Rocket Project Code Report

Table of Contents

This document contains the source code for the main components of the rocket simulation project.

Trajectory Simulation Code (trajectory.m)

The following code is from the file `trajectory.m`.

```
clc
clear
close all

%% ROCKET AND ENVIRONMENTAL PARAMETERS

g = 9.81;               % Gravity                      [m/s^2]
m_kit = 0.625;          % Dry mass                     [kg]
m_in = 0.226796;        % Inert mass of the motor      [kg]
A = 0.01081;            % Cross-sectional area         [m^2]
Cd = 0.63;              % Drag coefficient
v_w = 15.9;             % Wind speed in +x direction [km/h]
De = 1.5;               % Exit diameter                 [in]
launch_angle_deg = 2;% Launch angle from vertical   [deg]

v_w = v_w / 3.6;        % From km/h to m/s
launch_angle_rad = deg2rad(launch_angle_deg);

Ae = pi * (De * 0.0254 / 2)^2; % Nozzle exit area [m^2]

gamma = 1.2;            % Specific heat ratio (air)

De = 1.5;               % Exit diameter                 [in]
Dt = 0.25;              % Throat diameter               [in]
c_star = 5088;          % Characteristic velocity      [ft/s]

epsilon = De.^2 ./ Dt.^2;
Me = get_Me(epsilon, gamma);

%% GRAIN PARAMETERS AND THRUST CURVE CALCULATION

r1 = 0.30;                % Internal radius              [in]

% Thrust history [time (s), thrust (N)]
```

```
[thrust, mass_flow] = get_curves(r1, Me, epsilon, gamma, "Dt", Dt, "c_star",
c_star);
thrust_fn = @(t) interp1(thrust(:,1), thrust(:,2), t, 'linear', 0);

mdot_fn = @(t) interp1(mass_flow(:,1), mass_flow(:,2), t, 'linear', 0);
m_prop = trapz(mass_flow(:,1), mass_flow(:,2));
fprintf("Total Propellant Mass: %.2f kg\n", m_prop)
m0 = m_kit + m_prop + m_in;
m_dry = m_kit + m_in;

% Air Density and Pressure Model (ISA)
[rho_fn, pa_fn] = get_isa_props();

drag_fn = @(t, z, vz, vx) compute_drag_and_alpha(t, z, vz, vx, v_w, rho_fn,
A, Cd);

%%  INTEGRATION

% State vector = [x; z; vx; vz; m; v_loss_gravity]

x0 = [0; 0; 0; 0; m0; 0];
tspan = [0, 1000];

options = odeset('Events', @ground_hit_event); % Event to stop at ground hit

[t, x] = ode45(@(t, x) rocket_dynamics(t, x, thrust_fn, mdot_fn, drag_fn, g,
m_dry, pa_fn, Ae, launch_angle_rad), tspan, x0, options);

% Extract Maximum Altitude and Gravity Losses
z = x(:,2); % Altitude trajectory
[max_altitude, idx] = max(z);
max_time = t(idx);
v_loss = x(:,6); % Gravity loss trajectory
total_gravity_loss = v_loss(end); % Total gravity loss at end

% Compute Angle of Attack for Plotting
alpha = zeros(size(t));
for i = 1:length(t)
    vx = x(i,3); % Horizontal velocity
    vz = x(i,4); % Vertical velocity

    v_rocket_mag = sqrt(vx^2 + vz^2);
    v_r = [vx - v_w; vz];
    v_rel_mag = norm(v_r);

    if v_rocket_mag == 0
        if v_w ~= 0
            alpha(i) = pi/2;
        else
            alpha(i) = 0;
        end
    elseif v_rel_mag == 0
        alpha(i) = 0;
    else
```

```
        dot_product = vx * (vx - v_w) + vz * vz;
        alpha(i) = acos(dot_product / (v_rocket_mag * v_rel_mag));
    end
end

alpha_deg = rad2deg(alpha); % Convert to degrees for plotting

%% OUTPUT AND PLOTTING

%  Output Results
fprintf('Maximum Altitude: %.2f meters at t = %.2f seconds\n', max_altitude,
max_time);
fprintf('Maximum Altitude: %.2f ft at t = %.2f seconds\n', max_altitude /
0.3048, max_time);
fprintf('Total Gravity Loss: %.2f m/s\n', total_gravity_loss);
fprintf('Downrange Distance at Max Altitude: %.2f meters\n', x(idx,1));

% Plot Results
figure()
subplot(2,3,1)
plot(x(:,1), x(:,2), 'LineWidth', 2)
xlabel('Downrange Distance (m)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Altitude (m)', 'Interpreter', 'latex', 'FontSize', 15)
title('Rocket Trajectory', 'Interpreter', 'latex', 'FontSize', 20)
grid on
% axis equal
ax = gca;
ax.TickLabelInterpreter = 'latex';

subplot(2,3,2)
plot(t, x(:,2), 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Altitude (m)', 'Interpreter', 'latex', 'FontSize', 15)
title('Altitude vs Time', 'Interpreter', 'latex', 'FontSize', 20)
grid on
ax = gca;
ax.TickLabelInterpreter = 'latex';

subplot(2,3,3)
plot(t, x(:,4), 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Vertical Velocity (m/s)', 'Interpreter', 'latex', 'FontSize', 15)
title('Vertical Velocity vs Time', 'Interpreter', 'latex', 'FontSize', 20)
grid on
ax = gca;
ax.TickLabelInterpreter = 'latex';

subplot(2,3,4)
plot(t, x(:,3), 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Horizontal Velocity (m/s)', 'Interpreter', 'latex', 'FontSize', 15)
title('Horizontal Velocity vs Time', 'Interpreter', 'latex', 'FontSize', 20)
grid on
ax = gca;
```

```
ax.TickLabelInterpreter = 'latex';

subplot(2,3,5)
plot(t, x(:,5), 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Mass (kg)', 'Interpreter', 'latex', 'FontSize', 15)
title('Mass vs Time', 'Interpreter', 'latex', 'FontSize', 20)
grid on
ax = gca;
ax.TickLabelInterpreter = 'latex';

subplot(2,3,6)
plot(t, v_loss, 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Gravity Loss (m/s)', 'Interpreter', 'latex', 'FontSize', 15)
title('Cumulative Gravity Loss vs Time', 'Interpreter', 'latex', 'FontSize',
20)
grid on
ax = gca;
ax.TickLabelInterpreter = 'latex';

figure()
plot(t, alpha_deg, 'LineWidth', 2)
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
ylabel('Angle of Attack (deg)', 'Interpreter', 'latex', 'FontSize', 15)
title('Angle of Attack vs Time', 'Interpreter', 'latex', 'FontSize', 20)
grid on
ax = gca;
ax.TickLabelInterpreter = 'latex';

%% FUNCTION DEFINITIONS

% ISA Properties Wrapper
function [rho_fn, p_fn] = get_isa_props()
    % Returns function handles for atmospheric density and pressure.
    % It handles negative altitudes by clamping them to zero.
    function [rho, p] = isa_props_at_alt(altitude)
        altitude(altitude < 0) = 0;
        [~, ~, p, rho] = atmosisa(altitude);
    end
    rho_fn = @(z) isa_props_at_alt(z);
    p_fn = @(z) isa_props_at_alt(z);
end

%  Wind Effect and Drag Function
function [D, alpha] = compute_drag_and_alpha(t, z, vz, vx, v_w, rho_fn, A,
Cd)
    % Compute relative velocity and angle of attack

    v_r = [vx - v_w; vz]; % Relative velocity vector (rocket velocity - wind)
    v_rel_mag = norm(v_r);

    v_rocket_mag = sqrt(vx^2 + vz^2);
```

```
    if v_rocket_mag == 0
        if v_w ~= 0
            alpha = pi/2;
        else
            alpha = 0;
        end
    elseif v_rel_mag == 0
        alpha = 0;
    else
        % Angle between rocket velocity and relative wind
        dot_product = vx * (vx - v_w) + vz * vz;
        alpha = acos(dot_product / (v_rocket_mag * v_rel_mag));
    end

    rho = rho_fn(z);
    D_mag = 0.5 * rho * v_rel_mag^2 * A * Cd; % Drag magnitude
    if v_rel_mag == 0
        D = [0; 0]; % No drag if no relative velocity
    else
        D = -D_mag * v_r / v_rel_mag; % Drag opposes relative velocity
    end
end

% Equations of Motion
function dxdt = rocket_dynamics(t, x, thrust_fn, mdot_fn, drag_fn, g, m_dry,
pa_fn, Ae, launch_angle_rad)

    x_pos = x(1);  % Horizontal position [m]
    z = x(2);      % Altitude [m]
    vx = x(3);     % Horizontal velocity [m/s]
    vz = x(4);     % Vertical velocity [m/s]
    m = x(5);      % Mass [kg]
    v_loss = x(6); % Cumulative gravity loss [m/s]

    % Thrust
    T_vac = thrust_fn(t);
    Pa = pa_fn(z);
    T_mag = T_vac - Pa * Ae;
    if T_mag < 0
        T_mag = 0;
    end

    v_mag = sqrt(vz^2 + vx^2); % Rocket velocity magnitude
    if v_mag == 0
        T = T_mag * [sin(launch_angle_rad); cos(launch_angle_rad)]; % Thrust
at launch angle
    else
        T = T_mag * [vx; vz] / v_mag; % Thrust along velocity vector
    end

    % Drag and angle of attack
    [D, ~] = drag_fn(t, z, vz, vx);

    % Mass flow rate
```

```
    mdot = mdot_fn(t);

    % Prevent mass from going below dry mass
    if m <= m_dry
        mdot = 0;
        T = [0; 0];
        m = m_dry;
    end

    % Gravity loss rate (m/s per second, only during thrust)
    if T_mag > 0
        theta = atan2(vx, vz); % Trajectory angle from vertical
        g_loss = g * cos(theta); % Gravity loss along thrust direction
    else
        g_loss = 0; % No gravity loss after burnout
    end

    dxdt = [vx; vz; (T(1) + D(1)) / m; (T(2) + D(2) - m * g) / m; -mdot;
g_loss];
end

function [value, isterminal, direction] = ground_hit_event(t, x)
    % Function to detect when the rocket hits the ground (z = 0)
    %
    % Inputs:
    %   t - current time (not used)
    %   x - current state vector
    % Outputs:
    %   value      - value to be zero (altitude)
    %   isterminal - 1 to stop the integration
    %   direction  - -1 to detect only decreasing altitude

    value = x(2);       % Detect when altitude = 0
    isterminal = 1;     % Stop the integration
    direction = -1;     % Trigger only when altitude is decreasing

end

function Me = get_Me(epsilon, gamma)
% Calculates the exit Mach number for a given expansion ratio.
%
% Inputs:
%   epsilon - Nozzle expansion ratio (Ae/At)
%   gamma   - Ratio of specific heats
%
% Outputs:
%   Me      - Mach number at the exit of the nozzle

    options = optimoptions("fsolve", "Display", "none");

    func = @(M) -epsilon + (1 ./ M) .* ((2 + (gamma - 1) .* M.^2) ./ (gamma
+ 1)) .^ ((gamma + 1) ./ (2.*(gamma-1)));

    Me = fsolve(func, 2, options);
```

```
end
```

Thrust and Mass Flow Curve Generation (get_curves.m)

The following code is from the file `get_curves.m`.

```
function [thrust_curve, mass_flow_curve] = get_curves(r, Me, epsilon, gamma, 
options)
% Calculates the thrust and mass flow curves for a solid rocket motor.
%
% Inputs:
%   r                 - Initial radius parameter for the star points [in]
%   Me                - Mach number at the exit of the nozzle
%   epsilon           - Nozzle expansion ratio (Ae/At)
%   gamma             - Ratio of specific heats
% Name-Value pairs:
%   Dt                - Throat diameter [in]
%   c_star            - Characteristic velocity [ft/s]
%   g                 - Gravity acceleration [ft/s^2] (default: 32.2)
%   displayCurves     - Boolean to display plots (default: true)
%
% Outputs:
%   thrust_curve      - A [Nx2] matrix of [time, thrust] data in [s, N]
%   mass_flow_curve   - A [Nx2] matrix of [time, mass_flow] data in [s, kg/s]

    arguments
        r
        Me
        epsilon
        gamma
        options.Dt
        options.c_star
        options.g = 32.2;
        options.displayCurves = true;
    end

    At = pi * options.Dt^2 / 4; % [in^2]

    [t, Pc] = project_grain(r, "Dt", options.Dt, "c_star", options.c_star);

    if max(Pc) > 800
        warning('Maximum chamber pressure (Pc = %.2f psi) exceeds the 800 
psi limit.', max(Pc));
    end

    % Remove the last value from vectors to prevent NaN value in impulse 
calculation
    Pc = Pc(1:end-1);
    t = t(1:end-1);
```

```
    % Pe should be a vector, calculated for each value of Pc
    Pe = Pc ./ (1 + (gamma - 1)./2 .* Me.^2).^(gamma./(gamma-1));

    % Pressure ratio Pe/Pc for each time step
    pressure_ratio = Pe ./ Pc;

    cf_v = sqrt((2 .* gamma.^2 ./ (gamma - 1)) .* ((2 ./ (gamma +
1)).^((gamma + 1)./(gamma - 1))) .* ...
           (1 - pressure_ratio.^((gamma - 1)./gamma))) ...
          + (pressure_ratio).*epsilon;

    F_v = cf_v .* Pc .* At;
    F_v = F_v * 4.44822; % To Newtons

    m_dot = options.g .* Pc .* At ./ options.c_star;
    m_dot = m_dot * 0.453592; % To kg/s

    thrust_curve = [t', F_v'];
    mass_flow_curve = [t', m_dot'];

    if options.displayCurves

        % plot thrust curve
        figure()
        tiledlayout(1,3);

        nexttile
        plot(t, F_v, 'LineWidth', 2)
        title('Thrust vs. Time', 'Interpreter', 'latex', 'FontSize', 20)
        xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
        ylabel('Thrust (N)', 'Interpreter', 'latex', 'FontSize', 15)
        grid on;
        ax = gca;
        ax.TickLabelInterpreter = 'latex';

        % plot mass flow curve
        nexttile
        plot(t, m_dot, 'LineWidth', 2)
        title('Mass Flow Rate vs. Time', 'Interpreter', 'latex', 'FontSize',
20)
        xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
        ylabel('Mass Flow Rate (kg/s)', 'Interpreter', 'latex', 'FontSize',
15)
        grid on;
        ax = gca;
        ax.TickLabelInterpreter = 'latex';

        % plot chamber pressure curve
        nexttile
        plot(t, Pc, 'LineWidth', 2)
        title('Chamber Pressure vs. Time', 'Interpreter', 'latex',
'FontSize', 20)
        xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15)
```

```
        ylabel('Chamber Pressure (psi)', 'Interpreter', 'latex', 'FontSize', 
15)
        grid on;
        ax = gca;
        ax.TickLabelInterpreter = 'latex';
    end

end
```

Grain Burn Simulation (project_grain.m)

The following code is from the file `project_grain.m`.

```
function [t, Pc] = project_grain(r1, options)
% AUTHOR: CHARLIE OLSON
% This function tracks the propagation of the grain in a 6-sided star bore
% geometry, and provides a Chamber Pressure History.
arguments
    r1
    options.Dt
    options.c_star
end

% Rocket Dimensions of tube, throat, and exit
D = 1.212;  % [in]
Dt = 0.25;  % [in]
De = 1.5;  % [in]
z(1) = 4.6;  % [in]
% Defines second initial geometric conditions of 6-pointed star geometry
theta = 30;  % [deg]
% Defines initial propulsion conditions
rho_p = 0.06068;  % [lb/in^3]
a = 0.0318;  % [in/(s*Psi^n)]
n = 0.28;  % [unitless]
At = pi * Dt^2/4;  % [in^2]
g = 32.174 * 12;  % [in/s^2]
c_star = 5088 * 12;  % [in/s]

% t=0 Grain conditions:
% Length of a star face initially. The star points are a distance r1 from
% the centerline, while the star corners are a distance r1/2. The length
% of a star face is the hypotenuse of this triangle.
L(1) = r1/2 / cosd(theta);  % [in]
% Initial Burn area. Side area is A = L(w) * z(w). The code divides the 
grain into 12
% identical arcs, so this calculates the burn area of one face and then
% multiplies by a factor of 12. The top area is determined by dividing the
% star into 12 equal triangles, and subtracting their areas from the
% casing circle.
Ab_top(1) = pi * D^2 / 2 - 12 * r1 * L(1) * sind(theta);
Ab(1) = 12 * z(1) * L(1) + Ab_top(1);  % [in^2]
Pc(1) = (a * rho_p * Ab(1) * c_star / (g * At))^(1/(1-n));
```

```
% Calculates initial burning rate and sets up the time iteration
rb(1) = a * Pc(1)^n;
w(1) = 0;
t = 0;
tstep = 0.0001;
% Defines the limiting web distance of the first phase of the burn, the
% web distance where the points of the stars contact the outer casing.
w_lim = D / 4 - r1 / 2;
% Sets counter and checker variables for integration
j = 1;
Ab_checker = 0;
% Ab_checker breaks the loop when Ab ~ 0, i.e. when the fuel has been
% expended.
while Ab_checker == 0
    Pc(j) = (a * rho_p * Ab(j) * c_star / (g * At))^(1/(1-n));
    rb(j) = a * Pc(j)^n;
    % This first conditional is the first stage of the burn, where the star
    % points are propogating rapidly to the case boundary.
    if w(j) <= w_lim
        % Here is the iterative process, where the web distance at every
        % time t is determined by the time interval, the prior time step,
        % and the burning rate at the prior time step.
        j = j + 1;
        t(j) = t(j-1) + tstep;
        w(j) = w(j-1) + tstep * rb(j-1);
        % Calculates the new length and heights of the star faces, and
        % then determines the new burn area.The points of the stars
        % propogate twice as fast as the rest of the star surface due to
        % our assumptions of sharp corners.
        L(j) = (r1 / 2 + w(j)) / cosd(theta);
        z(j) = z(1) - 2 * w(j);
        Ab_top(j) = pi * D^2 / 2 - 12 * (r1+2*w(j)) * L(j) * sind(theta);
        Ab(j) = 12 * z(j) * L(j) + Ab_top(j);
    % The next area of the code is the second phase of the burn, when the
    % grain splits into 6 different triangles burning down to a point. Like
    % before, this solution zooms into one of the twelce sector arcs that
    % is representative of the problem.
    else
        j = j + 1;
        t(j) = t(j-1) + tstep;
        w(j) = w(j-1) + tstep * rb(j-1);
        z(j) = z(1) - 2 * w(j);
        % Solves for the distance xb between the centerline of the grain
        % chunk and where the grain intersects the circle casing.
        xb(j) = project_geosolver(w(j),D,theta,r1);
        % Solves for the length of the chunk face.
        L(j) = xb(j) / cosd(theta);
        Ab_triangle(j) = xb(j) * L(j) * sind(theta) / 2;
        beta(j) = asind(xb(j) / (D/2));
        Ab_arc(j) = 0.25 * (pi * D^2 * beta(j)/360 - xb(j) * D *
cosd(beta(j)));
        Ab_top(j) = 24 * (Ab_triangle(j) + Ab_arc(j));
        Ab(j) = 12 * z(j) * L(j) + Ab_top(j);
        % Condition where Ab ~ 0 and the grain burns out
```

```
        if Ab(j) <= 0.00001
            Ab_checker = 1;
            Pc(j) = (a * rho_p * Ab(j) * c_star / (g * At))^(1/(1-n));
            rb(j) = a * Pc(j)^n;
        end
    end
end
end
```

Grain Geometric Solver (project_geosolver.m)

The following code is from the file `project_geosolver.m`.

```
function xb = project_geosolver(w,D,theta, r1)
% Solves the geometric problem of the second stage of 6-pointed star grain
% propogation.

% The target is a constant value that allows the function to calculate the
% intersection of the circular casing and the receding grain-line at all
% time steps, and therefore the length of the grain chunk perpendicular to
% its centerline.
target = D^2 / 4;
checker = 0;
size = 100;
while checker == 0
    % Defines possible values that x could be for the iteration to check
    x = linspace(0,0.5,size);
    j = 1;
    for j = 1:1:length(x)
        % This is the equation of intersection between the casing and
        % grain-line equations.
        val = x(j)^2 + (-tand(theta)*x(j) - w / sind(2*theta) - r1/
(2*cosd(theta)))^2;
        % If the intersection is true, val will be equal to the target
        % (within tolerance).
        if abs(val - target) <= 0.0001
            % breaks loop and returns perpendicular grain dimension
            checker = 1;
            xb = x(j);
        end
    end
    % If a length has not been found, makes the x-values to be searched
    % smaller and repeats the loop.
    if checker == 0
        size = size * 10;
    end
end

return
```

Published with MATLAB® R2024b