Capítulo 1. Trajectory code\main.m

```
clc
clear
close all

%% Create a new rocket object
myRocket = Rocket("Rocket");

noseCone = NoseCone('MyNoseCone', 'ogive', 0.5, 0.1, 1.0, []);
myRocket = myRocket.add_part(noseCone);

bodyTube1 = BodyTube('MainBodyTube', 0.5, 0.1, 1.2, 0.003);
myRocket = myRocket.add_part(bodyTube1);

parachute = Parachute('MainParachute', 1.5, 0.8, 300, 0.1, 0.5,
bodyTube1);
parachute = parachute.update_position([0, 0, 0.1]); % Place it
10cm from the top of the tube
myRocket = myRocket.add_part(parachute);

avionics = Mass('AvionicsBay', 0.4, 0.15, 0.09, bodyTube1);
avionics = avionics.update_position([0, 0, 0.3]); % Place it
30cm from the top of the tube
myRocket = myRocket.add_part(avionics);

motorTube = BodyTube('MotorTube', 0.5, 0.05, 0.7, 0.002);
myRocket = myRocket.add_part(motorTube);


thrust_curve = [0.0, 0.0; 0.003, 281.69; 0.05, 1436.62; 0.121,
1363.38; 0.366, 1263.85; 0.59, 1230.05; 1.745, 1322.07; 2.835,
1166.2; 4.0, 974.648; 4.158, 839.437; 4.668, 82.629; 4.736,
0.0];
motor = Motor('MyMotor', 5.0, 2.8, 0.03, 0.6, thrust_curve,
motorTube);
motor = motor.update_position([0, 0, 0.05]); % Place it 5cm
from the top of the motor tube
myRocket = myRocket.add_part(motor);

finSet = FinSet('MyFins', 4, 0.15, 0.2, 0.1, 0.05, 0.004, 0.1,
motorTube);
finSet = finSet.update_position([0, 0, 0.4]); % Attach fins
starting 40cm down the motor tube
```

```
myRocket = myRocket.add_part(finSet);

% Debug calculations
time = 0;
total_mass = myRocket.get_total_mass(time);
cg = myRocket.get_cg(time);
inertia = myRocket.get_inertia(time);

fprintf('Rocket: %s\n', myRocket.rocketName);
fprintf('Total Mass at t=%.1f s: %.2f kg\n', time, total_mass);
fprintf('Center of Gravity (from nose tip) at t=%.1f s: %.2f
m\n', time, cg);
fprintf('Inertia Tensor at t=%.1f s (kg*m^2):\n', time);
disp(inertia);

%% Simulation Parameters

% Wind Profile
wind_model = PowerLawWindProfile();

altitudes = linspace(0, 1000, 200);
wind_vectors = zeros(length(altitudes), 2);

for i = 1:length(altitudes)
    wind_vectors(i, :) = wind_model.wind_at_h(altitudes(i));
end

% Plot the wind vector components in 3D
figure('Name', '3D Wind Profile');
plot3(wind_vectors(:,1), wind_vectors(:,2), altitudes, "b",
'LineWidth', 2, 'DisplayName', 'Wind Vector Tip');
hold on; % Keep the line plot

% Add arrows to the plot to show vector direction
arrow_indices = 1:20:length(altitudes); % Show an arrow every
20 points
quiver3(zeros(1, length(arrow_indices)), ... % Start arrows
from X=0 (as row vector)
    zeros(1, length(arrow_indices)), ... % Start arrows from
Y=0 (as row vector)
    altitudes(arrow_indices), ...         % Start arrows at
different altitudes (already a row vector)
    wind_vectors(arrow_indices, 1)', ... % X-component of wind
(transposed to row vector)
    wind_vectors(arrow_indices, 2)', ... % Y-component of wind
```

```
(transposed to row vector)
    zeros(1, length(arrow_indices)), ... % Z-component is zero
(as row vector)
    'b', 'LineWidth', 2, 'DisplayName', 'Wind Vectors',
'AutoScale', 'off', 'MaxHeadSize', 0.001);

set(gca, 'TickLabelInterpreter', 'latex');
xlabel('Wind $V_x$ (m/s)', 'Interpreter', 'latex', 'FontSize',
15);
ylabel('Wind $V_y$ (m/s)', 'Interpreter', 'latex', 'FontSize',
15);
zlabel('Altitude (m)', 'Interpreter', 'latex', 'FontSize', 15);
title('3D Wind Profile (0 to 1 km)', 'Interpreter', 'latex',
'FontSize', 20);
grid on;
view(45, 30); % Adjust view for better 3D perspective
legend('Interpreter', 'latex', 'Location', 'best');
hold off;

theta = deg2rad(45); % Launch rail elevation angle from
horizontal
phi = deg2rad(85);   % Launch rail azimuth angle from North

[time, state] = compute_trajectory(myRocket, ...
                                   "rail_length", 5.0, ...
                                   "theta", theta, ...
                                   "phi", phi, ...
                                   "wind_model", wind_model, ...
                                   "max_time", 600);

% Plot Trajectory
figure('Name', 'Rocket Trajectory');
plot3(state(:,1), state(:,2), state(:,3), 'LineWidth', 2);
grid on;
hold on;

% Mark apogee
[max_alt, apogee_idx] = max(state(:,3));
plot3(state(apogee_idx, 1), state(apogee_idx, 2), max_alt,
'ro', 'MarkerSize', 8, 'MarkerFaceColor', 'r');
text(state(apogee_idx, 1), state(apogee_idx, 2), max_alt,
sprintf(' Apogee: %.1f m', max_alt), 'Interpreter', 'latex',
'VerticalAlignment', 'bottom');

% Mark landing point
```

```
landing_pos = state(end, 1:3);
plot3(landing_pos(1), landing_pos(2), landing_pos(3), 'kx',
'MarkerSize', 10, 'LineWidth', 2);
text(landing_pos(1), landing_pos(2), landing_pos(3), '
Landing', 'Interpreter', 'latex', 'VerticalAlignment',
'bottom');

% Mark the launch point
plot3(state(1,1), state(1,2), state(1,3), 'g^', 'MarkerSize',
8, 'MarkerFaceColor', 'g');
text(state(1,1), state(1,2), state(1,3), ' Launch',
'Interpreter', 'latex', 'VerticalAlignment', 'bottom');

% Draw the launch rail
rail_length = 5.0; % meters
rail_end = [rail_length * cos(phi) * cos(theta), ...
            rail_length * cos(phi) * sin(theta), ...
            rail_length * sin(phi)];
plot3([0, rail_end(1)], [0, rail_end(2)], [0, rail_end(3)],
'm--', 'LineWidth', 2, 'DisplayName', 'Launch Rail');
legend('Trajectory', 'Apogee', 'Landing', 'Launch', 'Launch
Rail', 'Interpreter', 'latex', 'Location', 'best');

axis equal;
xlabel('X (m)', 'Interpreter', 'latex', 'FontSize', 15);
ylabel('Y (m)', 'Interpreter', 'latex', 'FontSize', 15);
zlabel('Altitude (m)', 'Interpreter', 'latex', 'FontSize', 15);
title('Rocket Trajectory', 'Interpreter', 'latex', 'FontSize',
20);
set(gca, 'TickLabelInterpreter', 'latex');
view(45, 25);

% Plot Velocities
figure('Name', 'Velocities vs. Time');
plot(time, state(:,4), 'r', 'LineWidth', 2, 'DisplayName',
'$V_x$');
hold on;
plot(time, state(:,5), 'g', 'LineWidth', 2, 'DisplayName',
'$V_y$');
plot(time, state(:,6), 'b', 'LineWidth', 2, 'DisplayName',
'$V_z$');
grid on;
hold off;
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15);
ylabel('Velocity (m/s)', 'Interpreter', 'latex', 'FontSize',
```

```
15);
title('Inertial Velocities vs. Time', 'Interpreter', 'latex',
'FontSize', 20);
legend('Interpreter', 'latex', 'Location', 'best');
set(gca, 'TickLabelInterpreter', 'latex');

% Plot Angular Velocities
figure('Name', 'Angular Velocities vs. Time');
plot(time, rad2deg(state(:,11)), 'r', 'LineWidth', 2,
'DisplayName', '$p$ (roll)');
hold on;
plot(time, rad2deg(state(:,12)), 'g', 'LineWidth', 2,
'DisplayName', '$q$ (pitch)');
plot(time, rad2deg(state(:,13)), 'b', 'LineWidth', 2,
'DisplayName', '$r$ (yaw)');
grid on;
hold off;
xlabel('Time (s)', 'Interpreter', 'latex', 'FontSize', 15);
ylabel('Angular Velocity (deg/s)', 'Interpreter', 'latex',
'FontSize', 15);
title('Body Angular Velocities vs. Time', 'Interpreter',
'latex', 'FontSize', 20);
legend('Interpreter', 'latex', 'Location', 'best');
set(gca, 'TickLabelInterpreter', 'latex');
```

Capítulo 2. Trajectory code\MissionSim\compute_trajectory.m

```
function [time, state] = compute_trajectory(rocket, options)
% Simulates the trajectory of a rocket through rail, free-
flight, and recovery phases.
%
% Inputs:
%   rocket  - An instance of the Rocket class.
%   options - A struct with simulation parameters (rail_length,
theta, phi, wind_model, etc.).
%
% Outputs:
%   time    - A column vector of time points (s).
%   state   - An N-by-13 matrix where each row corresponds to a
time point and
%             columns represent the rocket's state (position,
velocity, quaternion, angular velocity).
%

    arguments
        rocket
        options.rail_length
        options.theta % Rotation of the rail about the z-axis
        options.phi   % Inclination of the rail
        options.wind_model
        options.t_start = 0
        options.max_time = 300
    end

    %% PRE-CHECKS AND INITIALIZATION

    if isempty(options.wind_model)
        error("ERROR.\nA wind model must be defined in order to
perform the simulations")
    end

    time = [];
    state = [];

    %% INITIAL CONDITIONS

    % Position, Velocity, Angular Velocity
    r0 = [0; 0; 0];
    v0 = [0; 0; 0];
```

```
    omega0 = [0; 0; 0];

    % Orientation from rail angles
    phi = options.phi;
    theta = options.theta;

    % Rail direction vector in inertial frame (this is the
rocket's z-axis)
    xb_i = [cos(phi) * cos(theta); cos(phi) * sin(theta);
sin(phi)];
    % Body y-axis (perpendicular to rocket axis and horizontal)
    yb_i = [-sin(theta); cos(theta); 0];
    % Body x-axis
    zb_i = cross(yb_i, xb_i);


    C_b2i = [xb_i, yb_i, zb_i]; % Rotation from body to inertial
    C_i2b = C_b2i';             % Rotation from inertial to body

    % Initial quaternion [q0, q1, q2, q3] (w, x, y, z)
    q0 = dcm2quat(C_i2b);

    state0 = [r0; v0; q0'; omega0];

    %% RAIL

    fprintf('Simulating rail phase...\n');

    tspan_rail = [options.t_start, options.max_time];
    ode_options_rail = odeset('Events', @(t, y)
railExitEvent(t, y, options.rail_length));


    [t1, s1, te1, ye1, ie1] = ode45(@(t,y) rail(t, y, rocket,
xb_i), tspan_rail, state0, ode_options_rail);

    time = [time; t1];
    state = [state; s1];

    if isempty(ie1)
        fprintf('Warning: Rocket did not leave the rail.\n');
        return;
    end

    %% FREE-FLIGHT
```

```
    fprintf('Simulating free flight phase...\n');

    t_start_ff = te1(end);
    state0_ff = ye1(end, :)';

    tspan_ff = [t_start_ff, options.max_time];
    ode_options_ff = odeset('Events', @apogeeEvent);

    [t2, s2, te2, ye2, ie2] = ode45(@(t,y) freeflight(t, y,
rocket), tspan_ff, state0_ff, ode_options_ff);

    time = [time; t2(2:end)];
    state = [state; s2(2:end, :)];

    if isempty(ie2)
        fprintf('Warning: Rocket did not reach apogee.\n');
        return;
    end

    %% RECOVERY

    fprintf('Simulating recovery phase...\n');

    t_start_rec = te2(end);
    state0_rec = ye2(end, :)';

    tspan_rec = [t_start_rec, options.max_time];
    ode_options_rec = odeset('Events', @groundHitEvent);

    if ~isempty(rocket.parachute) % There is a parachute for
recovery

        odefun_recovery = @(t, y)
recovery(t, y, rocket, rocket.parachute.drag_coefficient,
pi*(rocket.parachute.diameter/2)^2, options.wind_model);

    else % No parachute -> continue in freeflight

        odefun_recovery = @(t, y) freeflight(t, y, rocket);

    end

    [t3, s3, te3, ye3, ~] = ode45(odefun_recovery, tspan_rec,
state0_rec, ode_options_rec);
```

```
    % Append the recovery phase trajectory up to the event
    time = [time; t3(2:end)];
    state = [state; s3(2:end, :)];

    fprintf('Simulation finished.\n');

end

% Event functions

function [value, isterminal, direction] = railExitEvent(t, y,
rail_length)
    % Event triggers when the rocket has traveled the length of
the rail

    value = norm(y(1:3)) - rail_length;
    isterminal = 1; % Stop integration
    direction = 1;  % Trigger when value is increasing

end

function [value, isterminal, direction] = apogeeEvent(~, y)
    % Event triggers at apogee (vertical velocity is zero)

    value = y(6);
    isterminal = 1; % Stop integration
    direction = -1; % Trigger when value is decreasing (i.e.,
at a peak)
end

function [value, isterminal, direction] = groundHitEvent(~, y)
    % Event triggers when the rocket hits the ground (altitude
is zero)

    value = y(3);
    isterminal = 1; % Stop integration
    direction = -1; % Trigger when approaching from above
end
```

Capítulo 3. Trajectory code\MissionSim\freeflight.m

```
function dstate_dt = freeflight(t, state, rocket)

    %% Extract Rocket and State parameters

    g = 9.81; % [m/s^2]
    rho_SL = 1.225; %! Might want to change to ISA

    I = rocket.get_inertia(t);
    mass = rocket.get_total_mass(t);

    thrust = rocket.get_thrust(t); % 3x1 vector in the body
frame. z axis is the rocket axis of symmetry
    F_g = [0; 0; -mass * g];

    % TODO: Change aero with aero model once it is finished
    % Aerodynamic Properties - Assumed for the moment
    ref_area = 0.0182; % Reference area for aerodynamic
coefficients (m^2)
    ref_length = 3.0;  % Reference length for aerodynamic
moments (m)
    % Simple aerodynamic model: Coefficients are assumed
constant for this example.
    % In a real simulation, these would be functions of Mach
number, AoA, etc.
    Ca = 0.3;    % Axial force coefficient (along body x-axis)
    Cn = 2.0;    % Normal force coefficient (along body z-axis)
    Cy = 0;      % Side force coefficient (along body y-axis)
    Cl = 0;      % Roll moment coefficient
    Cm = -5.0;   % Pitch moment coefficient
    Cn_yaw = 0;  % Yaw moment coefficient
    xcp = 2.0;   % Distance from nose cone tip to center of
pressure (m)
    xcg = 1.8;   % Distance from nose cone tip to center of
gravity (m)

    % Unpack state vector
    r_vec = state(1:3);         % Position (x, y, z)
    vel_inertial = state(4:6);         % Velocity (vx, vy, vz)
    q = state(7:10);            % Quaternion (q0, q1, q2, q3)
    omega_body = state(11:13); % Angular velocity (p, q ,r)

    q = q / norm(q);
```

```
    p = omega_body(1);
    q_rate = omega_body(2);
    r = omega_body(3);

    %% Intermediate variables

    C_i2b = quat2dcm(q'); % Rotation Matrix for inertial to body
    vel_body = C_i2b * vel_inertial;

    % Calculate angle of attack (alpha) and sideslip angle
(beta)
    % Note: Assumes no wind. For wind, use v_aero = vel_body -
v_wind_body
    v_aero_body = vel_body;
    if norm(v_aero_body) < 1e-6
        alpha = 0;
        beta = 0;
    else
        alpha = atan2(v_aero_body(3), v_aero_body(1));
        beta = asin(v_aero_body(2) / norm(v_aero_body));
    end

    % Dynamic pressure
    q_dyn = 0.5 * rho_SL * norm(v_aero_body)^2;

    %% Forces and Moments (Body RF)

    % Aerodynamic forces
    F_axial = -q_dyn * ref_area * Ca;          % Along -x body
axis
    F_normal = q_dyn * ref_area * Cn * alpha; % Along +z body
axis
    F_side = q_dyn * ref_area * Cy * beta;     % Along +y body
axis
    F_aero = [F_axial; F_side; F_normal];

    F_total_body = F_aero + thrust;

    % Aerodynamic moments
    roll_mom = q_dyn * ref_area * ref_length * Cl;
    pitch_mom = q_dyn * ref_area * ref_length * Cm * alpha;
    yaw_mom = q_dyn * ref_area * ref_length * Cn_yaw * beta;

    Moments_body = [roll_mom; pitch_mom; yaw_mom];
```

```
    %% Accelerations & Quaternion derivative

    C_b2i = C_i2b'; % From body to inertial
    F_total_inertial = C_b2i * F_total_body;

    pos_dot = vel_inertial;
    v_dot = (F_total_inertial + F_g) / mass;

    omega_cross_I_omega = cross(omega_body, I * omega_body);
    omega_dot = I \ (Moments_body - omega_cross_I_omega);

    % Quaternion derivative
    Omega = [ 0, -p, -q_rate, -r;
              p,  0,  r, -q_rate;
              q_rate, -r,  0,  p;
              r,  q_rate, -p,  0];

    q_dot = 0.5 * Omega * q;

    dstate_dt = [pos_dot; v_dot; q_dot; omega_dot];

end
```

Capítulo 4. Trajectory code\MissionSim\rail.m

```
function dstate_dt = rail(t, state, rocket, rail_direction)

    %% Extract Rocket and State parameters

    g = 9.81;        % [m/s^2]
    rho_SL = 1.225; % [kg/m^3]

    I = rocket.get_inertia(t);
    mass = rocket.get_total_mass(t);

    thrust = rocket.get_thrust(t);
    F_g = [0; 0; -mass * g];

    % Aerodynamic Properties (placeholders, as in freeflight.m)
    ref_area = 0.0182; % Reference area [m^2]
    Ca = 0.3;          % Axial force coefficient
    % Note: Normal and side forces are generated but
counteracted by the rail.
    % They are still needed to calculate the total force on the
system.
    Cn = 2.0;          % Normal force coefficient
    Cy = 0;            % Side force coefficient

    % Unpack state vector
    r_vec = state(1:3);        % Position (x, y, z)
    v_inertial = state(4:6);        % Velocity (vx, vy, vz)
    q = state(7:10);           % Quaternion (q0, q1, q2, q3)
    omega_body = state(11:13); % Angular velocity (p, q ,r)

    %% Intermediate variables

    C_i2b = quat2dcm(q'); % Rotation Matrix for inertial to body
    vel_body = C_i2b * v_inertial;

    % Angle of attack (alpha) and sideslip (beta)
    if norm(vel_body) < 1e-6
        alpha = 0;
        beta = 0;
    else
        alpha = atan2(vel_body(3), vel_body(1));
        beta = asin(vel_body(2) / norm(vel_body));
    end
```

```
    % Dynamic pressure
    q_dyn = 0.5 * rho_SL * norm(vel_body)^2;

    %% Forces (Body RF)

    % Aerodynamic forces
    F_axial = -q_dyn * ref_area * Ca;
    F_normal = q_dyn * ref_area * Cn * alpha;
    F_side = q_dyn * ref_area * Cy * beta;
    F_aero_body = [F_axial; F_side; F_normal];

    F_total_body = F_aero_body + thrust;

    %% Accelerations

    C_b2i = C_i2b'; % From body to inertial
    F_total_inertial = C_b2i * F_total_body;

    F_net_inertial = F_total_inertial + F_g;

    % Project the forces on the rail direction
    d_hat = rail_direction / norm(rail_direction);
    F_proj = dot(F_net_inertial, d_hat);

    % Acceleration along the rail
    accel_mag = F_proj / mass;

    % Position and velocity derivatives
    pos_dot = v_inertial;
    v_dot = accel_mag * d_hat;

    % Rotational motion constrained by the rail
    omega_dot = [0; 0; 0];
    q_dot = [0; 0; 0; 0];

    dstate_dt = [pos_dot; v_dot; q_dot; omega_dot];

end
```

Capítulo 5. Trajectory code\MissionSim\recovery.m

```
function dstate_dt = recovery(t, state, rocket, Cd_p, Ap,
wind_model)

    %% Extract Rocket and State parameters

    g = 9.81;        % [m/s^2]
    rho_SL = 1.225; % [kg/m^3]

    mass = rocket.get_total_mass(t);

    F_g = [0; 0; -mass * g];

    % Unpack state vector
    pos_inertial = state(1:3);
    vel_inertial = state(4:6);       % Velocity (vx, vy, vz) in
inertial frame
    q = state(7:10);                 % Quaternion (q0, q1, q2,
q3)
    omega_body = state(11:13);       % Angular velocity (p, q,
r) in body frame

    p = omega_body(1);
    q_rate = omega_body(2);
    r = omega_body(3);

    %% --- 2. Calculate Forces in Inertial Frame ---

    [vx_wind, vy_wind] = wind_model.wind_at_h(pos_inertial(3));

    vel_relative = vel_inertial - [vx_wind; vy_wind; 0];

    vel_rel_mag = norm(vel_relative);
    F_drag_inertial = -0.5 * rho_SL * Cd_p * Ap * vel_rel_mag *
vel_relative;

    F_net_inertial = F_g + F_drag_inertial;

    %% Accelerations & Quaternion derivative

    pos_dot = vel_inertial;
    v_dot = F_net_inertial / mass;
```

```
    % Rotational motion is unforced
    omega_dot = [0; 0; 0];

    Omega = [ 0, -p, -q_rate, -r;
              p,  0,  r, -q_rate;
              q_rate, -r,  0,  p;
              r,  q_rate, -p,  0];

    q_dot = 0.5 * Omega * q;

    dstate_dt = [vel_inertial; v_dot; q_dot; omega_dot];

end
```

Capítulo 6. Trajectory code\Models\AeroModel.m

```
classdef AeroModel

end
```

Capítulo 7. Trajectory code\Models\PowerLawWindProfile.m

```
classdef PowerLawWindProfile < WindModel
    % Implements a power law wind model.
    % This model describes how wind speed changes with altitude
according to a power law relationship.
    % Reference: Davenport, A. G. (1965). The relationship of
wind structure to wind loading

    properties
        ref_speed       % Wind speed at the reference height
(m/s)
        ref_height      % Reference height (m)
        alpha           % Power law exponent (dimensionless)
        direction_deg   % Wind direction in degrees (0=from
North, 90=from East)
    end

    methods
        function obj = PowerLawWindProfile(ref_speed,
ref_height, alpha, direction_deg)
            % Constructs an instance of the PowerLawWindProfile
class.
            %
            % Inputs:
            %   ref_speed - Wind speed at the reference height
(m/s). Default: 5.0.
            %   ref_height - Reference height (m). Default:
500.0.
            %   alpha - Power law exponent. Default: 1/7.
            %   direction_deg - Wind direction in degrees.
Default: 270 (from West).

            arguments
                ref_speed = 5.0;
                ref_height = 500.0;
                alpha = 1/7;
                direction_deg = 270;
            end

            obj.ref_speed = ref_speed;
            obj.ref_height = ref_height;
            obj.alpha = alpha;
            obj.direction_deg = direction_deg;
```

```
        end

        function [vx, vy] = wind_at_h(obj, height)
            % Calculates the wind vector components at a given
height.
            %
            % Inputs:
            %   height - Altitude above ground level (m).
            %
            % Outputs:
            %   vx - The x-component of the wind velocity (m/s).
            %   vy - The y-component of the wind velocity (m/s).

            if height <= 0
                vx = 0;
                vy = 0;
                return;
            end

            speed_at_h = obj.ref_speed * (height /
obj.ref_height)^obj.alpha;

            % 0 deg (North) -> 270 deg; 90 deg (East) -> 180
deg.
            direction_rad = deg2rad(270 - obj.direction_deg);

            vx = speed_at_h * cos(direction_rad);
            vy = speed_at_h * sin(direction_rad);
        end
    end
end
```

Capítulo 8. Trajectory code\Models\WindModel.m

```
classdef WindModel

    methods (Abstract)

        [vx, vy] = wind_at_h(obj, height);

    end

end
```

Capítulo 9. Trajectory code\Models\WindProfileInterp.m

```
classdef WindProfileInterp < WindModel

    properties
        heights
        vx_vec
        vy_vec
    end

    methods

        function obj = WindProfileInterp(file_path)
            % Constructs a WindProfileInterp object from a data
file.
            % The file is expected to have three columns:
height, vx, and vy,
            % and may contain a header row.
            %
            % Inputs:
            %   file_path - String path to the wind profile
data file.

            data = readmatrix(file_path);

            obj.heights = data(:, 1);
            obj.vx_vec = data(:, 2);
            obj.vy_vec = data(:, 3);
        end


        function [vx, vy] = wind_at_h(obj, height)
            % Interpolates wind velocity components at a
specific height using linear
            % interpolation and extrapolation on the stored
wind profile data.
            %
            %   Inputs:
            %       obj     - The object instance containing
the wind profile data
            %                 (properties: heights, vx_vec,
vy_vec).
            %       height  - Scalar or vector of altitudes [m]
at which to
```

```
            %                  interpolate the wind velocity.
            %
            %   Outputs:
            %       vx      - Interpolated x-component(s) of
wind velocity [m/s].
            %       vy      - Interpolated y-component(s) of
wind velocity [m/s].
            %

            vx = interp1(obj.heights, obj.vx_vec, height,
'linear', 'extrap');
            vy = interp1(obj.heights, obj.vy_vec, height,
'linear', 'extrap');
        end
    end

end
```

Capítulo 10. Trajectory code\Rocket\BodyTube.m

```
classdef BodyTube < RocketPart
    % Represents a cylindrical section of the rocket, such as
the main airframe.

    properties

        diameter;
        length;
        thickness;

    end

    methods

        function obj = BodyTube(name, mass, diameter, length,
thickness)
            % Constructs an instance of the BodyTube class.
            %
            % Inputs:
            %   name - Name of the body tube (string).
            %   mass - Mass of the body tube (kg).
            %   diameter - Outer diameter of the body tube (m).
            %   length - Length of the body tube (m).
            %   thickness - Wall thickness of the body tube (m).
            %

            obj@RocketPart(name, mass);

            obj.diameter = diameter;
            obj.length = length;
            obj.thickness = thickness;
        end

        function cg = compute_cg(obj)
            % Computes the center of gravity of the body tube.
            %
            % Outputs:
            %   cg - Center of gravity [x, y, z] in meters,
relative to the part's origin.

            cg = [0, 0, obj.length / 2];
```

```
        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the body tube.
            %
            % Outputs:
            %   I - Inertia tensor as a 3x3 matrix.
            r_out = obj.diameter / 2;
            r_in = r_out - obj.thickness; % Inner radius

            l = obj.length;
            m = obj.mass;

            Ixx = (1/4) * m * (r_in^2 + r_out^2) + (1/12) * m *
l^2;
            Iyy = Ixx;
            Izz = 0.5 * m * (r_in^2 + r_out^2);

            I = diag([Ixx, Iyy, Izz]);
        end

    end

end
```

Capítulo 11. Trajectory code\Rocket\FinSet.m

```
classdef FinSet < SubRocketPart
    % Represents a set of fins attached to the rocket.

    properties

        num_fins;
        span;
        root_chord;
        tip_chord;
        sweep;
        thickness;
        mass_per_fin;
    end

    methods

        function obj = FinSet(name, num_fins, span, root_chord,
tip_chord, sweep, thickness, mass_per_fin, parent_part)
            % Constructs an instance of the FinSet class.
            %
            % Inputs:
            %   name - Name of the fin set (string).
            %   num_fins - Number of fins in the set (integer).
            %   span - Span of a single fin (m).
            %   root_chord - Length of the fin chord at the
root (m).
            %   tip_chord - Length of the fin chord at the tip
(m).
            %   sweep - Sweep distance of the fin's leading
edge (m).
            %   thickness - Thickness of the fins (m).
            %   mass_per_fin - Mass of a single fin (kg).
            %   parent_part - The RocketPart this component is
attached to.

            total_mass = mass_per_fin * num_fins;

            obj@SubRocketPart(name, total_mass, parent_part);

            obj.num_fins = num_fins;
            obj.span = span;
            obj.root_chord = root_chord;
```

```
            obj.tip_chord = tip_chord;
            obj.sweep = sweep;
            obj.thickness = thickness;
            obj.mass_per_fin = mass_per_fin;

            obj.mass = obj.num_fins * mass_per_fin;

        end


        function cg = compute_cg(obj)
        % COMPUTE_CG Computes the longitudinal center of
gravity of the fin set.
        %
        %   Output Arguments:
        %       cg - The longitudinal position of the fin set's
center of
        %            gravity. [m]


            cg_long = (obj.sweep * (obj.root_chord +
2*obj.tip_chord)) / (3 * (obj.root_chord + obj.tip_chord))
+ (obj.root_chord^2 + obj.root_chord*obj.tip_chord +
obj.tip_chord^2) / (3 * (obj.root_chord + obj.tip_chord));

            cg = [0, 0, cg_long];

        end

        function d = get_fin_centroid(obj)
            % Computes the distance form the root chord to the
fin centroid

            d = (obj.span / 3) * ( (obj.root_chord +
2*obj.tip_chord) / (obj.root_chord + obj.tip_chord) );
        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the fin set about
its own center of gravity
            %
            % Outputs:
            %   I - Inertia tensor as a 3x3 matrix

            d = obj.parent_part.diameter / 2 +
```

```
obj.get_fin_centroid();
            cg = obj.compute_cg();
            z_cg = cg(3);

            Area_fin = (obj.span / 2) * (obj.root_chord +
obj.tip_chord);

            I_chordwise = obj.mass_per_fin * ( (obj.span^2 /
18) * (obj.root_chord^2 + 4*obj.root_chord*obj.tip_chord + ...
                          obj.tip_chord^2) / (obj.root_chord +
obj.tip_chord)^2 + (obj.thickness^2 / 12) );

            I_area_root = (obj.span / 12) * (obj.root_chord^3 +
obj.tip_chord^3 + (obj.root_chord + obj.tip_chord) * ...
                          (obj.root_chord*obj.tip_chord +
3*obj.sweep^2) + 3*obj.sweep*(obj.root_chord^2 +
obj.tip_chord^2));

            I_area_spanwise = I_area_root - Area_fin * z_cg^2;
            I_spanwise = obj.mass_per_fin * ( I_area_spanwise /
Area_fin + (obj.thickness^2 / 12) );

            I_xx = obj.num_fins * (I_chordwise +
obj.mass_per_fin * z_cg^2) + ...
                (obj.num_fins / 2) * (I_spanwise +
obj.mass_per_fin * d^2);

            I_yy = I_xx; % By symmetry

            I_zz = obj.num_fins * ( I_spanwise +
obj.mass_per_fin * d^2 );

            I = diag([I_xx, I_yy, I_zz]);

        end

    end

end
```

Capítulo 12. Trajectory code\Rocket\Mass.m

```
classdef Mass < SubRocketPart
    % Represents a simple mass component that can be modeled as
a point mass or simple cylinder.

    properties

        length = 0;
        diameter = 0;


    end

    methods

        function obj = Mass(name, mass, length, diameter,
parent_part)
            % Constructs an instance of the Mass class.
            %
            % Inputs:
            %   name - Name of the mass component (string).
            %   mass - Mass of the component (kg).
            %   length - Length of the mass (m).
            %   diameter - Diameter of the mass (m).
            %   parent_part - The RocketPart this component is
attached to.

            obj@SubRocketPart(name, mass, parent_part);
            obj.mass = mass;
            obj.length = length;
            obj.diameter = diameter;

        end

        function cg = compute_cg(obj)
            % Computes the center of gravity of the mass.
            %
            % Outputs:
            %   cg - Center of gravity [x, y, z] in meters,
relative to the part's origin.

            cg = [0, 0, obj.length / 2]; % Assuming uniform
density along length
```

```
        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the mass, modeled
as a cylinder.
            %
            % Outputs:
            %   I - Inertia tensor as a 3x3 matrix.

            r = obj.diameter / 2; % Approximate radius
            Ixx = (1/4) * obj.mass * r^2 + (1/12) * obj.mass *
obj.length^2;
            Iyy = Ixx;
            Izz = 0.5 * obj.mass * r^2;

            I = diag([Ixx, Iyy, Izz]);
        end
    end
end
```

Capítulo 13. Trajectory code\Rocket\Motor.m

```
classdef Motor < SubRocketPart
    % Represents the rocket motor, including propellant mass
and thrust characteristics.
    % TODO: Implement the evolution of CG and inertia as
propellant is consumed.

    properties (Access = private)

        exit_area;

    end

    properties

        p_sl = 101325; % Sea level atmospheric pressure (Pa)
        nozzle_diameter;
        length;
        total_mass;
        propellant_mass;
        thrust_curve; % Nx2 array: [time (s), thrust (N)]
    end

    methods

        function obj = Motor(name, total_mass, propellant_mass,
nozzle_diameter, length, thrust_curve, parent_part)
            % Constructs an instance of the Motor class.
            %
            % Inputs:
            %   name - Name of the motor (string).
            %   total_mass - Initial total mass of the motor
(casing + propellant) (kg).
            %   propellant_mass - Mass of the propellant (kg).
            %   nozzle_diameter - Diameter of the nozzle exit
(m).
            %   length - Length of the motor (m).
            %   thrust_curve - Nx2 array with time (s) and
thrust at sea level (N).
            %   parent_part - The RocketPart this component is
attached to.

            obj@SubRocketPart(name, total_mass, parent_part);
```

```
            obj.total_mass = total_mass;
            obj.propellant_mass = propellant_mass;
            obj.nozzle_diameter = nozzle_diameter;
            obj.length = length;
            obj.thrust_curve = thrust_curve;

            obj.exit_area = pi * (obj.nozzle_diameter / 2)^2;

        end

        function thrust = get_thrust_at_time(obj, time, p_atm)
            % Gets the thrust at a specific time, adjusted for
atmospheric pressure.
            %
            % Inputs:
            %   time - Time since ignition (s).
            %   p_atm - Ambient atmospheric pressure (Pa).
            %
            % Outputs:
            %   thrust - Thrust in Newtons (N).

            if time < 0 || time > obj.thrust_curve(end, 1)
                thrust = 0;
                return;
            end

            if nargin < 3 %! CHANGE ONCE THE ATMOSPHERE MODEL
IS IMPLEMENTED
                p_atm = 101325; % Default to sea level pressure
if not provided
            end

            thrust_sl = interp1(obj.thrust_curve(:, 1),
obj.thrust_curve(:, 2), time, 'linear', 0);

            thrust = thrust_sl + obj.exit_area * (obj.p_sl -
p_atm);

        end

        function mass = get_mass_at_time(obj, time)

            mass = 0;
            % TODO: Implement class equations assuming that the
```

```
propellant is BATES

        end

        function cg = get_cg_at_time(obj, time)

            cg = 0;
            % TODO: Implement class equations assuming that the
propellant is BATES

        end

        function I = get_inertia_at_time(obj, time)

            I = 0;
            % TODO: Implement class equations assuming that the
propellant is BATES

        end

    end
end
```

Capítulo 14. Trajectory code\Rocket\NoseCone.m

```
classdef NoseCone < RocketPart
    % Represents the nose cone of the rocket.
    % Calculates CG and inertia based on its shape.
    % TODO: Change once the shape is known for sure.

    properties

        shape; % 'conical', 'ogive', 'parabolic', 'elliptical'
        length; % Length of the nose cone [m]
        base_diameter; % Base diameter of the nose cone [m]
        parent_cylinder; % Reference to the parent rocket
cylinder

    end

    methods
        function obj = NoseCone(name, shape, length,
base_diameter, mass, parent_cylinder)
            % Constructs an instance of the NoseCone class.
            %
            % Inputs:
            %   name - Name of the nose cone (string).
            %   shape - Shape of the nose cone ('conical',
'ogive', etc.).
            %   length - Length of the nose cone (m).
            %   base_diameter - Diameter of the nose cone base
(m).
            %   mass - Mass of the nose cone (kg).
            %   parent_cylinder - Reference to the parent body
tube.

            obj@RocketPart(name, mass);

            obj.shape = shape;
            obj.length = length;
            obj.base_diameter = base_diameter;
            obj.parent_cylinder = parent_cylinder;

        end

        function cg = compute_cg(obj)
            % Computes the center of gravity of the nose cone.
```

```
            % CG is measured from the tip (z=0) of the nose
cone.
            %
            % Outputs:
            %   cg - Center of gravity [x, y, z] in meters,
relative to the part's origin.

            L = obj.length;

            switch lower(obj.shape)
                case 'conical'

                    cg_long = (2/3) * L;

                case 'ogive'

                    cg_long = 0.46 * L;

                case 'parabolic'

                    cg_long = (2/3) * L;

                case 'elliptical'

                    cg_long = (5/8) * L;

                otherwise

                    error('Unsupported nose cone shape: %s',
obj.shape);
            end

            cg = [0, 0, cg_long];
        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the nose cone
about its own center of gravity.
            %
            % Outputs:
            %   I - Inertia tensor as a 3x3 matrix.

            r = obj.base_diameter / 2;
            L = obj.length;
            m = obj.mass;
```

```
            switch lower(obj.shape)
                case 'conical'

                    Izz = (3/10) * m * r^2;
                    Ixx = m * ( (3/20)*r^2 + (3/80)*L^2 );

                case 'ogive'

                    % Uses conical as approximation
                    Izz = (3/10) * m * r^2;
                    Ixx = m * ( (3/20)*r^2 + (3/80)*L^2 );

                case 'parabolic'

                    Izz = (1/3) * m * r^2;
                    Ixx = m * ( (1/4)*r^2 + (1/9)*L^2 );

                case 'elliptical'

                    Izz = (2/5) * m * r^2;
                    Ixx = m * ( (1/5)*r^2 + (5/32)*L^2 );

                otherwise

                    error('Unsupported nose cone shape: %s',
obj.shape);
            end

            Iyy = Ixx;

            I = diag([Ixx, Iyy, Izz]);
        end
    end
end
```

Capítulo 15. Trajectory code\Rocket\Parachute.m

```
classdef Parachute < SubRocketPart
    % Represents a parachute, including its deployment
characteristics.

    properties
        diameter;
        drag_coefficient;
        deployment_altitude;
        length;
    end

    methods
        function obj = Parachute(name, diameter,
drag_coefficient, deployment_altitude, length, mass,
parent_part)
            % Constructs an instance of the Parachute class.
            %
            % Inputs:
            %   name - Name of the parachute (string).
            %   diameter - Deployed diameter of the parachute
(m).
            %   drag_coefficient - Drag coefficient when
deployed.
            %   deployment_altitude - Altitude at which the
parachute deploys (m).
            %   length - Packed length of the parachute
component (m).
            %   mass - Mass of the parachute (kg).
            %   parent_part - The part this component is
attached to.

            obj@SubRocketPart(name, mass, parent_part);

            obj.diameter = diameter;
            obj.drag_coefficient = drag_coefficient;
            obj.deployment_altitude = deployment_altitude;
            obj.length = length;

        end

        function cg = compute_cg(obj)
            % Computes the center of gravity of the packed
```

```
parachute.
            %
            % Outputs:
            %   cg - The center of gravity [x, y, z] in meters,
relative to the part's origin.

            cg = [0, 0, obj.length / 2]; % Center of gravity is
at half the length

        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the packed
parachute, modeled as a cylinder.
            %
            % Outputs:
            %   I - The inertia tensor as a 3x3 matrix.

            r = 0.05; % Assuming a packed radius of 5cm, since
diameter is for deployed state
            l = obj.length;
            m = obj.mass;

            Ixx = (1/12) * m * (3*r^2 + l^2);
            Iyy = Ixx;
            Izz = (1/2) * m * r^2;

            I = diag([Ixx, Iyy, Izz]);
        end
    end
end
```

Capítulo 16. Trajectory code\Rocket\Rocket.m

```
classdef Rocket
    % Represents the entire rocket assembly, composed of 
multiple RocketPart objects.
    % This class manages the collection of rocket parts and 
calculates
    % the rocket's overall mass, center of gravity (CG), and 
inertia at any given time.

    properties (Access = private)
        OVERWRITTEN_LENGTH = false;
    end

    properties
        rocketName;
        parts = {};
        length;
        parachute; % Implementation for a single parachute
    end

    methods

        function obj = Rocket(name, length)
            % Constructor for the Rocket class.
            obj.rocketName = name;

            if nargin > 1 && length > 0
                obj.OVERWRITTEN_LENGTH = true;
                obj.length = length;
            else
                obj.length = 0;
            end

        end

        function obj = add_part(obj, part)
            % Adds a RocketPart to the rocket.
            %
            % Inputs:
            %   part - An instance of a class that inherits 
from RocketPart.

            % Add the new part to the cell array
```

```
            obj.parts{end + 1} = part;

            if isa(part, "Parachute")
                obj.parachute = part;
            end

            if length(obj.parts) > 1 && ~isa(part,
"SubRocketPart")

                last_part = obj.parts{end - 1};

                part.position = [0, 0, last_part.position(3) +
last_part.length];
                obj.parts{end} = part;

                obj = obj.update_length(part);

            end
        end

        function total_mass = get_total_mass(obj, time)
            % Calculates the total mass of the rocket at a
given time.
            %
            % Inputs:
            %   time - Time in seconds (s).
            %
            % Outputs:
            %   total_mass - Total mass of the rocket (kg).

            total_mass = 0;

            for i = 1:length(obj.parts)
                part = obj.parts{i}; % Access part from cell
array
                if isa(part, "Motor")
                    total_mass = total_mass +
part.get_mass_at_time(time);
                else
                    total_mass = total_mass + part.mass;
                end

            end
        end
```

```
        function cg = get_cg(obj, time)
            % Calculates the center of gravity (CG) of the
rocket at a given time.
            %
            % Inputs:
            %   time - Time in seconds (s).
            %
            % Outputs:
            %   cg - Center of gravity position (m) from the
rocket's tip.

            total_mass = obj.get_total_mass(time);

            if total_mass == 0
                cg = 0;
                return;
            end

            moment_sum = 0;
            for i = 1:length(obj.parts)
                part = obj.parts{i}; % Access part from cell
array
                if isa(part, "Motor")
                    part_mass = part.get_mass_at_time(time);
                    part_cg_local = part.get_cg_at_time(time);
                    part_cg_wrt_tip = part.position(3) +
part_cg_local;
                else
                    part_mass = part.mass;
                    part_cg_local = part.compute_cg();
                    part_cg_wrt_tip = part.position(3) +
part_cg_local(3); % Assuming uniform density
                end

                moment_sum = moment_sum + part_mass *
part_cg_wrt_tip;

            end

            cg = moment_sum / total_mass;

        end

        function obj = update_length(obj, new_part)
```

```
            if obj.OVERWRITTEN_LENGTH
                return;
            end

            obj.length = obj.length + new_part.length;

        end

        function I = get_inertia(obj, time)

            % Calculates the moment of inertia of the rocket at
a given time.
            %
            % Inputs:
            %   time - Time in seconds (s).
            %
            % Outputs:
            %   I - Moment of inertia tensor (kg*m^2) about the
rocket's center of gravity.

            cg = obj.get_cg(time);
            I = zeros(3,3); % Initialize as a 3x3 zero matrix

            for i = 1:length(obj.parts)
                part = obj.parts{i}; % Access part from cell
array
                if isa(part, "Motor")

                    part_mass = part.get_mass_at_time(time);
                    part_cg_local = part.get_cg_at_time(time);
                    part_cg = part.position(3) + part_cg_local;
                    part_I = part.get_inertia_at_time(time);

                else

                    part_mass = part.mass;
                    part_cg_local = part.compute_cg();
                    part_cg = part.position(3) +
part_cg_local(3);
                    part_I = part.compute_inertia();

                end

                % Apply parallel axis theorem
                d_vec = [0, 0, part_cg - cg];
```

```
                d_sq = dot(d_vec, d_vec);
                I = I + part_I + part_mass * (d_sq * eye(3) -
d_vec' * d_vec);


            end
        end

        function F = get_thrust(obj, time)
            % Returns the total thrust produced by all motors
at a given time.
            %
            % Inputs:
            %   time - Time in seconds (s).
            %
            % Outputs:
            %   F - Total thrust force vector (N) in the
rocket's body frame.

            F = [0; 0; 0]; % Initialize thrust vector

            for i = 1:length(obj.parts)
                part = obj.parts{i}; % Access part from cell
array
                if isa(part, "Motor")
                    thrust_mag = part.get_thrust_at_time(time);
                    F = F + [0; 0; thrust_mag]; % Assuming
thrust along the z-axis
                end
            end
        end
    end
end
```

Capítulo 17. Trajectory code\Rocket\RocketPart.m

```
classdef RocketPart
    % Represents a generic component of a rocket.
    % This is an abstract base class that defines the common 
interface for all
    % rocket parts, including properties like name, mass, and 
position, and
    % methods for calculating center of gravity and inertia.

    properties
        name
        mass = 0.0; % Mass is overriden when instatiating the 
object
        position = [0, 0, 0]; % Position of the part in 3D 
space [x, y, z]
    end

    methods

        function obj = RocketPart(name, mass)

            obj.name = name;
            obj.mass = mass;

        end

        function cg = compute_cg(obj)
            % Computes the center of gravity of the rocket part.
            %
            % Outputs:
            %   cg - Center of gravity [x, y, z] in meters.

            cg = [0, 0, 0]; % Placeholder for center of gravity 
calculation
        end

        function I = compute_inertia(obj)
            % Computes the inertia tensor of the rocket part.
            %
            % Outputs:
            %   I - Inertia tensor as a 3x3 matrix.

            I = eye(3); % Placeholder for inertia tensor
```

```
calculation
        end

        function obj = update_position(obj, new_position)
            % Updates the position of the rocket part.
            %
            % Inputs:
            %   new_position - New position [x, y, z] in meters.

            obj.position = new_position;
        end

    end

end
```

Capítulo 18. Trajectory code\Rocket\SubRocketPart.m

```
classdef SubRocketPart < RocketPart
    % Represents a component that is attached to a parent part.
    % This class extends RocketPart to include a reference to a
parent
    % and provides a method to update its position relative to
that parent.

    properties
        parent_part % The part this component is attached to
    end

    methods
        function obj = SubRocketPart(name, mass, parent_part)
            % Constructs an instance of the SubRocketPart class.
            %
            % Inputs:
            %   name - Name of the sub-part (string).
            %   mass - Mass of the sub-part (kg).
            %   parent_part - The RocketPart this component is
attached to.

            obj@RocketPart(name, mass);
            obj.parent_part = parent_part;
        end

        function obj = update_position(obj, relative_position)
            % Updates the part's absolute position based on its
parent's position.
            %
            % Inputs:
            %   relative_position - The [x, y, z] position
relative to the parent part's origin (m).

            if ~isempty(obj.parent_part) &&
isprop(obj.parent_part, 'position')
                obj.position = obj.parent_part.position +
relative_position;
            else
                obj.position = relative_position;
            end
        end
```

```
    end
end
```