AE 5385: High Temperature Gasdynamics
Homework 6

James Grisham

December 3, 2015

Problem 1

Problem Statement

A sharp cone with a half-cone angle of $\theta_c = 15^\circ$ is tested in a hypersonic wind tunnel at a Mach number of 10.6 at zero degrees angle of attack. If the total pressure and total temperature at the stagnation chamber of the wind tunnel are 1200 psi and 2000°R, respectively, and assuming the temperature of the cone surface (wall temperature) is 560°R:

(a) Using the reference temperature method, estimate the shear stress and the heat transfer to the wall at a location of 15.7 inches away from the apex measured along the cone surface assuming laminar flow.

(b) Using the reference temperature method, estimate the shear stress and the heat transfer to the wall at a location of 15.7 inches away from the apex measured along the cone surface assuming turbulent flow.

Solution

At first glance, this problem seems complicated by the fact that it is a cone, not a wedge. This complication is mitigated by use of the Mangler transformation. This transformation allows the results to be calculated for a wedge and then transformed to a cone by multiplying by $\sqrt{3}$.

This problem was solved two ways, both assuming a calorically perfect gas. First, the problem was solved with $\gamma = 1.4$. Next, because the temperature is somewhat high in the stagnation chamber, an equilibrium calculation was performed using Cantera. The problem was then re-solved assuming that the flow as chemically and vibrationally frozen. The properties across the oblique shock were computed using the expressions for a calorically perfect gas. After that, the conditions behind the shock were used as the edge conditions required by the reference temperature method.

Given the edge conditions, the reference temperature method was applied as follows.

The recovery factor was computed using the following expressions.

$$r = \begin{cases} \sqrt{Pr}, & \text{Laminar} \\ Pr^{1/3}, & \text{Turbulent} \end{cases}$$

The adiabatic wall temperature can then be calculated:

$$T_{\text{aw}} = T_e \left(1 + r\frac{\gamma - 1}{2} M_e^2\right)$$

Finally, the reference temperature is determined:

$$T^* = T_e + 0.5(T_w - T_e) + 0.22(T_{\text{aw}} - T_e)$$

Reynolds number and Prandtl number are inputs to the correlations for c_f and C_H. The Reynolds number requires knowledge of the density and viscosity.

$$\rho^* = \frac{p_e}{RT^*}$$

The viscosity is determined using Sutherland's law. Now, Re_x^* can be determined.

$$Re_x^* = \frac{\rho^* U_e x}{\mu(T^*)}$$

After finding the Reynolds number, the skin friction coefficient at station x can be determined:

$$c_f^* = \begin{cases} \dfrac{0.664}{\sqrt{Re_x^*}}, & \text{Laminar} \\ \dfrac{0.455}{(\log_{10} Re_x^*)^{2.58}}, & \text{Turbulent} \end{cases}$$

The Stanton number for laminar flow is

$$C_H^* = \frac{0.332}{\sqrt{Re_x^*}} (Pr^*)^{-2/3}$$

For turbulent flow, the Reynolds analogy is used:

$$C_H^* = \frac{1}{2} c_f^* (Pr^*)^{-2/3}$$

After finding the skin friction coefficient and the Stanton number, the wall shear stress and heat transfer rate per unit area can be found:

$$\tau_w = \frac{1}{2}\rho^* U_e^2 c_f^*$$
$$q_w = \rho^* U_e c_p^* (T_{\text{aw}} - T_w) C_H^*$$

The procedure outlined above was implemented using C++. The code is available in the appendix. The results are reported next.

Calorically perfect ($\gamma = 1.4$)

Properties across the oblique shock:

$$
\begin{aligned}
p_1 &= 132.061 \text{ Pa} \\
p_2 &= 1955.57 \text{ Pa} \\
T_1 &= 47.3377 \text{ K} \\
T_2 &= 162.341 \text{ K} \\
\rho_1 &= 0.00972043 \text{ kg/m}^3 \\
\rho_2 &= 0.0419725 \text{ kg/m}^3 \\
M_1 &= 10.6 \\
M_2 &= 5.4057 \\
u_1 &= 1461.89 \text{ m/s} \\
u_2 &= 1380.61 \text{ m/s}
\end{aligned}
$$

Laminar reference temperature results:

$$
\begin{aligned}
T^* &= 413.222 \text{ K} \\
\rho^* &= 0.0164895 \text{ kg/m}^3 \\
Re_x^* &= 388056 \\
c_f^* &= 0.0017906 \\
C_H^* &= 0.00115447 \\
\tau_w &= 28.1397 \text{ N/m}^2 \\
q_w &= 1.72523 \text{ W/cm}^2
\end{aligned}
$$

Turbulent reference temperature results:

$$
\begin{aligned}
T^* &= 423.372 \text{ K} \\
\rho^* &= 0.0160942 \text{ kg/m}^3 \\
Re_x^* &= 372292 \\
c_f^* &= 0.00937752 \\
C_H^* &= 0.0101566 \\
\tau_w &= 143.837 \text{ N/m}^2 \\
q_w &= 15.86 \text{ W/cm}^2
\end{aligned}
$$

Calorically perfect ($\gamma = 1.32796$)

Properties across the oblique shock:

$$
\begin{aligned}
p_1 &= 50.2316 \text{ Pa} \\
p_2 &= 690.411 \text{ Pa} \\
T_1 &= 57.2 \text{ K} \\
T_2 &= 166.254 \text{ K} \\
\rho_1 &= 0.00304783 \text{ kg/m}^3 \\
\rho_2 &= 0.0144127 \text{ kg/m}^3 \\
M_1 &= 10.6 \\
M_2 &= 5.88706 \\
u_1 &= 1568.17 \text{ m/s} \\
u_2 &= 1484.82 \text{ m/s}
\end{aligned}
$$

Laminar reference temperature results:

$$
\begin{aligned}
T^* &= 419.552 \text{ K} \\
\rho^* &= 0.00571126 \text{ kg/m}^3 \\
Re_x^* &= 142999 \\
c_f^* &= 0.00294971 \\
C_H^* &= 0.00183061 \\
\tau_w &= 18.5707 \text{ N/m}^2 \\
q_w &= 1.22665 \text{ W/cm}^2
\end{aligned}
$$

Turbulent reference temperature results:

$$
\begin{aligned}
T^* &= 428.137 \text{ K} \\
\rho^* &= 0.00559673 \text{ kg/m}^3 \\
Re_x^* &= 138139 \\
c_f^* &= 0.0115403 \\
C_H^* &= 0.0120313 \\
\tau_w &= 71.1982 \text{ N/m}^2 \\
q_w &= 8.35543 \text{ W/cm}^2
\end{aligned}
$$

In the case of $\gamma = 1.32796$, the wall shear and heat flux are less than the case with $\gamma = 1.4$. The difference is about 50%.

Problem 2

Problem Statement

Consider the spherically blunted bi-conic nose geometry of a hypersonic ICBM shown in question 1 of homework 5. Assuming the same freestream conditions given in that question ($M_\infty = 18$, altitude is 15 km and $\alpha = 0$) with $T_w = 2000$ K,

(a) Calculate the stagnation point heat transfer for the nose cone geometry using calorically perfect gas assumption.

(b) Calculate the stagnation point heat transfer for the given nose cone geometry using air in thermochemical equilibrium. Assume that the term including the Lewis number is 1, i.e., neglect the heat transfer due to diffusion by assuming $h_D = 0$.

Solution

This problem is solved by using the correlation from Fay and Riddell:

$$q_w = 0.76Pr^{-0.6}(\rho_e\mu_e)^{0.4}(\rho_w\mu_w)^{0.1}\sqrt{\left(\frac{dU_e}{dx}\right)_s}(h_{0,e} - h_w)\left[1 + (Le^{0.52} - 1)\left(\frac{h_D}{h_{0,e}}\right)\right] \tag{1}$$

The stagnation point velocity gradient is

$$\left(\frac{dU_e}{dx}\right)_s = \frac{1}{R}\sqrt{\frac{2(p_e - p_\infty)}{\rho_e}}$$

Since we are neglecting heat transfer due to diffusion, (1) becomes

$$q_w = 0.76Pr^{-0.6}(\rho_e\mu_e)^{0.4}(\rho_w\mu_w)^{0.1}\sqrt{\left(\frac{dU_e}{dx}\right)_s}(h_{0,e} - h_w) \tag{2}$$

The correlation given in the notes is slightly different. The $(\rho_w\mu_w)^{-.1}$ term is dropped. This changes the results slightly. This problem was solved using C++. To make sure the C++ code was correct, it was validated using the example problems discussed in class. Using the correlation in (2), I get $q_w = 23.752$ W/cm^2 for the CPG example. Using the correlation in the notes, I get $q_w = 26.0285$ W/cm^2. Equation (2) will be used moving forward.

The freestream thermodynamic state was determined using the International Standard Atmosphere. Next, the properties across the shock were found using the expressions for a calorically perfect gas. After that, the properties at the boundary layer edge must be determined. Since we are dealing with a stagnation point, $p_e = p_{0,2}$, $T_e = T_{0,2}$. Now,

$$\rho_e = \frac{p_e}{RT_e}$$

The viscosity at the edge of the boundary layer is then determined using Sutherland's law.

Next, the properties at the wall must be determined. The problem statement says that $T_w = 2000$ K. From boundary layer theory, we know that $\partial p/\partial n \approx 0$ so, $p_w = p_e$. Furthermore, because

the flow is essentially incompressible, $\rho_w = \rho_e$. Again, the viscosity can be determined using Sutherland's law.

Since we are dealing with a calorically perfect gas, $h_w = c_p T_w$. Also, $h_{\text{aw}} \approx h_e$. This can be seen by looking at the definition of the adiabatic wall enthalpy:

$$h_{\text{aw}} = h_e + \frac{r}{2}\frac{U_e^2}{2} \quad \overset{\approx 0}{\nearrow} \quad \therefore \; h_{\text{aw}} \approx h_e$$

Again, remembering that we are dealing with a stagnation point, $h_e = h_{0,e}$, so $h_{\text{aw}} \approx h_{0,e}$.

Now, essentially everything is known and the heat flux can be evaluated. Doing so yields

$$\boxed{q_w = 2077.17 \text{ W/cm}^2}$$

This number seems quite high.

The main difference for the gas in thermo-chemical equilibrium is the calculation of the properties across the shock and evaluating the state at the different stations. To facilitate this task, a C++ code was developed which effectively wraps a Cantera model of air in equilibrium. The code is in the appendix. The tool was validated against CEA and Virginia Tech's EqAir applet. Also, the example problems from class were re-worked with and the results were in good agreement.

Out of curiosity, the Mach number behind the shock was calculated for the thermo-chemical equilibrium case. It was found to be approximately 0.28. So, it seems that the incompressible assumption is reasonable.

The result for a gas in thermo-chemical equilibrium is

$$\boxed{q_w = 110.034 \text{ W/cm}^2}$$

This value for the rate of heat transfer per unit area at the stagnation point is an order of magnitude less than the value assuming a calorically perfect gas. The temperatures at the various stations were compared to figure out if the above estimate is reasonable. At the edge of the boundary layer, the CPG model predicted a temperature that was twice as large as the model using air in thermo-chemical equilibrium.

Code listing

```
#ifndef SECANTHEADERDEF
#define SECANTHEADERDEF

#include <fstream>
#include <vector>

/**
 * This header contains a templated function for root finding.  A variable
 * argument list can be passed to the function using variadic templates.
 *
 * @param x0 initial guess.
 * @param tol tolerance used to monitor convergence.
 * @param max_iter max number of iterations to be used.
 * @param (*f)(T,Tn...) function pointer.
```

```
 * @param params parameter pack passed to *f.
 * @return the value of the root.
 *
 * \author James Grisham
 * \date 08/12/2015
 *
 */

template <typename T,typename... Tn>
T secant(T x0, T tol, const unsigned int max_iter, T (*f)(T,Tn...), Tn... params) {

  // Declaring some variables
  int i=2;
  std::vector<T> x(max_iter);
  std::vector<T> F(max_iter);
  x[0] = x0;
  F[0] = (*f)(x0,params...);

  // Figuring out the first step
  // This probably isn't a good way to do it
  x[1] = 1.1*x[0];
  F[1] = (*f)(x[1],params...);

  // Iterating
  while ((fabs(F[i-1])>tol)&&(i<max_iter)) {
    x[i] = x[i-1] - F[i-1]*(x[i-1]-x[i-2])/(F[i-1]-F[i-2]);
    F[i] = (*f)(x[i],params...);
#ifdef VERBOSE
    std::cout << "Iteration: " << i << " x = " << x[i] << " f = " << F[i] << std::endl;
#endif
    ++i;
  }

  return x[i-1];

}

#endif
```

```
#ifndef OBLIQUE_HEADER_DEF
#define OBLIQUE_HEADER_DEF

#include <cmath>
#include "normal_cperf.h"

namespace oblique {

  // Function to compute beta given theta (in radians) and M1
  template <typename T>
  T beta(const T M1, const T theta, const T gamma=(T) 1.4) {

    // Declaring variables
    T l,Chi,N,D;
    T delta = (T) 1.0;  // corresponds to weak solution. 0 is strong.

    // Equations
    l =sqrt(pow(M1*M1 - 1.0,2) - 3.0*(1.0 + (gamma - 1.0)/2.0*M1*M1)*\
        (1.0 + (gamma+1.0)/2.0*M1*M1)*pow(tan(theta),2));
    Chi = (pow(M1*M1 - 1.0,3) - 9.0*(1.0 + (gamma-1.0)/2.0*M1*M1)*\
        (1.0 + (gamma-1.0)/2.0*M1*M1 + (gamma+1.0)/4.0*pow(M1,4))*\
        pow(tan(theta),2))/pow(l,3);
    N = M1*M1 - 1.0 + 2.0*l*cos((4.0*M_PI*delta + acos(Chi))/3.0);
    D = 3.0*(1.0 + (gamma-1.0)/2.0*M1*M1)*tan(theta);
    return atan2(N,D);

  }

```

```
29    // Function to compute p2qp1 for given M1 and theta (in radians)
30    template <typename T>
31    T p2qp1(const T M1, const T theta, const T gamma=(T) 1.4) {
32
33      // Declaring variables
34      T b,Mn1;
35
36      // Finding beta and Mn1
37      b = beta<T>(M1,theta,gamma);
38      Mn1 = M1*sin(b);
39
40      // Using normal shock relations with Mn1
41      return normal::p2qp1(Mn1,gamma);
42
43    }
44
45    // Function to compute T2/T1 for given M1 and theta (in radians)
46    template <typename T>
47    T T2qT1(const T M1, const T theta, const T gamma=(T) 1.4) {
48
49      // Declaring variables
50      T b,Mn1;
51
52      // Finding beta and Mn1
53      b = beta<T>(M1,theta,gamma);
54      Mn1 = M1*sin(b);
55
56      // Using normal shock relations with Mn1
57      return normal::T2qT1(Mn1,gamma);
58
59    }
60
61    // Function to compute rho2/rho1 for given M1 and theta (in radians)
62    template <typename T>
63    T rho2qrho1(const T M1, const T theta, const T gamma=(T) 1.4) {
64
65      // Declaring variables
66      T b,Mn1;
67
68      // Finding beta and Mn1
69      b = beta(M1,theta,gamma);
70      Mn1 = M1*sin(b);
71
72      // Using normal shock relations with Mn1
73      return normal::rho2qrho1(Mn1,gamma);
74
75    }
76
77    // Function to compute the Mach number after the shock
78    template <typename T>
79    T M2(const T M1, const T theta , const T gamma=(T) 1.4) {
80
81      // Declaring variables
82      T b,Mn1,Mn2;
83
84      // Finding beta and Mn1
85      b = beta(M1,theta,gamma);
86      Mn1 = M1*sin(b);
87
88      // Using normal shock relations with Mn1
89      Mn2 = normal::M2(Mn1,gamma);
90
91      return Mn2/sin(b-theta);
92
93    }
94
95  }
96
```

```
#endif
```

```
#ifndef EQUILIBRIUMAIRHEADER
#define EQUILIBRIUMAIRHEADER

#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <initializer_list>
#include <exception>
#include "nlopt.hpp"
#include "cantera/IdealGasMix.h"
#include "cantera/transport/MixTransport.h"
#include "/home/James/codes/cpp-opt/include/secant.h"

// Overloading ostream operator
template <typename T> class air;
template <typename T>
std::ostream& operator<<(std::ostream& os, const air<T>& g) {
  if (g.found_equilibrium) {
    os << (g.get_gas()).report();
  }
  else {
    os << "Gas not in equilibrium.  Run find equilibrium method.";
  }
  return os;
}

/*****************************************************************\
 * Class definition                                              *
\*****************************************************************/

template <typename T=double>
class air {

  public:
    air();
    ~air();
    void set_TP(const T temperature, const T pressure);
    void set_rhoP(const T density, const T pressure);
    void find_equilibrium();
    inline Cantera::IdealGasPhase get_gas() const { return *gas; };
    inline T get_gamma() const { return gas->cp_mass()/gas->cv_mass(); };
    inline T get_p()     const { return gas->pressure(); };
    inline T get_h()     const { return gas->enthalpy_mass(); };
    inline T get_T()     const { return gas->temperature(); };
    inline T get_rho()   const { return gas->density(); };
    inline T get_s()     const { return gas->entropy_mass(); };
    inline T get_cp()    const { return gas->cp_mass(); };
    inline T get_cv()    const { return gas->cv_mass(); };
    inline T get_e()     const { return gas->intEnergy_mass(); };
    inline T get_mu()    const { return gas_transport->viscosity(); };
    inline T get_k()     const { return gas_transport->thermalConductivity(); };
    T get_MW();
    friend std::ostream& operator<< <>(std::ostream& os, const air<T>& g);

  private:
    T temp, press, dens;
    size_t num_species;
    Cantera::IdealGasPhase* gas;
    Cantera::MixTransport *gas_transport;
    std::vector<std::string> species_names;
    std::vector<T> X;  // mole fractions
    std::vector<T> c;  // mass fractions
    std::vector<T> MW; // molecular weights
    bool found_equilibrium;
```

```
66
67  };
68
69  /*****************************************************************\
70   * Class implementation                                          *
71  \*****************************************************************/
72
73  // Overriding default constructor
74  template <typename T>
75  air<T>::air() {
76    gas = new Cantera::IdealGasMix("air_below6000K.xml","air_below6000K");
77    gas_transport = new Cantera::MixTransport();
78    num_species = gas->nSpecies();
79    X.resize(num_species);
80    c.resize(num_species);
81    MW.resize(num_species);
82    found_equilibrium = false;
83  }
84
85  // Overriding default destructor
86  template <typename T>
87  air<T>::~air() {
88    delete gas;
89    delete gas_transport;
90  }
91
92  // Method for setting state using temperature and pressure
93  template <typename T>
94  void air<T>::set_TP(const T temperature, const T pressure) {
95    gas->setState_TP(temperature,pressure);
96  }
97
98  template <typename T>
99  T obj(const std::vector<T>& dvs, std::vector<T>& grad, void* data) {
100
101   // Creating a new mixture
102   Cantera::IdealGasMix* gas1 = new Cantera::IdealGasMix("air_below6000K.xml","air_below6000K");
103
104   // The IdealGasMix class is derived from ThermoPhase which is derived
105   // from Phase.
106   T temperature = dvs[0];
107   T* Tdata;
108   Tdata = (T *) data;
109   T density = Tdata[0];
110   T pressure = Tdata[1];
111   if (temperature <= (T) 0.0) {
112     return (T) 1.0e6;
113   }
114   gas1->setState_TP(temperature,pressure);
115
116   // Finding equilibrium
117   const std::string const_vars1("TP");
118   gas1->equilibrate(const_vars1);
119
120   // Computing residual
121   T rho1 = gas1->density();
122
123   delete gas1;
124
125   //std::cout << "obj_func = " << fabs(density-rho1)/density << std::endl;
126   return fabs(density-rho1)/density;
127 }
128
129 // Method for setting state using density and pressure
130 template <typename T>
131 void air<T>::set_rhoP(const T density, const T pressure) {
132
133   // Using an iterative procedure to determine the appropriate
```

```
134   // temperature.
135   T R_std = 287.0;
136   T Ti = pressure/(density*R_std);
137
138   //nlopt::opt opt(nlopt::LN_SBPLX,1);
139   nlopt::opt opt(nlopt::LN_COBYLA,1);
140   std::vector<T> lb({0.0});
141   std::vector<T> ub({10000.0});
142   std::vector<T> T_0({Ti});
143   T data[] = {density, pressure};
144   T dummy;
145 #ifdef DEBUG
146   static int i=0;
147   ++i;
148   std::cout << "Function call: " << i << " density = " << density << " pressure = " << pressure << std::endl;
149 #endif
150   opt.set_min_objective(&obj<T>,&data);
151   opt.set_ftol_abs((T) 1.0e-5);
152   //opt.set_lower_bounds(lb);
153   //opt.set_upper_bounds(ub);
154   try {
155     nlopt::result r = opt.optimize(T_0,dummy);
156     //std::cout << "result = " << r << std::endl;
157   }
158   catch (std::exception& e) {
159     std::cout << e.what() << std::endl;
160     //exit(1);
161   }
162   temp = T_0[0];
163
164   gas->setState_TP(temp,pressure);
165
166 }
167
168 template <typename T>
169 void air<T>::find_equilibrium() {
170   const std::string const_vars("TP");
171   gas->equilibrate(const_vars);
172 #ifdef DEBUG
173   std::cout << "Enthalpy at equilibrium: " << gas->enthalpy_mass() << std::endl;
174   std::cout << "Density: " << gas->density() << std::endl;
175 #endif
176   found_equilibrium = true;
177   gas->getMassFractions(&c[0]);  // this is done because the methods expect an array, not an STL vector
178   gas->getMoleFractions(&X[0]);
179   // Cantera is dumb... this doesn't work -> gas_transport->init(gas);
180 }
181
182 template <typename T>
183 T air<T>::get_MW() {
184   gas->getMolecularWeights(&MW[0]);
185   T MW_m = (T) 0;
186   for (unsigned int i=0; i<X.size(); ++i) {
187     MW_m += X[i]*MW[i];
188   }
189   return MW_m;
190 }
191
192 /*  The below doesn't work.
193 template <typename T>
194 T air<T>::get_Pr() {
195   return gas_transport->viscosity()*get_cp()/gas_transport->thermalConductivity();
196 }
197 */
198
199 #endif
```

```cpp
#include <iostream>
#include <string>
#include <cmath>
#include "cantera/IdealGasMix.h"
#include "equilibrium_air.h"
#include "../oblique_cperf.h"

// Structure which holds the solution to the inviscid problem
struct shock_properties {
  double p1,p2,T1,T2,rho1,rho2,u1,u2,M1,M2;
  double p01,p02,T01,T02;
  double a1,a2;
  double gamma,R;
  double theta,beta;
};

std::ostream& operator<< (std::ostream& os, shock_properties& s) {
  os << "\\begin{align*}\n";
  os << "p_1 &= "     << s.p1   << " \\ \\text{Pa}\\\\\n";
  os << "p_2 &= "     << s.p2   << " \\ \\text{Pa}\\\\\n";
  os << "T_1 &= "     << s.T1   << " \\ \\text{K}\\\\\n";
  os << "T_2 &= "     << s.T2   << " \\ \\text{K}\\\\\n";
  os << "\\rho_1 &= " << s.rho1 << " \\ \\text{kg/m$^3$}\\\\\n";
  os << "\\rho_2 &= " << s.rho2 << " \\ \\text{kg/m$^3$}\\\\\n";
  os << "M_1 &= "     << s.M1   << " \\\\\n";
  os << "M_2 &= "     << s.M2   << " \\\\\n";
  os << "u_1 &= "     << s.u1   << " \\ \\text{m/s}\\\\\n";
  os << "u_2 &= "     << s.u2   << " \\ \\text{m/s}\n";
  os << "\\end{align*}\n";
  return os;
}

// Structure which holds the properties for the ref. temp. method
struct ref_temp_properties {
  double Ue,Te,pe,rhoe,Me;
  double Tstar,rhostar,pstar;
  double Tw,Taw;
  double Pr;
  double Prstar,Rexstar,Recstar,Chstar,Cfstar,cfstar;
  double gamma,R,cp;
  double tau_w,q_w;
  bool is_turbulent;
};

// Overloading ostream operator to write out info
std::ostream& operator<< (std::ostream& os, ref_temp_properties& r) {
  os << "\\begin{align*}\n";
  os << "T^*     &= " << r.Tstar   << " \\ \\text{K}\\\\\n";
  os << "\\rho^* &= " << r.rhostar << " \\ \\text{kg/m$^3$}\\\\\n";
  os << "Re_x^*  &= " << r.Rexstar << " \\\\\n";
  os << "c_f^*   &= " << r.cfstar  << " \\\\\n";
  os << "C_H^*   &= " << r.Chstar  << " \\\\\n";
  os << "\\tau_w &= " << r.tau_w   << " \\ \\text{N/m$^2$}\\\\\n";
  os << "q_w     &= " << r.q_w/1e4 << " \\ \\text{W/cm$^2$}\\\\\n";
  os << "\\end{align*}\n";
  return os;
}

// Function prototypes
void oblique_shock(shock_properties& s);
void inv_to_visc(const double Tw, const double Pr, const shock_properties& s, ref_temp_properties& r);
void ref_temp_laminar(ref_temp_properties& rtp, const double x);
void ref_temp_turbulent(ref_temp_properties& rtp, const double x);
double mu(double T);

// Main
int main() {
```

```
68
69    // Declaring variables
70    air<double> eqair;
71    double R = 287.0;
72    double R_u = 8.314462;
73    double Pr_eq, Pr_cpg;
74    double Tw = 311.11;
75    Pr_cpg = 0.715;
76    Pr_eq = 0.7571;  // <-- this is from CEA because Cantera is dumb.
77
78    // Setting state
79    eqair.set_TP(1111.11,8273708.4);
80
81    // Finding equilibrium
82    eqair.find_equilibrium();
83
84    // Outputing info
85    std::ofstream state_file("hw6prob1_state.txt");
86    state_file << eqair << std::endl;
87    state_file.close();
88
89    // CPG properties
90    shock_properties sp_cpg;
91    sp_cpg.gamma = 1.4;
92    sp_cpg.p01 = 8273709.0;
93    sp_cpg.T01 = 1111.111;
94    sp_cpg.theta = 15.0*M_PI/180.0;
95    sp_cpg.M1 = 10.6;
96    sp_cpg.R = R;
97
98    // Equilibrium properties
99    shock_properties sp_eq(sp_cpg);
100   sp_eq.gamma = eqair.get_gamma();
101   sp_eq.R = R_u/(eqair.get_MW()*1.0e-3);
102
103   // Finding properties across a shock wave for a wedge
104   oblique_shock(sp_cpg);
105   oblique_shock(sp_eq);
106
107   // Setting properties at the boundary layer edge required for
108   // the reference temperature method
109   ref_temp_properties rt_cpg;
110   ref_temp_properties rt_eq;
111   rt_cpg.is_turbulent = false;
112   rt_eq.is_turbulent = false;
113   inv_to_visc(Tw,Pr_cpg,sp_cpg,rt_cpg);
114   inv_to_visc(Tw,Pr_eq,sp_eq,rt_eq);
115   ref_temp_properties rt_cpgt(rt_cpg);
116   ref_temp_properties rt_eqt(rt_eq);
117   rt_cpgt.is_turbulent = true;
118   rt_eqt.is_turbulent = true;
119
120   // Using reference temperature method to find tau_w and q_w
121   ref_temp_laminar(rt_cpg,0.39878);
122   ref_temp_laminar(rt_eq,0.39878);
123   ref_temp_turbulent(rt_cpgt,0.39878);
124   ref_temp_turbulent(rt_eqt,0.39878);
125
126   // Writing outputs to file
127   std::ofstream outfile("hw6prob1-results.tex");
128   outfile << "\\subsubsection*{Calorically perfect ($\\gamma=1.4$)}" << std::endl;
129   outfile << "Properties across the oblique shock:" << std::endl;
130   outfile << sp_cpg << std::endl;
131   outfile << "\nLaminar reference temperature results:" << std::endl;
132   outfile << rt_cpg << std::endl;
133   outfile << "\nTurbulent reference temperature results:" << std::endl;
134   outfile << rt_cpgt << std::endl;
135
```

```
136   outfile << "\\subsubsection*{Calorically perfect ($\\gamma=" << sp_eq.gamma << "$)}" << std::endl;
137   outfile << "Properties across the oblique shock:" << std::endl;
138   outfile << sp_eq << std::endl;
139   outfile << "\nLaminar reference temperature results:" << std::endl;
140   outfile << rt_eq << std::endl;
141   outfile << "\nTurbulent reference temperature results:" << std::endl;
142   outfile << rt_eqt << std::endl;
143
144   outfile.close();
145
146   return 0;
147
148 }
149
150 // Function for computing the properties across the oblique shock
151 void oblique_shock(shock_properties& s) {
152
153   // Computing static pressure and temp using isentropic
154   // relations
155   s.p1 = s.p01/pow(1.0 + (s.gamma-1.0)/2.0*s.M1*s.M1,s.gamma/(s.gamma-1.0));
156   s.T1 = s.T01/(1.0 + (s.gamma-1.0)/2.0*s.M1*s.M1);
157
158   // Computing density
159   s.rho1 = s.p1/(s.R*s.T1);
160
161   // Finding initial velocity
162   s.a1 = sqrt(s.gamma*s.R*s.T1);
163   s.u1 = s.M1*s.a1;
164
165 #ifdef DEBUG
166   std::cout << "theta = " << s.theta << "\n";
167   std::cout << "gamma = " << s.gamma << "\n";
168   std::cout << "p2/p1 = " << oblique::p2qp1(s.M1,s.theta,s.gamma) << std::endl;
169   std::cout << "T2/T1 = " << oblique::T2qT1(s.M1,s.theta,s.gamma) << std::endl;
170   std::cout << "rho2/rho1 = " << oblique::rho2qrho1(s.M1,s.theta,s.gamma) << std::endl;
171   std::cout << "M2 = " << oblique::M2(s.M1,s.theta,s.gamma) << std::endl;
172 #endif
173
174   // Finding properties across the shock
175   s.p2 = s.p1*oblique::p2qp1(s.M1,s.theta,s.gamma);
176   s.T2 = s.T1*oblique::T2qT1(s.M1,s.theta,s.gamma);
177   s.a2 = sqrt(s.gamma*s.R*s.T2);
178   s.rho2 = s.rho1*oblique::rho2qrho1(s.M1,s.theta,s.gamma);
179   s.M2 = oblique::M2(s.M1,s.theta,s.gamma);
180   s.u2 = s.M2*s.a2;
181
182 }
183
184 void inv_to_visc(const double Tw, const double Pr, const shock_properties& s, ref_temp_properties& r) {
185
186   r.Te = s.T2;
187   r.pe = s.p2;
188   r.Tw = Tw;
189   r.Pr = Pr;
190   r.Me = s.M2;
191   r.Ue = s.u2;
192   r.gamma = s.gamma;
193   r.R = s.R;
194
195 }
196
197 // Function for Sutherland's law
198 double mu(double T) {
199   double T0,S,mu0;
200   T0 = 273.1;
201   S = 110.6;
202   mu0 = 1.716e-5;
203   return mu0*(pow(T/T0,1.5)*(T0+S)/(T+S));
```

```
}

void ref_temp_laminar(ref_temp_properties& rtp, const double x) {

  // Computing the adiabatic wall temperature
  double r = sqrt(rtp.Pr);
  rtp.Taw = rtp.Te*(1 + r*(rtp.gamma-1.0)/2.0*rtp.Me*rtp.Me);

  // Computing the reference temperature
  rtp.Tstar = rtp.Te + 0.5*(rtp.Tw - rtp.Te) + 0.22*(rtp.Taw-rtp.Te);

  // Evaluating properties at the reference temperature
  rtp.rhostar = rtp.pe/(rtp.R*rtp.Tstar);
  rtp.Rexstar = rtp.rhostar*rtp.Ue*x/mu(rtp.Tstar);
  rtp.cfstar = 0.644/sqrt(rtp.Rexstar)*sqrt(3.0);
  rtp.tau_w = 0.5*rtp.rhostar*rtp.Ue*rtp.Ue*rtp.cfstar;
  rtp.Chstar = 0.332/sqrt(rtp.Rexstar)*pow(rtp.Pr,-2.0/3.0)*sqrt(3.0);
  rtp.cp = rtp.gamma*rtp.R/(rtp.gamma-1.0);
  rtp.q_w = rtp.Chstar*rtp.rhostar*rtp.Ue*rtp.cp*(rtp.Taw-rtp.Tw);

}

void ref_temp_turbulent(ref_temp_properties& rtp, const double x) {

  // Computing the adiabatic wall temperature
  double r = pow(rtp.Pr,1.0/3.0);
  rtp.Taw = rtp.Te*(1 + r*(rtp.gamma-1.0)/2.0*rtp.Me*rtp.Me);

  // Computing the reference temperature
  rtp.Tstar = rtp.Te + 0.5*(rtp.Tw - rtp.Te) + 0.22*(rtp.Taw-rtp.Te);

  // Evaluating properties at the reference temperature
  rtp.rhostar = rtp.pe/(rtp.R*rtp.Tstar);
  rtp.Rexstar = rtp.rhostar*rtp.Ue*x/mu(rtp.Tstar);
  // book -> rtp.cfstar = 0.0592/pow(rtp.Rexstar,0.2)*sqrt(3);
  rtp.cfstar = 0.455/pow(log10(rtp.Rexstar),2.58)*sqrt(3.0);
  rtp.tau_w = 0.5*rtp.rhostar*rtp.Ue*rtp.Ue*rtp.cfstar;
  rtp.Chstar = 0.5*rtp.cfstar*pow(rtp.Pr,-2.0/3.0)*sqrt(3.0);
  rtp.cp = rtp.gamma*rtp.R/(rtp.gamma-1.0);
  rtp.q_w = rtp.Chstar*rtp.rhostar*rtp.Ue*rtp.cp*(rtp.Taw-rtp.Tw);

}
```

```
#include <iostream>
#include <cmath>
#include "../viscosity.h"
#include "../normal_cperf.h"

double q_w(double Pr, double rhoe, double mue, double rhow, double muw, double duedx, double h0e, double hw);

int main() {

  // Declaring variables;
  double gamma,R,cp,Pr;
  double T1,p1,M1,rho1,T01,p01,u1,a1;
  double T2,p2,M2,rho2,T02,p02,u2,a2;
  double Te,pe,Me,rhoe,Ue,mue;
  double Tw,pw,Mw,rhow,muw;
  double duedx,Radius;

  // Constants
  R = 287.0;
  gamma = 1.4;
  Pr = 0.715;
  cp = gamma*R/(gamma-1.0);

  // Inputs
```

```
  M1 = 18.0;
  Tw = 2000.0;
  Radius = 0.15;

  // Inputs from ISA
  T1 = 216.650;
  p1 = 12044.6;
  rho1 = 0.193674;
  a1 = 295.070;
  T01 = T1*(1.0 + (gamma-1.0)/2.0*M1*M1);

  // Properties across the normal shock (CPG)
  T02 = T01;
  T2 = T1*normal::T2qT1(M1);
  p2 = p1*normal::p2qp1(M1);
  rho2 = rho1*normal::rho2qrho1(M1);
  M2 = normal::M2(M1);
  p02 = p2*pow(1+(gamma-1.0)/2.0*M2*M2,gamma/(gamma-1.0));

  // Properties at the boundary layer edge
  pe = p02;
  Te = T02;
  std::cout << "Te = " << Te << std::endl;
  rhoe = pe/(R*Te);
  mue = mu(Te);

  // Properties at the wall
  pw = pe;
  rhow = rhoe;  // because flow is incompressible
  muw = mu(Tw);

  // dUe/dx at the stagnation point
  duedx = 1.0/Radius*sqrt(2.0*(pe - p1)/rhoe);

  // Heat flux
  std::cout << "q_w = " <<
    q_w(Pr,rhoe,mue,rhow,muw,duedx,cp*Te,cp*Tw)/1.0e4 <<
    " W/cm^2\n";

  return 0;

}

double q_w(double Pr, double rhoe, double mue, double rhow, double muw, double duedx, double h0e, double hw) {
  return 0.76*pow(Pr,-0.6)*pow(rhoe*mue,0.4)*pow(rhow*muw,0.1)*sqrt(duedx)*(h0e-hw);
}
```

```
#include <iostream>
#include <cmath>
#include "../viscosity.h"
#include "normal_tperf.h"
#include "equilibrium_air.h"

double q_w(double Pr, double rhoe, double mue, double rhow, double muw, double duedx, double h0e, double hw);

int main() {

  // Declaring variables;
  air<double> eqair;
  double gamma,R,cp,Pr,R_u;
  double T1,p1,M1,rho1,T01,p01,h01,h1,u1,a1;
  double T2,p2,M2,rho2,T02,p02,h02,h2,u2,a2;
  double Te,pe,Me,rhoe,Ue,mue,he;
  double Tw,pw,Mw,rhow,muw,hw;
  double duedx,Radius;

  // Constants
```

```
  R_u = 8.314;
  Pr = 0.715;
  gamma = 1.4;

  // Inputs
  M1 = 18.0;
  Tw = 2000.0;
  Radius = 0.15;

  // Inputs from ISA
  a1 = 295.070;
  T1 = 216.650;
  p1 = 12044.6;
  rho1 = 0.193674;
  eqair.set_TP(T1,p1);
  eqair.find_equilibrium();
  u1 = M1*a1;
  h1 = eqair.get_h();
  h01 = h1 + 0.5*rho1*u1*u1;

  // Properties across the normal shock (using iterative procedure)
  h02 = h01;
  normal::iterate(u1,p1,rho1,T1,h1,p2,rho2,h2);
  u2 = rho1/rho2*u1;
  eqair.set_rhoP(rho2,p2);
  eqair.find_equilibrium();
  R = R_u/(eqair.get_MW()*1.0e-3);
  p02 = p2 + 0.5*rho2*u2*u2;
  T2 = eqair.get_T();

  // Checking conditions after the shock
  a2 = sqrt(eqair.get_gamma()*R*T2);
  M2 = u2/a2;
  std::cout << "M_2 = " << M2 << std::endl;

  // Properties at the boundary layer edge
  pe = p02;
  he = h02;
  rhoe = rho2;
  eqair.set_rhoP(rhoe,pe);
  eqair.find_equilibrium();
  Te = eqair.get_T();
  std::cout << "Te = " << Te << std::endl;
  mue = mu(Te);

  // Properties at the wall
  pw = pe;      // because dp/dn = 0
  rhow = rhoe;  // because flow is incompressible
  muw = mu(Tw);
  eqair.set_TP(Tw,pw);
  eqair.find_equilibrium();
  hw = eqair.get_h();

  // dUe/dx at the stagnation point
  duedx = 1.0/Radius*sqrt(2.0*(pe - p1)/rhoe);

  // Heat flux
  std::cout << "q_w = " <<
    q_w(Pr,rhoe,mue,rhow,muw,duedx,he,hw)/1.0e4 <<
    " W/cm^2\n";

  return 0;

}

double q_w(double Pr, double rhoe, double mue, double rhow, double muw, double duedx, double h0e, double hw) {
  return 0.76*pow(Pr,-0.6)*pow(rhoe*mue,0.4)*pow(rhow*muw,0.1)*sqrt(duedx)*(h0e-hw);
```

```
88 }
```