```
FUNCTION SYNTAX:
    [p3,UR,gas3] = reflected_eq(gas1,gas2,gas3,UI)

INPUT:
    gas1 = gas object at initial state
    gas2 = gas object at post-incident-shock state (already computed)
    gas3 = working gas object
    UI = incident shock speed (m/s)

OUTPUT:
    p3 = post-reflected-shock pressure (Pa)
    UR = reflected shock speed (m/s)
    gas3 = gas object at equilibrium post-reflected-shock state
```

reflected_fr Calculates frozen post-reflected-shock state.

reflected_fr.m

```
FUNCTION SYNTAX:
    [p3,UR,gas3] = reflected_fr(gas1,gas2,gas3,UI)

INPUT:
    gas1 = gas object at initial state
    gas2 = gas object at post-incident-shock state (already computed)
    gas3 = working gas object
    UI = incident shock speed (m/s)

OUTPUT:
    p3 = post-reflected-shock pressure (Pa)
    UR = reflected shock speed (m/s)
    gas3 = gas object at frozen post-reflected-shock state
```

`reflected_fr` in reflections.py

```
FUNCTION SYNTAX:
    [p3,UR,gas3] = reflected_fr(gas1,gas2,gas3,UI)

INPUT:
    gas1 = gas object at initial state
    gas2 = gas object at post-incident-shock state (already computed)
    gas3 = working gas object
    UI = incident shock speed (m/s)

OUTPUT:
    p3 = post-reflected-shock pressure (Pa)
    UR = reflected shock speed (m/s)
    gas3 = gas object at frozen post-reflected-shock state
```

ZND Model Detonation Structure Computation

zndsolve.m

```
FUNCTION SYNTAX:
[output] = zndsolve(gas,gas1,U1)

INPUT:
gas = Cantera gas object - postshock state
```

```
gas1 = Cantera gas object - initial state
U1 = Shock Velocity

OPTIONAL INPUT (name-value pairs):
     t_end = end time for integration, in sec
     rel_tol = relative tolerance
     abs_tol = absolute tolerance
     advanced_output = calculates optional extra parameters
     such as induction lengths
     max_step = maximum step size (in time) that solver is allowed to take

OUTPUT:
    output = a dictionary containing the following results:
        time = time array
        distance = distance array

        T = temperature array
        P = pressure array
        rho = density array
        U = velocity array
        thermicity = thermicity array
        species = species mass fraction array

        M = Mach number array
        af = frozen sound speed array
        g = gamma (cp/cv) array
        wt = mean molecular weight array
        sonic = sonic parameter (c^2-U^2) array

        tfinal = final target integration time
        xfinal = final distance reached

        gas1 = a copy of the input initial state
        U1 = shock velocity

        and, if advanced_output=True:
        ind_time_ZND = time to maximum thermicity gradient
        ind_len_ZND = distance to maximum thermicity gradient
        exo_time_ZND = pulse width (in secs) of thermicity  (using 1/2 max)
        ind_time_ZND = pulse width (in meters) of thermicity (using 1/2 max)
        max_thermicity_width_ZND = according to Ng et al definition
```

`zndsolve` in znd.py

This function is included as a module in the toolbox script `znd.py` .

```
    FUNCTION SYNTAX:
    output = zndsolve(gas,gas1,U1,**kwargs)

    INPUT
        gas = Cantera gas object - postshock state
        gas1 = Cantera gas object - initial state
        U1 = shock velocity (m/s)

    OPTIONAL INPUT:
            t_end = end time for integration, in sec
            max_step = maximum time step for integration, in sec
            t_eval = array of time values to evaluate the solution at.
                        If left as 'None', solver will select values.
                        Sometimes these may be too sparse for good-looking plots.
            relTol = relative tolerance
            absTol = absolute tolerance
            advanced_output = calculates optional extra parameters such as induction lengths
            Method = method of integration, 'LSODA' is default.

        OUTPUT:
            output = a dictionary containing the following results:
                time = time array
                distance = distance array

                T = temperature array
                P = pressure array
                rho = density array
                U = velocity array
                thermicity = thermicity array
                species = species mass fraction array

                M = Mach number array
                af = frozen sound speed array
                g = gamma (cp/cv) array
                wt = mean molecular weight array
                sonic = sonic parameter (c^2-U^2) array

                tfinal = final target integration time
                xfinal = final distance reached

                gas1 = a copy of the input initial state
                U1 = shock velocity

                and, if advanced_output=True:
                ind_time_ZND = time to maximum thermicity gradient
                ind_len_ZND = distance to maximum thermicity gradient
                exo_time_ZND = pulse width (in secs) of thermicity  (using 1/2 max)
                ind_time_ZND = pulse width (in meters) of thermicity (using 1/2 max)
                max_thermicity_width_ZND = according to Ng et al definition
```

CV Model Explosion Computation

cvsolve.m

```
FUNCTION SYNTAX:
    output = cvsolve(gas,varargin)

INPUT:
    gas = working gas object

OPTIONAL INPUT (name-value pairs):
    t_end = end time for integration, in sec. If not included
            as an input, set to 10^-3 by default.
     rel_tol = relative tolerance
     abs_tol = absolute tolerance
     max_step = maximum step size (in time) that solver is allowed to take

OUTPUT:
    output = a structure containing the following results:
         time = time array
         T = temperature profile array
         P = pressure profile array
         speciesY = species mass fraction array
         speciesX = species mole fraction array

         gas = working gas object

         exo_time = pulse width (in secs) of temperature gradient (using 1/2 max)
         ind_time = time to maximum temperature gradient
         ind_time_10 = time to 10% of maximum temperature gradient
         ind_time_90 = time to 90% of maximum temperature gradient
```

cvsolve in cv.py

This function is included as a module in the toolbox script cv.py.

```
    FUNCTION SYNTAX:
        output = cvsolve(gas,**kwargs)

    INPUT:
        gas = working gas object

    OPTIONAL INPUT:
        t_end = end time for integration, in sec
        max_step = maximum time step for integration, in sec
        t_eval = array of time values to evaluate the solution at.
                    If left as 'None', solver will select values.
                    Sometimes these may be too sparse for good-looking plots.
        relTol = relative tolerance
        absTol = absolute tolerances

    OUTPUT:
        output = a dictionary containing the following results:
            time = time array
            T = temperature profile array
            P = pressure profile array
            speciesY = species mass fraction array
            speciesX = species mole fraction array

            gas = working gas object

            exo_time = pulse width (in secs) of temperature gradient (using 1/2 max)
            ind_time = time to maximum temperature gradient
            ind_time_10 = time to 10% of maximum temperature gradient
            ind_time_90 = time to 90% of maximum temperature gradient
```

CP Model Explosion Computation

cpsolve.m

```
FUNCTION SYNTAX:
    output = cpsolve(gas,varargin)

INPUT:
    gas = working gas object

OPTIONAL INPUT (name-value pairs):
    t_end = end time for integration, in sec. If not included
            as an input, set to 10^-3 by default.%
    rel_tol = relative tolerance
    abs_tol = absolute tolerance
    max_step = maximum step size (in time) that solver is allowed to take

OUTPUT:
    output = a structure containing the following results:
         time = time array
         T = temperature profile array
         P = pressure profile array
         speciesY = species mass fraction array
         speciesX = species mole fraction array

         gas = working gas object

         exo_time = pulse width (in secs) of temperature gradient (using 1/2 max)
         ind_time = time to maximum temperature gradient
         ind_time_10 = time to 10% of maximum temperature gradient
         ind_time_90 = time to 90% of maximum temperature gradient
```

cpsolve in cp.py

```
   FUNCTION SYNTAX:
       output = cpsolve(gas,**kwargs)

    INPUT:
       gas = working gas object

    OPTIONAL INPUT:
       t_end = end time for integration, in sec
       max_step = maximum time step for integration, in sec
       t_eval = array of time values to evaluate the solution at.
                   If left as 'None', solver will select values.
                   Sometimes these may be too sparse for good-looking plots.
       relTol = relative tolerance
       absTol = absolute tolerances
       Method = method of integration, 'LSODA' is default.

    OUTPUT:
       output = a dictionary containing the following results:
           time = time array
           T = temperature profile array
           D = density profile array
           speciesY = species mass fraction array
           speciesX = species mole fraction array
```

```
              gas = working gas object

              exo_time = pulse width (in secs) of temperature gradient (using 1/2 max)
              ind_time = time to maximum temperature gradient
              ind_time_10 = time to 10% of maximum temperature gradient
              ind_time_90 = time to 90% of maximum temperature gradient
```

Stagnation Reaction zone structure computation for blunt body flow using the approximation of linear gradient in mass flux (ρw)

stgsolve.m

```
SYNTAX
[output] = stgsolve(gas,gas1,U1,Delta)

INPUT
gas = Cantera gas object - postshock state
gas1 = Cantera gas object - initial state
U1 = Shock Velocity
Delta = shock standoff distance

OPTIONAL INPUT (positional argument):
t_end = end time for integration, in sec. If not included
        as an input, set to 10^-3 by default.

OUTPUT
Structure
  output.time = time array
  output.distance = distance array

  output.T = temperature array
  output.P = pressure array
  output.rho = density array
  output.U = velocity array
  output.thermicity = thermicity array

  output.M = Mach number array
  output.af = frozen sound speed array
  output.g = gamma (cp/cv) array
  output.wt = mean molecular weight array
  output.sonic = sonic parameter (c^2-U^2) array
```

`stgsolve` in stagnation.py

This function is included as a module in the toolbox script `stagnation.py` .

```
    FUNCTION SYNTAX:
    output = stgsolve(gas,gas1,U1,Delta,**kwargs)

    INPUT
        gas = Cantera gas object - postshock state
        gas1 = Cantera gas object - initial state
        U1 = shock velocity (m/s)
        Delta = shock standoff distance (m)

    OPTIONAL INPUT:
```

```
    t_end = end time for integration, in sec
    max_step = maximum time step for integration, in sec
    t_eval = array of time values to evaluate the solution at.
                If left as 'None', solver will select values.
                Sometimes these may be too sparse for good-looking plots.
    relTol = relative tolerance
    absTol = absolute tolerance


OUTPUT:
    output = a dictionary containing the following results:
        time = time array
        distance = distance array

        T = temperature array
        P = pressure array
        rho = density array
        U = velocity array
        thermicity = thermicity array
        distance = distance array
        species = species mass fraction array

        M = Mach number array
        af = frozen sound speed array
        g = gamma (cp/cv) array
        wt = mean molecular weight array
        sonic = sonic parameter (c^2-U^2) array

        gas1 = a copy of the input initial state
        U1 = shock velocity
        Delta = shock standoff distance
```

Thermo Computation of sound speed and Grüneisen coefficent.

soundspeed_eq Computes the equilibrium sound speed by using a centered finite difference approximation. Directly evaluating pressure at two density/specific volume states along an isentrope requires use of `equilibrate('SV')`. However, this may not always converge at high pressure. Instead, a more robust method using `equilibrate('TP')` is used that employs thermodynamic identities detailed further in Appendix G2 of the report.

soundspeed_eq.m

```
FUNCTION SYNTAX:
    aequil =  soundspeed_eq(gas)

INPUT:
    gas = working gas object (restored to original state at end of function)

OUTPUT:
    aequil = equilibrium sound speed = sqrt({d P/d rho)_s, eq) (m/s)
```

`soundspeed_eq` in thermo.py

```
FUNCTION SYNTAX:
    ae =  soundspeed_eq(gas)
    INPUT:
        gas = working gas object (restored to original state at end of function)

    OUTPUT:
        ae = equilibrium sound speed = sqrt({d P/d rho)_s, eq) (m/s)
```

soundspeed_fr Computes the frozen sound speed by using a centered finite difference approximation and evaluating frozen composition states on the isentrope passing through the reference (S, V) state supplied by the gas object passed to the function.

soundspeed_fr.m

```
FUNCTION SYNTAX:
    afrozen =  soundspeed_fr(gas)

INPUT:
    gas = working gas object (restored to original state at end of function)

OUTPUT:
    afrozen = frozen sound speed = sqrt({d P/d rho)_{s,x0})
```

`soundspeed_fr` in thermo.py

```
    FUNCTION SYNTAX:
        afrz =  soundspeed_fr(gas)

    INPUT:
        gas = working gas object (restored to original state at end of function)

    OUTPUT:
        afrz = frozen sound speed = sqrt({d P/d rho)_{s,x0})
```

gruneisen_eq Computes the equilibrium Grüneisen coefficient by using a centered finite difference approximation and evaluating equilibrium states on the isentrope passing through the reference (S, V) state supplied by the gas object passed to the function.

gruneisen_eq.m

```
FUNCTION SYNTAX:
    G_eq =  gruneisen_eq(gas)

INPUT:
    gas = working gas object (restored to original state at end of function)

OUTPUT:
    G_eq = equilibrium Gruneisen coefficient [-de/dp)_{v,eq} =
            -(v/T)dT/dv)_{s,eq} = + (rho/T)(dT/d rho)_{s,eq}]
```

`gruneisen_eq` in thermo.py

```
    FUNCTION SYNTAX:
        G_eq =  gruneisen_eq(gas)

    INPUT:
        gas = working gas object (restored to original state at end of function)

    OUTPUT:
        G_eq = equilibrium Gruneisen coefficient [-de/dp)_{v,eq} =
                -(v/T)dT/dv)_{s,eq} = + (rho/T)(dT/d rho)_{s,eq}]
```

gruneisen_fr Computes the frozen Grüneisen coefficient by using a centered finite difference approximation and evaluating frozen states on the isentrope passing through the reference (S, V) state supplied by the gas object passed to the function.

gruneisen_fr.m

```
FUNCTION SYNTAX:
    G_fr =  gruneisen_fr(gas)

INPUT:
    gas = working gas object (not modified in function)

OUTPUT:
    G_fr = frozen Gruneisen coefficient [-de/dp)_{v,x0} =
            -(v/T)dT/dv)_{s,x0} = + (rho/T)(dT/d rho)_{s,x0}]
```

`gruneisen_fr` in thermo.py

```
     FUNCTION SYNTAX:
        G_fr =  gruneisen_fr(gas)

    INPUT:
        gas = working gas object (not modified in function)

    OUTPUT:
        G_fr = frozen Gruneisen coefficient [-de/dp)_{v,x0} =
                -(v/T)dT/dv)_{s,x0} = + (rho/T)(dT/d rho)_{s,x0}]
```

Internal Functions called as part of iteration process.

shk_calc Calculates frozen post-shock state using Reynolds iterative method (see Section 8.2).
MATLAB Function - shk_calc.m
Python Function - shk_calc (in postshock.py)

```
SYNTAX:
[gas] = shk_calc(U1,gas,gas1,ERRFT,ERRFV)
INPUT:
U1 = shock speed (m/s)
gas = working gas object
gas1 = gas object at initial state
ERRFT,ERRFV = error tolerances for iteration
OUTPUT:
gas = gas object at frozen post-shock state
```

shk_eq_calc Calculates equilibrium post-shock state using Reynolds iterative method (see Section 8.2).
MATLAB Function - shk_eq_calc.m
Python Function - shk_calc (in postshock.py)

```
SYNTAX: [gas] = shk_eq_calc(U1,gas,gas1,ERRFT,ERRFV)
INPUT:
U1 = shock speed (m/s)
gas = working gas object
gas1 = gas object at initial state
ERRFT,ERRFV = error tolerances for iteration
OUTPUT:
gas = gas object at equilibrium post-shock state
```

FHFP

Uses the momentum and energy conservation equations to calculate error in current pressure and the enthalpy guesses (see (8.16) & (8.15)). In this case, state 2 is frozen.
MATLAB Function - FHFP.m
Python Function - FHFP (in postshock.py)

```
SYNTAX:
[FH,FP] = FHFP(U1,gas,gas1)
INPUT:
U1 = shock speed (m/s)
gas = working gas object
gas1 = gas object at initial state
OUTPUT:
FH,FP = error in enthalpy and pressure
```

FHFP_reflected_fr

Uses the momentum and energy conservation equations to calculate error in current pressure and the enthalpy guesses (see (8.16) & (8.15)). In this case, state 3 is frozen.
MATLAB Function - FHFP_reflected_fr.m
Python Function - FHFP_reflected_fr (in reflections.py)

```
SYNTAX:
[FH,FP] = FHFP_reflected_fr(u2,gas3,gas2)
INPUT:
u2 = current post-incident-shock lab frame particle speed
gas3 = working gas object
gas2 = gas object at post-incident-shock state (already computed)
OUTPUT:
FH,FP = error in enthalpy and pressure
```

CJ_calc

Calculates the wave speed for the Chapman-Jouguet case using Reynolds' iterative method (see Section 8.2).
MATLAB Function - CJ_calc.m
Python Function - CJ_calc (in postshock.py)

```
SYNTAX:
[gas,w1] = CJ_calc(gas,gas1,ERRFT,ERRFV,x)
INPUT:
gas = working gas object
gas1 = gas object at initial state
ERRFT,ERRFV = error tolerances for iteration
x = density ratio
OUTPUT:
gas = gas object at equilibrium state
w1 = initial velocity to yield prescribed density ratio
```

state

Calculates frozen state given T and ρ.
MATLAB Function - state.m
Python Function - state (in thermo.py)

```
SYNTAX:
[P,H] = state(gas,r1,T1)
INPUT:
gas = working gas object
r1,T1 = desired density and temperature
OUTPUT:
```

```
P,H = pressure and enthalpy
```

eq_state

Calculates equilibrium state given T and ρ.
MATLAB Function - eq_state.m
Python Function - eq_state (in thermo.py)

```
SYNTAX:
[P,H] = eq_state(gas,r1,T1)
INPUT:
gas = working gas object
r1,T1 = desired density and temperature
OUTPUT:
P,H = equilibrium pressure and enthalpy at constant temperature and specific volume
```

hug_eq

Algebraic expressions of equilibrium (product) Hugoniot pressure and enthalpy. Passed to root solver 'fsolve'.
MATLAB Function - hug_eq.m
Python Function - hug_eq (in postshock.py)

```
SYNTAX:
[x,fval] = fsolve(@hug_eq,Ta,options,gas,array)
INPUT:
Ta = initial guess for equilibrium Hugoniot temperature (K)
options = options string for fsolve
gas = working gas object
array = array with the following values
   vb = desired equilibrium Hugoniot specific volume (m^3/kg)
   h1 = enthalpy at state 1 (J/kg)
   P1 = pressure at state 1 (Pa)
   v1 = specific volume at state 1 (m^3/kg)
OUTPUT:
x = equilibrium Hugoniot temperature corresponding to vb (K)
fval = value of function at x
```

hug_fr

Algebraic expressions of frozen (reactant) Hugoniot pressure and enthalpy. Passed to root solver 'fsolve'.
MATLAB Function - hug_fr.m
Python Function - hug_fr (in postshock.py)

```
SYNTAX:
[x,fval] = fsolve(@hug_fr,Ta,options,gas,array)
INPUT:
Ta = initial guess for frozen Hugoniot temperature (K)
options = options string for fsolve
gas = working gas object
array = array with the following values
   vb = desired frozen Hugoniot specific volume (m^3/kg)
   h1 = enthalpy at state 1 (J/kg)
   P1 = pressure at state 1 (Pa)
   v1 = specific volume at state 1 (m^3/kg)
OUTPUT:
x = frozen Hugoniot temperature corresponding to vb (K)
fval = value of function at x
```

LSQ_CJspeed

Determines least squares fit of parabolic data.
MATLAB Function - N/A
Python Function - LSQ_CJspeed (in postshock.py)

```
SYNTAX:
[a,b,c,R2,SSE,SST] = LSQ_CJspeed(x,y)
INPUT:
x = independent data points
y = dependent data points
OUTPUT:
a,b,c = coefficients of quadratic function (ax^2 + bx + c = 0)
R2 = R-squared value
SSE = sum of squares due to error
SST = total sum of squares
```

PostReflectedShock_eq

Calculates equilibrium post-reflected-shock state for a specified shock velocity.
MATLAB Function - PostReflectedShock_eq.m
Python Function - PostReflectedShock_eq (in reflections.py)

```
FUNCTION SYNTAX:
    [gas3] = PostReflectedShock_eq(u2,gas2,gas3)

INPUT:
    u2 = current post-incident-shock lab frame particle speed
    gas2 = gas object at post-incident-shock state (already computed)
    gas3 = working gas object

OUTPUT:
    gas3 = gas object at equilibrium post-reflected-shock state
```

PostReflectedShock_fr

Calculates frozen post-reflected-shock state for a specified shock velocity.
MATLAB Function - PostReflectedShock_fr.m
Python Function - PostReflectedShock_fr (in reflections.py)

```
SYNTAX:
[gas3] = PostReflectedShock_fr(u2,gas2,gas3)
INPUT:
u2 = current post-incident-shock lab frame particle speed
gas2 = gas object at post-incident-shock state (already computed)
gas3 = working gas object
OUTPUT:
gas3 = gas object at frozen post-reflected-shock state
```

Utilities Plotting and output routines

znd_plot Creates four plots from the solution to a ZND detonation: temperature, pressure, Mach number, and thermicity vs. distance. Optionally, also creates plots of species mass fraction vs. time, for given lists of major or minor species. If major_species= 'All', all species will be plotted together.

znd_fileout Creates 2 formatted text files to store the output of the solution to a ZND detonation. Includes a timestamp of when the file was created, input conditions, and tab-separated columns of output data.

cv_plot Creates two subplots from the solution to a CV explosion: temperature vs. time, and pressure vs. time. Optionally, also creates plots of species mass fraction vs. time, for given lists of major or minor species. If major_species='All', all species will be plotted together.

CJspeed_plot Creates two plots of the CJspeed fitting routine: both display density ratio vs. speed. The first is very "zoomed in" around the minimum, and shows the quadratic fit plotted through the calculated points. The second shows the same fit on a wider scale, with the minimum and its corresponding speed indicated.

Error Control and Limits Setting iteration error and volume limits

Three parameters control the convergence and bounds on the specific volume for the Newton-Raphson iteration used to solve the jump conditions. These are specified in files located in the SDToolbox directory:

MATLAB Function - SDTconfig.m
Python Function - config.py

The default values of these parameters are:

```
ERRFT = 1e-4;
ERRFV = 1e-4;
volumeBoundRatio = 5;
```

The values of the error parameters represent the maximum relative errors allowed for convergence of shock and detonation jump condition computations, see the discussion in Section 8.1. Iteration ceases and the solution is returned when the conditions $\Delta T/T <$ `ERRFT` and $\Delta v/v <$ `ERRFV` are both met.

The value of `volumeBoundRatio` is the lower bound on specific volume ratio v_1/v_2 used as a starting point for the iteration. For shock waves in gases with a high specific heat, higher values of `volumeBoundRatio` may be required in order to get solutions but care must be taken not to select `volumeBoundRatio` larger than the maximum value possible on the Hugoniot. The perfect gas analytical solution for strong shock is a useful estimate if the ratio of specific heats γ is known.

$$\frac{v_1}{v_{2,min}} \geq \frac{\gamma+1}{\gamma-1} \tag{12.1}$$

Chapter 13

Demonstration Programs

A number of demonstration programs are provided with the Shock and Detonation Toolbox. These show how Cantera and the SDT routines can be used to carry out a variety of calculations. The programs are available in the `demos` subdirectories in the Python and MATLAB branches of the distribution. The links to the MATLAB versions are in given in the following list. Python version of all demonstration programs are also available and have identical names except for the extension `.py` instead of `.m`.

demo_CJ.py demo_CJ.m Computes CJ speed.

demo_CJ_and_shock_state.py demo_CJ_and_shock_state.m Computes 2 reflection conditions. 1) equilibrium post-initial-shock state behind a shock traveling at CJ speed (CJ state) followed by equilibrium post-reflected-shock state 2) frozen post-initial-shock state behind a CJ wave followed by frozen post-reflected-shock state

demo_CJstate.py demo_CJstate.m Computes CJ speed and CJ state.

demo_CJstate_isentrope.py demo_CJstate_isentrope.m Computes CJ speed, CJ state, isentropic expansion in 1-D Taylor wave, plateau state conditions.

demo_cv.m Generates plots for a constant volume explosion simulation with specified initial conditions. Outputs metrics on induction time, reaction pulse, effective activation energy and reaction order.

demo_cv_comp.py demo_cv_comp.m Generates plots and output files for a constant volume explosion simulation where the initial conditions are adiabaically compressed reactants.

demo_cvCJ.py demo_cvCJ.m Generates plots and output files for a constant volume explosion simulation where the initial conditions are given by the postshock conditions for a CJ speed shock wave.

demo_cvshk.py demo_cvshk.m Generates plots and output files for a constant volume explosion simulation where the initial conditions are given by the postshock conditions for shock wave traveling at a user specified speed.

demo_detonation_pu.py demo_detonation_pu.m Computes the Hugoniot and pressure-velocity $(P-U)$ relationship for shock waves centered on the CJ state. Generates an output file.

demo_equil.py demo_equil.m Computes the equilibrium state at constant (T, P) over a range of temperature for a fixed pressure and plots composition.

demo_EquivalenceRatioSeries.py demo_EquivalenceRatioSeries.m - An example of how to vary the equivalence ratio over a specified range and for each resulting composition, compute constant volume explosion and ZND detonation structure. This example creates a set of plots and an output file.

demo_exp_state.py demo_exp_state.m Calculates mixture properties for explosion states (UV,HP, TP).

demo_ExplosionSeries.py demo_ExplosionSeries.m How to compute basic explosion parameters as a function of concentration of one component for given mixture. Creates plots and output file.

demo_g.py demo_g.m Compares methods of computing ratio of specific heats and logarithmic isentrope slope using several approaches and compares the results graphically.

demo_GasPropAll.py demo_GasPropAll.m Mixture thermodynamic and transport properties of gases at fixed pressure as a function of temperature. Edit to choose either frozen or equilibrium composition state. The mechanism file must contain transport parameters for each species and specify the transport model 'Mix'.

demo_oblique.py demo_oblique.m Calculates shock polar using FROZEN post-shock state based the initial gas properties and the shock speed. Plots shock polar using three different sets of coordinates.

demo_overdriven.py demo_overdriven.m Computes detonation and reflected shock wave pressure for overdriven waves. Both the post-initial-shock and the post-reflected-shock states are equilibrium states. Creates output file.

demo_OverdriveSeries.py demo_OverdriveSeries.m This is a demonstration of how to vary the Overdrive (U/U_{CJ})) in a loop for constant volume explosions and ZND detonation simulations.

demo_PrandtlMeyer.py demo_PrandtlMeyer.m Calculates Prandtl-Meyer function and polar. Creates plots of polars.

demo_PrandtlMeyer_CJ.py demo_PrandtlMeyer_CJ.m Calculates Prandtl-Meyer function and polar expanded from CJ state. Creates plots of polars and fluid element trajectories.

demo_PrandtlMeyerDetn.py demo_PrandtlMeyerDetn.m Calculates Prandtl-Meyer function and polar originating from CJ state. Calculates oblique shock wave moving into expanded detonation products or a specified bounding atmosphere. Creates a set of plots, evaluates axial flow model for rotating detonation engine.

demo_PrandtlMeyerLayer.py demo_PrandtlMeyerLayer.m Calculates Prandtl-Meyer function and polar originating from lower layer postshock state. Calculates oblique shock wave moving into expanded detonation products or a specified bounding atmosphere. Two-layer version with arbitrary flow in lower layer (1), oblique wave in upper layer (2). Upper and lower layers can have various compositions as set by user.

demo_precompression_detonation.py demo_precompression_detonation.m Computes detonation and reflected shock wave pressure for overdriven waves. Varies density of initial state and detonation wave speed. Creates an output file.

demo_PressureSeries.py demo_PressureSeries.m Properties computed as a function of initial pressure for a constant volume explosions and ZND detonation simulations Creates a set of plots and an output file.

demo_PSeq.py demo_PSeq.m Calculates the equilibrium post shock state based on the initial gas state and the shock speed.

demo_PSfr.py demo_PSfr.m Calculates the frozen postshock state based on the initial gas state and the shock speed.

demo_quasi1d_eq.py demo_quasi1d_eq.m Computes ideal quasi-one dimensional flow using equilibrium properties to determine exit conditions for expansion to a specified pressure. Carries out computation for a range of helium dilutions.

demo_reflected_eq.py demo_reflected_eq.m Calculates post-relected-shock state for a specified shock speed speed and a specified initial mixture. In this demo, both shocks are reactive, i.e. the computed states behind both the incident and reflected shocks are equilibrium states.

demo_reflected_fr.py demo_reflected_fr.m Calculates post-relected-shock state for a specified shock speed speed and a specified initial mixture. In this demo, both shocks are frozen, i.e. there is no composition change across the incident and reflected shocks.

demo_RH.py demo_RH.m Creates arrays for Rayleigh Line with specified shock speed, Reactant, and Product Hugoniot Curves for H_2-air mixture. Options to creates output file and plots.

demo_RH_air.py demo_RH_air.m Creates arrays for Rayleigh Line with specified shock speed and frozen Hugoniot Curve for a shock wave in air. Options to create output file and plot.

demo_RH_air_eq.py demo_RH_air_eq.m Creates arrays for Rayleigh Line with specified shock speed in air, frozen and equilibrium Hugoniot curves. Options to create output file and plot.

demo_RH_air_isentropes.py demo_RH_air_isentropes.m Creates arrays for frozen Hugoniot curve for shock wave in air, Rayleigh Line with specified shock speed, and four representative isentropes. Options to create plot and output file.

demo_RH_CJ_isentropes.py demo_RH_CJ_isentropes.m Creates plot for equilibrium product Hugoniot curve near CJ point, Shows Rayleigh Line with slope U_{CJ} and four isentropes bracketing CJ point. Creates plot showing Gruneisen coefficient, denominator in Jouguet's rule, isentrope slope.

demo_rocket_impulse.py demo_rocket_impulse.m Computes rocket performance using quasi-one dimensional isentropic flow using both frozen and equilibrium properties for a range of helium dilutions in a hydrogen-oxygen mixture. Plots impulse as a function of dilution.

demo_RZshock.py demo_RZshock.m Generate plots and output files for a reaction zone behind a shock front traveling at a user specified speed.

demo_shock_adiabat.py demo_shock_adiabat.m Generates the points on a frozen shock adiabat and creates an output file.

demo_shock_point.py demo_shock_point.m This is a demonstration of how to compute frozen and equilibrium postshock conditions for a single shock Mach number. Computes transport properties and thermodynamic states.

demo_shock_state_isentrope.m Computes frozen post-shock state and isentropic expansion for specified shock speed. Create plots and output file.

demo_ShockTube.py demo_ShockTube.m Calculates the solution to ideal shock tube problem. Three cases possible: conventional nonreactive driver (gas), constant volume combustion driver (uv), CJ detonation (initiate at diaphragm) driver (cj).

demo_STGshk.py demo_STGshk.m Generate plots and output files for a steady reaction zone between a shock and a blunt body using the model of linear profile of mass flux ρw on stagnation streamline.

demo_STG_RZ.py demo_STG_RZ.m Compare propagating shock and stagnation point profiles using transformation methodology of Hornung.

demo_TP.py demo_TP.m Explosion computation simulating constant temperature and pressure reaction. Reguires function `tpsys.m` for ODE solver

demo_TransientCompression.py demo_TransientCompression.m Explosion computation simulating adiabatic compression ignition with control volume approach and effective piston used for compression. Requires `adiasys.m` function for ODE solver.

demo_vN_state.py demo_vN_state.m Calculates the frozen shock (vN = von Neumann) state of the gas behind the leading shock wave in a CJ detonation.

demo_ZNDCJ.py demo_ZNDCJ.m Solves ODEs for ZND model of detonation structure. Generate plots and output files for a for a shock front traveling at the CJ speed.

demo_ZNDshk.py demo_ZNDshk.m Solves ODEs for ZND model of detonation structure. Generate plots and output files for a for a shock front traveling at a user specified speed U.

demo_ZND_CJ_cell.py demo_ZND_CJ_cell.m Computes ZND and CV models of detonation with the shock front traveling at the CJ speed. Evaluates various measures of the reaction zone thickness and exothermic pulse width, effective activation energy and Ng stability parameter.

Chapter 14

Utility Programs

Checking and Updating Data

thermo_check.py This Python script scans a Cantera .cti mechanism file to determine the size of jumps in thermodynamic properties and derivatives. Identifies species with largest Cp/R jump. Provides routines for finding all jumps and plotting thermodynamic properties of individual species. Only works for NASA-7 polynomials with the current version of Cantera 2.3 and 2.4

thermo_refit.m Refits thermodynamic data to eliminate jumps in properties at midpoint temperature. Works with a list of species created by thermo_check.py or individual species specified by user. Creates a new NASA-7 fit and data structure for polynomial coefficients, writes output files in three formats (cti, NASA-7 and NASA-9).

thermo_replace.m Reads new thermodynamic data fits generated by thermo_refit.m and batch processes replaces the data in the NASA format data file using the list generated by check_thermo.py. Currently only works for NASA-7 polynomials.

thermo_fit.m fit tabular thermodynamic data to generate NASA-7 polynomial fits and writes files in three formats. An example input file is provided for **2-butenal**

Statistical Thermodynamics

partition_rotvib.m. Evaluation of the partition function for heteronuclear diatomic molecules and the resulting thermodynamic properties. Creates plots and output files, data files in the form of Cantera cti file, NASA 7 and NASA 9 formats.

The spectroscopic data needed to compute the energy levels is provided in files for three molecules.

NO_rotvib.m Nitric oxide (NO), first 15 electronic states.

OH_rotvib.m Hydroxyl (OH), first 4 electronic states.

CH_rotvib.m Methylidyne (CH), first 6 electronic states.

Chapter 15

Hints and Tips

The routines provided in the Toolbox are reasonably robust but do not always yield the correct answer or even result in errors that cause the programs halt with error messages. Usually these issues can be resolved by adjusting the parameters that control the routine algorithms. In some cases, you may have to investigate the innards of the toolbox and do some diagnostic work. The routines are not particularly sophisticated and should be considered "research" grade software which does not attempt to prevent user errors or provide particularly verbose error messages. On the other hand, these have been used by successfully by many generations of students and professional researchers. Based on these experiences here are some observations about possible problems and tips for solutions.

Jump Conditions

For certain cases, the jump conditions will not converge. In those instances, it may be necessary to adjust the error bounds and convergence parameters.

Three parameters control the convergence and bounds on the specific volume for the Newton-Raphson iteration used to solve the jump conditions. These are specified in files located in the SDToolbox directory:

MATLAB Function - SDTconfig.m
Python Function - config.py

The default values of these parameters are:

```
ERRFT = 1e-4;
ERRFV = 1e-4;
volumeBoundRatio = 5;
```

The values of the error parameters represent the maximum relative errors allowed for convergence of shock and detonation jump condition computations, see the discussion in Section 8.1. Iteration ceases and the solution is returned when the conditions $\Delta T/T <$ `ERRFT` and $\Delta v/v <$ `ERRFV` are both met.

The value of `volumeBoundRatio` is the lower bound on specific volume ratio v_1/v_2 used as a starting point for the iteration. For shock waves in gases with a high specific heat, higher values of `volumeBoundRatio` may be required in order to get solutions but care must be taken not to select `volumeBoundRatio` larger than the maximum value possible on the Hugoniot. The perfect gas analytical solution for strong shock is a useful estimate if the ratio of specific heats γ is known.

$$\frac{v_1}{v_{2,min}} \geq \frac{\gamma+1}{\gamma-1} \tag{15.1}$$

In rare instances, Cantera may fail to converge to an equilibrium composition. The equilibrium solvers are fairly robust but you may find that there are particular combinations of stoichiometry and thermodynamic state, particularly for exothermic mixtures, that halt with error messages. This can often be solved by