shock tube.

20. Shock-processed accelerator gas that is pushed along, in front of the expanded test gas.

The final expansion process is regulated by the fill pressure of the acceleration tube and the test-gas conditions are determined by balancing the expanded gas pressure against the post shock pressure of the acceleration gas. When computing this balance iteratively, we guess the pressure and compute the two velocities. As done in the classic shock-tube example, we use the difference between the two velocities as the measure of error for the guessed pressure.

```
#!/usr/bin/env python
"""
classic_expansion_tube.py -- Hadas' 8.5 expansion-tube condition.

Done as an example of using gas_flow functions.
```

```
PJ, 21-Mar-2012
"""

import sys, os
sys.path.append(os.path.expandvars("$HOME/e3bin"))

from cfpylib.gasdyn.cea2_gas import Gas
from cfpylib.gasdyn.gas_flow import normal_shock, finite_wave_dp, normal_shock_p2p1
from cfpylib.nm.zero_solvers import secant

def main():
    print "Titan gas"
    state1 = Gas({'N2':0.95, 'CH4':0.05}, inputUnits='moles', outputUnits='moles')
    state1.set_pT(2600.0, 300.0)
    print "state1:"
    state1.write_state(sys.stdout)
    #
    print "Air accelerator gas"
    state10 = Gas({'Air':1.0})
    state10.set_pT(10.0, 300.0)
    print "state10:"
    state10.write_state(sys.stdout)
    #
    print "Incident shock"
    state2 = state1.clone()
    V2,V2g = normal_shock(state1, 4100.0, state2)
    print "V2=", V2, "Vg=", V2g, "expected 3670.56"
    print "state2:"
    state2.write_state(sys.stdout)
    print "Checks:"
    print "p2/p1=", state2.p/state1.p, "expected 166.4"
    print "rho2/rho1=", state2.rho/state1.rho, "expected 9.5474"
    print "T2/T1=", state2.T/state1.T, "expected 14.9"
    #
    print "\nNow do unsteady expansion..."
    # For the unsteady expansion of the test gas, regulation of the amount
    # of expansion is determined by the shock-processed accelerator gas.
    # Across the contact surface between these gases, the pressure and velocity
    # have to match so we set up some trials of various pressures and check
    # that velocities match.
```

```
46     def error_in_velocity(p5p2, state2=state2, V2g=V2g, state10=state10):
47         "Compute the velocity mismatch for a given pressure ratio across the expansion."
48         # Across the expansion, we get a test-gas velocity, V5g.
49         V5g, state5 = finite_wave_dp('cplus', V2g, state2, p5p2*state2.p)
50         # Across the contact surface, p20 == p5
51         p20 = p5p2 * state2.p
52         print "current guess for p5 and p20=", p20
53         V10, V20, V20g, state20 = normal_shock_p2p1(state10, p20/state10.p)
54         return (V5g - V10)/V5g # V10 was V20g - lab speed of accelerator gas - we now make the assumption
               that this is the same as the shock speed
55     p5p2 = secant(error_in_velocity, 0.01, 0.011, tol=1.0e-3)
56     print "From secant solve: p5/p2=", p5p2
57     # It would have been faster and the code closer to Hadas' spreadsheet if we had
58     # stepped down in pressure until we found the point where the velocities matched.
59     # The expansion along the u+a wave would have appeared in the code here.
60     V5g, state5 = finite_wave_dp('cplus', V2g, state2, p5p2*state2.p)
61     print "Expanded test gas, at end of acceleration tube:"
62     print "V5g=", V5g
63     print "state5:"
64     state5.write_state(sys.stdout)
65     V10, V20, V20g, state20 = normal_shock_p2p1(state10, state5.p/state10.p)
66     print V10
67     print "Done."
68     return
69
70 if __name__ == '__main__':
71     main()
```

References

[1] M. K. McIntosh. A computer program for the numerical calculation of equilibrium and perfect gas conditions in shock tunnels. Technical Note CPD 169, Australian Defence Scientific Service, Department of Supply, Salisbury, South Australia, 1970.

[2] R. M. Krek and P. A. Jacobs. STN, shock tube and nozzle calculations for equilibrium air. Department of Mechanical Engineering Report 2/93, The University of Queensland, February 1993.

[3] P. A. Jacobs, A. D. Gardner, and K. Hannemann. Gas-dynamic modelling of the HEG shock tunnel. Report DLR-IB 224-2003 A02, Deutsches Zentrum für Luft- und Raumfahrt E.V., Göttingen, Germany, January 2003.

[4] S. Gordon and B. J. McBride. Computer program for calculation of complex chemical equilibrium compositions and applications. part 1: Analysis. Reference Publication 1311, NASA, 1994.

[5] 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.

[6] L. Doherty, W. Y. K. Chan, P. A. Jacobs, F. Zander, R. M. Kirchhartz, and R. J. Gollan. Nenzfr: Non-equilibrium nozzle flow, reloaded. School of Mechanical and Mining Engineering Technical Report 2012/08, The University of Queensland, Brisbane, June 2012.

[7] Paul W. Huber. Hypersonic shock-heated flow parameters for velocities to 46000 ft/s and altitudes to 323000 feet. Technical Report R-163, NASA, December 1963.

[8] James L. Hunt and Sue W. Souders. Normal- and oblique-shock flow parameters in equilibrium air including attached-shock solutions for surfaces at angle of attack, sweep and dihedral. Special Publication NASA-SP-3093, NASA, 1975.

[9] J. D. Anderson. *Modern Compressible Flow: with Historical Perspective.* McGraw-Hill, New York, 1982.

[10] R. L. Trimpi. A preliminary theoretical study of the expansion tube, a new device for producing high-enthalpy short-duration hypersonic gas flows. NASA Technical Report R-133, 1962.

A Source code for gas models

A.1 ideal_gas.py

Thermodynamic functions for an ideal gas.

```
#! /usr/bin/env python
"""
ideal_gas.py: Thermodynamic properties of an ideal gas.

This module provides a look-alike Gas class for use in
the gas flow functions.  Whereever cea2_gas works, so should this.

.. Author:
   PA Jacobs
   School of Mechanical Engineering
   The University of Queensland

.. Versions:
   02-Apr-12: first cut from cea2_gas.py
"""

import sys, math

R_universal = 8314.0;  # J/kgmole.K

class Gas(object):
    """
    Provides the place to keep property data for the ideal gas.
    """
    def __init__(self, Mmass=28.96, gamma=1.4, name='air',
                 s1=0.0, T1=298.15, p1=101.325e3,
                 mu_ref=1.716e-5, T_ref=273.0, S_mu=111.0,
                 Prandtl=0.71):
        """
        Set up a new object, from either a name of species list.

        :param Mmass: molecular mass, g/mole
        :param gamma: ratio of specific heats
        :param name: string name of gas (something like a species name in cea2_gas)
```

```
35          :param s1: reference entropy, J/kg/K
36          :param T1: temperature for reference entropy, K
37          :param p1: pressure for reference entropy, Pa
38          :param mu_ref: reference viscosity for Sutherland expression, Pa.s
39          :param T_ref: reference temperature for Sutherland expression, degree K
40          :param S_mu: constant (degree K) in Sutherlans expression
41          :param Prandtl: mu.C_p/k
42          """
43          assert gamma > 1.0 and gamma <= 2.0, ('odd value: gamma=%g' % gamma)
44          assert Mmass > 1.0 and Mmass < 1000.0, ('odd value: Mmass=%g' % Mmass)
45          self.Mmass = Mmass
46          self.R = R_universal / Mmass
47          self.gam = gamma
48          self.C_v = self.R / (gamma - 1)
49          self.C_p = self.R + self.C_v
50          self.name = name
51          # reference entropy
52          self.s1 = s1
53          self.T1 = T1
54          self.p1 = p1
55          # Data for transport properties, based on Sutherland variation.
56          self.mu_ref = mu_ref
57          self.T_ref = T_ref
58          self.S_mu = S_mu
59          self.Prandtl = Prandtl
60          # set default thermo conditions
61          self.set_pT(100.0e3, 300.0)
62          return
63
64      def clone(self):
65          """
66          Clone the current Gas object to make another, just the same.
67
68          :returns: the new Gas object.
69          """
70          other = Gas(self.Mmass, self.gam, self.name,
71                      s1=self.s1, T1=self.T1, p1=self.p1,
72                      mu_ref=self.mu_ref, T_ref=self.T_ref, S_mu=self.S_mu,
73                      Prandtl=self.Prandtl)
74          other.set_pT(self.p, self.T)
```

```
75            return other
76
77        def set_pT(self, p, T, transProps=True):
78            """
79            Fills out gas state from given pressure and temperature.
80
81            :param p: pressure, Pa
82            :param T: temperature, K
83            :param transProps: if True, compute transport properties as well.
84            """
85            self.p = p
86            self.T = T
87            self.rho = p / (self.R * T)
88            self.a = math.sqrt(self.gam * self.R * T)
89            self.e = self.C_v * T
90            self.h = self.C_p * T
91            self.s = self.s1 + self.C_p * math.log(T/self.T1) - self.R * math.log(p/self.p1)
92            if transProps:
93                self.mu = self.mu_ref * (T/self.T_ref)**1.5 * (self.T_ref+self.S_mu)/(T+self.S_mu)
94                self.k = self.mu * self.C_p / self.Prandtl
95            else:
96                self.mu = 0.0
97                self.k = 0.0
98            return
99
100       def set_rhoT(self, rho, T, transProps=True):
101           """
102           Fills out gas state from given density and temperature.
103
104           :param rho: density, kg/m**3
105           :param T: temperature, K
106           """
107           p = rho * self.R * T
108           return self.set_pT(p, T, transProps)
109
110       def set_ps(self, p, s, transProps=True):
111           """
112           Fills out gas state from given pressure and specific entropy.
113
114           :param p: pressure, Pa
```

```
115         :param s: entropy, J/(kg.K)
116         """
117         cp_ln_TT1 = s - self.s1 + self.R * math.log(p/self.p1)
118         T = self.T1 * math.exp(cp_ln_TT1 / self.C_p)
119         return self.set_pT(p, T, transProps)
120
121     def set_ph(self, p, h, transProps=True):
122         """
123         Fills out gas state from given pressure and enthalpy.
124
125         :param p: pressure, Pa
126         :param h: enthalpy, J/kg
127         """
128         T = h / self.C_p
129         return self.set_pT(p, T, transProps)
130
131     def write_state(self, strm):
132         """
133         Writes the gas state data to the specified stream.
134         """
135         strm.write('    p: %g Pa, T: %g K, rho: %g kg/m**3, e: %g J/kg, h: %g J/kg, a: %g m/s, s: %g J/(kg.K
            )\n'
136                    % (self.p, self.T, self.rho, self.e, self.h, self.a, self.s) )
137         strm.write('    R: %g J/(kg.K), gam: %g, Cp: %g J/(kg.K), mu: %g Pa.s, k: %g W/(m.K)\n'
138                    % (self.R, self.gam, self.C_p, self.mu, self.k) )
139         strm.write('    name: %s\n' % self.name)
140         return
141
142 def make_gas_from_name(gasName):
143     """
144     Manufacture a Gas object from a small library of options.
145
146     :param gasName: one of the names for the special cases set out below
147     """
148     if gasName in ['air', 'Air', 'air5species']:
149         return Gas()
150     elif gasName in ['n2', 'N2', 'nitrogen']:
151         return Gas(Mmass=28.0, gamma=1.4, name='N2',
152                    s_1=0.0, T_1=298.15, p_1=101.325e3,
153                    mu_ref=1.663e-5, T_ref=273.0, S_mu=107.0,
```

```
154                     Prandtl=0.71)
155     elif gasName in ['co2', 'CO2', 'carbon dioxide', 'carbon-dioxide']:
156         return Gas(Mmass=44.0, gamma=1.301, name='CO2',
157                    s_1=0.0, T_1=298.15, p_1=101.325e3,
158                    mu_ref=1.370e-5, T_ref=273.0, S_mu=222.0,
159                    Prandtl=0.72)
160     else:
161         raise Exception, 'make_gas_from_name(): unknown gasName: %s' % gasName
162 
163 def list_gas_names():
164     """
165     :returns: the list of gases available in make_gas_from_name()
166     """
167     return ['air', 'n2', 'co2']
168 
169 # --------------------------------------------------------------
170 
171 if __name__ == '__main__':
172     print 'Test/demonstrate the Gas class...'
173     print 'gases available in make_gas_from_name():'
174     for name in list_gas_names():
175         print "   ", name
176     #
177     print '\nDefault constructor with Air as the test gas.'
178     a = Gas()
179     a.set_pT(100.0e3, 300.0)
180     a.write_state(sys.stdout)
181     print 'and the same Air at a higher temperature'
182     a.set_pT(100.0e3, 4000.0)
183     a.write_state(sys.stdout)
184     #
185     print '\nCheck enthalpy specification'
186     b = make_gas_from_name('air')
187     b.set_ph(a.p, a.h)
188     b.write_state(sys.stdout)
189     #
190     print '\nCheck entropy specification'
191     b = make_gas_from_name('air')
192     b.set_ps(a.p, a.s)
193     b.write_state(sys.stdout)
```

```
194     #
195     print 'End of test.'
```

A.2 libgas_gas.py

Thermodynamic functions for the gas model used by Eilmer3.

```
#! /usr/bin/env python
"""
libgas_gas.py: access the gas models from the libgas library using the
cfpylib/gasdyn interface.

.. Author: Peter J Blyton
.. Version: 21/06/2012
.. Version: 11-Dec-2013 generalised a little by PeterJ
"""

from ..nm.zero_solvers import secant
import sys, os
sys.path.append(os.path.expandvars("$HOME/e3bin"))
try:
    from gaspy import *
    libgas_ok = True
except:
    libgas_ok = False

class Gas(object):
    """
    Provides the place to hold the libgas gas data object and gas model object.
    """
    def __init__(self, fname='gas-model.lua', massf=None, molef=None):
        """
        Set up the libgas model from the generic input file.

        :param fname: gas-model config file
        :param massf: optional dictionary of mass fractions
        :param molef: optional dictionary of mole fractions

        Rowan's thermochemistry module uses the Lua file to define
        the gas model, in detail.  There are so many options for
        this input file that we whimp out and delegate the construction
        of a suitable file to other tools.  One such tool is gasmodel.py
        which, in turn, delegates all of it's work to Rowan's Lua
        program gasfile.lua.
```

```
38          """
39          if not libgas_ok:
40              raise ImportError("Cannot use libgas_gas model because gaspy cannot be found.")
41          self.fname = fname
42          self.gasModel = create_gas_model(fname)
43          self.gasData = Gas_data(self.gasModel)
44          if massf is None and molef is None:
45              name0 = self.gasModel.species_name(0)
46              if name0 == "LUT": # [todo] we really need to fix the look-up-table code.
47                  set_massf(self.gasData, self.gasModel, [1.0,])
48              else:
49                  set_molef(self.gasData, self.gasModel, {name0:1.0})
50          elif (type(massf) is dict) or (type(massf) is list):
51              set_massf(self.gasData, self.gasModel, massf)
52          elif (type(molef) is dict) or (type(molef) is list):
53              set_molef(self.gasData, self.gasModel, molef)
54          self.set_pT(100.0e3, 300.0)
55          return
56
57      def clone(self):
58          """
59          Clone the current Gas object to make another, just the same.
60
61          :returns: the new Gas object.
62          """
63          other = Gas(self.fname)
64          nsp = self.gasModel.get_number_of_species()
65          other.gasData.massf = self.gasData.massf
66          other.set_pT(self.p, self.T)
67          return other
68
69      def set_pT(self, p, T, transProps=True):
70          """
71          Compute the thermodynamic state from given pressure and temperature.
72
73          :param p: pressure, Pa
74          :param T: temperature, K
75          :param transProps: if True, compute transport properties as well.
76          """
77          self.p = p
```

```
78          self.gasData.p = p
79          self.T = T
80          self.gasData.T[0] = T # [todo] consider all modes
81          # Calculate density, sound speed, internal energy and quality if available
82          self.gasModel.eval_thermo_state_pT(self.gasData)
83          self.rho = self.gasData.rho
84          self.a = self.gasData.a
85          self.e = self.gasModel.mixture_internal_energy(self.gasData, 0.0)
86          self.quality = self.gasData.quality
87          # Manually call methods to calculate other thermodynamic properties
88          self.h = self.gasModel.mixture_enthalpy(self.gasData, 0.0)
89          self.s = self.gasModel.mixture_entropy(self.gasData)
90          self.R = self.gasModel.R(self.gasData)
91          self.C_p = self.gasModel.Cp(self.gasData)
92          self.C_v = self.gasModel.Cv(self.gasData)
93          self.gam = self.gasModel.gamma(self.gasData)
94          if transProps:
95              self.gasModel.eval_transport_coefficients(self.gasData)
96              self.mu = self.gasData.mu
97              self.k = self.gasData.k[0] # [todo] sum over all modes
98          else:
99              self.mu = 0.0
100             self.k = 0.0
101         return
102
103     def set_rhoT(self, rho, T, transProps=True):
104         """
105         Compute the thermodynamic state from given density and temperature.
106
107         :param rho: density, kg/m**3
108         :param T: temperature, K
109         :param transProps: if True, compute transport properties as well.
110         """
111         self.gasData.rho = rho
112         self.gasData.T[0] = T
113         self.gasModel.eval_thermo_state_rhoT(self.gasData)
114         return self.set_pT(self.gasData.p, T, transProps)
115
116     def set_ps(self, p, s, transProps=True):
117         """
```

```
118         Compute the thermodynamic state from given pressure and entropy
119
120         :param p: pressure, Pa
121         :param s: entropy, J/(kg.K)
122         :param transProps: if True, compute transport properties as well.
123         """
124         # The libgas library does not have a pressure-entropy thermodynamic
125         # state solver, so we need to do the iterative calculation ourselves.
126         gasData2 = Gas_data(self.gasModel)
127         for isp in range(self.gasModel.get_number_of_species()):
128             gasData2.massf[isp] = self.gasData.massf[isp]
129         def entropy_solve(temp):
130             gasData2.p = p
131             gasData2.T[0] = temp # [todo] consider all modes
132             self.gasModel.eval_thermo_state_pT(gasData2) # calculate density
133             entropy = self.gasModel.mixture_entropy(gasData2)
134             # print "debug p=", p, "s=", s, "temp=", temp, "entropy=", entropy
135             return s - entropy
136         # expecting values of entropy of several thousand
137         # so we don't want the tolerance too small
138         T = secant(entropy_solve, 250.0, 260.0, tol=1.0e-4)
139         if T == "FAIL": raise Exception("set_ps(): Secant solver failed.")
140         return self.set_pT(p, T, transProps)
141
142     def write_state(self, strm):
143         """
144         Writes the gas state data to the specified stream.
145         """
146         strm.write('    p: %g Pa, T: %g K, rho: %g kg/m**3, e: %g J/kg, h: %g J/kg, a: %g m/s, s: %g J/(kg.K
               )\n'
147                    % (self.p, self.T, self.rho, self.e, self.h, self.a, self.s) )
148         strm.write('    R: %g J/(kg.K), gam: %g, Cp: %g J/(kg.K), mu: %g Pa.s, k: %g W/(m.K)\n'
149                    % (self.R, self.gam, self.C_p, self.mu, self.k) )
150         strm.write('    filename: %s\n' % self.fname)
151         return
152
153 def make_gas_from_name(gasName):
154     """
155     Manufacture a Gas object from a small library of options.
156
```

```
157     :param gasName: one of the names for the special cases set out below.
158         We might also specify the details of the gas via a Lua gas-model file
159         or via a compressed look-up table, again in Lua format.
160     """
161     if gasName.lower() in ['co2-refprop']:
162         os.system('gasmodel.py --model="real gas REFPROP"'+
163                   ' --species="CO2.FLD" --output="co2-refprop.lua"')
164         return Gas('co2-refprop.lua')
165     elif gasName.lower() in ['co2-bender']:
166         os.system('gasmodel.py --model="real gas Bender"'+
167                   ' --species="CO2" --output="co2-bender.lua"')
168         return Gas('co2-bender.lua')
169     elif gasName.lower() in ['air-thermally-perfect']:
170         os.system('gasmodel.py --model="thermally perfect gas" --species="N2 O2"')
171         return Gas('gas-model.lua', molef={'O2':0.21, 'N2':0.79})
172     elif gasName.lower() in ['r134a-refprop']:
173         os.system('gasmodel.py --model="real gas REFPROP"'+
174                   ' --species="R134A.FLD" --output="r134a-refprop.lua"')
175         return Gas('r134a-refprop.lua')
176     elif gasName.lower().find('.lua') >= 0:
177         # Look-up tables are contained in files with names like cea_lut_xxxx.lua.gz
178         # and previously-constructed gas models may be supplied in a gas-model.lua file.
179         fname = gasName
180         if os.path.exists(fname):
181             return Gas(fname)
182         else:
183             raise RuntimeError('make_gas_from_name(): gas model file %s does not exist.' % fname)
184     else:
185         raise RuntimeError('make_gas_from_name(): unknown gasName: %s' % gasName)
186
187 def list_gas_names():
188     """
189     :returns: the list of gases available in make_gas_from_name()
190     """
191     return ['co2-refprop', 'co2-bender', 'air-thermally-perfect', 'r134a-refprop',
192             '<gas-model-filename>']
```

A.3 cea2_gas.py

Thermodynamic functions for the thermochemical-equilibrium gas model backed by CEA2.

```
#! /usr/bin/env python
"""
cea2_gas.py: Thermodynamic properties of a gas mixture in chemical equilibrium.

It interfaces to the CEA code by writing a small input file,
running the CEA code as a child process and then reading the results
from the CEA plot file.

See the report::

    Bonnie J. McBride and Sanford Gordon
    Computer Program for Calculation of Complex Chemical Equilibrium
    Compositions and Applications II. Users Manual and Program
    Description. NASA Reference Publication 1311, June 1996.

for details of the input and output file formats.

.. Author:
   PA Jacobs RJ Gollan and DF Potter
   Institute of Aerodynamics and Flow Technology
   The German Aerospace Center, Goettingen.
   and
   School of Mechanical Engineering
   The University of Queensland

.. Versions:
   24-Dec-02: First code.
   10-May-04: Updated for a mix of species.
   06-Feb-05: renamed to cea_gas.py
   28-Feb-08: Added a get_eq_massf() access function.
   28-Fed-08: Major changes to allow proper calculation at high temps.
   11-Dec-08: Addition of basic incident Shock function
   19-Feb-12: some refactoring, simplification and general clean-up
"""

import sys, string, math, os, subprocess, re
from copy import copy
```

```
# ------------------------------------------------------------------
# First, global data.

DEBUG_GAS = 0
R_universal = 8314.0;  # J/kgmole.K

# Set name for cea executable. If we are not on a Windows
# machine then assume we are on a Linux-like machine
if sys.platform.startswith('win'):
    CEA_COMMAND_NAME = 'fcea2.exe'
else:
    CEA_COMMAND_NAME = 'cea2'

# ------------------------------------------------------------------
# Second, utility functions.

def locate_executable_file(name):
    """
    Locates an executable file, if available somewhere on the PATH.

    :param name: may be a simple file name or fully-qualified path.
    :returns: the full program name, if is is found and is executable,
        else None.
    """
    def is_exe(path):
        return os.path.exists(path) and os.access(path, os.X_OK)

    head, tail = os.path.split(name)
    if head:
        # If there is a head component, we may have been given
        # full path to the exe_file.
        if is_exe(name): return name
    else:
        # We've been given the name of the program
        # without the fully-qualified path in front,
        # now search the PATH for the program.
        #
        # At the highest level of estcj we have added
        # e3bin and local estcj path to sys.path. Searching
```

```
78        # over sys.path  ensures that estcj/cea2_gas will
79        # work on Windows machines. Luke D. 24-May-12
80        for path in sys.path:
81            fullName = os.path.join(path, name)
82            if is_exe(fullName): return fullName
83        # Note that sys.path is initialized from PYTHONPATH,
84        # at least on linux machines,
85        # so we might need to search the PATH as well.  PJ 25-Jul-12
86        for path in os.environ["PATH"].split(os.pathsep):
87            fullName = os.path.join(path, name)
88            if is_exe(fullName): return fullName
89    return None
90
91 def run_cea_program(jobName,checkTableHeader=True):
92    """
93    Runs the CEA program on the specified job.
94
95    :param jobName: string that is used to construct input and output file names
96    :param checkTableHeader: boolean flag to activate checking of output file
97        table header.  We use this as a test to see if the cea2 program has run
98        the job succesfully.
99    """
100   inpFile = jobName + '.inp'
101   outFile = jobName + '.out'
102   pltFile = jobName + '.plt'
103   if os.path.exists(inpFile):
104       if DEBUG_GAS >= 2:
105           print 'cea2_gas: Start cea program on job %s...' % jobName
106       # We should remove the results files from previous runs.
107       if os.path.exists(pltFile): os.remove(pltFile)
108       if os.path.exists(outFile): os.remove(outFile)
109       p = subprocess.Popen(CEA_COMMAND_NAME, stdin=subprocess.PIPE,
110                            stdout=subprocess.PIPE, stderr=subprocess.PIPE)
111       out, err = p.communicate(jobName + '\n')
112       return_code = p.wait()
113       if DEBUG_GAS >= 2:
114           print('cea2_gas: %s finished job %s.' % (CEA_COMMAND_NAME,jobName))
115       if return_code != 0:
116           print('cea2_gas: return-code from cea2 program is nonzero.')
117           raise Exception,  'cea2-return-code = %d' % return_code
```

```
118        fp = open(outFile, 'r')
119        outFileText = fp.read()
120        outFileIsBad = False
121        if checkTableHeader:
122            # Look for the summary table header
123            if outFileText.find('THERMODYNAMIC PROPERTIES') == -1:
124                outFileIsBad = True
125        if outFileIsBad:
126            print('cea2_gas: the output file seems incomplete; you should go check.')
127            raise Exception, 'cea2_gas: detected badness in cea2 output file.'
128    else:
129        raise Exception, 'cea2_gas: The file %s is not present.' % inpFile
130
131def get_cea2_float(token_list):
132    """
133    Clean up the CEA2 short-hand notation for exponential format.
134
135    CEA2 seems to write exponential-format numbers in a number of ways:
136
137    | 1.023-2
138    | 1.023+2
139    | 1.023 2
140    """
141    if len(token_list) == 0:
142        value_str = '0.0'
143    elif len(token_list) == 1:
144        value_str = token_list[0]
145        if value_str.find("****") >= 0:
146            # We have one of the dodgy strings such as ******e-3
147            # CEA2 seems to write such for values like 0.0099998
148            # We should be able to tolerate one of these, at most,
149            # because we should be able to back out the intended
150            # value from knowledge of the rest of the list.
151            return None
152        if value_str.find("-") > 0:
153            value_str = value_str.replace("-","e-")
154        if value_str.find("+") > 0:
155            value_str = value_str.replace("+","e+")
156    elif len(token_list) == 2:
157        value_str = token_list[0] + 'e+' + token_list[1]
```

```
158     else:
159         print "get_cea2_float(): too many tokens (expected one or two, only):", token_list
160         value_str = '0.0'
161     try:
162         value = float(value_str)
163     except:
164         print "Cannot make a float from this string: ", value_str
165         sys.exit(-1)
166     return value
167 
168 # ----------------------------------------------------------------
169 
170 class Gas(object):
171     """
172     Provides the equation of state for the gas.
173     """
174     def __init__(self, reactants={}, onlyList=[],
175                  inputUnits='massf', outputUnits='massf',
176                  with_ions=False, trace=1.0e-6):
177         """
178         Set up a new object, from either a name of species list.
179 
180         :param reactants: dictionary of reactants and their mixture fractions
181             The keys used to specify the reactants in the mix
182             and the (float) values are their mass- or mole-fractions.
183             The names are as per the CEA database.
184             Note that other chemical species may be added to the mix by cea2.
185         :param onlyList: list of strings limiting the species in the mix.
186         :param inputUnits: string 'moles' or 'massf'
187         :param outputUnits: string 'moles' or 'massf'
188         :param with_ions: boolean flag indicating whether electrons and ions
189             should be included in the mix
190         :param trace: fraction below which a species will be neglected in CEA
191         """
192         if locate_executable_file(CEA_COMMAND_NAME) is None:
193             print "Could not find the executable program %s" % CEA_COMMAND_NAME
194             print "The chemical equilibrium-analysis program is external"
195             print "to the cfcfd3 code collection and needs to be obtained from NASA Glenn."
196             print "Quitting the current program because we really can't do anything further."
197             sys.exit()
```

```
198         # ------------------------------------------------------------------
199         assert inputUnits == 'moles' or inputUnits == 'massf'
200         assert outputUnits == 'moles' or outputUnits == 'massf'
201         self.reactants = copy(reactants)
202         self.inputUnits = inputUnits
203         self.outputUnits = outputUnits
204         self.onlyList = copy(onlyList)
205         self.species = {} # will be read from CEA2 output
206         self.with_ions = with_ions or ('e-' in self.reactants.keys()) or ('e-' in self.onlyList)
207         self.trace = trace
208         self.Us = 0.0 # m/s
209         self.have_run_cea = False
210         return
211 
212     def clone(self, newOutputUnits=None):
213         """
214         Clone the current Gas object to make another, just the same.
215 
216         :returns: the new Gas object.
217         """
218         if newOutputUnits == None: newOutputUnits = self.outputUnits
219         other = Gas(self.reactants, self.onlyList, self.inputUnits,
220                     newOutputUnits, self.with_ions, self.trace)
221         if self.have_run_cea:
222             other.p = self.p
223             other.T = self.T
224             other.Us = self.Us
225             other.trace = self.trace
226             other.EOS(problemType='pT', transProps=True)
227         return other
228 
229     def set_pT(self, p, T, transProps=True):
230         """
231         Fills out gas state from given pressure and temperature.
232 
233         :param p: pressure, Pa
234         :param T: temperature, K
235         """
236         self.p = p; self.T = T
237         return self.EOS(problemType='pT', transProps=transProps)
```

```
238
239     def set_rhoT(self, rho, T, transProps=True):
240         """
241         Fills out gas state from given density and temperature.
242
243         :param rho: density, kg/m**3
244         :param T: temperature, K
245         """
246         self.rho = rho; self.T = T
247         return self.EOS(problemType='rhoT', transProps=transProps)
248
249     def set_rhoe(self, rho, e, transProps=True):
250         """
251         Fills out gas state from given density and internal energy.
252
253         :param rho: density, kg/m**3
254         :param e: internal energy of mixture, J/kg
255         """
256         self.rho = rho; self.e = e
257         return self.EOS(problemType='rhoe', transProps=transProps)
258
259     def set_ps(self, p, s, transProps=True):
260         """
261         Fills out gas state from given pressure and specific entropy.
262
263         :param p: pressure, Pa
264         :param s: entropy, J/(kg.K)
265         """
266         self.p = p; self.s = s
267         return self.EOS(problemType='ps', transProps=transProps)
268
269     def set_ph(self, p, h, transProps=True):
270         """
271         Fills out gas state from given pressure and enthalpy.
272
273         :param p: pressure, Pa
274         :param h: enthalpy, J/kg
275         """
276         self.p = p; self.h = h
277         return self.EOS(problemType='ph', transProps=transProps)
```

```
278
279     def write_state(self, strm):
280         """
281         Writes the gas state data to the specified stream.
282         """
283         strm.write('    p: %g Pa, T: %g K, rho: %g kg/m**3, e: %g J/kg, h: %g J/kg, a: %g m/s, s: %g J/(kg.K
               )\n'
284                    % (self.p, self.T, self.rho, self.e, self.h, self.a, self.s) )
285         strm.write('    R: %g J/(kg.K), gam: %g, Cp: %g J/(kg.K), mu: %g Pa.s, k: %g W/(m.K)\n'
286                    % (self.R, self.gam, self.cp, self.mu, self.k) )
287         strm.write('    species %s: %s\n' % (self.outputUnits, str(self.species)) )
288         return
289
290     def get_fractions(self, speciesList):
291         """
292         Gets a list of mole- or mass-fractions for the specified species.
293
294         :param speciesList: the species names for which we want a list of fractions.
295         :returns: list of floats representing the fractions of each species in the mix
296             Note that the mass-fractions or mole-fractions are returned, based on
297             the value of outputUnits in the Gas object.
298         """
299         fractionList = []
300         for s in speciesList:
301             if s in self.species.keys():
302                 fractionList.append(self.species[s])
303             else:
304                 fractionList.append(0.0)
305         return fractionList
306
307     def write_cea2_input_file(self, problemType, transProps):
308         """
309         Set up a problem-description file for CEA2.
310
311         :param problemType: a string specifying type of CEA analysis that is requested:
312             'pT', 'rhoT', 'rhoe', 'ps', 'shock'
313         :param transProps: a boolean flag:
314             False=don't request transport props, True=request viscosity and thermal-conductivity
315         :returns: None
316         """
```

```
317         if DEBUG_GAS >= 2:
318             print 'EOS: Write temporary input file.'
319         inp_file_name = 'tmp.inp'
320         fp = open(inp_file_name, 'w')
321         fp.write('# %s generated by cea2_gas.py\n' % inp_file_name)
322         if problemType == 'rhoT':
323             if self.with_ions:
324                 fp.write('problem case=estcj tv ions\n')
325             else:
326                 fp.write('problem case=estcj tv\n')
327             assert self.rho > 0.0
328             assert self.T > 0.0
329             fp.write('   rho,kg/m**3 %e\n' % self.rho)
330             fp.write('   t(k)        %e\n' % self.T)
331             if DEBUG_GAS >= 2:
332                 print 'EOS: input to CEA2 rho: %g, T: %g' % (self.rho, self.T)
333         elif problemType == 'rhoe':
334             if self.with_ions:
335                 fp.write('problem case=estcj vu ions\n')
336             else:
337                 fp.write('problem case=estcj vu\n')
338             assert self.rho > 0.0
339             fp.write('   rho,kg/m**3 %e\n' % self.rho)
340             fp.write('   u/r         %e\n' % (self.e / R_universal) )
341             if DEBUG_GAS >= 2:
342                 print 'EOS: input to CEA2 rho: %g, e: %g' % (self.rho, self.e)
343         elif problemType == 'pT':
344             if self.with_ions:
345                 fp.write('problem case=estcj tp ions\n')
346             else:
347                 fp.write('problem case=estcj tp\n')
348             assert self.p > 0.0, self.T > 0.0
349             fp.write('   p(bar)      %e\n' % (self.p / 1.0e5) )
350             fp.write('   t(k)        %e\n' % self.T)
351             if DEBUG_GAS >= 2:
352                 print 'EOS: input to CEA2 p: %g, T: %g' % (self.p, self.T)
353         elif problemType == 'ps':
354             if self.with_ions:
355                 fp.write('problem case=estcj ps ions\n')
356             else:
```

```
357                 fp.write('problem case=estcj ps\n')
358             assert self.p > 0.0
359             fp.write('   p(bar)      %e\n' % (self.p / 1.0e5) )
360             fp.write('   s/r         %e\n' % (self.s / R_universal) )
361             if DEBUG_GAS >= 2:
362                 print 'EOS: input to CEA2 p: %g, s/r: %g' % (self.p, self.s)
363         elif problemType == 'ph':
364             if self.with_ions:
365                 fp.write('problem case=estcj ph ions\n')
366             else:
367                 fp.write('problem case=estcj ph\n')
368             assert self.p > 0.0
369             fp.write('   p(bar)      %e\n' % (self.p / 1.0e5) )
370             fp.write('   h/r         %e\n' % (self.h / R_universal) )
371             if DEBUG_GAS >= 2:
372                 print 'EOS: input to CEA2 p: %g, h/r: %g' % (self.p, self.s)
373         elif problemType == 'shock':
374             if self.with_ions:
375                 fp.write('problem shock inc eq ions\n')
376             else:
377                 fp.write('problem shock inc eq\n')
378             assert self.p > 0.0, self.T > 0.0
379             fp.write('   p(bar)      %e\n' % (self.p / 1.0e5) )
380             fp.write('   t(k)        %e\n' % self.T)
381             fp.write('   u1          %e\n' % self.Us)
382         else:
383             raise Exception, 'cea2_gas: Invalid problemType: %s' % problemType
384         # Select the gas components.
385         fp.write('reac\n')
386         for s in self.reactants.keys():
387             f = self.reactants[s]
388             if f > 0.0:
389                 if self.inputUnits == 'moles':
390                     fp.write('   name= %s  moles=%g' % (s, f))
391                 else:
392                     fp.write('   name= %s  wtf=%g' % (s, f))
393                 if problemType in ['ph', 'rhoe']: fp.write(' t=300')
394                 fp.write('\n')
395         #
396         if len(self.onlyList) > 0:
```

```
397             fp.write('only %s\n' % (' '.join(self.onlyList)))
398         #
399         fp.write('output')
400         if self.outputUnits == 'massf': fp.write(' massf')
401         fp.write(' trace=%e' % self.trace)
402         if transProps: fp.write(' trans')
403         fp.write('\n')
404         #
405         fp.write('end\n')
406         fp.close()
407         return
408 
409     def scan_cea2_dot_out_file(self, transProps):
410         """
411         Scan the output text file generated by CEA2 and extract our gas-properties data.
412 
413         :param transProps: a boolean flag:
414             False=don't request transport props, True=request viscosity and thermal-conductivity
415         :returns: None, but does update the contents of the gas state as a side-effect.
416         """
417         # use the .out file as this allows more species to be included
418         fp = open('tmp.out', 'r')
419         lines = fp.readlines()
420         fp.close()
421         thermo_props_found = False
422         conductivity_found = False
423         incident_shock_data = False
424         for line in lines:
425             if line=="\n": continue
426             if line.find("PRODUCTS WHICH WERE CONSIDERED BUT WHOSE")>=0: break
427             if (line.find("THERMODYNAMIC EQUILIBRIUM PROPERTIES AT ASSIGNED")>=0 or
428                 line.find("THERMODYNAMIC EQUILIBRIUM COMBUSTION PROPERTIES AT ASSIGNED")>=0):
429                 thermo_props_found = True
430             elif line.find("SHOCKED GAS (2)--INCIDENT--EQUILIBRIUM")>=0:
431                 incident_shock_data = True
432             elif thermo_props_found or incident_shock_data:
433                 tokens = line.split()
434                 # Fill out thermo properties
435                 if line.find("H, KJ/KG")>=0:
436                     self.h = get_cea2_float(tokens[2:]) * 1.0e3
```

```
437                 elif line.find("U, KJ/KG")>=0:
438                     self.e = get_cea2_float(tokens[2:]) * 1.0e3
439                 elif line.find("S, KJ/(KG)(K)")>=0:
440                     self.s = get_cea2_float(tokens[2:]) * 1.0e3
441                 elif line.find("Cp, KJ/(KG)(K)")>=0:
442                     self.cp = get_cea2_float(tokens[2:]) * 1.0e3
443                     self.C_p = self.cp
444                 elif line.find("GAMMAs")>=0:
445                     self.gam = get_cea2_float(tokens[1:])
446                 elif line.find("M, (1/n)")>=0:
447                     self.Mmass = get_cea2_float(tokens[2:])
448                 elif line.find("SON VEL,M/SEC")>=0:
449                     self.a = get_cea2_float(tokens[2:])
450                 elif line.find("P, BAR")>=0:
451                     self.p = get_cea2_float(tokens[2:]) * 1.0e5
452                     # print "p = ", self.p
453                 elif line.find("T, K")>=0:
454                     self.T = get_cea2_float(tokens[2:])
455                     # print "T = ", self.T
456                 elif line.find("RHO, KG/CU M")>=0:
457                     self.rho = get_cea2_float(tokens[3:])
458                     # print "rho = ", self.rho
459                 # Fill out transport properties if requested
460                 if transProps:
461                     if line.find("VISC,MILLIPOISE")>=0:
462                         self.mu = get_cea2_float(tokens[1:]) * 1.0e-4
463                         # print "mu = ", self.mu
464                     elif conductivity_found==False and line.find("CONDUCTIVITY")>=0 and len(tokens)==2:
465                         self.k = get_cea2_float(tokens[1:]) * 1.0e-1
466                         # print "k = ", self.k
467                         # want to use the first conductivity value (for equilibrium reaction)
468                         conductivity_found = True
469                 else:
470                     self.mu = 0.0
471                     self.k = 0.0
472                 # Get the shock specific parameters if appropriate
473                 if incident_shock_data:
474                     if line.find("U2, M/SEC")>=0:
475                         self.u2 = get_cea2_float(tokens[2:])
476         # Calculate remaining thermo properties
```

```
477         self.R = R_universal / self.Mmass  # gas constant, J/kg.K
478         self.C_v = self.C_p - self.R       # specific heat, const volume
479         # Check for small or zero pressure value printed by CEA2;
480         # it may have underflowed when printed in bars.
481         if self.p < 1000.0:
482             self.p = self.rho * self.R * self.T
483         #
484         # Scan lines again, this time looking for species fractions.
485         species_fractions_found = False
486         # Re-initialise the species list/fractions so that we ensure that there is
487         # no 'left-over' information from last time
488         self.species = {}
489         for line in lines:
490             line = line.strip()
491             if len(line) == 0: continue
492             if line.find('MOLE FRACTIONS') >= 0:
493                 species_fractions_found = True
494                 continue
495             if line.find('MASS FRACTIONS') >= 0:
496                 species_fractions_found = True
497                 continue
498             if line.find('* THERMODYNAMIC PROPERTIES FITTED') >= 0: break
499             if species_fractions_found:
500                 tokens = line.split()
501                 s = tokens[0].replace('*', '')
502                 self.species[s] = get_cea2_float(tokens[1:])
503                 # print "%s = %e" % (s, self.species[s])
504         # Now check for any None values, where CEA2 wrote a dodgy format float.
505         dodgyCount = 0
506         sumFractions = 0.0
507         for s in self.species.keys():
508             if self.species[s] == None:
509                 dodgyCount += 1
510                 dodgySpecies = s
511             else:
512                 sumFractions += self.species[s]
513         if dodgyCount > 1:
514             print "Cannot evaluate species fractions"
515             print "because there are too many dodgy values"
516             sys.exit(-1)
```

```
517         # but we can recover one missing value.
518         if dodgyCount == 1:
519             self.species[dodgySpecies] = 1.0 - sumFractions
520         return
521 
522     def EOS(self, problemType='pT', transProps=True):
523         """
524         Computes the gas state, taking into account the high-temperature effects.
525 
526         It does this by writing a suitable input file for the CEA code,
527         calling that code and then extracting the relevant results from
528         the CEA output or plot file.
529 
530         :param self: the gas state to be filled in
531         :param problemType: a string specifying the type of CEA analysis:
532             'pT', 'rhoT', 'rhoe', 'ps', shock
533         :param transProps: a boolean flag:
534             False=don't request transport props, True=request viscosity and thermal-conductivity
535         :returns: None, but does update the contents of the gas state as a side-effect.
536         """
537         # Make sure that the database input files are in the working dir
538         if not os.path.exists('thermo.inp'):
539             print 'Copying thermo.inp to the current working directory'
540             os.system("cp %s/e3bin/thermo.inp ." % ( os.getenv("HOME") ) )
541             print 'Copying trans.inp to the current working directory'
542             os.system("cp %s/e3bin/trans.inp ." % ( os.getenv("HOME") ) )
543         # Make sure that binary versions of the database files exist.
544         if not os.path.exists('thermo.lib'):
545             print 'Make the binary database for thermodynamic properties'
546             run_cea_program('thermo',checkTableHeader=False)
547             print 'Make the binary database for transport properties'
548             run_cea_program('trans',checkTableHeader=False)
549         # Now, run the cea program on the actual job.
550         self.write_cea2_input_file(problemType, transProps)
551         run_cea_program('tmp')
552         self.scan_cea2_dot_out_file(transProps)
553         self.have_run_cea = True
554         return
555 
556     def shock_process(self, Us):
```

```
557         """
558         Compute the gas state after being processed by an incident shock.
559 
560         :param Us: shock speed into quiescent gas, m/s
561         :returns: a reference to the post-shock gas state (self)
562 
563         .. This recovers (approximately) Dan's original Shock function.
564         """
565         self.Us = Us
566         self.EOS(problemType='shock', transProps=True)
567         return self
568 
569 # --------------------------------------------------------------
570 
571 def make_gas_from_name(gasName, outputUnits='massf'):
572     """
573     Manufacture a Gas object from a small library of options.
574 
575     :param gasName: one of the names for the special cases set out below
576     :returns: a Gas object
577     """
578     if gasName.lower() == 'air':
579         return Gas({'Air':1.0,}, outputUnits=outputUnits, trace=1.0e-4)
580     elif gasName.lower() == 'air-ions':
581         return Gas({'Air':1.0,}, outputUnits=outputUnits, trace=1.0e-4,
582                    with_ions=True)
583     elif gasName.lower() == 'air5species':
584         return Gas(reactants={'N2':0.79, 'O2':0.21}, inputUnits='moles',
585                    onlyList=['N2','O2','N','O','NO'],
586                    outputUnits=outputUnits)
587     elif gasName.lower() == 'air7species':
588         return Gas(reactants={'N2':0.79, 'O2':0.21}, inputUnits='moles',
589                    onlyList=['N2','O2','N','O','NO','NO+','e-'],
590                    outputUnits=outputUnits, with_ions=True)
591     elif gasName.lower() == 'air11species':
592         return Gas(reactants={'N2':0.79, 'O2':0.21}, inputUnits='moles',
593                    onlyList=['N2','O2','N','O','NO','N+','O+','N2+','O2+','NO+','e-'],
594                    outputUnits=outputUnits, with_ions=True, trace=1.0e-30)
595     elif gasName.lower() == 'air13species':
596         return Gas(reactants={'N2':0.7811, 'O2':0.2095, 'Ar':0.0093}, inputUnits='moles',
```

```
597                     onlyList=['N2','O2','Ar','N','O','NO','Ar+','N+','O+','N2+','O2+','NO+','e-'],
598                     outputUnits=outputUnits, with_ions=True, trace=1.0e-30)
599     elif gasName.lower() == 'n2':
600         return Gas(reactants={'N2':1.0, 'N':0.0}, onlyList=['N2','N'],
601                     outputUnits=outputUnits)
602     elif gasName.lower() == 'n2-ions':
603         return Gas(reactants={'N2':1.0, 'N':0.0},
604                     onlyList=['N2','N','N2+','N+','e-'],
605                     outputUnits=outputUnits, with_ions=True)
606     elif gasName.lower() == 'co2':
607         return Gas(reactants={'CO2':1.0},
608                     onlyList=['CO2','C2','C','CO','O2','O'],
609                     outputUnits=outputUnits)
610     elif gasName.lower() == 'co2-ions':
611         return Gas(reactants={'CO2':1.0},
612                     onlyList=['CO2','C2','C','CO','O2','O','C+','CO+','O2+','O+','e-'],
613                     outputUnits=outputUnits, with_ions=True)
614     elif gasName.lower() == 'mars-basic':
615         return Gas(reactants={'CO2':0.97,'N2':0.03}, inputUnits='massf',
616                     onlyList=['C','C2','CN','CO','CO2','N','N2','NO','O','O2'],
617                     outputUnits=outputUnits)
618     elif gasName.lower() == 'mars-trace':
619         return Gas(reactants={'CO2':0.9668,'N2':0.0174,'O2':0.0011,'Ar':0.0147}, inputUnits='massf',
620                     onlyList=['C','C2','CN','CO','CO2','N','N2','NO','O','O2','Ar'],
621                     outputUnits=outputUnits)
622     elif gasName.lower() == 'mars-trace-ions':
623         return Gas(reactants={'CO2':0.9668,'N2':0.0174,'O2':0.0011,'Ar':0.0147}, inputUnits='massf',
624                     onlyList=['C','C2','CN','CO','CO2','N','N2','NO','O','O2','Ar',
625                               'C+','CO+','NO+','O+','O2+','e-'],
626                     outputUnits=outputUnits, with_ions=True)
627     elif gasName.lower() == 'h2ne':
628         return Gas(reactants={'H2':0.85, 'Ne':0.15}, inputUnits='moles',
629                     onlyList=['H2','H','Ne'],
630                     outputUnits=outputUnits)
631     elif gasName.lower() == 'h2ne-ions':
632         return Gas(reactants={'H2':0.85, 'Ne':0.15}, inputUnits='moles',
633                     onlyList=['H2','H','Ne','H+','e-'],
634                     outputUnits=outputUnits, with_ions=True)
635     elif gasName.lower() == 'jupiter-like':
636         return Gas(reactants={'H2':0.15, 'Ne':0.85}, inputUnits='moles',
```

```
                        onlyList=['H2','H','Ne','H+','e-'],
                        outputUnits=outputUnits, with_ions=True)
    elif gasName.lower() == 'titan-like':
        return Gas(reactants={'N2':0.95,'CH4':0.05}, inputUnits='moles',
                        onlyList=['N2','CH4','CH3','CH2','CH','C2','H2','CN','NH','HCN','N','C','H'],
                        outputUnits=outputUnits, with_ions=False)
    elif gasName.lower() == 'titan-like-ions':
        return Gas(reactants={'N2':0.95,'CH4':0.05}, inputUnits='moles',
                        onlyList=['N2','CH4','CH3','CH2','CH','C2','H2','CN','NH','HCN','N','C','H',
                                  'N2+','CN+','N+','C+','H+','e-'],
                        outputUnits=outputUnits, with_ions=True)
    elif gasName.lower() == 'ar':
        return Gas(reactants={'Ar':1.0, 'Ar+':0.0, 'e_minus':0.0},
                        inputUnits='moles', outputUnits=outputUnits,
                        with_ions=True, trace=1.0e-16)
    elif gasName.lower() == 'kr':
        return Gas(reactants={'Kr':1.0, 'Kr+':0.0, 'e_minus':0.0},
                        inputUnits='moles', outputUnits=outputUnits,
                        with_ions=True, trace=1.0e-16)
    else:
        raise Exception, 'make_gas_from_name(): unknown gasName: %s' % gasName

def list_gas_names():
    """
    :returns: the list of gases available in make_gas_from_name()
    """
    return ['air', 'air-ions', 'air5species', 'air7species', 'air11species',
            'air13species', 'n2', 'n2-ions', 'co2', 'co2-ions', 'mars-trace', 'mars-basic',
            'h2ne', 'h2ne-ions', 'jupiter-like', 'titan-like', 'titan-like-ions', 'ar', 'kr']

def make_reactants_dictionary( species_list ):
    """
    Creates the CEA reactants dictionary from a list of species
    in the lib/gas format
    :param species_list: lib/gas species list
    """
    nsp = len(species_list)
    reactants = dict()
    for sp in species_list:
        # replace names containing '_plus' with '+'
```

```
        sp = sp.replace("_plus","+")
        # replace names containing '_minus' with '-'
        sp = sp.replace("_minus","-")
        reactants.setdefault(sp,0.0)
    return reactants

def get_species_composition( sp, species_data ):
    """
    Creates a list of mass or mole fractions for a species
    in lib/gas form from the CEA species_data dictionary
    :param sp: a single lib/gas species
    :param species_data: the CEA species_data dictionary
    """
    # replace names containing '_plus' with '+'
    if ( sp.find("_plus")>=0 ): sp = sp[0:sp.find("_plus")] + "+"
    # replace names containing '_minus' with '-'
    if ( sp.find("_minus")>=0 ): sp = sp[0:sp.find("_minus")] + "-"
    if sp in species_data.keys():
            return species_data[sp]
    else:
            return 0.0

def get_with_ions_flag( species_list ):
    """
    Determines the 'with_ions' flag from a list of species
    in the lib/gas format
    :param species_list: lib/gas species list
    """
    for sp in species_list:
        if sp.find("_plus")>=0: return True
        if sp.find("_minus")>=0: return True
    return False

# --------------------------------------------------------------

if __name__ == '__main__':
    print 'Test/demonstrate the Gas class...'
    #
    print '\nDefault constructor with Air as the test gas.'
    a = Gas({'Air':1.0,}, outputUnits='moles')
```

```
717    a.set_pT(100.0e3, 300.0)
718    a.write_state(sys.stdout)
719    print 'and the same Air at a higher temperature'
720    a.set_pT(100.0e3, 4000.0)
721    a.write_state(sys.stdout)
722    #
723    print '\nCheck enthalpy specification'
724    b = make_gas_from_name('air', outputUnits='moles')
725    b.set_ph(a.p, a.h)
726    b.write_state(sys.stdout)
727    #
728    print '\nCheck internal-energy specification'
729    b = make_gas_from_name('air', outputUnits='moles')
730    b.set_rhoe(a.rho, a.e)
731    b.write_state(sys.stdout)
732    #
733    print '\nAir-5-species for nenzfr: 79% N2, 21% O2 by mole fraction.'
734    a = Gas(reactants={'N2':0.79, 'O2':0.21, 'N':0.0, 'O':0.0, 'NO':0.0},
735            inputUnits='moles', outputUnits='massf',
736            onlyList=['N2','O2','N','O','NO'])
737    a.set_pT(100.0e3, 300.0)
738    a.write_state(sys.stdout)
739    print 'and isentropically compress to a higher pressure'
740    a.set_ps(10.0e6, a.s)
741    a.write_state(sys.stdout)
742    #
743    print '\nTry an odd mix of Helium, N2 and N'
744    b = Gas({'N2':1.0, 'N':0.0, 'He':0.0})
745    b.set_pT(100.0e3, 300.0)
746    b.write_state(sys.stdout)
747    print 'and the same initial mix and volume at a higher temperature'
748    b.set_rhoT(b.rho, 5000.0)
749    b.write_state(sys.stdout)
750    #
751    print '\nStart again with low-T air as the test gas'
752    a = Gas({'Air':1.0,}); a.set_pT(100.0e3, 300.0)
753    a.write_state(sys.stdout)
754    print 'clone it, changing species-fraction units'
755    c = a.clone(newOutputUnits='moles')
756    c.write_state(sys.stdout)
```

```
757     print 'and shock process it'
758     c.shock_process(4000.0)
759     c.write_state(sys.stdout)
760     #
761     print 'End of test.'
```

B Source code for flow process calculations

B.1 ideal_gas_flow.py

Basic flow relations for an ideal gas.

```
"""
ideal_gas_flow.py: One-dimensional steady flow of an ideal gas.

.. Author:
   PA Jacobs
   Centre for Hypersonics, School of Engineering
   The University of Queensland

.. Versions:
   1.1 30-Sep-94: Xplore version
   2.0 16-May-04: Python equivalent adapted from the Xplore version.
   27-Feb-2012: use relative import in cfpylib

Contents:

* One-dimensional flows:

   * Isentropic flow relations.
     State zero (0) refers to the stagnation condition.
     State star is the sonic (throat) condition.
   * 1D (Normal) Shock Relations
     State 1 is before the shock and state 2 after the shock.
     Velocities are in a shock-stationary frame.
   * 1-D flow with heat addition (Rayleigh-line)
     State star is the (hypothetical) sonic condition.

* Two-dimensional flows:

   * Prandtl-Meyer functions
   * Oblique-shock relations
   * Taylor-Maccoll conical flow
"""

from math import *
```

```
35 import numpy
36 from ..nm.secant_method import solve
37 from ..nm.zero_solvers import secant
38 
39 # ---------------------------------------------------------------
40 # Isentropic flow
41 
42 def A_Astar(M, g=1.4):
43     """
44     Area ratio A/Astar for an isentropic, quasi-one-dimensional flow.
45 
46     :param M: Mach number at area A
47     :param g: ratio of specific heats
48     :returns: A/Astar
49     """
50     t1 = (g + 1.0) / (g - 1.0)
51     m2 = M**2
52     t2 = 1.0 / m2 * (2.0 / (g + 1.0) * (1.0 + (g - 1.0) * 0.5 * m2))**t1
53     t2 = sqrt(t2)
54     return t2
55 
56 def T0_T(M, g=1.4):
57     """
58     Total to static temperature ratio for an adiabatic flow.
59 
60     :param M: Mach number
61     :param g: ratio of specific heats
62     :returns: T0/T
63     """
64     return 1.0 + (g - 1.0) * 0.5 * M**2
65 
66 def p0_p(M, g=1.4):
67     """
68     Total to static pressure ratio for an isentropic flow.
69 
70     :param M: Mach number
71     :param g: ratio of specific heats
72     :returns: p0/p
73     """
74     return (T0_T(M, g))**( g / (g - 1.0) )
```

```
def r0_r(M, g=1.4):
    """
    Stagnation to free-stream density ratio for an isentropic flow.

    :param M: Mach number
    :param g: ratio of specific heats
    :returns: r0/r
    """
    return (T0_T(M, g))**(1.0 / (g - 1.0))

# ---------------------------------------------------------------
# 1-D normal shock relations.

def m2_shock(M1, g=1.4):
    """
    Mach number M2 after a normal shock.

    :param M1: Mach number of incoming flow
    :param g: ratio of specific heats
    :returns: M2
    """
    numer = 1.0 + (g - 1.0) * 0.5 * M1**2
    denom = g * M1**2 - (g - 1.0) * 0.5
    return sqrt(numer / denom)

def r2_r1(M1, g=1.4):
    """
    Density ratio r2/r1 across a normal shock.

    :param M1: Mach number of incoming flow
    :param g: ratio of specific heats
    :returns: r2/r1
    """
    numer = (g + 1.0) * M1**2
    denom = 2.0 + (g - 1.0) *M1**2
    return numer / denom

def u2_u1(M1, g=1.4):
    """
```

```
115     Velocity ratio u2/u1 across a normal shock.
116
117     :param M1: Mach number of incoming flow
118     :param g: ratio of specific heats
119     :returns: u2/u1
120     """
121     return 1 / r2_r1(M1, g)
122
123 def p2_p1(M1, g=1.4):
124     """
125     Static pressure ratio p2/p1 across a normal shock.
126
127     :param M1: Mach number of incoming flow
128     :param g: ratio of specific heats
129     :returns: p2/p1
130     """
131     return 1.0 + 2.0 * g / (g + 1.0) * (M1**2 - 1.0)
132
133 def T2_T1(M1, g=1.4):
134     """
135     Static temperature ratio T2/T1 across a normal shock.
136
137     :param M1: Mach number of incoming flow
138     :param g: ratio of specific heats
139     :returns: T2/T1
140     """
141     return  p2_p1(M1, g) / r2_r1(M1, g)
142
143 def p02_p01(M1, g=1.4):
144     """
145     Stagnation pressure ratio p02/p01 across a normal shock.
146
147     :param M1: Mach number of incoming flow
148     :param g: ratio of specific heats
149     :returns: p02/p01
150     """
151     t1 = (g + 1.0) / (2.0 * g * M1**2 - (g - 1.0))
152     t2 = (g + 1.0) * M1**2 / (2.0 + (g - 1.0) * M1**2)
153     return t1**(1.0/(g-1.0)) * t2**(g/(g-1.0))
154
```

```
155 def DS_Cv(M1, g=1.4):
156     """
157     Nodimensional entropy change ds across a normal shock.
158 
159     :param M1: Mach number of incoming flow
160     :param g: ratio of specific heats Cp/Cv
161     :returns: ds/Cv
162     """
163     t1 = p2_p1(M1, g)
164     t2 = r2_r1(M1, g)
165     return log(t1 * t2**g)
166 
167 def pitot_p(p1, M1, g=1.4):
168     """
169     Pitot pressure for a specified Mach number free-stream flow.
170 
171     Will shock the gas if required.
172 
173     :param M1: Mach number of incoming flow
174     :param g: ratio of specific heats
175     :returns: Pitot pressure (absolute)
176     """
177     if M1 > 1.0:
178         p2 = p2_p1(M1,g)*p1
179         M2 = m2_shock(M1, g)
180         return p0_p(M2, g)*p2
181     else:
182         return p0_p(M1, g)*p1
183 
184 
185 # ---------------------------------------------------------------
186 # 1-D flow with heat addition (Rayleigh-line)
187 
188 def T0_T0star(M, g=1.4):
189     """
190     Total temperature ratio for flow with heat addition.
191 
192     :param M: initial Mach number
193     :param g: ratio of specific heats
194     :returns: T0/T0star where T0 is the total temperature of the initial flow
```

```
195                and T0star is the total temperature that would be achieved
196                if enough heat is added to get to sonic conditions.
197     """
198     term1 = (g + 1.0) * M**2
199     term2 = (1.0 + g * M**2)**2
200     term3 = 2.0 + (g - 1.0) * M**2
201     return term1 / term2 * term3
202
203 def M_Rayleigh(T0T0star, g=1.4):
204     """
205     Computes M from Total Temperature ratio for Rayleigh-line flow.
206
207     :param T0T0star: total temperature ratio (star indicating sonic conditions)
208     :param g: ratio of specific heats
209     :returns: initial Mach number of flow
210
211     Note that supersonic flow is assumed for the initial guess.
212     """
213     def f_to_solve(m): return T0_T0star(m, g) - T0T0star
214     return solve(f_to_solve, 2.5, 2.4)
215
216 def T_Tstar(M, g=1.4):
217     """
218     Static temperature ratio T/Tstar for Rayleigh-line flow.
219
220     :param M: initial Mach number
221     :param g: ratio of specific heats
222     :returns: T/Tstar where T is the static temperature of the initial flow
223                and Tstar is the static temperature that would be achieved
224                if enough heat is added to get to sonic conditions.
225     """
226     return M**2 * ( (1.0 + g) / (1.0 + g * M**2) )**2
227
228 def p_pstar(M, g=1.4):
229     """
230     Static pressure ratio p/pstar for Rayleigh-line flow.
231
232     :param M: initial Mach number
233     :param g: ratio of specific heats
234     :returns: p/pstar where p is the static pressure of the initial flow
```

```
235                 and pstar is the static pressure that would be achieved
236                 if enough heat is added to get to sonic conditions.
237     """
238     return (1.0 + g) / (1.0 + g * M**2)
239
240 def r_rstar(M, g=1.4):
241     """
242     Density ratio r/rstar for Rayleigh-line flow.
243
244     :param M: initial Mach number
245     :param g: ratio of specific heats
246     :returns: r/rstar where r is the density of the initial flow
247               and rstar is the density that would be achieved
248               if enough heat is added to get to sonic conditions.
249     """
250     return 1.0 / M**2 / (1.0 + g) * (1.0 + g * M**2)
251
252 def p0_p0star(M, g=1.4):
253     """
254     Stagnation pressure ratio p0/p0star for Rayleigh-line flow.
255
256     :param M: initial Mach number
257     :param g: ratio of specific heats
258     :returns: p0/p0star where p0 is the total pressure of the initial flow
259               and p0star is the total pressure that would be achieved
260               if enough heat is added to get to sonic conditions.
261     """
262     term1 = (2.0 + (g - 1.0) * M**2) / (g + 1.0)
263     term2 = g / (g - 1.0)
264     return (1.0 + g) / (1.0 + g * M**2) * term1**term2
265
266 # ------------------------------------------------------------------
267 # Prandtl-Meyer functions
268
269 def deg_to_rad(d): return d / 180.0 * pi
270 def rad_to_deg(r): return r * 180.0 / pi
271
272 def PM1(M, g=1.4):
273     """
274     Prandtl-Meyer function.
```

```
275
276     :param M: Mach number
277     :param g: ratio of specific heats
278     :returns: Prandtl-Meyer function value (in radians)
279     """
280     if M > 1.0:
281         t1 = M**2 - 1.0
282         t2 = sqrt((g - 1.0) / (g + 1.0) * t1)
283         t3 = sqrt(t1)
284         t4 = sqrt((g + 1.0) / (g - 1.0))
285         nu = t4 * atan(t2) - atan(t3)
286     else:
287         nu = 0.0
288     return nu
289
290 def PM2(nu, g=1.4):
291     """
292     Inverse Prandtl-Meyer function.
293
294     :param nu: Prandtl-Meyer function value (in radians)
295     :param g: ratio of specific heats
296     :returns: Mach number
297
298     Solves the equation PM1(m, g) - nu = 0, assuming supersonic flow.
299     """
300     def f_to_solve(m): return PM1(m, g) - nu
301     return solve(f_to_solve, 2.0, 2.1)
302
303 # -----------------------------------------------------------------
304 # Oblique shock relations
305 # beta is shock angle wrt on-coming stream direction (in radians)
306 # theta is flow deflection wrt on-coming stream (in radians)
307
308 def beta_obl(M1, theta, g=1.4):
309     """
310     Oblique shock wave angle.
311
312     :param M1: upstream Mach number
313     :param theta: flow deflection angle (radians)
314     :returns: shock angle with respect to initial flow direction (radians)
```

```
315     """
316     b1 = asin(1.0/M1); b2 = b1 * 1.05
317     def f_to_solve(beta): return theta_obl(M1, beta, g) - theta
318     return solve(f_to_solve, b1, b2)
319 
320 def theta_obl(M1, beta, g=1.4):
321     """
322     Compute the deflection angle given the shock wave angle.
323 
324     :param M1: upstream Mach number
325     :param beta: shock angle with respect to initial flow direction (radians)
326     :returns: theta, flow deflection angle (radians)
327     """
328     m1sb = M1 * sin(beta)
329     t1 = 2.0 / tan(beta) * (m1sb**2 - 1.0)
330     t2 = M1**2 * (g + cos(2.0 * beta)) + 2.0
331     theta = atan(t1/t2)
332     return theta
333 
334 def M2_obl(M1, beta, theta, g=1.4):
335     """
336     Mach number after an oblique shock.
337 
338     :param M1: upstream Mach number
339     :param beta: shock angle with respect to initial flow direction (radians)
340     :returns: M2, Mach number in flow after the shock
341     """
342     m1sb = M1 * sin(beta)
343     numer = 1.0 + (g - 1.0) * 0.5 * m1sb**2
344     denom = g * m1sb**2 - (g - 1.0) * 0.5
345     m2 = sqrt(numer / denom / (sin(beta - theta))**2 )
346     return m2
347 
348 def r2_r1_obl(M1, beta, g=1.4):
349     """
350     Density ratio r2/r1 across an oblique shock.
351 
352     :param M1: upstream Mach number
353     :param beta: shock angle with respect to initial flow direction (radians)
354     :returns: r2/r1
```

```
355     """
356     m1sb = M1 * sin(beta)
357     numer = (g + 1.0) * m1sb**2
358     denom = 2.0 + (g - 1.0) * m1sb**2
359     return numer / denom
360
361 def u2_u1_obl(M1, beta, g=1.4):
362     """
363     Flow-speed ratio u2/u1 across an oblique shock.
364
365     :param M1: upstream Mach number
366     :param beta: shock angle with respect to initial flow direction (radians)
367     :returns: u2/u1
368     """
369     return sqrt((sin(beta) / r2_r1_obl(M1, beta, g))**2 + (cos(beta))**2)
370
371 def p2_p1_obl(M1, beta, g=1.4):
372     """
373     Static pressure ratio p2/p1 across an oblique shock.
374
375     :param M1: upstream Mach number
376     :param beta: shock angle with respect to initial flow direction (radians)
377     :returns: p2/p1
378     """
379     m1sb = M1 * sin(beta)
380     return 1.0 + 2.0 * g / (g + 1.0) * (m1sb**2 - 1.0)
381
382 def T2_T1_obl(M1, beta, g=1.4):
383     """
384     Static temperature ratio T2/T1 across an oblique shock.
385
386     :param M1: upstream Mach number
387     :param beta: shock angle with respect to initial flow direction (radians)
388     :returns: T2/T1
389     """
390     return p2_p1_obl(M1, beta, g) / r2_r1_obl(M1, beta, g)
391
392 def p02_p01_obl(M1, beta, g=1.4):
393     """
394     Ratio of stagnation pressures p02/p01 across an oblique shock.
```

```
395
396     :param M1: upstream Mach number
397     :param beta: shock angle with respect to initial flow direction (radians)
398     :returns: p02/p01
399     """
400     m1sb = M1 * sin(beta)
401     t1 = (g + 1.0) / (2.0 * g * m1sb**2 - (g - 1.0))
402     t2 = (g + 1.0) * m1sb**2 / (2.0 + (g - 1.0) * m1sb**2)
403     return t1**(1.0/(g-1.0)) * t2**(g/(g-1.0))
404
405 #------------------------------------------------------------------------
406 # Taylor-Maccoll cone flow.
407
408 def taylor_maccoll_odes(z, theta, g=1.4):
409     """
410     The ODEs from the Taylor-Maccoll formulation.
411
412     See PJ's workbook for Feb 2012 for details.
413     We've packaged them formally so that we might one day use
414     a more sophisticated ODE integrator requiring fewer steps.
415     """
416     rho, V_r, V_theta, h, p = z
417     # Assemble linear system for determining the derivatives wrt theta.
418     A = numpy.zeros((5,5), float)
419     b = numpy.zeros((5,), float)
420     A[0,0] = V_theta; A[0,2] = rho; b[0] = -2.0*rho*V_r - rho*V_theta/tan(theta)
421     A[1,1] = 1.0; b[1] = V_theta
422     A[2,1] = rho*V_r; A[2,2] = rho*V_theta; A[2,4] = 1.0
423     A[3,1] = V_r; A[3,2] = V_theta; A[3,3] = 1.0
424     A[4,0] = h*(g-1)/g; A[4,3] = rho*(g-1)/g; A[4,4] = -1.0
425     dzdtheta = numpy.linalg.solve(A,b)
426     return dzdtheta
427
428 def theta_cone(V1, p1, T1, beta, R=287.1, g=1.4):
429     """
430     Compute the cone-surface angle and conditions given the shock wave angle.
431
432     :param V1: speed of gas into shock
433     :param p1: free-stream pressure
434     :param T1: free-stream static temperature
```

```
435     :param beta: shock wave angle wrt stream direction (in radians)
436     :param R: gas constant
437     :param g: ratio of specific heats
438     :returns: tuple of theta_c, V_c, p_c, T_c:
439         theta_c is stream deflection angle in radians
440         V_c is the cone-surface speed of gas in m/s
441         p_c is the cone-surface pressure
442         T_c is the cone-surface static temperature
443 
444     The computation starts with the oblique-shock jump and then integrates
445     across theta until V_theta goes through zero.
446     The cone surface corresponds to V_theta == 0.
447 
448     .. Versions: This ideal-gas version adapted from the cea2_gas_flow version, 08-Mar-2012.
449        24-Jun-2012 : RJG added checks to catch the limiting case when beta < mu
450                    : and a linear interpolation when beta is only slightly larger
451                    : than mu (1% larger)
452     """
453     # When beta is only this fraction larger than mu,
454     # we'll apply a linear interpolation
455     LINEAR_INTERP_SWITCH = 1.01
456     # Free-stream properties and gas model.
457     a1 = sqrt(g*R*T1)
458     M1 = V1 / a1
459     C_p = R * g / (g-1)
460     h1 = C_p * T1
461     rho1 = p1 / (R * T1)
462     # Test beta in relation to the Mach angle, mu
463     mu = asin(1.0/M1)
464     beta2 = LINEAR_INTERP_SWITCH*mu
465     #print "beta= ", beta, "mu= ", mu, " beta2= ", beta2
466     if beta <= mu:
467         # An infinitely weak shock angle
468         return 0.0, V1, p1, T1
469     if beta < beta2:
470         # It is difficult to integrate between the shock and cone body
471         # when the shock angle is only slightly larger than the Mach
472         # angle. In this instance, find the value at LINEAR_INTER_SWITCH*mu
473         # and linearly interpolate to find the value at beta
474         (theta2, V2, p2, T2) = theta_cone(V1, p1, T1, beta2, R, g)
```

```
475        frac = (beta - mu)/(beta2 - mu)
476        theta_c = frac*theta2
477        V = (1.0 - frac)*V1 + frac*V2
478        p = (1.0 - frac)*p1 + frac*p2
479        T = (1.0 - frac)*T1 + frac*T2
480        return theta_c, V, p, T
481    #
482    # Start at the point just downstream the oblique shock.
483    theta_s = theta_obl(M1, beta, g)
484    M2 = M2_obl(M1, beta, theta_s, g)
485    assert M2 > 1.0
486    rho2 = rho1 * r2_r1_obl(M1, beta, g)
487    V2 = V1 * u2_u1_obl(M1, beta, g)
488    p2 = p1 * p2_p1_obl(M1, beta, g)
489    T2 = T1 * T2_T1_obl(M1, beta, g)
490    h2 = T2 * C_p
491    #
492    # Initial conditions for Taylor-Maccoll integration.
493    dtheta = -0.05 * pi / 180.0  # fraction-of-a-degree steps
494    theta = beta
495    V_r = V2 * cos(beta - theta_s)
496    V_theta = -V2 * sin(beta - theta_s)
497    # For integrating across the shock layer, the state vector is:
498    z = numpy.array([rho2, V_r, V_theta, h2, p2])
499    while V_theta < 0.0:
500        # Keep a copy for linear interpolation at the end.
501        z_old = z.copy(); theta_old = theta
502        # Do the update using a low-order method (Euler) for the moment.
503        dzdtheta = taylor_maccoll_odes(z, theta, g)
504        z += dtheta * dzdtheta; theta += dtheta
505        rho, V_r, V_theta, h, p = z
506        if False: print "DEBUG theta=", theta, "V_r=", V_r, "V_theta=", V_theta
507    # At this point, V_theta should have crossed zero so
508    # we can linearly-interpolate the cone-surface conditions.
509    V_theta_old = z_old[2]
510    frac = (0.0 - V_theta_old)/(V_theta - V_theta_old)
511    z_c = z_old*(1.0-frac) + z*frac
512    theta_c = theta_old*(1.0-frac) + theta*frac
513    # At the cone surface...
514    rho, V_r, V_theta, h, p = z_c
```

```
515     T = h / C_p
516     assert abs(V_theta) < 1.0e-6
517     #
518     return theta_c, V_r, p, T
519 
520 def beta_cone(V1, p1, T1, theta, R=287.1, g=1.4):
521     """
522     Compute the conical shock wave angle given the cone-surface deflection angle.
523 
524     :param V1: speed of gas into shock
525     :param p1: free-stream pressure
526     :param T1: free-stream static temperature
527     :param theta: stream deflection angle (in radians)
528     :param R: gas constant
529     :param g: ratio of specific heats
530     :returns: shock wave angle wrt incoming stream direction (in radians)
531 
532     .. This ideal-gas version adapted from the cea2_gas_flow version, 08-Mar-2012.
533     """
534     # Free-stream properties and gas model.
535     a1 = sqrt(g*R*T1)
536     M1 = V1 / a1
537     C_p = R * g / (g-1)
538     h1 = C_p * T1
539     rho1 = p1 / (R * T1)
540     # Initial guess
541     M1 = V1 / a1
542     b1 = asin(1.0 / M1) * 1.01 # to be stronger than a Mach wave
543     b2 = b1 * 1.05
544     def error_in_theta(beta_guess):
545         theta_guess, V_c, p_c, T_c = theta_cone(V1, p1, T1, beta_guess, R, g)
546         return theta_guess - theta
547     return secant(error_in_theta, b1, b2, tol=1.0e-4, limits=[asin(1.0/M1), pi/2.0])
548 
549 def beta_cone2(M1, theta, R=287.1, g=1.4):
550     """
551     Compute the conical shock wave angle given the cone-surface deflection angle and free stream Mach number
        .
552 
553     :param M1: free stream Mach number
```

```
554     :param theta: stream deflection angle (in radians)
555     :param R: gas constant
556     :param g: ratio of specific heats
557     :returns: shock wave angle wrt incoming stream direction (in radians)
558 
559     .. This version basically delegates work to beta_cone().
560     """
561     # Compute free stream velocity assuming unit value temperature
562     T1 = 1.0
563     a1 = sqrt(g*R*T1)
564     V1 = M1*a1
565     # Set free stream pressure to unit value
566     p1 = 1.0
567     # Now ready to call beta_cone()
568     return beta_cone(V1, p1, T1, theta, R, g)
569 
570 # ------------------------------------------------------------------
571 
572 def demo():
573     print "Begin test of isentropic flow ratios..."
574     M = 2.0
575     print "Computed: M=%g: A/Astar=%g, T0/T=%g, p0/p=%g, r0/r=%g" % \
576           (M, A_Astar(M), T0_T(M), p0_p(M), r0_r(M))
577     print "Expected: M=2, A/Astar=1.687, T0/T=1.80, p0/p=7.824, r0/r=4.347"
578     print ""
579     print "Normal shock jump..."
580     print "Computed: M=%g: M2=%g, T2/T1=%g, p2/p1=%g, r2/r1=%g" % \
581           (M, m2_shock(M), T2_T1(M), p2_p1(M), r2_r1(M))
582     print "Expected: M1=2, M2=0.5774, T2/T1=1.687, p2/p1=4.50, r2/r1=2.667"
583     print ""
584     print "Rayleigh-line flow..."
585     print "Computed: M=%g: T0/Tstar=%g, T/Tstar=%g, p/pstar=%g, r/rstar=%g" % \
586           (M, T0_T0star(M), T_Tstar(M), p_pstar(M), r_rstar(M))
587     print "Expected: M=2, T0/T0star=0.7934, T/Tstar=0.5289, p/pstar=0.3636, r/rstar=0.6875"
588     print "Inverse calculation: T0/T0star=%g --> M=%g" % \
589           (T0_T0star(M), M_Rayleigh(T0_T0star(M)))
590     print ""
591     print "Prandtl-Meyer function..."
592     print "Computed: M=%g --> nu=%g; Inverse: M=%g <-- nu=%g" % \
593           (M, PM1(M), PM2(1.1481), 1.1481)
```

```
    print "Expected: M=2 --> nu=0.4604; Inverse: M=4 <-- nu=1.1481"
    print ""
    print "Oblique shock relations may not quite match (data is from chart)..."
    beta = deg_to_rad(44.0); theta = deg_to_rad(14.0); # from chart, M=2
    print "Computed: M1=%g, theta(beta=%g)=%g, beta(theta=%g)=%g" % \
          (M, beta, theta_obl(M, beta), theta, beta_obl(M, theta))
    print "Conditions behind shock:"
    print "M2=%g, expected 1.482 (from chart, 14 degree deflection)" % \
          M2_obl(M, beta, theta)
    print "Computed: T2/T1=%g, p2/p1=%g, r2/r1=%g" % \
          (T2_T1_obl(M, beta), p2_p1_obl(M, beta), r2_r1_obl(M, beta))
    print "Expected: T2/T1=1.249, p2/p1=2.088, r2/r1=1.673 (approx. normal-shock table M=1.390)"
    print "u2/u1=%g, p02/p01=%g" % \
          (u2_u1_obl(M, beta), p02_p01_obl(M, beta))
    print "Expected: u2/u1=0.8304=sin(B)/sin(B-d)*r1/r2"
    print ""
    M1 = 1.5; p1 = 100.0e3; T1 = 300.0; R = 287.1; g = 1.4; rho1 = p1/(R*T1)
    print "Taylor-Maccoll cone flow demo with M1=%g" % M1
    print "for M1=1.5, beta=49deg, expect theta=20deg from NACA1135."
    a1 = sqrt(1.4*287*T1)
    V1 = M1 * a1
    beta = 49.0 * pi/180
    theta_c, V_c, p_c, T_c = theta_cone(V1, p1, T1, beta)
    print "theta_c(deg)=", theta_c*180.0/pi, "expected 20deg, surface speed V_c=", V_c
    print "surface pressure coefficient=", (p_c - p1)/(0.5*rho1*V1*V1), "expected 0.385"
    print "p_c: %g, T_c: %g" % (p_c, T_c)
    print ""
    print "Conical shock from cone with half-angle 20deg in M1=", M1
    beta = beta_cone(V1, p1, T1, 20.0*pi/180)
    print "sigma(deg)=", beta*180/pi, "expected 49deg"
    print "Repeat above test, but call beta_cone2()"
    beta = beta_cone2(M1, 20.0*pi/180)
    print "sigma(deg)=", beta*180/pi, "expected 49deg"
    #
    print "Done."
    return
```

B.2 gas_flow.py

Basic flow relations for a more general gas.

```
"""
gas_flow.py -- Gas flow calculations using CEA2 or ideal Gas objects.

.. Author:
   PA Jacobs

.. Version:
   26-Feb-2012 : functions moved out of estcj.py to this module.
   02-May-2013" added more expansion_to_throat_calculation function from
        Matt McGilvray's gun tunnel version of nenzfr. -Chris James
"""

import sys, math, numpy
from ..nm.zero_solvers import secant

DEBUG_GAS_FLOW = False

#---------------------------------------------------------------------------
# 1-D flow functions abstracted from estcj.py
# and made a little more generic.

def shock_ideal(state1, Vs, state2):
    """
    Computes post-shock conditions in the shock frame, assuming ideal gas.

    :param state1: pre-shock Gas state
    :param Vs: speed of gas coming into shock
    :param state2: post-shock Gas state
    :returns: the post-shock gas speed, V2 in the shock-reference frame, Vg in the lab frame.
    """
    #
    M1 = Vs / state1.a
    V1 = Vs
    gam = state1.gam
    R = state1.R
    C_v = state1.C_v
    #
```

```
38     state2.rho = state1.rho * (gam + 1.0) * M1 * M1 \
39              / (2.0 + (gam - 1.0) * M1 * M1)
40     state2.p = state1.p * (2.0 * gam * M1 * M1 - (gam - 1.0)) / (gam + 1.0)
41     state2.T = state2.p / (R * state2.rho)
42     state2.e = state2.T * C_v
43     #
44     V2 = state1.rho / state2.rho * V1
45     Vg = V1 - V2
46     state2.a = state1.a * math.sqrt(state2.T / state1.T)
47     #
48     state2.R = state1.R
49     state2.gam = state1.gam
50     state2.C_v = state1.C_v
51     #
52     return (V2, Vg)
53
54
55 def my_limiter(delta, orig, frac=0.5):
56     """
57     Limit the magnitude of delta to no more than a fraction of the original.
58
59     It occasionally happens that the Newton iterations go badly.
60     It is worth trying to take smaller steps in these situations,
61     assuming that the computed direction is still a fair guess.
62     """
63     if delta >= 0.0:
64         sign = 1
65     else:
66         sign = -1
67     abs_delta = min(abs(delta), frac*abs(orig))
68     return sign * abs_delta
69
70
71 def normal_shock(state1, Vs, state2,ideal_gas_guess=None):
72     """
73     Computes post-shock conditions, using high-temperature gas properties
74     and a shock-stationary frame.
75
76     :param state1: pre-shock gas state
77     :param Vs: speed of gas coming into shock
```

```
78      :param state2: post-shock gas state
79      :param ideal_gas_guess: defaulting to None, otherwise a dictionary of the
80          form {'gam':gam,'R':R} thatis used for the ideal guess at the start of
81          the function when Vs is too high and CEA can't deal with the ideal guess
82          for state 2
83      :returns: the post-shock gas speed, V2 in the shock-reference frame, Vg in the lab frame.
84      """
85      #
86      # Initial guess via ideal gas relations.
87      #
88      if ideal_gas_guess: #if we're worried the ideal gas guess will not work,
89                          #store the original state and use our own guess gam and R for now
90          original_state1 = state1.clone()
91          state1.gam = ideal_gas_guess['gam']
92          state1.R = ideal_gas_guess['R']
93      (V2,Vg) = shock_ideal(state1, Vs, state2)
94      if DEBUG_GAS_FLOW:
95          print 'normal_shock(): post-shock condition assuming ideal gas'
96          state2.write_state(sys.stdout)
97          print '    V2: %g m/s, Vg: %g m/s' % (V2,Vg)
98      #
99      # We assume that p1 and T1 are correct
100     # and that state2 contains a fair initial guess.
101     V1 = Vs
102     state1.set_pT(state1.p, state1.T);
103     if DEBUG_GAS_FLOW:
104         print 'normal_shock(): pre-shock condition assuming real gas and original pT'
105         state1.write_state(sys.stdout)
106     state2.set_pT(state2.p, state2.T);
107     if DEBUG_GAS_FLOW:
108         print 'normal_shock(): post-shock condition assuming real gas and ideal pT'
109         state2.write_state(sys.stdout)
110     #
111     momentum = state1.p + state1.rho * V1 * V1
112     total_enthalpy = state1.h + 0.5 * V1 * V1
113     #
114     def Fvector(rho2, T2):
115         """
116         Constraint equations for state2 from the normal shock relations.
117
```

```
118        The correct post-shock values allow this vector to evaluate to zeros.
119        """
120        state2.set_rhoT(rho2, T2)
121        V2 = V1 * state1.rho / rho2  # mass conservation
122        f1 = momentum - state2.p - state2.rho * V2 * V2
123        f2 = total_enthalpy - state2.h - 0.5 * V2 * V2
124        return f1, f2
125    #
126    A = numpy.zeros((2,2), float)
127    b = numpy.zeros((2,), float)
128    #
129    rho_delta = 1.0
130    T_delta = 1.0
131    rho_tol = 1.0e-3; # tolerance in kg/m^3
132    T_tol = 0.25;  # tolerance in degrees K
133    #
134    # Update the estimates using the Newton-Raphson method.
135    #
136    for count in range(20):
137        rho_save = state2.rho
138        T_save = state2.T
139        f1_save, f2_save = Fvector(rho_save, T_save)
140        # Use finite differences to compute the Jacobian.
141        d_rho = rho_save * 0.01
142        d_T = T_save * 0.01
143        f1, f2 = Fvector(rho_save + d_rho, T_save)
144        df1drho = (f1 - f1_save) / d_rho
145        df2drho = (f2 - f2_save) / d_rho
146        f1, f2 = Fvector(rho_save, T_save + d_T)
147        df1dT = (f1 - f1_save) / d_T
148        df2dT = (f2 - f2_save) / d_T
149        A = numpy.array([[df1drho, df1dT],
150                         [df2drho, df2dT]])
151        b = numpy.array([-f1_save, -f2_save])
152        rho_delta, T_delta = numpy.linalg.solve(A, b)
153        # Possibly limit the increments so that the Newton iteration is
154        # less inclined to go crazy.
155        rho_delta = my_limiter(rho_delta, rho_save)
156        T_delta = my_limiter(T_delta, T_save)
157        rho_new = rho_save + rho_delta
```

```
158         T_new   = T_save + T_delta
159         if DEBUG_GAS_FLOW:
160             print('normal_shock(): rho_save=%e, T_save=%e' % (rho_save, T_save))
161             print('normal_shock(): rho_delta=%e, T_delta=%e' % (rho_delta, T_delta))
162             print('normal_shock(): rho_new=%e, T_new=%e' % (rho_new, T_new))
163         state2.set_rhoT(rho_new, T_new)
164         # Check convergence.
165         if abs(rho_delta) < rho_tol and abs(T_delta) < T_tol: break
166     #
167     if DEBUG_GAS_FLOW:
168         print ('normal_shock(): count = %d, drho=%e, dT=%e' %
169                (count, rho_delta, T_delta) )
170     if ideal_gas_guess: #if we did this, restore the original state before we finish
171         state1 = original_state1.clone()
172     #
173     # Back-out velocities via continuity.
174     V2 = V1 * state1.rho / state2.rho
175     Vg = V1 - V2
176     return (V2, Vg)
177 
178 
179 def normal_shock_p2p1(state1, p2p1):
180     """
181     Computes post-shock conditions, using high-temperature gas properties
182     and a shock-stationary frame.
183 
184     :param state1: pre-shock gas state
185     :param p2p1: ration of pressure across the shock
186     :returns: a tuple of the incident shock speed, V1;
187         the post-shock gas speed, V2 in the shock-reference frame;
188         Vg in the lab frame; and the post shock state state2.
189     """
190     state2 = state1.clone()
191     # Initial guess via ideal gas relations.
192     g = state1.gam
193     Ms = math.sqrt(1+(g+1)/2/g*(p2p1-1.0))
194     V1ideal = Ms * state1.a
195     def error_in_p2p1(Vs, state1=state1, state2=state2, p2p1=p2p1):
196         "Set up error function that will be zero when we have the correct V1"
197         V2, Vg = normal_shock(state1, Vs, state2)
```

```
198         return (state2.p/state1.p - p2p1)/p2p1
199     V1 = secant(error_in_p2p1, V1ideal, 1.01*V1ideal, tol=1.0e-3)
200     if V1 == 'FAIL':
201         raise Exception, ("normal_shock_p2p1: secant method failed p2p1=%g, V1ideal=%g"
202                           % (p2p1, V1ideal))
203     V2, Vg = normal_shock(state1, V1, state2)
204     return (V1, V2, Vg, state2)
205 
206 
207 def reflected_shock(state2, Vg, s5):
208     """
209     Computes state5 which has brought the gas to rest at the end of the shock tube.
210 
211     :param state2: the post-incident-shock gas state
212     :param Vg: the lab-frame velocity of the gas in state 2
213     :param s5: the stagnation state that will be filled in
214         (as a side effect of this function)
215     :returns: Vr, the reflected shock speed in the lab frame.
216     """
217     #
218     # As an initial guess,
219     # assume that we have a very strong shock in an ideal gas.
220     density_ratio = (state2.gam + 1.0)/(state2.gam - 1.0)
221     Vr_a = Vg / density_ratio;
222     V5, Vjunk = normal_shock(state2, Vr_a+Vg, s5)
223     # The objective function is the difference in speeds,
224     # units are m/s.  A value of zero for this function means
225     # that, as the shock propagates upstream with speed ur,
226     # the processed test gas is left in the end of the tube
227     # with a velocity of zero in the laboratory frame.
228     f_a = V5 - Vr_a
229     if DEBUG_GAS_FLOW:
230         print 'Reflected shock: Vr_a: %g, V5: %g' % (Vr_a, V5)
231     #
232     # Now, we need to update this guess...use a secant update.
233     #
234     Vr_b = 1.1 * Vr_a
235     V5, Vjunk = normal_shock(state2, Vr_b+Vg, s5)
236     f_b = V5 - Vr_b
237     if DEBUG_GAS_FLOW:
```

```
238         print 'Reflected shock: Vr_b: %g, V5: %g' % (Vr_b, V5)
239     if abs(f_a) < abs(f_b):
240         f_a, f_b = f_b, f_a
241         Vr_a, Vr_b = Vr_b, Vr_a
242     count = 0
243     while abs(f_b) > 0.5 and count < 20:
244         slope = (f_b - f_a) / (Vr_b - Vr_a)
245         Vr_c = Vr_b - f_b / slope
246         V5, Vjunk = normal_shock(state2, Vr_c+Vg, s5)
247         f_c = V5 - Vr_c
248         if abs(f_c) < abs(f_b):
249             Vr_b = Vr_c; f_b = f_c
250         else:
251             Vr_a = Vr_c; f_a = f_c
252         count = count + 1
253     #
254     # At this point, ur_b should be out best guess.
255     # Update the gas state data and return the best-guess value.
256     #
257     if count >= 20:
258         print 'Reflected shock iteration did not converge.'
259     V5, Vjunk = normal_shock(state2, Vr_b+Vg, s5)
260     return Vr_b
261 
262 
263 def expand_from_stagnation(p_over_p0, state0):
264     """
265     Given a stagnation condition state0, expand to a new pressure.
266 
267     :param p_over_p0: pressure ratio
268     :param state0: Gas object specifying stagnation conditions
269     :returns: new gas state and the corresponding velocity (in m/s)
270         of the expanded stream.
271     """
272     new_state = state0.clone()
273     new_state.set_ps(state0.p * p_over_p0, state0.s)
274     # Matt McGilvray had a note about CEA giving bad entropy values
275     # so we'll assert things are OK before proceeding.
276     assert abs(new_state.s - state0.s)/abs(state0.s) < 0.001
277     h = new_state.e + new_state.p/new_state.rho  # static enthalpy
```

```
278     H = state0.e + state0.p/state0.rho  # stagnation enthalpy
279     V = math.sqrt(2.0*(H-h))
280     return new_state, V
281 
282 def expansion_to_throat_calculation(state1, p0, T0, PRINT_STATUS = 1):
283     """
284     Given a starting state and stagnation pressure and temperature (p0 and T0)
285     find the throat conditions.
286 
287     A more generalised version of a function written by Matt McGilvray for his
288     gun tunnel version of nenzfr.
289 
290     :param state1: starting gas object
291     :param p0: stagnation pressure (in Pa)
292     :param T0: stagnation temperature (in K)
293     :param PRINT_STATUS: tells the program to print or not, turned on by default
294     :returns: a dictionary including state start, enthalpy, throat state,
295         throat velocity, and throat mass flux.
296 
297     """
298     if PRINT_STATUS: print 'Write stagnation conditions.'
299     state1.set_pT(p0, T0)
300     H1 = state1.e + state1.p/state1.rho
301     result = {'state1':state1, 'H1':H1}
302     if PRINT_STATUS: print 'print state1.s =', state1.s
303     #
304     if PRINT_STATUS: print 'Start isentropic relaxation to throat (Mach 1)'
305     def error_at_throat(x, s1s=state1):
306         "Returns Mach number error as pressure is changed."
307         state, V = expand_from_stagnation(x, s1s)
308         return (V/state.a) - 1.0
309     x6 = secant(error_at_throat, 0.95, 0.90, tol=1.0e-4)
310     if x6 == 'FAIL':
311         print "Failed to find throat conditions iteratively."
312         x6 = 1.0
313     state6, V6 = expand_from_stagnation(x6, state1)
314     mflux6 = state6.rho * V6  # mass flux per unit area, at throat
315     result['state6'] = state6
316     result['V6'] = V6
317     result['mflux6'] = mflux6
```

```
318     print 'M6 =', V6/state6.a, ', V6 =', V6, 'm/s and a6 =', state6.a, 'm/s'
319     #
320     return result
321
322
323 def total_condition(state1, V1):
324     """
325     Given a free-stream condition and velocity,
326     compute the corresponding stagnant condition
327     at which the gas is brought to rest isentropically.
328
329     :param state1: Gas object specifying free-stream condition
330     :param V1: free-stream velocity, m/s
331     :returns: Gas object specifying gas total conditions (isentropic, stagnant)
332     """
333     H1 = state1.p/state1.rho + state1.e + 0.5*V1*V1
334     def error_in_total_enthalpy(x, state1=state1, H1=H1):
335         """
336         The enthalpy at the stagnation condition should match
337         the total enthalpy of the stream.
338         """
339         new_state = state1.clone()
340         new_state.set_ps(x * state1.p, state1.s)
341         h = new_state.p/new_state.rho + new_state.e
342         return (H1 - h)/abs(H1)
343     x_total = secant(error_in_total_enthalpy, 1.0, 1.01, tol=1.0e-4)
344     if x_total == 'FAIL':
345         print "Failed to find total conditions iteratively."
346         x_total = 1.0
347     new_state = state1.clone()
348     new_state.set_ps(x_total * state1.p, state1.s)
349     return new_state
350
351
352 def pitot_condition(state1, V1):
353     """
354     Given a free-stream condition, compute the corresponding Pitot condition
355     at which the gas is brought to rest, possibly through a shock.
356
357     :param state1: Gas object specifying free-stream condition
```

```
358     :param V1: free-stream velocity, m/s
359     :returns: Gas object specifying gas impact conditions,
360         possibly after processing be a normal shock.
361     """
362     if V1 > state1.a:
363         # Supersonic free-stream; process through a shock first.
364         state2 = state1.clone()
365         (V2,Vg) = normal_shock(state1, V1, state2)
366         return total_condition(state2, V2)
367     else:
368         # Subsonic free-stream
369         return total_condition(state1, V1)
370
371
372 def steady_flow_with_area_change(state1, V1, A2_over_A1):
373     """
374     Given station 1 condition, velocity and area-ratio A2/A1,
375     compute the steady, isentropic condition at station 2.
376
377     :param state1: Gas object specifying condition at station 1
378     :param V1: velocity at station 1, m/s
379     :param A2_over_A1: area ratio between stations A2/A1
380     :returns: tuple (V2, state2) of conditions at station 2
381     """
382     M1 = abs(V1)/state1.a
383     # When setting up the initial guess for pressure ratio,
384     # we could probably do better with the ideal relation between M and A/Astar.
385     # Note that we'll have trouble heading toward the sonic condition.
386     # For the moment, just don't do that.
387     if M1 > 1.0:
388         if A2_over_A1 > 1.0:
389             # For a supersonic expansion, we might start at the high Mach number end.
390             p2p1_guess_1 = 0.001
391             p2p1_guess_2 = 1.01 * p2p1_guess_1
392         else:
393             # For a supersonic compression, we probably can't go far in area ratio.
394             p2p1_guess_1 = 1.01
395             p2p1_guess_2 = 1.01 * p2p1_guess_1
396     else:
397         if A2_over_A1 < 1.0:
```

```
398             # Subsonic nozzle will accelerate to lower pressures.
399             p2p1_guess_1 = 0.95
400             p2p1_guess_2 = 1.01 * p2p1_guess_1
401         else:
402             # Subsonic diffuser will decelerate to higher pressure.
403             total_cond = total_condition(state1, V1)
404             p2p1_guess_1 = 0.99 * total_cond.p/state1.p
405             p2p1_guess_2 = 0.99 * p2p1_guess_1
406     # Set up constraint data and the error-function to be given to the solver.
407     H1 = state1.p/state1.rho + state1.e + 0.5*V1*V1
408     mdot1 = state1.rho * V1  # assuming unit area at station 1
409     def error_in_mass_flux(p2p1, state1=state1, A2=A2_over_A1, H1=H1, mdot1=mdot1):
410         """
411         The mass flux should be the same at each station.
412         """
413         # print "p2/p1=", p2p1
414         state2 = state1.clone()
415         state2.set_ps(p2p1 * state1.p, state1.s)
416         h2 = state2.p/state2.rho + state2.e
417         V2 = math.sqrt(2*(H1 - h2))
418         mdot2 = state2.rho * V2 * A2
419         return (mdot2 - mdot1)/abs(mdot1)
420     p2p1 = secant(error_in_mass_flux, p2p1_guess_1, p2p1_guess_2, tol=1.0e-4)
421     if p2p1 == 'FAIL':
422         print "Failed to find area-change conditions iteratively."
423         p2p1 = 1.0
424     state2 = state1.clone()
425     state2.set_ps(p2p1 * state1.p, state1.s)
426     h2 = state2.p/state2.rho + state2.e
427     V2 = math.sqrt(2*(H1 - h2))
428     return V2, state2
429 
430 #------------------------------------------------------------------------
431 # Finite-strength waves along characteristic lines.
432 
433 def finite_wave_dp(characteristic, V1, state1, p2, steps=100):
434     """
435     Process the gas isentropically, following a characteristic line.
436 
437     See Section 7.6 Finite Nonlinear Waves in JD Anderson's text
```

```
438     Modern Compressible Flow.
439 
440     :param characteristic: is either 'cplus' or 'cminus'
441     :param V1: initial gas velocity, in m/s
442     :param state1: initial gas state
443     :param p2: new pressure after processing, in Pa
444     :param steps: number of small steps to take through the process
445     :returns: flow condition after processing, as tuple (V2, state2)
446     """
447     V2 = V1
448     p1 = state1.p; s1 = state1.s
449     state2 = state1.clone()
450     dp = (p2 - state1.p)/steps
451     # I'm putting stuff in here that will make the function use more steps
452     # if p2 < dp, to prevent an overshoot into -ve pressure. (Chris James)
453     while p2 < dp:
454         steps *= 2
455         dp = (p2 - state1.p)/steps
456     p = p1+0.5*dp   # effectively mid-point of next step
457     state2.set_ps(p, s1)
458     for i in range(steps):
459         rhoa = state2.rho * state2.a
460         if characteristic == 'cminus':
461             dV = dp / rhoa
462         else:
463             dV = -dp / rhoa
464         V2 += dV
465         p += dp  # prepare for next step
466         state2.set_ps(p, s1)
467     # back up to the correct end-point
468     p -= 0.5 * dp
469     state2.set_ps(p, s1)
470     return V2, state2
471 
472 def finite_wave_dv(characteristic, V1, state1, V2_target, steps=100, Tmin=200.0):
473     """
474     Process the gas isentropically, following a characteristic line.
475 
476     See Section 7.6 Finite Nonlinear Waves in JD Anderson's text
477     Modern Compressible Flow.
```

```
478
479     :param characteristic: is either 'cplus' or 'cminus'
480     :param V1: initial gas velocity, in m/s
481     :param state1: initial gas state
482     :param V2_target: desired velocity after processing, in m/s
483         Note that we may not reach the requested velocity before pressure
484         and temperature become too small.
485     :param steps: number of small steps to take through the process
486     :param Tmin: temperature (in Kelvin) below which we terminate the process.
487         We have this minimum to avoid problems with the thermodynamic
488         polynomials of CEA2 program.  If you really want to work with very low
489         temperatures, it's probably best to use an ideal gas model.
490     :returns: flow condition after processing, as tuple (V2, state2)
491     """
492     V2 = V1
493     dV = (V2_target - V1)/steps
494     p = state1.p
495     s1 = state1.s
496     state2 = state1.clone()
497     for i in range(steps):
498         rhoa = state2.rho * state2.a
499         if characteristic == 'cminus':
500             dp = dV * rhoa
501         else:
502             dp = -dV * rhoa
503         V2 += dV
504         p += dp
505         state2.set_ps(p, s1)
506         if state2.T < Tmin: break
507     return V2, state2
508
509 #-----------------------------------------------------------------------
510 # Oblique shock relations
511
512 def theta_oblique(state1, V1, beta):
513     """
514     Compute the deflection angle and post-shock conditions given the shock wave angle.
515
516     :param state1: upstream gas condition
517     :param V1: speed of gas into shock
```

```
518     :param beta: shock wave angle wrt stream direction (in radians)
519     :returns: tuple of theta, V2 and state2:
520         theta is stream deflection angle in radians
521         V2 is post-shock speed of gas in m/s
522         state2 is post-shock gas state
523     """
524     V1_n = V1 * math.sin(beta)
525     V_t = V1 * math.cos(beta)
526     M1_n = V1 / state1.a
527     if M1_n < 1.0:
528         raise Exception, 'theta_oblique(): subsonic inflow M1_n=%e' % M1_n
529     state2 = state1.clone()
530     V2_n, Vg_n = normal_shock(state1, V1_n, state2)
531     V2 = math.sqrt(V2_n * V2_n + V_t * V_t)
532     theta = beta - math.atan2(V2_n, V_t)
533     return theta, V2, state2
534 
535 
536 def beta_oblique(state1, V1, theta):
537     """
538     Compute the oblique shock wave angle given the deflection angle.
539 
540     :param state1: upstream gas condition
541     :param V1: speed of gas into shock
542     :param theta: stream deflection angle (in radians)
543     :returns: shock wave angle wrt incoming stream direction (in radians)
544     """
545     M1 = V1 / state1.a
546     b1 = max(math.asin(1.0/M1), 1.1*theta)
547     b2 = b1 * 1.05
548     def error_in_theta(beta_guess):
549         theta_guess, V2, state2 = theta_oblique(state1, V1, beta_guess)
550         error_value = theta_guess - theta
551         # print "beta_guess=", beta_guess, "error_value=", error_value
552         return error_value
553     beta_result = secant(error_in_theta, b1, b2, tol=1.0e-4)
554     if beta_result == 'FAIL':
555         raise RuntimeError('beta_oblique(): failed to converge on a shock-wave angle.')
556     return beta_result
557 
```

```
558 #--------------------------------------------------------------------------
559 # Taylor-Maccoll cone flow.
560
561 def EOS_derivatives(state):
562     """
563     Compute equation-of-state derivatives at the specified state.
564
565     :param state: a complete state (with valid data)
566     :returns: tuple of approximations (drho/dp, drho/dh)
567     """
568     rho_0 = state.rho
569     # Choose relatively-small increments in enthalpy (J/kg) and pressure (Pa).
570     dh = abs(state.h) * 0.01 + 1000.0
571     dp = state.p * 0.01 + 1000.0
572     # Use finite-differences to get the partial derivative.
573     state_new = state.clone()
574     state_new.set_ph(state.p + dp, state.h)
575     drhodp = (state_new.rho - rho_0) / dp
576     # and again, for the other.
577     state_new.set_ph(state.p, state.h + dh)
578     drhodh = (state_new.rho - rho_0) / dh
579     # Assume that these first-order differences will suffice.
580     return drhodp, drhodh
581
582 def taylor_maccoll_odes(z, theta, gas_state):
583     """
584     The ODEs from the Taylor-Maccoll formulation.
585
586     See PJ's workbook for Feb 2012 for details.
587     We've packaged them formally so that we might one day use
588     a more sophisticated ODE integrator requiring fewer steps.
589     """
590     rho, V_r, V_theta, h, p = z
591     dfdp, dfdh = EOS_derivatives(gas_state)
592     if DEBUG_GAS_FLOW: print "DEBUG dfdp=", dfdp, "dfdh=", dfdh
593     # Assemble linear system for determining the derivatives wrt theta.
594     A = numpy.zeros((5,5), float)
595     b = numpy.zeros((5,), float)
596     A[0,0] = V_theta; A[0,2] = rho; b[0] = -2.0*rho*V_r - rho*V_theta/math.tan(theta)
597     A[1,1] = 1.0; b[1] = V_theta
```

```
598     A[2,1] = rho*V_r; A[2,2] = rho*V_theta; A[2,4] = 1.0
599     A[3,1] = V_r; A[3,2] = V_theta; A[3,3] = 1.0
600     A[4,0] = 1.0; A[4,3] = -dfdh; A[4,4] = -dfdp
601     dzdtheta = numpy.linalg.solve(A,b)
602     return dzdtheta
603
604 def theta_cone(state1, V1, beta):
605     """
606     Compute the cone-surface angle and conditions given the shock wave angle.
607
608     :param state1: upstream gas condition
609     :param V1: speed of gas into shock
610     :param beta: shock wave angle wrt stream direction (in radians)
611     :returns: tuple of theta_c, V_c and state_c:
612         theta_c is stream deflection angle in radians
613         V_c is cone-surface speed of gas in m/s
614         state_c is cone-surface gas state
615
616     The computation starts with the oblique-shock jump and then integrates
617     across theta until V_theta goes through zero.
618     The cone surface corresponds to V_theta == 0.
619     """
620     # Start at the point just downstream the oblique shock.
621     theta_s, V2, state2 = theta_oblique(state1, V1, beta)
622     #
623     # Initial conditions.
624     dtheta = -0.5 * math.pi / 180.0  # fraction-of-a-degree steps
625     theta = beta
626     V_r = V2 * math.cos(beta - theta_s)
627     V_theta = -V2 * math.sin(beta - theta_s)
628     rho = state2.rho; h = state2.h; p = state2.p
629     gas_state = state2.clone()
630     # For integrating across the shock layer, the state vector is:
631     z = numpy.array([rho, V_r, V_theta, h, p])
632     while V_theta < 0.0:
633         # Keep a copy for linear interpolation at the end.
634         z_old = z.copy(); theta_old = theta
635         # Do the update using a low-order method (Euler) for the moment.
636         dzdtheta = taylor_maccoll_odes(z, theta, gas_state)
637         z += dtheta * dzdtheta; theta += dtheta
```

```
638             rho, V_r, V_theta, h, p = z
639             gas_state.set_ph(p, h)
640             if DEBUG_GAS_FLOW: print "DEBUG theta=", theta, "V_r=", V_r, "V_theta=", V_theta
641         # At this point, V_theta should have crossed zero so
642         # we can linearly-interpolate the cone-surface conditions.
643         V_theta_old = z_old[2]
644         frac = (0.0 - V_theta_old)/(V_theta - V_theta_old)
645         z_c = z_old*(1.0-frac) + z*frac
646         theta_c = theta_old*(1.0-frac) + theta*frac
647         # At the cone surface...
648         rho, V_r, V_theta, h, p = z_c
649         gas_state.set_ph(p, h)
650         assert abs(V_theta) < 1.0e-6
651         #
652         return theta_c, V_r, gas_state
653
654
655     def beta_cone(state1, V1, theta):
656         """
657         Compute the conical shock wave angle given the cone-surface deflection angle.
658
659         :param state1: upstream gas condition
660         :param V1: speed of gas into shock
661         :param theta: stream deflection angle (in radians)
662         :returns: shock wave angle wrt incoming stream direction (in radians)
663         """
664         M1 = V1 / state1.a
665         b1 = max(math.asin(1.0/M1), theta) * 1.01 # to be stronger than a Mach wave
666         b2 = b1 * 1.05
667         def error_in_theta(beta_guess):
668             theta_guess, V_c, state_c = theta_cone(state1, V1, beta_guess)
669             return theta_guess - theta
670         beta_result = secant(error_in_theta, b1, b2, tol=1.0e-4)
671         if beta_result == 'FAIL':
672             raise RuntimeError('beta_cone(): failed to converge on a shock-wave angle.')
673         return beta_result
674
675     #-----------------------------------------------------------------------
676
677     def demo():
```

```
678    print "gas_flow Demonstration -- reflected shock tunnel."
679    from cea2_gas import Gas
680    s1 = Gas({'Air':1.0})
681    s1.set_pT(1.0e5, 300.0)
682    print "s1:"
683    s1.write_state(sys.stdout)
684    print "Incident shock"
685    s2 = s1.clone()
686    V2,Vg = normal_shock(s1, 3000.0, s2)
687    print "V2=", V2, "Vg=", Vg
688    print "s2:"
689    s2.write_state(sys.stdout)
690    #
691    print "Reflected shock"
692    s5 = s1.clone()
693    Vr_b = reflected_shock(s2, Vg, s5)
694    print "Vr_b=", Vr_b
695    print "s5:"
696    s5.write_state(sys.stdout)
697    #
698    print "Expand from stagnation"
699    s6, V = expand_from_stagnation(0.0025, s5)
700    print "V=", V, "Mach=", V/s6.a, "s6:"
701    s6.write_state(sys.stdout)
702    #
703    print "Total condition"
704    s7 = total_condition(s6, V)
705    print "s7:"
706    s7.write_state(sys.stdout)
707    print "Pitot condition from state 6"
708    s8 = pitot_condition(s6, V)
709    print "pitot-p/total-p=", s8.p/s5.p, "s8:"
710    s8.write_state(sys.stdout)
711    #
712    print "\nSteady, isentropic flow with area change."
713    s8a = Gas({'Air':1.0})
714    s8a.set_pT(1.0e5, 320.0)
715    V8a = 1.001 * s8a.a
716    V8b, s8b = steady_flow_with_area_change(s8a, V8a, 10.72) # something like M4 nozzle
717    print "M=", V8b/s8b.a, "expected 4,  p2/p1=", s8b.p/s8a.p, "expected", 0.006586/0.5283
```

```
718     V8b, s8b = steady_flow_with_area_change(s8a, V8a, 1.030) # slightly supersonic
719     print "M=", V8b/s8b.a, "expected 1.2,  p2/p1=", s8b.p/s8a.p, "expected", 0.4124/0.5283
720     V8a = 0.999 * s8a.a
721     V8b, s8b = steady_flow_with_area_change(s8a, V8a, 2.9635) # sonic to M=0.2
722     print "M=", V8b/s8b.a, "expected 0.2,  p2/p1=", s8b.p/s8a.p, "expected", 0.9725/0.5283
723     V8a = 0.2 * s8a.a
724     V8b, s8b = steady_flow_with_area_change(s8a, V8a, 1.3398/2.9635) # M=0.2 to M=0.5
725     print "M=", V8b/s8b.a, "expected 0.5,  p2/p1=", s8b.p/s8a.p, "expected", 0.8430/0.9725
726     #
727     print "\nFinite wave process along a cplus characteristic, stepping in p."
728     V1 = 0.0
729     s9 = Gas({'Air':1.0})
730     s9.set_pT(1.0e5, 320.0)
731     Jplus = V1 + 2*s9.a/(1.4-1)
732     V2, s10 = finite_wave_dp('cplus', V1, s9, 60.0e3)
733     print "V2=", V2, "s10:"
734     s10.write_state(sys.stdout)
735     print "ideal V2=", Jplus - 2*s10.a/(1.4-1)
736     #
737     print "\nFinite wave process along a cplus characteristic, stepping in V."
738     V1 = 0.0
739     s9.set_pT(1.0e5, 320.0)
740     Jplus = V1 + 2*s9.a/(1.4-1)
741     V2, s10 = finite_wave_dv('cplus', V1, s9, 125.0)
742     print "V2=", V2, "s10:"
743     s10.write_state(sys.stdout)
744     print "ideal Jplus=", Jplus, " actual Jplus=", V2 + 2*s10.a/(1.4-1)
745     #
746     M1 = 1.5
747     print "\nOblique-shock demo for M1=%g." % M1
748     from ideal_gas_flow import theta_obl
749     s1.set_pT(100.0e3, 300.0)
750     beta = 45.0 * math.pi/180
751     V1 = 1.5 * s1.a
752     print "s1:"
753     s1.write_state(sys.stdout)
754     theta, V2, s2 = theta_oblique(s1, V1, beta)
755     print "theta=", theta, "V2=", V2, "s2:"
756     s2.write_state(sys.stdout)
757     print "c.f. ideal gas angle=", theta_obl(M1, beta)
```

```
758     #
759     print "Oblique shock angle from deflection."
760     beta2 = beta_oblique(s1, V1, theta)
761     print "beta2(degrees)=", beta2*180/math.pi
762     #
763     M1 = 1.5
764     print "\nTaylor-Maccoll cone flow demo with M1=%g" % M1
765     print "for M1=1.5, beta=49deg, expect theta=20deg from NACA1135."
766     V1 = M1 * s1.a
767     beta = 49.0 * math.pi/180
768     theta_c, V_c, s_c = theta_cone(s1, V1, beta)
769     print "theta_c(deg)=", theta_c*180.0/math.pi, "expected 20deg, surface speed V_c=", V_c
770     print "surface pressure coefficient=", (s_c.p - s1.p)/(0.5*s1.rho*V1*V1), "expected 0.385"
771     print "s_c:"
772     s_c.write_state(sys.stdout)
773     #
774     M1 = 1.5
775     print "\nTaylor-Maccoll cone flow demo with M1=%g" % M1
776     print "for M1=1.5, beta=49.0404423512deg, expect theta=20deg from NACA1135."
777     V1 = M1 * s1.a
778     beta = 49.0404423512 * math.pi/180
779     theta_c, V_c, s_c = theta_cone(s1, V1, beta)
780     print "theta_c(deg)=", theta_c*180.0/math.pi, "expected 20deg, surface speed V_c=", V_c
781     print "surface pressure coefficient=", (s_c.p - s1.p)/(0.5*s1.rho*V1*V1), "expected 0.385"
782     print "s_c:"
783     s_c.write_state(sys.stdout)
784     #
785     M1 = 1.8
786     print "\nTaylor-Maccoll cone flow demo with M1=%g" % M1
787     print "for M1=1.8, beta=45deg, theta=24deg from NACA1135."
788     V1 = M1 * s1.a
789     beta = 45.0 * math.pi/180
790     theta_c, V_c, s_c = theta_cone(s1, V1, beta)
791     print "theta_c(deg)=", theta_c*180.0/math.pi, "expected 24deg, surface speed V_c=", V_c
792     print "surface pressure coefficient=", (s_c.p - s1.p)/(0.5*s1.rho*V1*V1), "expected 0.466"
793     print "s_c:"
794     s_c.write_state(sys.stdout)
795     #
796     M1 = 1.5
797     print "\nConical shock from cone with half-angle 20deg in M1=", M1
```

```
798     V1 = M1 * s1.a
799     beta = beta_cone(s1, V1, 20.0*math.pi/180)
800     print "sigma(deg)=", beta*180/math.pi, "expected 49deg"
801     #
802     print "Done."
```

C Source code for ESTCj application

Top-level application code.

```
#! /usr/bin/env python
"""
estcj.py: Equilibrium Shock Tube Conditions, Junior

This program can be used to estimate flow conditions
for shock-processed flows typical of high-performance
shock-tunnels and expansion tubes.
The gas is assumed to remain in thermochemical equilibrium
and the flow processing is done in decoupled quasi-one-dimensional
wave processes such as shock waves and expansion fans.
For the reflected shock tunnel, this means that the initial,
quiescent test gas is first processed by the incident shock and
subsequently by the reflected shock.
The incident shock sets the inflow conditions for the reflected shock
but there is no further interaction.

The program can do a number of calculations:

* flow in a reflected shock tube with or without a nozzle
* pitot pressure from free-stream flow condition
* stagnation (total) condition from free-stream condition
* code surface condition from free-stream condition

When run as an application, this program takes its input as
command line arguments, performs the requested calculations and outputs
the gas-state results.
To see what specific inputs are required, start the program as::

$ estcj.py --help

Which particular input parameters you need to supply depends on the
chosen task, however, a typical flow condition for the T4 shock tunnel
with the Mach 4 nozzle may be computed using::

$ estcj.py --task=stn --gas=air --T1=300 --p1=125.0e3 --Vs=2414 --pe=34.37e6 --ar=27.0

```

```
37 The full output is a bit too much to include here, but you should see that
38 this condition has an enthalpy of 5.43 MJ/kg and the nozzle-exit condition
39 has a pressure of 93.6 kPa and a static temperature of 1284 degrees K,
40 with a flow speed of 2.95 km/s.
41
42 The default gas model is based on calling the NASA CEA2 program to compute
43 thermochemical properties of the gas, however, there is the option to select
44 the thermochemical gas model used by Eilmer3 (libgas) and an ideal gas.
45 Note that the libgas model is essentially a "frozen" gas model but equilibrium
46 chemistry can be obtained implicitly via a look-up table gas description.
47 To repeat the T4 calculation with a libgas look-up table for air, use::
48
49 $ estcj.py --task=stn --model=libgas --gas=cea-lut-air-ions.lua.gz \
50 --T1=300 --p1=125.0e3 --Vs=2414 --pe=34.37e6 --ar=27.0
51
52 To see the available gases for a particular gas model,
53 use the --list-gas-names option.
54
55
56 Getting the program set up
57 --------------------------
58 estcj.py is not a stand-alone file.
59 It comes as part of the cfcfd3 compressible-flow collection and
60 depends upon functions from the cfpylib library to do the specific
61 calculations.
62 The easiest way to get started is to build and install from the
63 nenzfr directory where this source file resides::
64
65 $ cd app/nenzfr/
66 $ make install
67
68 You may then call upon estcj.py so long as you have suitable
69 enviroment variables set, as per the installation instructions
70 for Eilmer3.
71
72
73 Some History
74 ------------
75 Since 1968, we have been using the ESTC code by Malcolm McIntosh
76 to compute the conditions in the end of the reflected shock tubes
```

```
T1--T5 and HEG.  There are a number of problems in using the ESTC
code, including uncertainty in updating the chemistry coefficients.
This program, ESTCj, moves away from the old chemistry model
by making use of the CEA code from the NASA Glenn Research Center.

.. Author: PA Jacobs
   Institute of Aerodynamics and Flow Technology
   The German Aerospace Center, Goettingen.

.. Versions:
   24-Dec-02 PJ: First code.
   2010 PJ : ported to run with Rowan's cea2_gas module.
   2011 PJ : Added isentropic expansions so that we now have
       a full replacement for stn.f
   01-June-2011 LukeD: Separated the code which writes an output
       file into its own function to allow for better integration with nenzfr.py
   30-June-2011 LukeD: Decreased the starting guess for secant
       when solving for the exit flow
   22-July-2011 LukeD: Added stnp option which allows us to expand
       to a nominated pitot-to-supply pressure ratio. The calculated pitot
       pressure and pitot-to-supply pressure ratio are included in the values
       printed out for the nozzle exit
   24-Feb-2012 PJ: update to use the new cea2_gas.py arrangement.
   31-Dec-2013 PJ: added libgas_gas.py option.
   14-Jan-2014 PJ: included ideal gas option.
"""

VERSION_STRING = "14-Jan-2014"
DEBUG_ESTCJ  = False  # some detailed data is output to help debugging

import sys, os, math
sys.path.append(os.path.expandvars("$HOME/e3bin")) # installation directory
sys.path.append("") # so that we can find user's scripts in current directory
from cfpylib.nm.zero_solvers import secant
# We base our calculation of gas properties upon calls to the NASA Glenn CEA code.
import cfpylib.gasdyn.cea2_gas as cea2
import cfpylib.gasdyn.libgas_gas as libgas
import cfpylib.gasdyn.ideal_gas as ideal
gas_models = {'cea2':cea2, 'libgas':libgas, 'ideal':ideal}
from cfpylib.gasdyn.gas_flow import *
```

```
# ---------------------------------------------------------------------------

def reflected_shock_tube_calculation(gasModel, gasName, p1, T1, Vs, pe,
                                     pp_on_pe, area_ratio, task):
    """
    Runs the reflected-shock-tube calculation from initial fill conditions
    observed shock speed and equilibrium pressure.

    This function may be imported into other applications (such as nenzfr).

    :param gasModel: pointer to the gas model (cea2_gas or libgas_gas)
    :param gasName: name of the specific gas model to create via make_gas_from_name()
    :param p1: fill pressure of gas initially filling shock tube
    :param T1: fill temperature of gas initially filling shock tube
    :param Vs: observed incident shock speed
    :param pe: observed pressure once shock-reflected region reaches equilibrium
    :param pp_on_pe: specify this ratio if we want the supersonic nozzle expansion to
        terminate at a particular Pitot pressure
    :param area_ratio: specify this ratio if we want the supersonic nozzle expansion
        to proceed to a particular quasi-one-dimensional area ratio.
    :param task: one of 'ishock', 'st', 'stn', 'stnp'
    """
    PRINT_STATUS = True  # the start of each stage of the computation is noted.
    #
    if PRINT_STATUS: print 'Write pre-shock condition.'
    state1 = gasModel.make_gas_from_name(gasName)
    state1.set_pT(p1, T1)
    H1 = state1.e + state1.p/state1.rho
    result = {'state1':state1, 'H1':H1}
    #
    if PRINT_STATUS: print 'Start incident-shock calculation.'
    state2 = gasModel.make_gas_from_name(gasName)
    (V2,Vg) = normal_shock(state1, Vs, state2)
    result['state2'] = state2
    result['V2'] = V2
    result['Vg'] = Vg
    #
    if task == 'ishock':
        # We want post-incident-shock conditions only.
```

```
        return result
    #
    if PRINT_STATUS: print 'Start reflected-shock calculation.'
    state5 = gasModel.make_gas_from_name(gasName)
    Vr = reflected_shock(state2, Vg, state5)
    result['state5'] = state5
    result['Vr'] = Vr
    #
    if PRINT_STATUS: print 'Start calculation of isentropic relaxation.'
    state5s = gasModel.make_gas_from_name(gasName)
    # entropy is set, then pressure is relaxed via an isentropic process
    if pe==None:
        state5s.set_ps(state5.p, state5.s)
    else:
        state5s.set_ps(pe, state5.s);
    result['state5s'] = state5s
    H5s = state5s.e + state5s.p/state5s.rho # stagnation enthalpy
    result['H5s'] = H5s
    #
    if task in ['stn','stnp']:
        if PRINT_STATUS: print 'Start isentropic relaxation to throat (Mach 1)'
        def error_at_throat(x, s5s=state5s, gasName=gasName):
            "Returns Mach number error as pressure is changed."
            state, V = expand_from_stagnation(x, s5s)
            return (V/state.a) - 1.0
        x6 = secant(error_at_throat, 0.95, 0.90, tol=1.0e-4)
        if x6 == 'FAIL':
            print "Failed to find throat conditions iteratively."
            x6 = 1.0
        state6, V6 = expand_from_stagnation(x6, state5s)
        mflux6 = state6.rho * V6  # mass flux per unit area, at throat
        result['state6'] = state6
        result['V6'] = V6
        result['mflux6'] = mflux6
        #
        if task == 'stn':
            if PRINT_STATUS: print 'Start isentropic relaxation to nozzle exit.'
            # The mass flux going through the nozzle exit has to be the same
            # as that going through the nozzle throat.
            def error_at_exit(x, s5s=state5s, s6=state6, mflux_throat=mflux6,
```

```
197                            area_ratio=area_ratio):
198                 "Returns mass_flux error as pressure is changed."
199                 state, V = expand_from_stagnation(x, s5s)
200                 mflux = state.rho * V * area_ratio
201                 if DEBUG_ESTCJ: print "x=", x, "p=", state.p, "T=", state.T, "V=", V, \
202                         "mflux=", mflux, "mflux_throat=", mflux_throat
203                 return (mflux-mflux_throat)/mflux_throat
204             # It appears that we need a pretty good starting guess for the pressure ratio.
205             # Maybe a low value is OK.
206             x7 = secant(error_at_exit, 0.001*x6, 0.00005*x6, tol=1.0e-4,
207                         limits=[1.0/state5s.p,1.0])
208             if x7 == 'FAIL':
209                 print "Failed to find exit conditions iteratively."
210                 x7 = x6
211             state7, V7 = expand_from_stagnation(x7, state5s)
212             mflux7 = state7.rho * V7 * area_ratio
213             result['area_ratio'] = area_ratio
214             state7_pitot = pitot_condition(state7, V7)
215             result['state7'] = state7
216             result['V7'] = V7
217             result['mflux7'] = mflux7
218             result['pitot7'] = state7_pitot.p
219         elif task == 'stnp':
220             if PRINT_STATUS: print 'Start isentropic relaxation to nozzle exit pitot pressure.'
221             # The exit pitot pressure has to be the same as that measured
222             def error_at_exit(x, s5s=state5s, s6=state6, pp_pe=pp_on_pe):
223                 "Returns pitot pressure error as static pressure is changed."
224                 state1, V = expand_from_stagnation(x, s5s)
225                 state2 = pitot_condition(state1, V)
226                 if DEBUG_ESTCJ: print "x=", x, "pitot_to_supply=", state2.p/s5s.p, \
227                     "relative error=", (state2.p/s5s.p - pp_pe)/pp_pe
228                 return (state2.p/s5s.p - pp_pe)/pp_pe
229             # We need a low starting guess for the pressure ratio.
230             #x7 = secant(error_at_exit, 0.001*x6, 0.00005*x6, tol=1.0e-4)
231             # Changed the tolerance on 25/07/2011 in order to get the M8 nozzle to work (shot 10803)
232             x7 = secant(error_at_exit, 0.001*x6, 0.00005*x6, tol=2.0e-4,
233                         limits=[1.0/state5s.p,1.0])
234             if x7 == 'FAIL':
235                 print "Failed to find exit conditions iteratively."
236                 x7 = x6
```

```
237             state7, V7 = expand_from_stagnation(x7, state5s)
238             result['area_ratio'] = mflux6/(state7.rho * V7)
239             state7_pitot = pitot_condition(state7, V7)
240             #mflux7 = mflux6
241             result['state7'] = state7
242             result['V7'] = V7
243             result['mflux7'] = mflux6
244             result['pitot7'] = state7_pitot.p
245             if DEBUG_ESTCJ: print "area_ratio=", area_ratio, "pitot7=", state7_pitot.p
246     #
247     if PRINT_STATUS: print 'Done with reflected shock tube calculation.'
248     return result
249 
250 #---------------------------------------------------------------------
251 
252 def main():
253     """
254     The application gets information from the command options,
255     does some calculation (depending on the specified task)
256     and writes the results to the console or a file.
257     """
258     import optparse
259     op = optparse.OptionParser(version=VERSION_STRING)
260     op.add_option('--task', dest='task', default='st',
261                   choices=['st', 'stn', 'stnp', 'ishock', 'total', 'pitot', 'cone'],
262                   help=("particular calculation to make: "
263                         "st = reflected shock tube; "
264                         "stn = reflected shock tube with nozzle; "
265                         "stnp = reflected shock tube with nozzle expanded to pitot; "
266                         "ishock = incident shock only; "
267                         "total = free-stream to total condition; "
268                         "pitot = free-stream to Pitot condition; "
269                         "cone = free-stream to Taylor-Maccoll cone flow"))
270     op.add_option('--model', dest='gasModelName', default='cea2',
271                   choices=['cea2', 'libgas', 'ideal'],
272                   help=("type of gas model: "
273                         "cea2: equilibrium thermochemistry provided by NASA CEA2 code; "
274                         "libgas: thermochemistry provided by Rowan's libgas module; "
275                         "ideal: fixed species with fixed thermodynamic coefficients."))
276     op.add_option('--gas', dest='gasName', default='air',
```

```
277                    help=("name of specific gas; "
278                          "To see the available gases, use the option --list-gas-names"))
279     op.add_option('--list-gas-names', action="store_true", dest="listGasNames", default=False,
280                   help=("list the gas names available for the current gas model"))
281     op.add_option('--p1', dest='p1', type='float', default=None,
282                   help=("shock tube fill pressure or static pressure, in Pa"))
283     op.add_option('--T1', dest='T1', type='float', default=None,
284                   help=("shock tube fill temperature, in degrees K"))
285     op.add_option('--V1', dest='V1', type='float', default=None,
286                   help=("initial speed of gas in lab frame [default: %default], in m/s"))
287     op.add_option('--Vs', dest='Vs', type='float', default=None,
288                   help=("incident shock speed, in m/s"))
289     op.add_option('--pe', dest='pe', type='float', default=None,
290                   help=("equilibrium pressure (after shock reflection), in Pa"))
291     op.add_option('--pp_on_pe', dest='pp_on_pe', type='float', default=None,
292                   help=("nozzle supply to exit pitot pressure ratio"))
293     op.add_option('--ar', dest='area_ratio', type='float', default=None,
294                   help=("exit-to-throat area ratio of the nozzle"))
295     op.add_option('--sigma-deg', dest='cone_half_angle_deg', type='float', default=None,
296                   help=("half-angle of the cone, in degrees"))
297     op.add_option('--ofn', dest='outFileName', default=None,
298                   help="name of file in which to accumulate output."
299                       " file name will be: outFileName-estcj.dat"
300                       " (Note that output defaults to stdout.)")
301     opt, args = op.parse_args()
302     #
303     task = opt.task
304     gasName = opt.gasName
305     gasModel = gas_models[opt.gasModelName]
306     if opt.listGasNames:
307         print "For gas model %s, these gases are available:" % opt.gasModelName
308         for name in gasModel.list_gas_names():
309             print "    %s" % name
310         return 0
311     p1 = opt.p1
312     T1 = opt.T1
313     V1 = opt.V1
314     Vs = opt.Vs
315     pe = opt.pe
316     pp_on_pe = opt.pp_on_pe
```

```
317     area_ratio = opt.area_ratio
318     cone_half_angle_deg = opt.cone_half_angle_deg
319     outFileName = opt.outFileName
320     if DEBUG_ESTCJ:
321         print 'estcj:', opt.gasModelName, gasName, p1, T1, V1, Vs, pe, area_ratio, outFileName
322     #
323     bad_input = False
324     if p1 is None:
325         print "Need to supply a float value for p1."
326         bad_input = True
327     if T1 is None:
328         print "Need to supply a float value for T1."
329         bad_input = True
330     if Vs is None and task in ['stn', 'stnp', 'st', 'ishock']:
331         print "Need to supply a float value for Vs."
332         bad_input = True
333     if V1 is None and task in ['pitot', 'total', 'cone']:
334         print "Need to supply a free-stream velocity."
335         bad_input = True
336     if cone_half_angle_deg is None and task in ['cone',]:
337         print "Need to supply a cone half-angle (in degrees)."
338         bad_input = True
339     if pe is None and task in ['stn', 'stnp', 'st']:
340         print "Need to supply a float value for pe."
341         bad_input = True
342     if pp_on_pe is None and task in ['stnp']:
343         print "Need to supply a float value for pp_on_pe."
344         bad_input = True
345     if area_ratio is None and task in ['stn']:
346         print "Need to supply a float value for ar=area_ratio."
347         bad_input = True
348     if bad_input:
349         return -2
350     #
351     if outFileName is None:
352         fout = sys.stdout
353     else:
354         fout = open(outFileName+'-estcj.dat','w')
355     fout.write('estcj: Equilibrium Shock Tube Conditions\n')
356     fout.write('Version: %s\n' % VERSION_STRING)
```

```
357     #
358     if task in ['st', 'stn', 'stnp', 'ishock']:
359         fout.write('Input parameters:\n')
360         fout.write('    gasModel is %s, Gas is %s, p1: %g Pa, T1: %g K, Vs: %g m/s\n'
361                    % (opt.gasModelName,gasName,p1,T1,Vs) )
362         result = reflected_shock_tube_calculation(gasModel, gasName, p1, T1, Vs,
363                                                   pe, pp_on_pe, area_ratio,
364                                                   task=task)
365         fout.write('State 1: pre-shock condition\n')
366         result['state1'].write_state(fout)
367         fout.write('State 2: post-shock condition.\n')
368         result['state2'].write_state(fout)
369         fout.write('    V2: %g m/s, Vg: %g m/s\n' % (result['V2'],result['Vg']) )
370         if task in ['st', 'stn', 'stnp']:
371             fout.write('State 5: reflected-shock condition.\n')
372             result['state5'].write_state(fout)
373             fout.write('    Vr: %g m/s\n' % (result['Vr'],) )
374             fout.write('State 5s: equilibrium condition (relaxation to pe)\n')
375             result['state5s'].write_state(fout)
376             fout.write('Enthalpy difference (H5s - H1): %g J/kg\n' %
377                        ((result['H5s'] - result['H1']),) )
378             if task in ['stn','stnp']:
379                 # shock tube plus nozzle, expand gas isentropically, stopping at area_ratio
380                 fout.write('State 6: Nozzle-throat condition (relaxation to M=1)\n')
381                 result['state6'].write_state(fout)
382                 fout.write('    V6: %g m/s, M6: %g, mflux6: %g kg/s/m**2\n' %
383                            (result['V6'], result['V6']/result['state6'].a, result['mflux6'],) )
384                 fout.write('State 7: Nozzle-exit condition (relaxation to correct mass flux)\n')
385                 result['state7'].write_state(fout)
386                 fout.write('    V7: %g m/s, M7: %g, mflux7: %g kg/s/m**2, area_ratio: %g, pitot: %g Pa\n' %
387                            (result['V7'], result['V7']/result['state7'].a, result['mflux7'],
388                             result['area_ratio'], result['pitot7'],) )
389                 fout.write('    pitot7_on_p5s: %g\n' % (result['pitot7']/result['state5s'].p,) )
390     elif task in ['total', 'TOTAL', 'Total']:
391         fout.write('Input parameters:\n')
392         fout.write('    gasModel is %s, Gas is %s, p1: %g Pa, T1: %g K, V1: %g m/s\n'
393                    % (opt.gasModelName,gasName,p1,T1,V1) )
394         state1 = gasModel.make_gas_from_name(gasName)
395         state1.set_pT(p1, T1)
396         state0 = total_condition(state1, V1)
```

```
397         fout.write('Total condition:\n')
398         state0.write_state(fout)
399     elif task in ['pitot', 'PITOT', 'Pitot']:
400         fout.write('Input parameters:\n')
401         fout.write('    gasModel is %s, Gas is %s, p1: %g Pa, T1: %g K, V1: %g m/s\n'
402                    % (opt.gasModelName,gasName,p1,T1,V1) )
403         state1 = gasModel.make_gas_from_name(gasName)
404         state1.set_pT(p1, T1)
405         state0 = pitot_condition(state1, V1)
406         fout.write('Pitot condition:\n')
407         state0.write_state(fout)
408     elif task in ['cone', 'CONE', 'Cone']:
409         fout.write('Input parameters:\n')
410         fout.write('    gasModel is %s, Gas is %s, p1: %g Pa, T1: %g K, V1: %g m/s, sigma: %g degrees\n'
411                    % (opt.gasModelName,gasName,p1,T1,V1,cone_half_angle_deg) )
412         state1 = gasModel.make_gas_from_name(gasName)
413         state1.set_pT(p1, T1)
414         fout.write('Free-stream condition:\n')
415         state1.write_state(fout)
416         cone_half_angle_rad = cone_half_angle_deg*math.pi/180.0
417         beta_rad = beta_cone(state1, V1, cone_half_angle_rad)
418         theta_c, V_cone_surface, state2 = theta_cone(state1, V1, beta_rad)
419         assert abs(theta_c - cone_half_angle_rad) < 0.001
420         fout.write('Shock angle: %g (rad), %g (deg)\n' % (beta_rad, beta_rad*180.0/math.pi))
421         fout.write('Cone-surface velocity: %g m/s\n' % (V_cone_surface,))
422         fout.write('Cone-surface condition:\n')
423         state2.write_state(fout)
424     #
425     if outFileName is None:
426         pass
427     else:
428         fout.close()
429     return 0
430
431 # --------------------------------------------------------------------
432
433 if __name__ == '__main__':
434     if len(sys.argv) <= 1:
435         print "Equilibrium Shock Tube Conditions"
436         print "   Version:", VERSION_STRING
```

```
437         print "   To see some useful hints, invoke this program with option --help."
438         sys.exit(0)
439     return_flag = main()
440     sys.exit(return_flag)
```

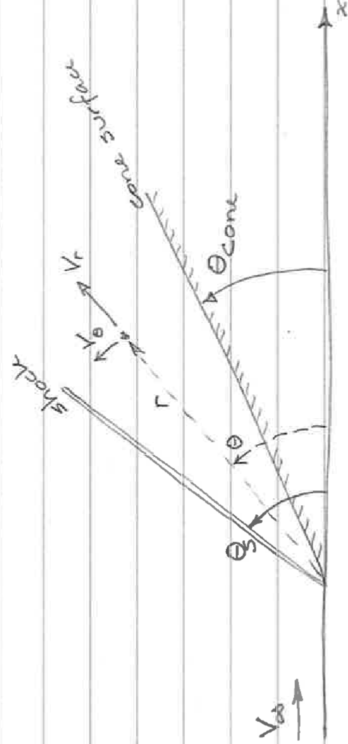
D Notes on conical flow

Scanned straight from PJ's workbook, warts and all.

Feb 2012

Taylor - Maccoll Cone Flow

- formulation as per §10.3 (page 296)
  JD Anderson Modern Compressible Flow (2nd Ed)



polar coordinates with origin at cone apex r, θ

all properties are functions of θ only

continuity equation $\nabla \cdot (\rho \vec{V}) = 0$ becomes, for axisymmetric flow,

$$2 \rho V_r + \rho V_\theta \cot\theta + \rho \frac{\partial V_\theta}{\partial \theta} + V_\theta \frac{\partial \rho}{\partial \theta} = 0 \qquad (10.2)$$

condition that flow is irrotational

$$V_\theta = \frac{\partial V_r}{\partial \theta} \qquad (10.5)$$

Euler's equation for isentropic flow $\quad dp = -\rho V dV \quad$ (apply in any direction)

$$\frac{dp}{d\theta} = -\rho \left(V_r \frac{dV_r}{d\theta} + V_\theta \frac{dV_\theta}{d\theta} \right) \qquad (10.6)$$

(sort of)

Feb 2012

Taylor-Maccoll cone flow.

Total Enthalpy is constant

$$H_0 = h_\infty + \frac{V_\infty^2}{2} = h + \frac{V^2}{2} \qquad \text{where } V^2 = V_r^2 + V_\theta^2$$

$$0 = \frac{\partial h}{\partial \theta} + \frac{1}{2}\frac{\partial}{\partial \theta}(V^2)$$

$$0 = \frac{\partial h}{\partial \theta} + \frac{1}{2}\left[\frac{\partial V_r^2}{\partial \theta} + \frac{\partial V_\theta^2}{\partial \theta}\right]$$

$$0 = \frac{\partial h}{\partial \theta} + V_r \frac{\partial V_r}{\partial \theta} + V_\theta \frac{\partial V_\theta}{\partial \theta}$$

Connections between thermo variables via cea2-gas.py

$$\rho = f(p, h)$$

$$d\rho = dp\left(\frac{\partial f}{\partial p}\right) + dh\left(\frac{\partial f}{\partial h}\right)$$

↖ can probably evaluate these numerically at specified p, h values

Feb 2012

Taylor–Maccoll cone flow

collect the vector of variables

$$\begin{bmatrix} \rho \\ V_r \\ V_\theta \\ h \\ p \end{bmatrix}$$

} Anderson's set for ideal gas

collect constraint equations into matrix form

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} V_\theta & 0 & \rho & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & \rho V_r & \rho V_\theta & 0 & 1 \\ 0 & V_r & V_\theta & 1 & 0 \\ 1 & 0 & 0 & -\partial f/\partial h & -\partial f/\partial p \end{bmatrix} \begin{bmatrix} \partial\rho/\partial\theta \\ \partial V_r/\partial\theta \\ \partial V_\theta/\partial\theta \\ \partial h/\partial\theta \\ \partial p/\partial\theta \end{bmatrix} = \begin{bmatrix} -2\rho V_r - \rho V_\theta \cot\theta \\ V_\theta \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

(columns numbered 0 1 2 3 4)

Test cases from NACA 1135
Chart 5, Chart 6 for $\gamma = 1.405$ (low T air)

	Case 1	Case 2
cone half angle δ	20°	24°
free stream Mach	1.5	1.8
shock wave angle Θ	49°	45°
surface pressure coeff $(p_c - p_1)/q_1$	0.385	0.466

convergence for case 1

$\Delta\Theta$	δ	$(p_c-p_1)/q_1$	
0.5°	19.95°	0.390	
			> 0.6%
0.2°	19.96°	0.3875	
			> 0.2%
0.1°	19.96°	0.3866	