Table of Contents

WI Space Race Rocket Parameterization Code

Author: Brandon Wilson (bwilson8@wisc.edu)

This script solves the equation of motion of a sounding rocket during the vertical ascent phase of flight. The eom_rocket function is called to solve the EOM using several linear and non-linear equations to provide relatively accurate predictions with minimal paramaters

```
clear; clc; close all;

global ft2m psi2Pa

% Simulation time
simTime = 230;
```

Rocket Parameters

```
%---------------- feel free to change these parameters
 -------------------

m_veh_structure = 120;    % dry mass of rocket [kg]
m_prop = 300;            % Initial propellant mass, kg
r_veh = 7/12*ft2m;       % radius of vehicle [m]
epsilon =8;            % expansion ratio of nozzle
```

```
F_T_max = 15e3;          % Thrust when exit pressure = atm pressure
r_tank = 6/12;           % tank radius [ft]

% feed pressure parameters
inj_percentDrop = 25;  % pressure drop across injector [% of chamber
 press]
catBed_pressDrop = 100*psi2Pa;  % pressure drop across catalyst bed
 [Pa]
ox_lineDiameter = 1.5-2*0.083;   % ox feed line diameter [in]
f_lineDiameter = 3/8-2*0.035; % fuel feed line diameter [in]

% from CEARUN
P0_eng = 2.76e6;
T0_eng = 2734;
M_bar_prop = 21.9;
gamma_prop = 1.14;

% call altitude prediction model
for i = 1:length(epsilon)
[xx, R_x, Pe(i)] =
 altitude_prediction(simTime,r_veh,r_tank,m_veh_structure,...
    m_prop,F_T_max,epsilon(i),P0_eng,T0_eng,M_bar_prop,gamma_prop,...
    inj_percentDrop,catBed_pressDrop,ox_lineDiameter,f_lineDiameter);
end
% for i = 1:9
% figure(i)
% legend(['Expansion Ratio: ',num2str(epsilon(1))],['Expansion Ratio:
 ',num2str(epsilon(2))],...
%     ['Expansion Ratio: ',num2str(epsilon(3))],['Expansion Ratio:
 ',num2str(epsilon(4))],...
%     ['Expansion Ratio: ',num2str(epsilon(5))],['Expansion Ratio:
 ',num2str(epsilon(6))],...
%     ['Expansion Ratio: ',num2str(epsilon(7))],['Expansion Ratio:
 ',num2str(epsilon(8))])
% end
%
% figure(10)
% plot(epsilon,Pe/psi2Pa,'d')
% xlabel('Expansion Ratio')
% ylabel('Exit Pressure [psia]')
```

Rocket Altitude Predictions

main function that calls sub functions and does plotting

```
function [xx, R_x, Pe] =
 altitude_prediction(sim_t,R,r_tank,m_struct,m_propellant,...
    F_T,epsilon,P0,T0,M_bar,gamma,inj_percentDrop,catBed_pressDrop,...
    ox_lineDiameter,f_lineDiameter)

global A_veh g m_veh_empty eng_time m_dot r_veh R_bar M_bar_prop Ae...
    At gamma_prop P0_eng Pe_eng T0_eng n_rao expansionRatio ue Isp...
    m_veh_wet ft2m psi2Pa in2m Pa2psi
```

Universal Constants

```
g = 9.807;                  % gravitational constant [m/s^2]
R_bar = 8314.3;             % universal gas constant [J/kmol-K]
kg2lbm = 2.20462;           % kg to lbm conversion
lbm2kg = 0.453592;          % lbm to kg conversion
Pa2psi = 0.000145038;       % Pascal to psi conversion
psi2Pa = 6894.76;           % psi to Pascal conversion
m2ft = 3.28084;             % meter to foot conversion
ft2m = 0.3048;              % foot to meter conversion
in2m = 0.0254;              % inch to meter conversion
kN2lbf = 224.81;            % kN to lbf conversion
```

Defining variable from input

```
m_veh_structure = m_struct;
simTime = sim_t;
m_prop = m_propellant;
P0_eng = P0;
T0_eng = T0;
M_bar_prop = M_bar;
F_T_max = F_T;
gamma_prop = gamma;
expansionRatio = epsilon;
r_veh = R;
```

Initial Calculations

```
% mass flow, throat area, and exit area for engine
[Pe_eng, ue, m_dot] = GetMassFlowAndNozzleDimensions(F_T_max);
eng_time = m_prop/m_dot; % burn time
Pe = Pe_eng

% Propellant tank calculations
rho_ox = 1400; % oxidizer density [kg/m^3]
rho_f = 810;   % fuel density [kg/m^3]
OF = 7; % ox:fuel ratio

m_dot_ox = (OF * m_dot) / (1 + OF);  % mass flow of oxidizer [kg/s]
m_dot_f = m_dot - m_dot_ox;  % mass flow of fuel [kg/s]

inj_PressDrop = inj_percentDrop/100*P0_eng; % pressure drop across
 injector

ox_lineLength = 24; % guess for ox run line length [in]
% oxidizer tank pressure
ox_tankPressure = 2*GetFeedPress(ox_lineDiameter*in2m,P0_eng,...
    catBed_pressDrop,inj_PressDrop,m_dot_ox,rho_ox,ox_lineLength);

f_lineLength = 144; % guess for fuel run line length [in]
% fuel tank pressure
f_tankPressure = 2*GetFeedPress(f_lineDiameter*in2m,P0_eng,...
```

```
    catBed_pressDrop*0,inj_PressDrop,m_dot_f,rho_f,f_lineLength);

% get tank volumes
[vol_ox,vol_f,m_ox,m_f] = GetPropTankVol(OF,m_prop,rho_ox,rho_f);

% get ox tank size and weight estimates
[ox_tankWt,ox_tank_length,ox_t_cyl,ox_t_hemi,ox_l_tank_cyl]...
    = propTankWt_Size(vol_ox*m2ft^3,r_tank,ox_tankPressure*Pa2psi);

% get fuel tank size and weight estimates
[f_tankWt,f_tank_length,f_t_cyl,f_t_hemi,f_l_tank_cyl]...
    = propTankWt_Size(vol_f*m2ft^3,r_tank,f_tankPressure*Pa2psi);
tankWt = ox_tankWt + f_tankWt;

m_tanks = tankWt*lbm2kg; % propellant tank total

m_veh_empty = m_veh_structure+m_tanks;  %empty mass of vehicle [kg]
m_veh_wet = m_veh_empty+m_prop;      % wet mass of vehicle [kg]
A_veh = pi*r_veh^2;                % Cross sectional area of vehicle
 [m^2]
```

Solving EOM

Initial Conditions

```
x0 = 0;
v0 = 0;
IC = [x0;v0];
t_span = [0,simTime];

% Call ode45 to solve EOM
options = odeset('RelTol',1e-13,'AbsTol',1e-13);
[tout, yout] = ode45('eom_rocket',t_span,IC);

% only want data up to max altitude
for i = 1:length(tout)
    if(yout(i,2) >= 0)
        alt(i) = yout(i,1);
        vel(i) = yout(i,2);
        time(i) = tout(i);
    else
        break;
    end
end

% Back Solving for accleration, Force of Drag, Cd, Ma, Re
acc_num = zeros(length(time),1);
F_d = zeros(length(time),1);
F_T = zeros(length(time),1);
Ma = zeros(length(time),1);
Re = zeros(length(time),1);
rho = zeros(length(time),1);
Cd = zeros(length(time),1);
```

```
% back out acceleration, drag, Mach and Reynolds
for i = 1:length(time)
    [acc_num(i),F_T(i),F_d(i),Ma(i),Re(i),rho(i), Cd(i)] =
 BackoutPerformance...
        (time(i),alt(i), vel(i));
end
```

Nozzle geometry

```
n_rao = 200;            % number of mesh points
R_t = sqrt(At*4/pi)/2;  % throat radius [m]
epsilon = Ae/At;        % expansion ratio
Lf_ratio = 0.9;         % conical nozzle fraction
theta_E = 8*pi/180;     % exit angle
theta_N = 23*pi/180;    % diverging angle
theta_C = 30*pi/180;    % converging angle

% Rao parabolic approximation
[xx, coeffs, R_x] = Rao(R_t, epsilon,
 Lf_ratio,theta_E,theta_N,theta_C);

% converting to inches
R_x_in = R_x*39.3701;
xx_in = xx*39.3701;
```

Plotting

```
figure(1)
hold on;
plot(time,alt/1000,'LineWidth',1.7)
title('Altitude Vs Time','FontSize',13)
xlabel('Time [s]','FontSize',12)
ylabel('Altitude [km]','FontSize',12)
grid on;

figure(2)
hold on;
plot(time,vel,'LineWidth',1.7)
grid on;
title('Vertical Velocity Vs Time', 'FontSize',13)
xlabel('Time [s]', 'FontSize',12)
ylabel('Velocity [m/s]', 'FontSize',12)

figure(3)
hold on;
plot(time,acc_num/g,'LineWidth',1.7)
grid on;
title('Acceleration Vs Time', 'FontSize',13)
xlabel('Time [s]', 'FontSize',12)
ylabel('Acceleration [g]', 'FontSize',12)

figure(4)
```

```
hold on;
plot(time,F_d,'LineWidth',1.7)
grid on;
title('Force of Drag Vs time', 'FontSize',13)
xlabel('time [s]', 'FontSize',12)
ylabel('Force of Drag [N]', 'FontSize',12)

figure(5)
hold on;
plot(time,Ma,'LineWidth',1.7)
grid on;
title('Mach # Vs Time', 'FontSize',13)
xlabel('Time [s]', 'FontSize',12)
ylabel('Mach #', 'FontSize',12)

figure(6)
hold on;
plot(alt,F_T/1000,'LineWidth',1.7)
grid on;
title('Thurst Vs Altitude', 'FontSize',13)
xlabel('Altitude [km]', 'FontSize',12)
ylabel('Thrust [kN]', 'FontSize',12)

figure(7)
hold on;
plot(xx_in,R_x_in,'b-','LineWidth',1.7)
plot(xx_in,-R_x_in,'b-','LineWidth',1.7)
grid on;
title('Nozzle Dimensions', 'FontSize',13)
xlabel('x [in]', 'FontSize',12)
ylabel('Radius [in]', 'FontSize',12)
axis equal

figure(8)
hold on;
plot(Ma,Cd,'d','LineWidth',1.2)
grid on;
title('Drag Coefficient vs Mach #', 'FontSize',13)
xlabel('Mach #', 'FontSize',12)
ylabel('Cd [-]', 'FontSize',12)

figure(9)
hold on;
plot(alt/1000,F_d/1000,'d','LineWidth',1.2)
grid on;
title('Drag vs Altitude', 'FontSize',13)
xlabel('altitude [km]', 'FontSize',12)
ylabel('Drag [kN]', 'FontSize',12)
```

Results output

```
% max drag force
max_Fd = max(F_d);
```

```
% max dynamic pressure
q = zeros(length(time),1);
for i = 1:length(vel)
   q(i) = 1/2*rho(i)*vel(i)^2;
end
maxQ = max(q);

% max velocity
maxVelocity = max(yout(:,2));

% max Mach #
maxMach = max(Ma);

% specific impulse
Isp = ue/g;

clc;
disp('------------------------ Rocket Parameters
 ------------------------')
fprintf('Vehicle dry weight: %3.2f [lb]', (m_veh_empty)*kg2lbm)
fprintf('\nVehicle wet weight: %3.2f [lb]', (m_veh_wet)*kg2lbm)
fprintf('\nVehicle diameter: %3.2f [ft]', 2*r_veh*m2ft)
fprintf('\nPropellant mass (total): %3.2f [kg], %3.2f [lb]',...
    m_prop,m_prop*kg2lbm)
fprintf('\nPeroxide mass: %3.2f [kg], %3.2f [lb]',...
    m_ox,m_ox*kg2lbm)
fprintf('\nKerosene mass: %3.2f [kg], %3.2f [lb]',...
    m_f,m_f*kg2lbm)
fprintf('\nPropellant mass fraction: %3.2f [-]', m_prop/m_veh_wet)
fprintf('\nPropellant tank Weight : %3.2f [lb]', tankWt)
fprintf('\nPeroxide tank Wall thickness : %3.2f [in]', ox_t_cyl)
fprintf('\nKerosene tank Wall thickness : %3.2f [in]', f_t_cyl)
fprintf('\nPeroxide tank length (inner dim): %3.2f [ft]',
 ox_tank_length)
fprintf('\nKerosene tank length (inner dim): %3.2f [ft]',
 f_tank_length)
fprintf('\nPeroxide tank volume: %3.2f [m^3]', vol_ox)
fprintf('\nKerosene tank volume: %3.2f [m^3]', vol_f)
fprintf('\nPeroxide tank max operating pressure: %3.2f [psi]',...
    ox_tankPressure*Pa2psi)
fprintf('\nKerosene tank max operating pressure: %3.2f [psi]',...
    f_tankPressure*Pa2psi)
fprintf('\nPeroxide tank mean operating pressure: %3.2f [psi]',...
    ox_tankPressure*Pa2psi/2)
fprintf('\nKerosene tank mean operating pressure: %3.2f [psi]',...
    f_tankPressure*Pa2psi/2)
fprintf('\n')
```

Engine Paramaters

```
disp('---------------------- Propulsion Parameters
 ---------------------')
```

```
fprintf('Sea level thrust: %3.2f [kN], %3.2f [lbf]', F_T(1)/1000,...
    F_T(1)/1000*kN2lbf)
fprintf('\nExpansion ratio: %3.2f [-]', expansionRatio)
fprintf('\nChamber Pressure: %3.2f [MPa], %3.2f [psia]',
 P0_eng*1e-6,...
    P0_eng*Pa2psi)
fprintf('\nExit Pressure: %3.2f [MPa], %3.2f [psia]', Pe_eng*1e-6,...
    Pe_eng*Pa2psi)
fprintf('\nChamber temperature %3.2f [K]', T0_eng)
fprintf('\nMolar mass of propellant: %3.2f [kg/kmol]', M_bar_prop)
fprintf('\nSpecific heat ratio of propellant (frozen-flow) %3.2f
 [-]',...
    gamma_prop)
fprintf('\nPressure at catalyst bed: %3.2f [psia]',...
    (P0_eng+catBed_pressDrop+inj_PressDrop)*Pa2psi)
fprintf('\nPressure at injector: %3.2f [psia]',...
    (P0_eng+catBed_pressDrop)*Pa2psi)
fprintf('\nMass flow rate (total): %3.2f [kg/s]', m_dot)
fprintf('\nMass flow rate (oxidizer): %3.2f [kg/s], %3.2f [lb/s]',...
    m_dot_ox,m_dot_ox*kg2lbm)
fprintf('\nMass flow rate (fuel): %3.2f [kg/s], %3.2f [lb/s]',
 m_dot_f,...
    m_dot_f*kg2lbm)
fprintf('\nNozzle exit diameter: %3.2f [in]', 2*R_x(length(R_x))/in2m)
fprintf('\nThroat Diameter: %3.2f [in]', 2*R_t/in2m)
fprintf('\nPeroxide feed line diameter %3.2f [in]',ox_lineDiameter)
fprintf('\nKerosene feed line diameter %3.2f [in]',f_lineDiameter)
fprintf('\nBurn time %3.2f [s]',eng_time)
fprintf('\n\n')

disp('----------------------------- Results
 -----------------------------')
fprintf('Max Altitude: %3.2f [m], %3.2f [ft]',...
    max(yout(:,1)), 3.28*max(yout(:,1)))
fprintf('\nMax Velocity: %3.2f [m/s]', maxVelocity)
fprintf('\nMax Mach #: %3.2f [-]', maxMach)
fprintf('\nMax Thrust: %3.2f [kN], %3.2f [lbf]', max(F_T)/1000,...
max(F_T)/1000*kN2lbf)
fprintf('\nSpecific Impulse: %3.2f [s]', Isp)
fprintf('\nMax Drag Force: %3.2f [kN], %3.2f [lbf]', max_Fd/1000,...
    max_Fd/1000*kN2lbf)
fprintf('\nMax dynamic pressure: %3.2f [kPa]', maxQ/1000)
fprintf('\nActual expansion ratio %3.2f [-]',pi*R_x(length(R_x))^2/At)
fprintf('\n\n\n')
 total_impulse = max(cumtrapz(F_T)/4.448/1000)
 R_t/in2m*sqrt(3)

end
```

GetMassFlowAndNozzleDimensions Function

returns mass flow based on initial thrust

```
function [Pe,ue,m_dot] = GetMassFlowAndNozzleDimensions(F_T_0)
```

```
global P0_eng Pe_eng T0_eng expansionRatio gamma_prop R_bar M_bar_prop
 At...
    Ae Isp

epsilon = expansionRatio;
P0 = P0_eng;
gamma = gamma_prop;
T0 = T0_eng;
R = R_bar/M_bar_prop;
P_atm = 101.3e3;

% calculate Me from expansion ratio
Ae_Astar_expr = @(Me) 1./Me.*(2/(gamma+1)*(1+(gamma-1)/2*Me.^2))...
    .^((gamma+1)/(2*(gamma-1))) - epsilon;
Me = newtzero(Ae_Astar_expr, 1);

% 2 solutions, only want supersonic
if(length(Me) == 2)
    Me = Me(2);
end
% get exit pressure and temp
Pe = P0*(1+(gamma-1)/2*Me^2)^(gamma/(1-gamma));
Pe_eng = Pe;
Te = T0*(1+(gamma-1)/2*Me^2)^(-1);

% solving for throat area and m_dot
rho0 = P0/(R*T0);
rho_t = rho0*(1+(gamma-1)/2)^(-1/(gamma-1));
Tt = T0*(1+(gamma-1)/2)^(-1);
At = F_T_0/((Pe-P_atm)*epsilon+Me*rho_t*gamma*R*sqrt(Tt*Te));
m_dot = rho_t*At*sqrt(gamma*R*Tt);

% exit area, velocity and ISP
Ae = epsilon*At;
ue = Me*sqrt(gamma*R*Te);
Isp = ue/9.807;
end
```

eom_rocket Function

2nd order non-linear differential equation for vertical ascent and constant thrust

```
function [ydot] = eom_rocket(t,y)


global A_veh g m_veh_empty eng_time m_dot Ae Pe_eng ue m_veh_wet

massFlow = m_dot;
Pe = Pe_eng;
% altitude, temp and pressure as a function of altitude
[rho, T, P] = Get_rho_T_P(y(1));
```

```
% Get Cd, Ma and Re
[Cd, Ma, Re] = GetFlow(T,rho,y(2));

% Force due to drag [N]
F_d = 0.5* Cd * A_veh * rho * y(2)^2;

% Thrust is constant piecwise
if t < max(eng_time)

    % thrust
    F_T = massFlow*ue+(Pe-P)*Ae;
%     F_T = 20e3;

    % rocket mass decreases linearly with m_dot (assumed ~constant
 m_dot)
    m_veh = m_veh_wet-massFlow*t;

else
    F_T = 0;                % Thrust = 0 after motor burnout
    m_veh = m_veh_empty;    % Mass is constant after burnout
    massFlow = 0;
end


% returned ydot values
ydot(1,1) = y(2);
ydot(2,1) = F_T/m_veh-F_d/m_veh-massFlow*y(2)/m_veh-g;

end
```

Get_rho_T_P Function

returns temperature, pressure and density from an altitude input

```
function [rho,T,P] = Get_rho_T_P(alt)
% Model based on NASA.gov:
% https://www.grc.nasa.gov/www/k-12/airplane/atmosmet.html

%atmospheric definitions
Tropopause = 11000;     % Altitude of initial Tropopause reg [m]
lowerStrat = 25000;     % Altitude of lower stratosphere [m]

% Temp linearly decreases then becomes constant at tropopause
if(alt > lowerStrat)
    T = -131.21 + 0.00299*alt; %C
    P = 2.488*((T+273.15)/216.6)^(-11.388); % kPa
elseif(alt > Tropopause && alt < lowerStrat)
    T = -56.46; % C
    P = 22.65*exp(1.73-0.000157*alt); %kPa
else
    T = 15.04 - 0.00649*alt;    % C
    P = 101.29*((T+273.15)/288.08)^5.256; % kPa
```

```
end

% Air density as a function of pressure and temp [kg /m^3]
rho = P/(0.2869*(T+273.15));

% convert back to SI
P = P*1000; %Pa
T = T + 273.15; % K


end
```

Get_flow Function

returns drag coefficient, Mach and Reynolds #

```
function [Cd, Ma, Re] = GetFlow(T,rho,V)


global r_veh R_bar


gamma = 1.4;            % Ratio of specific heat for air
M_bar_air = 28.97;      % kg/kmol
R = R_bar/M_bar_air;    % Ideal gas constant [j/(Kg-K)]
c = sqrt(gamma*R*T);    % Speed of sound
Ma = norm(V/c);         % Mach Number
mu0 = 18.27e-6;         % Dynamic viscosity at 291.15 K [Pa-s]
T0 = 291.15;
C = 120;                % Sutherland constant
mu = mu0*((T0+C)/(T+C))*(T/T0)^(3/2);   % Dynamic viscosity
 (Sutherland's eq)
Re = rho*V*r_veh*2/mu; % Reynold's number

% approximationg drag coefficient from "rocket ans spacecraft
 propulsion"
% from Marting Turner (page 150). Esssentiall Cd is roughly constant
 in
% subsonic regime, then exponentially increases to mach 1, then
% exponentiall decreases.
a = 0.15;
b = 0.35;
if (Ma < 1)
    Cd = a + b*Ma^6;
else
    Cd = a + b/Ma^2;
end


end
```

GetPropTankVol Function

Returns tank volumes

```
function [vol_ox_total,vol_f_total,m_ox,m_f] = GetPropTankVol(OF,
 propellantMass,...
    rho_ox, rho_f)

% Inputs: OF: ox:fuel ratio, propellantMass: kg, rho_ox: kg/m^3,
 rho_f:
% kg/m^3
```

Volume Calcs

```
m_ox = (OF * propellantMass) / (1 + OF);  % mass oxidizer [kg]
m_f = propellantMass - m_ox;              % mass fuel [kg]
vol_ox = m_ox / rho_ox;                   % Volume of oxidizer [m^3]
vol_f = m_f / rho_f;                      % Volume of fuel [m^3]
vol_ox_boilOff = vol_ox * 0.01;           % boil off volume (rough
 approx)
vol_f_boilOff = vol_f * 0;                % non cryo, wont boil off
vol_ox_ullage = (vol_ox_boilOff + vol_ox) * 0.05;
vol_f_ullage = (vol_f_boilOff + vol_f) * 0.05;

vol_ox_total = vol_ox + vol_ox_boilOff + vol_ox_ullage;
vol_f_total = vol_f + vol_f_boilOff + vol_f_ullage;

end
```

propTankWt_Size function

returns overall tank weight, overall length, thickness of cylinder, thickness of hemispherical ends, thickness of junction, and length of tank

```
function [tankWt,tank_length,t_cyl,t_hemi,l_tank_cyl]...
    = propTankWt_Size(vol_total, r_tank, operatingPressure)

% Parameters are:  outer radius of tank [ft], and operating pressure
 [psi]


%%%%%%%%%%%%%%%%%%%%%%% Tank size and weight %%%%%%%%%%%%%%%%%%%%%%%
%%%%%%

rho_Al = 170;                         % Density of 6066 t-6 aluminum,
 lbm/ft^3
al_yield = 39000;                     % Yield strength of 6066 t-6 al
al_ultimate = 45000;                  % Ultimate strength of 6066 t-6 al
K = 1/0.67;                           % Knuckle factor for stress
 concentrations

% Max allowable stress per MIL-STD1522A (USAF)
maxAllowStress = min(al_yield/1.25, al_ultimate/1.5);
```

Tank thickness calcs

Required wall thickness at the weld [in]

```
t_k = K * operatingPressure * r_tank * 12 / maxAllowStress;

% Required wall thickness of the hemispheircal ends [in]
t_cr = operatingPressure * r_tank * 12 / (2 * maxAllowStress);

% Hemispherical end thickness [in]
t_hemi = (t_k + t_cr) / 2;

% Required thickness of cylindrical section [in]
t_cyl = operatingPressure * r_tank * 12 / maxAllowStress;
```

Tank sizing calcs

Inner radii of hemi and cyl [ft]

```
r_inner = r_tank - t_hemi/12;

% volume of end cap
vol_hemi = 2/3 * pi * r_inner^3;

% Volume required for cylinders
vol_cyl = vol_total - 2 * vol_hemi;

% oxider tank length cylinder only
l_tank_cyl = vol_cyl / (pi * r_inner^2);

% tank overall Length (inner dimensions)
tank_length = l_tank_cyl + 2 * r_inner;
```

Weight Calcs

Weight of components

```
weight_tank_cyl = rho_Al * pi * l_tank_cyl * (r_tank^2 - r_inner^2);
weight_Tank_Hemis = 2/3 * pi * rho_Al * (r_tank^3 - r_inner^3);

% Overall weight
tankWt = weight_tank_cyl + 2 * weight_Tank_Hemis;

end
```

GetFeedPress Function

returns tank pressure based on pressure drops, losses, and line diam and mass flow, density, and line length [m]

```
function [tank_press] =
 GetFeedPress(lineDiam,P0_eng,catBed_pressDrop,...
```

```
    inj_PressDrop, m_dot, rho, L_in)

global in2m


A = pi/4*lineDiam^2; % area [m^2]
v = m_dot/(rho*A); % velocity [m/s]
mu = 1.003e-3;  % viscosity (using water)
Re = rho*v*lineDiam/mu; % reynolds #
epsilon = 0.005/1000; % guess for pipe roughness [m]
L= L_in*in2m; % line length [m]

if(Re > 2200)
   f_expr = @(f) -2*log(epsilon/lineDiam/3.7+2.51./(Re*sqrt(f)))-1./
sqrt(f);
   f = fsolve(f_expr, 0.05);
else
    disp('laminar') % shouldnt ever be laminar, if it is I want to
 know
end

deltaP = rho*f*L/lineDiam*v^2/2; % TODO: calculate friction losses and
% pressure drops from relief valves, etc

backPress = P0_eng+catBed_pressDrop+inj_PressDrop; % pressure at inlet
tank_press = rho*v^2 + backPress + deltaP;


end
```

Rao Function

Rao Parabolic Approximation Method for sizing nozzles

```
function [xx, coeffs, R_x] = Rao(R_t, epsilon,
 Lf_ratio,theta_E,theta_N,theta_C)

% returns coefficients for parabola, and the radius vector
% note: using fixed exit and entrance angles, need to get numeric
% data for plots so these can be chosen automatically


global n_rao

n = n_rao;
% *************** Don't change anything in this section
 ******************
R_dt = 0.382*R_t;
R_ut = 1.5*R_t;
R_cu = R_ut + R_t;
R_cd = R_dt + R_t;

% geometry (see notes for figure)
L_n = Lf_ratio*(R_t*(sqrt(epsilon)-1)+...
```

```
    R_dt*(sec(15*pi/180)-1))/tan(15*pi/180);

% solving for parabola coeffs
x_N = tan(theta_N)*R_dt/(sqrt(1+tan(theta_N))^2);
R_N = R_cd - sqrt(R_dt^2-x_N^2);
c = (tan(theta_N)-tan(theta_E))/(2*(x_N-L_n));
b = tan(theta_N)-2*c*x_N;
a = R_N-x_N*(b+c*x_N);

coeffs = [a;b;c];

x_co = tan(theta_C)*R_ut/sqrt(1+tan(theta_C)^2);

xx = linspace(-1.5*x_co,L_n,n);
for i = 1:n
    if(xx(i) >= x_N)
        R_x(i) = a + b*xx(i) + c*xx(i)^2;
    elseif (xx(i) < x_N && xx(i) >= 0)
        R_x(i) = R_cd-sqrt(R_dt^2-xx(i)^2);
    else
        R_x(i) = R_cu-sqrt(R_ut^2-xx(i)^2);
    end
end
```

newtzero Function

```
function root = newtzero(f,xr,mx,tol)
%NEWTZERO finds roots of function using unique approach to Newton's
 Method.
% May find more than one root, even if guess is way off.  The function
 f
% should be an inline object or function handle AND capable of taking
% vector arguments.  The user can also pass in a string function.
% NEWTZERO takes four arguments. The first argument is the function to
 be
% evaluated, the second is an optional initial guess, the third is an
% optional number of iterations, and the fourth is an absolute
 tolerance.
% If the optional arguments are not supplied, the initial guess will
 be set
% to 1, the number of iterations will be set to 30, and the tolerance
 will
% be set to 1e-13.  If the initial guess is a root, no iterations will
 take
% place.
%
% EXAMPLES:
%
%         % Use any one of these three equivalent function
 definitions.
%         f = inline('cos(x+3).^3+(x-1).^2'); % Inline object.
%         f = 'cos(x+3).^3+(x-1).^2'; % String function.
%         f = @(x) cos(x+3).^3+(x-1).^2; % Anonymous function.
```

```
%         newtzero(f,900) % A very bad initial guess!
%
%         fb = @(x) besselj(2,x)
%         rt = newtzero(fb); % Finds roots about the origin.
%         rt = rt(rt>=-100 & rt<=100); % Plot about origin.
%         x = -100:.1:100;
%         plot(x,real(fb(x)),rt,abs(fb(rt)),'*r')
%
%         f = @(x) x.^3 +1;
%         newtzero(f,1) % Finds the real root;
%         newtzero(f,i) % Finds the real root and two imaginary roots.
%
%         f = @(x) x.^3 + 2*x.^2 + 3*x -exp(x)
%         newtzero(f)  % Finds two roots.
%
%         % Try it with Wilkinson's famous polynomial.
%         g = @(x)prod(bsxfun(@(x,y)(x-y),[1:20]',x.')).'; % For
 brevity
%         newtzero(g,0)
%
%
% May work when the initial guess is outside the range of fzero,
% for example compare:
%
%            fzero(f,900)   % for f as in the above example.
%
% This function may also find complex roots when fzero fails.  For
 example,
% try to find the roots of the following using NEWTZERO and FZERO:
%
%            f1 = @(x) x.^(2*cos(x)) -x.^3 - sin(x)+1;
%            ntz1 = newtzero(f1,[],[],1e-15) % NEWTZERO finds 5 roots
%            fzrt1 = fzero(f1,0) % FZERO aborts.
%
%            f2 = @(x) x.^2 + (2 + .1*i);  % could use 'roots' here.
%            ntz2 = newtzero(f2,1)  % NEWTZERO finds 2 roots
%            fzrt2 = fzero(f2,1) % FZERO fails.
%
% See also fzero, roots
%
% Author:  Matt Fig
% Contact:  popkenai@yahoo.com

defaults = {1,30,1e-13};% Initial guess, max iterations, tolerance.

switch nargin  % Error checking and default assignments.
    case 1
        [xr,mx,tol] = defaults{:};
    case 2
        [mx,tol] = defaults{2:3};
    case 3
        tol = defaults{3};
end
```

```
if isempty(xr)   % User called newtzero(f,[],50,1e-3) for example.
    xr = defaults{1};
end

if isempty(mx)
    mx = 30;
end

if ~isa(xr,'double')
    error('Only double values allowed.  See help examples.')
end

if tol < 0 || ~isreal(tol)
    error('Tolerance must be greater than zero.')
end

if mx < 1 || ~isreal(mx)
    error('Maximum number of iterations must be real and >0.')
end

[f,err] = fcnchk(f,'vectorized'); % If user passed in a string.

if ~isempty(err)
    error(['Error using NEWTZERO:',err.message])
end

if abs(f(xr))< tol
    root = xr; % The user supplied a root as the initial guess.
    return  % Initial guess correct.
end

LGS = logspace(0,3,220); % Can be altered for wider range or denser
 search.
LNS = 0:1/19:18/19;  % Search very close to initial guess too.
xr = [xr-LGS xr+LGS xr-LNS(2:end) xr+LNS].';  % Make vector of
 guesses.
iter = 1;  % Initialize the counter for the while loop.
mn1 = .1; % These will store the norms of the converging roots.
mn2 = 1; % See last comment.
sqrteps = sqrt(eps); % Used to make h.  See loop.
warning off MATLAB:divideByZero % WILL BE RESET AT THE END OF WHILE
 LOOP.

while iter <= mx && abs(mn2-mn1) > 5*eps
    h = sqrteps*xr; % From numerical recipes, make h = h(xr)
    xr = xr-f(xr)./((f(xr+h)-f(xr-h))./(2*h)); % Newton's method.
    xr(isnan(xr) | isinf(xr)) = []; % No need for these anymore.
    mn1 = mn2; % Store the old value first.
    mn2 = norm(xr,'fro'); % This could make the loop terminate early!
    iter = iter+1;  % Increment the counter.
end

if abs(f(0)) < tol % The above method will tend to send zero root to
 Inf.
```

```
    xr = [xr;0];  % So explicitly check.
end

warning on MATLAB:divideByZero % Reset this guy, as promised.

% Filtering.  We want to filter out certain common results.
idxi = abs(imag(xr)) < 5e-15;  % A very small imag term is zero.
xr(idxi) = real(xr(idxi));  % Discard small imaginary terms.
idxr = abs(real(xr)) < 5e-15;  % A very small real term is zero.
xr(idxr) = complex(0,imag(xr(idxr))); % Discard small real terms.
root = xr(abs(f(xr)) < tol); % Apply the tolerance.

% Next we are going to delete repeat roots.  unique does not work in
% this case because many repeats are very close to each other but not
% equal.  For loops are fast enough here, most root vectors are
 short(ish).

if ~isempty(root)
    cnt = 1;  % Counter for while loop.

    while ~isempty(root)
        vct = abs(root - root(1))<5e-6; % Minimum spacing between
 roots.
        C = root(vct);  %C has roots grouped close together.
        [idx,idx] = min(abs(f(C)));  % Pick the best root per group.
        rt(cnt) = C(idx);  %#ok<AGROW>  Most root vectors are small.
        root(vct) = []; % Deplete the pool of roots.
        cnt = cnt + 1;  % Increment the counter.
    end

    root = sort(rt).';  % return a nice, sorted column vector

end
```

Published with MATLAB® R2016a