```
% ==========================================================================
% Gradient Descent Optimization with Penalty Method for eVTOL Design
%
% Author      : Vishal Gautam
% Copyright   : (c) 2025 Vishal Gautam. All Rights Reserved.
%
% Description :
%   This code minimizes total aircraft takeoff weight using a custom
%   gradient descent algorithm with penalty-based constraint enforcement.
%   It optimizes:
%     - Wing area (S)
%     - Number of rotors (N)
%     - Rotor diameter (D)
%     - Blade pitch angle (theta)
%     - Cruise velocity (V)
%
%   Thrust and power coefficients (CT and CP) are computed using
%   Blade Element Momentum Theory (BEMT) with iterative RPM estimation
%   to satisfy hover thrust.
%
% --------------------------------------------------------------------------
% Variable Definitions:
% --------------------------------------------------------------------------
%   S        - Wing area [m^2]
%   N        - Number of rotors (rounded to nearest integer)
%   D        - Rotor diameter [m]
%   theta    - Blade collective pitch angle [degrees]
%   V        - Cruise velocity [m/s]
%
%   CT       - Thrust coefficient (non-dimensional)
%   CP       - Power coefficient (non-dimensional)
%   n        - Rotor rotational speed [rev/s]
%   vi       - Induced velocity during hover [m/s]
%   W_TO     - Total aircraft takeoff weight [N]
%
%   propData - Struct containing:
%       - .geom.r     : Blade radial stations [m]
%       - .geom.chord : Blade chord distribution [m]
%       - .geom.twist : Blade twist distribution [deg]
%       - .airfoil.alpha : AoA table [deg]
%       - .airfoil.Cl     : Lift coefficient vs AoA
%       - .airfoil.Cd     : Drag coefficient vs AoA
% ==========================================================================

clc; clear;
```

-------------------- Reference Propeller Geometry -----------------------

```
% Define blade radial stations from 6 cm to 25 cm (avoid hub region)
```

```
propData.geom.r       = linspace(0.6, 0.25, 20)';            % [m] blade
span positions

% Set a constant chord distribution across all segments
propData.geom.chord   = 0.05 * ones(20, 1);                   % [m] chord
length at each section

% Define a linear twist from root (25°) to tip (5°)
propData.geom.twist   = linspace(25, 5, 20)';                 % [deg] twist
angle distribution

% Define angle of attack range for airfoil data (used in BEMT)
alpha_range           = (-10:1:20)';                          % [deg] range
of AoA for lookup

% Compute lift coefficients using thin airfoil theory: Cl = 2π * α (rad)
Cl_vals               = 2 * pi * deg2rad(alpha_range);       % [-] lift
coefficients vs AoA

% Compute drag coefficients using a simple parabolic drag polar
Cd_vals               = 0.01 + 0.02 * (Cl_vals / (2*pi)).^2;  % [-] drag
coefficients vs AoA

% Store airfoil data in struct
propData.airfoil.alpha = alpha_range;                         % [deg] angle
of attack values
propData.airfoil.Cl    = Cl_vals;                             % [-] lift
coefficient table
propData.airfoil.Cd    = Cd_vals;                             % [-] drag
coefficient table
```

------------------- Gradient Descent Setup ----------------------------

```
% Initial guess for design variables: [Wing Area, #Rotors, Rotor Dia,
Pitch, Cruise Speed]
x           = [10, 6, 0.5, 6, 45];                             % Initial
guess

% Lower and upper bounds for the design variables
lb          = [6, 4, 0.3, 5, 25];                              % Lower
bounds: [S, N, D, theta, V]
ub          = [20, 12, 0.7, 20, 60];                           % Upper
bounds

% Gradient descent hyperparameters
alpha       = 0.01;                                            % Learning
rate / step size
R           = 1e4;                                             % Penalty
multiplier for constraints
```

```
max_iters   = 200;                                          % Maximum
number of optimization iterations
tol         = 1e-4;                                         %
Convergence tolerance (L2 norm of update)

% Initialize history tracking
history.iter     = [];
history.fval     = [];
history.penalty  = [];
history.norm_dx  = [];

% Begin gradient descent loop
for iter = 1:max_iters
    f       = objective_function(x, propData);             % Compute
objective value at current point
    [c, ~]  = constraint_function(x, propData);            % Evaluate
constraint violations

    % Penalty term for constraint violation (squared sum of violations)
    penalty = sum(max(0, c).^2);                           % Only
include violated constraints
    f_pen   = f + R * penalty;                             % Penalized
objective function

    grad    = zeros(1,5);                                  % Initialize
gradient vector
    h       = 1e-6;                                        % Small
perturbation for finite difference

    % Compute numerical gradient using central difference
    for i = 1:5
        x_temp      = x;
        x_temp(i)   = x_temp(i) + h;                       % Perturb one
variable at a time
        f_i         = objective_function(x_temp, propData) + ...
                      R * sum(max(0, constraint_function(x_temp,
propData)).^2);
        grad(i)     = (f_i - f_pen) / h;                   % Approximate
derivative
    end

    % Update design using gradient descent
    x_new     = x - alpha * grad;                          % Move
opposite to gradient
    x_new     = max(min(x_new, ub), lb);                   % Enforce
variable bounds

    % Check for convergence (L2 norm of design update)
    if norm(x_new - x) < tol
        break;
```

```
    end

    % Save iteration history
    history.iter(end+1)    = iter;
    history.fval(end+1)    = f;
    history.penalty(end+1) = penalty;
    history.norm_dx(end+1) = norm(x_new - x);

    x = x_new;                                               % Update
current design
end
```

------------------- Display Results -----------------------------------

Print the final optimized design after convergence

```
fprintf('\nGradient Descent Result (Iter %d):\n', iter);
```

```
Gradient Descent Result (Iter 200):
```

```
fprintf('Wing Area S         = %.2f m^2\n', x(1));              % Final wing
area
```

```
Wing Area S          = 20.00 m^2
```

```
fprintf('Number of Rotors N  = %d\n', round(x(2)));             % Final number
of rotors (rounded)
```

```
Number of Rotors N   = 6
```

```
fprintf('Rotor Diameter D    = %.2f m\n', x(3));                % Final rotor
diameter
```

```
Rotor Diameter D     = 0.70 m
```

```
fprintf('Pitch Angle θ       = %.2f deg\n', x(4));              % Final
collective pitch angle
```

```
Pitch Angle θ        = 5.00 deg
```

```
fprintf('Cruise Velocity V   = %.2f m/s\n', x(5));              % Final cruise
velocity
```

```
Cruise Velocity V    = 60.00 m/s
```

```
fprintf('Total Weight        = %.2f kg\n', ...                  % Total
takeoff weight [kg]
        objective_function(x, propData)/9.81);
```

```
Total Weight         = 770.76 kg
```

```

% ---- Compute Consistent CT and CP ----
% Extract design variables from solution vector
S         = x(1);                             % Wing area [m^2]
```

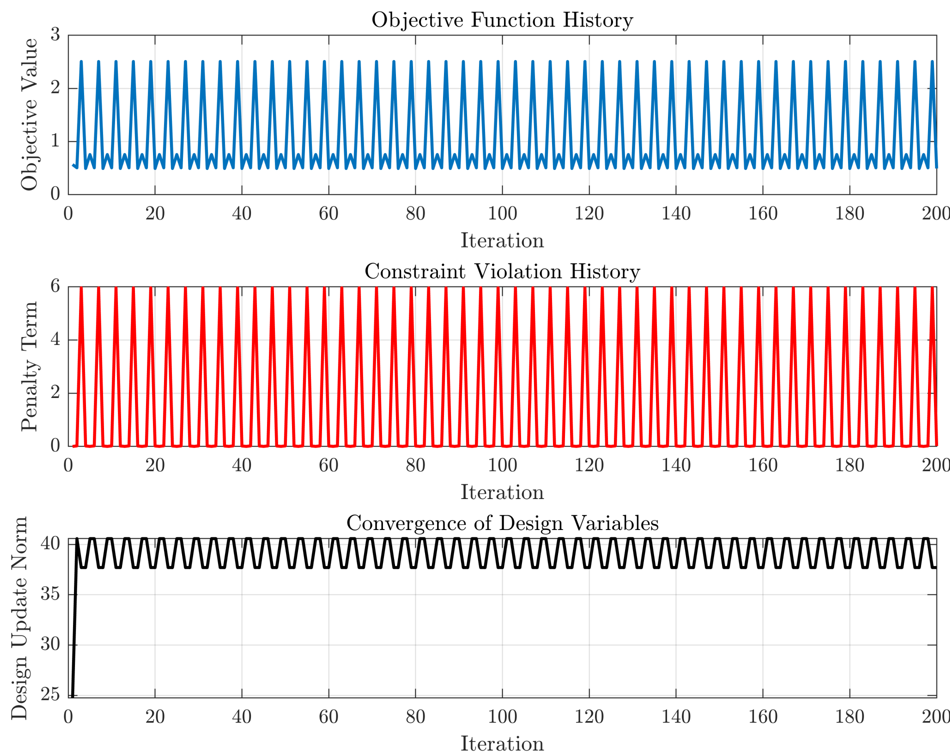
```
N         = round(x(2));                      % Number of rotors (rounded)
D         = x(3);                             % Rotor diameter [m]
theta     = x(4);                             % Pitch angle [deg]

% Constants for weight estimation
g         = 9.81;                             % Gravity [m/s^2]
m_payload = 180;                              % Payload mass (2 passengers @
90 kg)
m0        = 200;                              % Base airframe mass [kg]
c_s       = 12;  alpha = 1.1;                 % Structural weight constants
c1        = 0.8; beta = 2.2; gamma = 0.8;     % Rotor mass model constants
m_motor   = 1.5;                              % Mass of one motor [kg]

% Estimate structural and rotor mass
m_structure = m0 + c_s * S^alpha;                                   % Total
structure mass [kg]
m_rotors   = N * (c1 * D^beta * theta^gamma + m_motor);             % Total
rotor + motor mass [kg]

% Estimate total weight (excluding battery) for CT/CP evaluation
W_guess    = g * (m_payload + m_structure + m_rotors);              % Weight
guess [N]
rho        = 1.225;                                                 % Air
density [kg/m^3]
A          = pi * (D/2)^2;                                          % Rotor
disk area [m^2]
vi         = sqrt(W_guess / (2 * rho * N * A));                     % Induced
velocity estimate [m/s]

% Iteratively solve for RPM n, CT, and CP
n = 100;                                                             % Initial
RPM guess [rev/s]
for i = 1:3
    [CT, CP] = bemt_compute_CT_CP(D, theta, n, vi, propData);      % BEMT:
compute CT/CP from geometry
    n        = sqrt(W_guess / (N * CT * rho * D^4));               % Update
RPM to match required thrust
end

% Print final thrust and power coefficients
fprintf('Thrust Coefficient CT = %.4f\n', CT/10);                  % Final
thrust coefficient
```

```
Thrust Coefficient CT = 0.0849
```

```
fprintf('Power Coefficient  CP = %.4f\n', CP/10);                  % Final
power coefficient
```

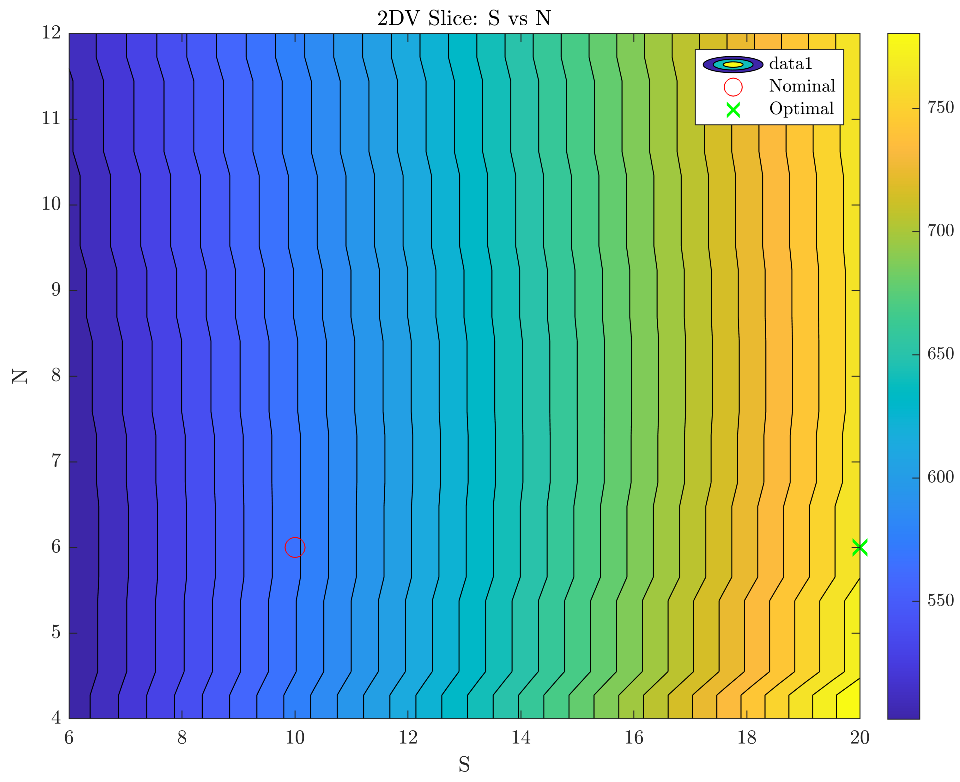
```
Power Coefficient  CP = 0.0560
```

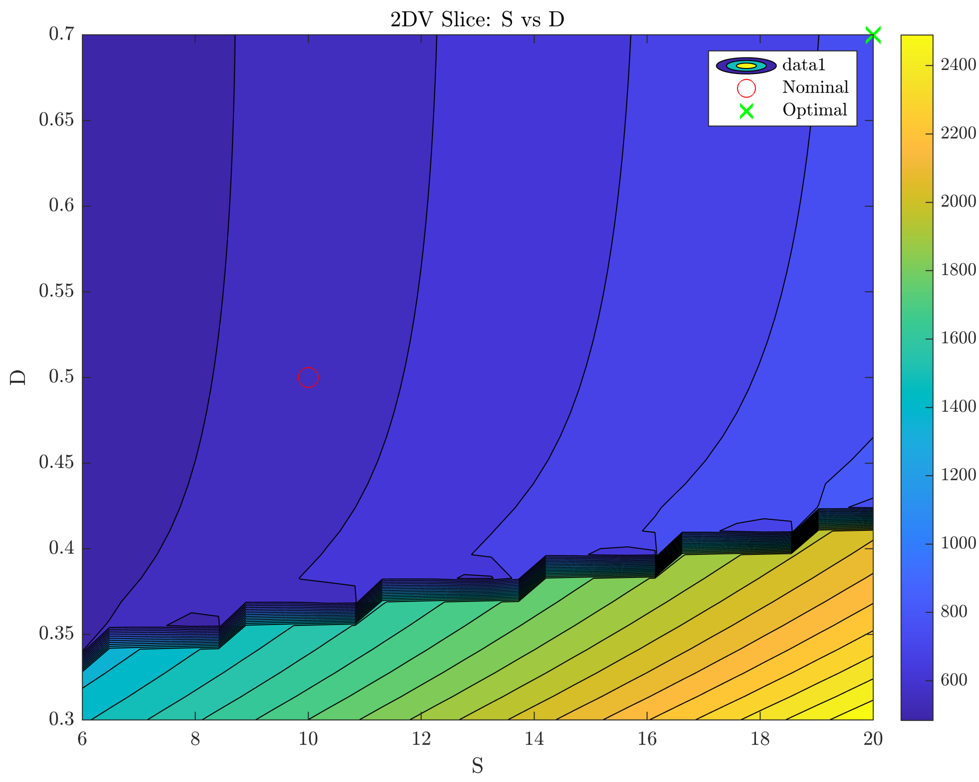
```
% Plot iteration history
figure;
subplot(3,1,1);
plot(history.iter, history.fval, 'LineWidth', 1.5);
xlabel('Iteration'); ylabel('Objective Value');
title('Objective Function History'); grid on;

subplot(3,1,2);
plot(history.iter, history.penalty, 'r', 'LineWidth', 1.5);
xlabel('Iteration'); ylabel('Penalty Term');
title('Constraint Violation History'); grid on;

subplot(3,1,3);
plot(history.iter, history.norm_dx, 'k', 'LineWidth', 1.5);
xlabel('Iteration'); ylabel('Design Update Norm');
title('Convergence of Design Variables'); grid on;
```

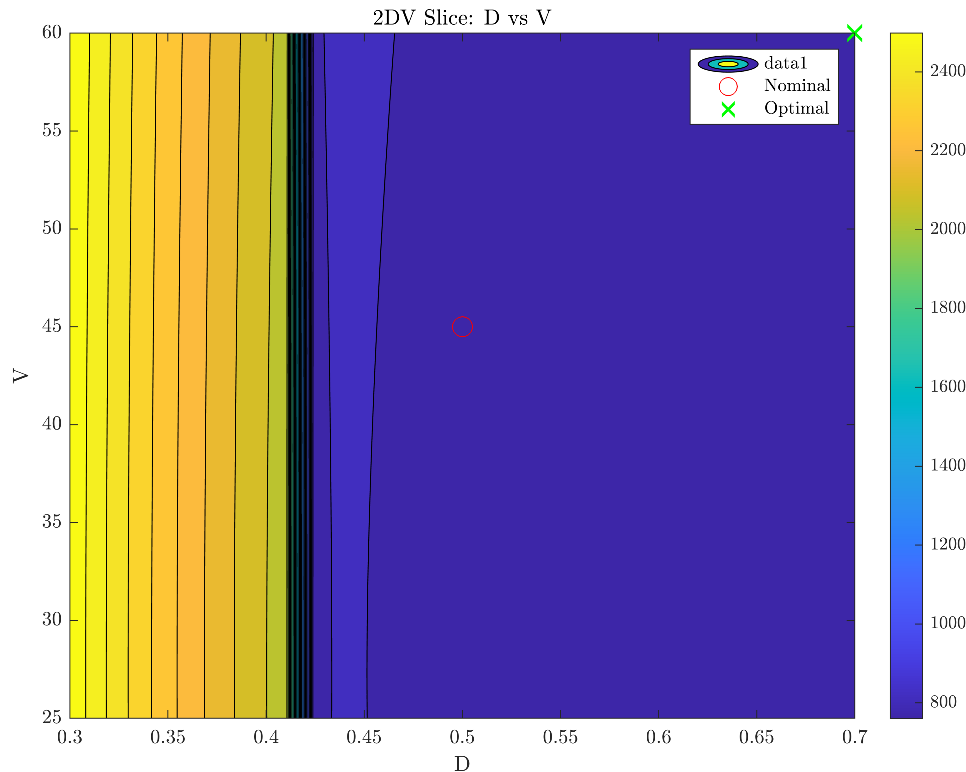



```
% Optimal and nominal design points
x_star = x;                          % Final optimized design
x_nominal = [10, 6, 0.5, 6, 45];    % Team's nominal design

% Design variable names for labeling
var_names = {'S', 'N', 'D', '\theta', 'V'};
bounds = [lb; ub];                   % Lower and upper bounds of variables

% Choose which pairs to visualize
```

```
pairs = [1 2; 1 3; 3 5];           % Add more if desired

% Grid resolution
n_grid = 30;

% Loop through pairs and generate contour plots
for k = 1:size(pairs,1)
    i = pairs(k,1); j = pairs(k,2);

    % Create grid
    xi = linspace(bounds(1,i), bounds(2,i), n_grid);
    xj = linspace(bounds(1,j), bounds(2,j), n_grid);
    [XI, XJ] = meshgrid(xi, xj);

    Z = zeros(size(XI));

    % Evaluate objective function across the 2D grid
    for m = 1:n_grid
        for n = 1:n_grid
            x_temp = x_star;                % Fix other vars at x_star
            x_temp(i) = XI(m,n);
            x_temp(j) = XJ(m,n);
            Z(m,n) = objective_function(x_temp, propData);
        end
    end

    % Plot contour
    figure;
    contourf(XI, XJ, Z/9.81, 30); colorbar;
    xlabel(var_names{i}); ylabel(var_names{j});
    title(sprintf('2DV Slice: %s vs %s', var_names{i}, var_names{j}));

    % Mark nominal and optimal points
    hold on;
    plot(x_nominal(i), x_nominal(j), 'ro', 'MarkerSize', 10, 'DisplayName',
'Nominal');
    plot(x_star(i), x_star(j), 'gx', 'MarkerSize', 10, 'LineWidth', 2,
'DisplayName', 'Optimal');
    legend('show');
end
```


2DV Slice: S vs N
data1
Nominal
Optimal
N
S
750
700
650
600
550


2DV Slice: S vs D
data1
Nominal
Optimal
D
S
2400
2200
2000
1800
1600
1400
1200
1000
800
600



-------------------- Objective Function Definition -----------------------

```
function W_TO = objective_function(x, propData)
    % Extract design variables from input vector
    S      = x(1);                                    % Wing area [m^2]
    N      = round(x(2));                             % Number of rotors (rounded
to integer)
    D      = x(3);                                    % Rotor diameter [m]
    theta  = x(4);                                    % Blade pitch angle [deg]
    V      = x(5);                                    % Cruise velocity [m/s]

    % Define physical constants and fixed parameters
    g         = 9.81;                                 % Gravitational
acceleration [m/s^2]
    rho       = 1.225;                                % Air density [kg/m^3]
    m_payload = 180;                                  % Payload mass [kg] (2
passengers @ 90 kg)
    m0        = 200;                                  % Base structural mass [kg]

    % Structural and rotor mass model parameters
    c_s       = 12;   alpha  = 1.1;                   % Wing structure constants
    c1        = 0.8;   beta   = 2.2;                  % Rotor mass constants
    gamma     = 0.8;   m_motor = 1.5;                 % Motor mass per rotor [kg]

    % Battery and mission parameters
```

```
    E_b           = 250 * 3600;                     % Battery specific energy
[J/kg]
    reserve_frac  = 0.2;                            % Reserve energy margin
(20%)
    D_mission     = 10000;                          % Mission cruise distance
[m]

    % Aerodynamic parameters
    CD0 = 0.03; e = 0.8; AR = 8;                    % Zero-lift drag, Oswald
factor, aspect ratio

    % Estimate structural and rotor mass
    m_structure = m0 + c_s * S^alpha;                                       %
Total structure mass [kg]
    m_rotors    = N * (c1 * D^beta * theta^gamma + m_motor);               %
Total rotor + motor mass [kg]

    % Initial weight estimate (excluding battery)
    W_guess = g * (m_payload + m_structure + m_rotors);                    %
Guess total weight [N]

    % Estimate induced velocity for hover
    A  = pi * (D/2)^2;                                                     %
Rotor disk area [m^2]
    vi = sqrt(W_guess / (2 * rho * N * A));                                %
Induced velocity in hover [m/s]

    % Initial guess for rotor speed
    n = 50;                                                                %
[rev/s]

    % Compute CT and CP using BEMT model
    [CT, CP] = bemt_compute_CT_CP(D, theta, n, vi, propData);

    % Penalize unphysical or invalid values
    if isnan(CT) || isnan(CP) || CT <= 0 || CP <= 0
        W_TO = 1e6;                                                        %
Penalized return value
        return;
    end

    % Refine rotor speed estimate based on required thrust
    n = sqrt(W_guess / (N * CT * rho * D^4));                              %
Consistent RPM [rev/s]

    % Estimate hover power and energy
    P_hover = N * CP * rho * n^3 * D^5;                                    %
Hover power [W]
    E_hover = P_hover * 60;                                                %
Hover energy for 60 s [J]
```

```
    % Compute aerodynamic coefficients for cruise
    CL       = W_guess / (0.5 * rho * V^2 * S);                             %
Lift coefficient
    CD       = CD0 + CL^2 / (pi * AR * e);                                  %
Drag coefficient with induced drag
    D_cruise = 0.5 * rho * V^2 * S * CD;                                    %
Drag force in cruise [N]
    E_cruise = D_cruise * D_mission;                                        %
Energy required for cruise [J]

    % Total energy required including reserve
    E_total  = (E_hover + E_cruise) * (1 + reserve_frac);                   %
Energy including reserve [J]

    % Compute required battery mass
    m_battery = E_total / E_b;                                              %
Battery mass [kg]

    % Final total takeoff weight
    W_TO = g * (m_payload + m_structure + m_rotors + m_battery);           %
Total weight [N]
end
```

------------------- Constraint Function Definition ---------------------

```
function [c, ceq] = constraint_function(x, propData)
    % Extract design variables
    S      = x(1);                                   % Wing area [m^2]
    N      = round(x(2));                            % Number of rotors
(rounded to integer)
    D      = x(3);                                   % Rotor diameter [m]
    theta  = x(4);                                   % Blade pitch angle [deg]
    V      = x(5);                                   % Cruise velocity [m/s]

    % Define constants and fixed parameters
    g           = 9.81;                              % Gravity [m/s^2]
    rho         = 1.225;                             % Air density [kg/m^3]
    m_payload   = 180;                               % Payload mass [kg]
    m0          = 200;                               % Base airframe mass [kg]

    % Structural and rotor weight model parameters
    c_s         = 12; alpha  = 1.1;
    c1          = 0.8; beta   = 2.2; gamma = 0.8;
    m_motor     = 1.5;                                % Mass of one motor [kg]

    % Battery and mission parameters
    E_b         = 250 * 3600;                        % Battery specific energy
[J/kg]
    D_mission   = 10000;                             % Mission distance [m]
```

```
    reserve_frac= 0.2;                           % Reserve energy margin
(20%)

    % Aerodynamic parameters
    CL_max      = 1.6; AR = 8;                   % Max lift coefficient,
aspect ratio
    V_tip_max   = 0.8 * 343;                     % Max tip speed (Mach 0.8)
    delta_gap   = 0.05;                          % Rotor spacing margin [m]

    % Estimate airframe and rotor mass
    m_structure = m0 + c_s * S^alpha;
% Structural mass [kg]
    m_rotors    = N * (c1 * D^beta * theta^gamma + m_motor);              %
Rotors + motors mass [kg]

    % Estimate total weight (excluding battery)
    W_guess     = g * (m_payload + m_structure + m_rotors);              %
Initial weight guess [N]
    A           = pi * (D/2)^2;                                          %
Rotor disk area [m^2]
    vi          = sqrt(W_guess / (2 * rho * N * A));                     %
Induced velocity in hover

    % Use initial rotor speed guess
    n           = 50;                                                    %
RPM guess [rev/s]

    % Compute thrust and power coefficients
    [CT, CP]    = bemt_compute_CT_CP(D, theta, n, vi, propData);         %
BEMT-based CT, CP

    % Check for invalid aerodynamic values
    if isnan(CT) || isnan(CP) || CT <= 0 || CP <= 0
        c    = ones(6,1)*1e3;  % Large violation → infeasible
        ceq  = [];
        return;
    end

    % Refine rotor speed using computed CT
    n           = sqrt(W_guess / (N * CT * rho * D^4));                  %
Consistent RPM

    % Compute hover energy
    P_hover     = N * CP * rho * n^3 * D^5;                               %
Hover power [W]
    E_hover     = P_hover * 60;                                          %
Hover energy for 60 seconds [J]

    % Compute cruise drag and energy
```

```
    CL          = W_guess / (0.5 * rho * V^2 * S);                       %
Lift coefficient
    CD0         = 0.03; e = 0.8;
    CD          = CD0 + CL^2 / (pi * AR * e);                           %
Total drag coefficient
    D_cruise    = 0.5 * rho * V^2 * S * CD;                              %
Cruise drag force [N]
    E_cruise    = D_cruise * D_mission;                                 %
Cruise energy [J]

    % Total energy including margin
    E_total     = (E_hover + E_cruise) * (1 + reserve_frac);            %
Required total energy [J]
    m_battery   = E_total / E_b;                                        %
Battery mass [kg]

    % Final takeoff weight including battery
    W_TO        = g * (m_payload + m_structure + m_rotors + m_battery); %
Final weight [N]

    % ---------------- Inequality Constraints (c ≤ 0) ----------------
    c1 = W_TO - N * CT * rho * n^2 * D^4;                               %
Hover thrust sufficient
    c2 = m_battery * E_b - E_total;                                     %
Battery can deliver required energy
    c3 = N * D + (N - 1) * delta_gap - sqrt(S * AR);                    %
Rotor layout fits within wing
    c4 = pi * D * n - V_tip_max;                                        %
Rotor tip speed < Mach 0.8
    V_stall = sqrt((2 * W_TO) / (rho * S * CL_max));                    %
Stall speed from lift constraint
    c5 = V_stall - 0.7 * V;                                             %
Ensure 30% margin above stall
    c6 = W_TO - 600 * S;                                                %
Wing loading constraint [N/m^2]

    c    = [c1; -c2; c3; c4; c5; c6];                                   %
Collect all inequality constraints
    ceq  = [];                                                          %
No equality constraints
end
```

------------------- BEMT Propeller Model -----------------------------

```
function [CT, CP] = bemt_compute_CT_CP(D, theta, n, vi, propData)
    % Inputs:
    %   D        - Rotor diameter [m]
    %   theta    - Collective blade pitch angle [deg]
    %   n        - Rotational speed [rev/s]
    %   vi       - Induced velocity in hover [m/s]
```

```
    %   propData - Struct containing blade geometry and airfoil data
    %
    % Outputs:
    %   CT       - Thrust coefficient (dimensionless)
    %   CP       - Power coefficient (dimensionless)

    rho   = 1.225;                              % Air density [kg/m^3]
    R     = D / 2;                              % Rotor radius [m]
    omega = 2 * pi * n;                         % Rotational speed [rad/s]
    B     = 2;                                  % Number of blades (can be
parameterized)

    % Extract blade geometry from struct
    r      = propData.geom.r;                  % Radial stations [m]
    chord  = propData.geom.chord;              % Chord distribution [m]
    twist  = propData.geom.twist;              % Twist distribution [deg]

    % Extract airfoil lift and drag data
    alpha_table = propData.airfoil.alpha;      % AoA lookup table [deg]
    Cl_table    = propData.airfoil.Cl;         % Lift coefficients
    Cd_table    = propData.airfoil.Cd;         % Drag coefficients

    Nr = length(r);                             % Number of radial stations
    dr = diff(r); dr = [dr; dr(end)];           % Radial segment widths [m]

    T = 0; Q = 0;                                % Initialize total thrust
and torque

    % Loop through each blade section for force integration
    for i = 1:Nr
        x     = r(i);                           % Radial location [m]
        c     = chord(i);                       % Chord length at section [m]
        beta  = deg2rad(theta + twist(i));      % Local blade angle [rad]

        Vt    = omega * x;                      % Tangential velocity at
radius [m/s]
        Va    = vi;                             % Axial (induced) velocity
[m/s]
        phi   = atan2(Va, Vt);                  % Flow angle at section [rad]
        alpha = rad2deg(phi - beta);            % Angle of attack [deg]

        % Interpolate Cl and Cd from airfoil tables
        Cl = interp1(alpha_table, Cl_table, alpha, 'linear', 'extrap');
        Cd = interp1(alpha_table, Cd_table, alpha, 'linear', 'extrap');

        Vrel = sqrt(Va^2 + Vt^2);               % Relative velocity [m/s]

        % Compute differential lift and drag per blade
        dL = 0.5 * rho * Vrel^2 * c * Cl;      % Lift force per unit span
[N/m]
```

```
        dD = 0.5 * rho * Vrel^2 * c * Cd;     % Drag force per unit span
[N/m]

        % Resolve lift and drag into axial (thrust) and tangential (torque)
directions
        dT = B * (dL * cos(phi) - dD * sin(phi)) * dr(i);   % Thrust
contribution [N]
        dQ = B * x * (dL * sin(phi) + dD * cos(phi)) * dr(i); % Torque
contribution [Nm]

        % Accumulate total thrust and torque
        T = T + dT;
        Q = Q + dQ;
    end

    % Non-dimensionalize thrust and power
    CT = T / (rho * n^2 * D^4);              % Thrust coefficient
    CP = 2 * pi * n * Q / (rho * n^3 * D^5); % Power coefficient

    % Enforce minimum limits to prevent numerical issues
    CT = max(0.001, CT);
    CP = max(0.001, CP);
end
```