Jesse Wynn
ME 575
Genetic Algorithm Project
April 10, 2018

Genetic Optimization for Multirotor UAV Design

Section I. Project Summary

Introduction

Multirotor unmanned air vehicles (UAVs) are becoming an increasingly valuable asset in a variety of professional fields. Their vertical take-off and landing capability makes them particularly well-suited for a broad range of tasks. A primary weakness of multirotor aircraft however is limited flight time. Typical multirotor UAVs have a flight time of 20 to 35 minutes. While the greatest factor in limited flight time is battery energy density, great improvements in flight time can be achieved through optimal multirotor power systems configurations. In this project I implement a genetic optimization algorithm to find the optimal set of motors, propellers, and batteries in order to maximize multirotor flight time.

Objective (Primary)

Maximize Total Flight Time

Chromosome Fitness: `fitness = flight_time`

Algorithm Parameters

Generation Size: `N = 50`
Number of Generations: `n_gen = 20`
Crossover Probability: `p_crossover = 0.50`
Mutation Probability: `p_mutation = 0.10`
Roulette Exponent: roulette_exponent = `= 2`
Dynamic Mutation parameter, β: `beta = 1`

Algorithm Methods

Selection: Roulette Wheel Selection
Crossover: Blend Crossover & Point Crossover
Mutation: Dynamic Mutation
Elitism: Most-fit of current and previous generation

Optimal Design

```
optimal_design = [1, 2, 1, 4, 1, 1]
optimal_fitness = 0.9643 (hours)
```

Section II. Procedure

Chromosome and Fitness

The first step in implementing the genetic algorithm was to define the algorithm chromosome or set of design variables. For this problem, the chromosome consists of six integer variables that each represent an aspect of the multirotor's design. A complete description of the chromosome is given in Figure 1.

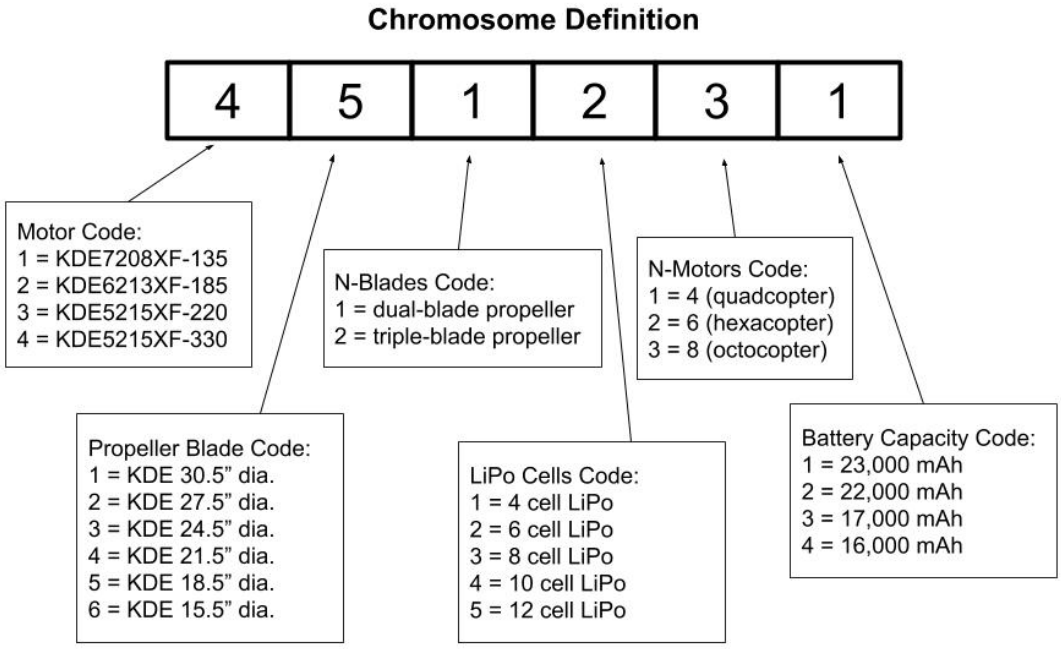

Figure 1. Chromosome representing the six design variables

After defining the chromosome, the next step is to define chromosome fitness and to construct a model to evaluate the fitness of any given chromosome. Since the goal of the optimization is to maximize flight time, fitness was simply defined as total predicted flight time for the given design. Evaluation of chromosome fitness proceeded as follows and can be seen in *compute_fitness.m* given in the Appendix:

1) Compute total mass of the design defined by the chromosome
   - *mass = f(motor, prop-dia, n-motors, battery, ...)*
   - See *compute_mass.m* in Appendix
2) Compute required thrust from each motor needed to hover
3) Look-Up the current draw for the given motor, propeller, and battery at the required thrust
   - For the given design, fit a polynomial curve to the motor, propeller, and battery data obtained from KDE Direct (brushless DC motor manufacturer)
   - Evaluate the individual motor current draw using the polynomial fit
   - See *get_amps.m* in Appendix
4) Compute the total current draw at hover for the given design
   - *total_current = individual_current * n_motors*
5) Compute total predicted flight time (fitness)

- *predicted_flight_time = battery_capacity / total_current*

Note: As mentioned in step 3 above, the current draw for a given design was approximated according to published motor testing data from the motor manufacturer. Although there is bound to be some error in the polynomial curve fit (since the manufacturer only supplies limited data points), using actual test data and constructing 'Look-Up' routines was identified as being a more accurate approach than to attempt to fully define a mathematical model to predict motor performance. Motor-propeller performance is particularly difficult to model due to many sources of uncertainty (actual aerodynamic forces and coefficients, actual motor output, drag, friction, viscous effects, turbulent flow, etc.). Because the results of this optimization were intended to be realistic and usable, using actual test data was determined to be the best route to success.

With the ability to evaluate fitness of any given design, the genetic process can now begin.

Genetic Process

The genetic process consists of four primary steps namely selection, crossover, mutation, and elitism. Details on each of these steps are given below:

Selection—For this algorithm a 'roulette wheel' selection routine was chosen and implemented. In this method, parent chromosomes are randomly selected from the current generation with chromosomes having greater fitness having a greater chance of being selected. To slightly increase fitness pressure, a roulette exponent of 2 was used. Further details of this method can be seen in *roulette_selection.m* given in the Appendix.

Crossover—Two types of crossover were implemented and experimented with in the tuning of the algorithm. The first was 'blend crossover'. This method has the advantage of being able to mix the genes of the parents in order to produce a child. The second was 'single point crossover'. In this method crossover takes place by simply swapping the genetic material of the two parent chromosomes after a randomly chosen crossover point. In both cases the crossover probability was dialed in to be 0.5. After several trials, blend crossover was found to be slightly more robust at producing fit children than point crossover (although in most cases, both methods produced good results). See *blend_crossover.m* and *point_crossover.m* in the Appendix for more details.

Mutation—To introduce variety into each generation, a mutation step was performed. In this genetic algorithm, a 'dynamic mutation' routine was implemented. Dynamic mutation has the advantage of allowing greater mutations in earlier generations, and less mutation in later generations—which is oftentimes favorable. Through experimentation, a mutation probability of 0.10 was found to give good results paired with the mutation parameter $\beta = 1$. See *dynamic_mutation.m* in the Appendix.

Elitism—To ensure that only the most-fit chromosomes survived to the next generation, an elitism step was performed on each generation. After a new generation has been formed through several iterations of selection, crossover, and mutation, only the N most-fit designs from the new-generation *and* the previous generation were actually chosen to represent the next generation. Implementation details of this step can be seen in *elitism.m* in the Appendix.

A generation size of 50 with a total of 20 generations was found to be a robust and efficient set of global algorithm parameters for consistently finding the global optimum design.

Section III. Results

Single Objective Results

The optimal design found by the genetic process is given below:

```
optimal_design = [1, 2, 1, 4, 1, 1]
Optimal Design Specifications:
motor: KDE7208XF-135
propeller: KDE 27.5 inch diameter dual-blade prop
motor configuration: 4 motors (quadcopter)
battery: 10S 37.0V 23,000 mAh LiPo
```

Which had a fitness of **0.9343 hours** or **56.1 minutes**. Since the space of this particular problem was not excessively large (approximately 3,000 possible designs), and all parameters have discrete values, a brute-force search for the optimum was also performed. This search verified that the optimal design found by the algorithm represented the **global optimum** of the design space.

The inspiration for this project came from a past internship where I was working with a pair of industrial multirotor UAVs. A chromosome representing their design is:

```
previous_design = [4, 5, 1, 2, 3, 1]
```

The optimal design given by the genetic algorithm represents a **119.8%** (more than double) increase in total flight time over this previous design! Evaluating the previous design was also a good validity check for the objective model. Using the objective model, the previous design resulted in a flight time (fitness) of 25.5 minutes. This almost exactly matches the performance I experienced with this particular design on my internship, and further increases confidence in the developed model.

In addition to finding the optimum, the genetic algorithm performed very consistently and steadily increased the overall fitness of designs with each successive generation. The algorithm found the global optimum ~95% of the time and found a solution within 5% of the global optimum 99% of the time. The plots in Figure 2 and Figure 3 show the average fitness as a function of generation number for two separate executions of the algorithm.

Dual objective Results

Maximizing flight time is an important factor in multirotor design but there are also other objectives to consider. One possible objective is total rotor-disc area. Rotor-disc area is simply the combined area of the lifting discs formed by the spinning rotors. In many cases reducing rotor-disc area is important so that the multirotor can maintain a lower aerodynamic profile—which naturally increases its resistance to wind disturbances. To explore the effect of also trying to minimize rotor disc area, I defined a second fitness function as:

```
fitness = rotor_disc_area
```

I then used this function along with the previous objective of maximizing flight time to compute the *Maximin* fitness of each design in the population (see *pareto.m* in the Appendix). Using this new fitness, I was able to identify the Pareto front of the dual objective space and a plot of this is given in Figure 4.

Based on the dual objective Pareto front, I selected the following as the ‘best’ overall design:

```
pareto_design* = [3, 4, 1, 5, 1, 1]
```

Selecting this design results in only a **10.3%** decrease in total flight time when compared to the single-objective optimum, and also gives a **38.9%** decrease in rotor disc area.

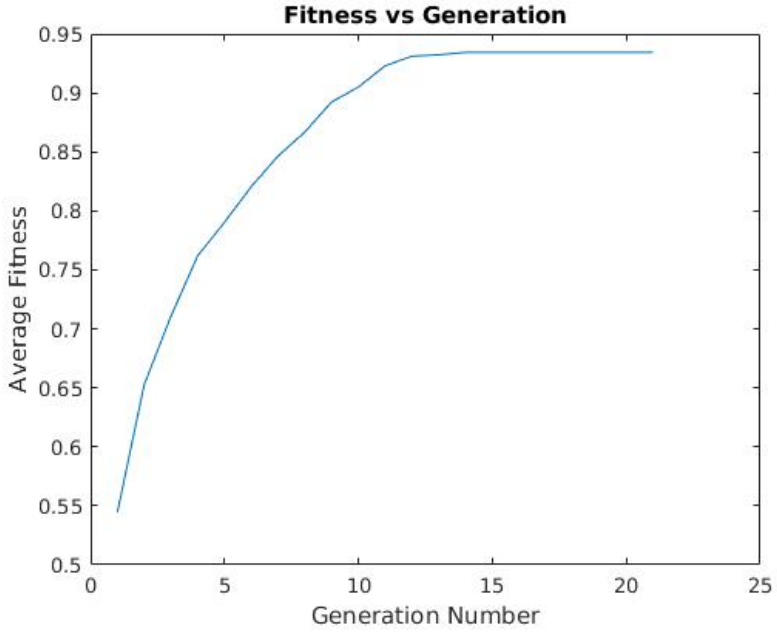

Figure 2

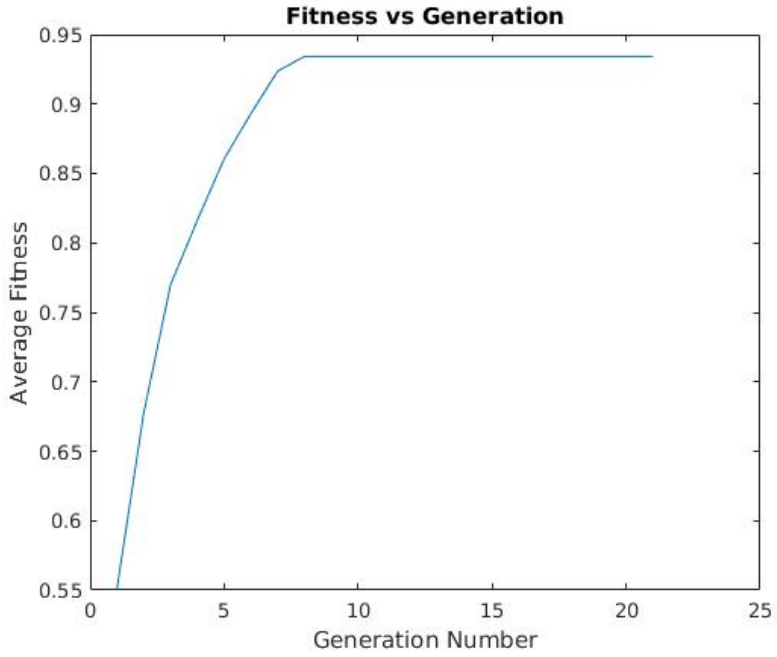

Figure 3

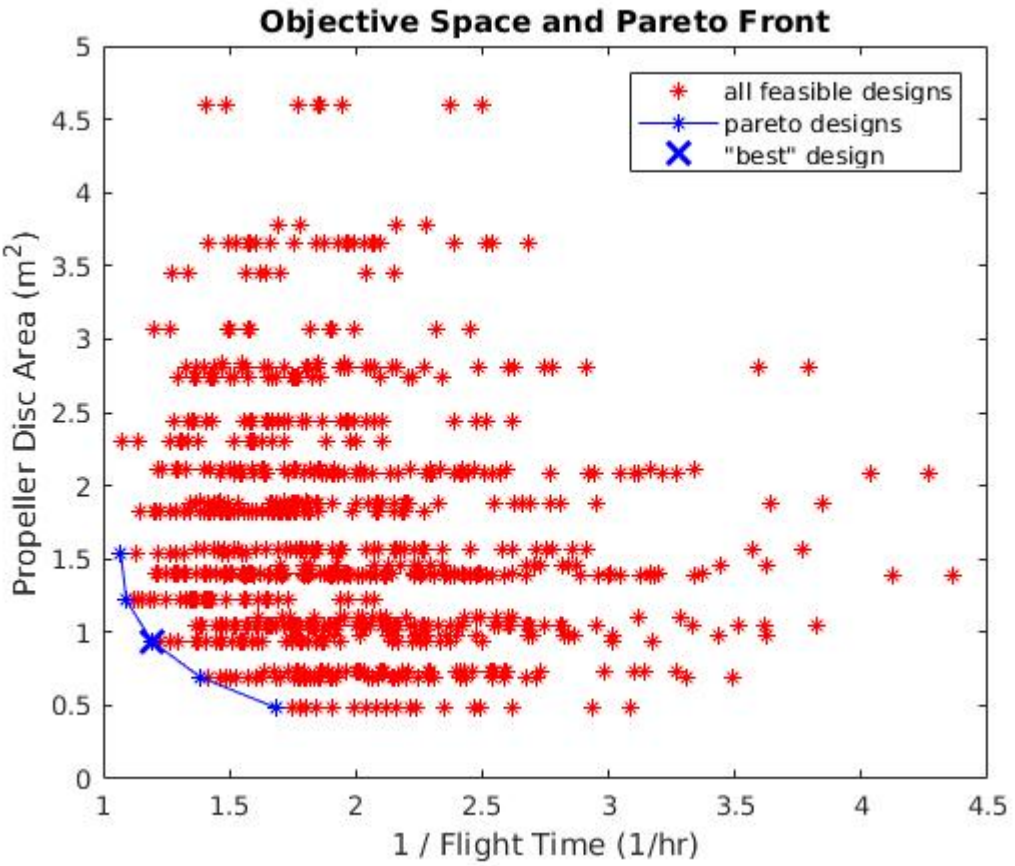

Figure 4

Section IV. Discussion

The results given previously provide good evidence that the developed algorithm is indeed an effective implementation of the genetic optimization approach. Although the design space for this particular problem is relatively small (~3,000) possible designs, I believe that the algorithm would also perform well in much larger design spaces. The results of the optimization also seem to be very reasonable and support the notion that the developed model accurately represents reality. In general, larger propellers at lower rotational speeds are known to be more efficient than smaller propellers at higher speeds, and the results also support this.

Although the algorithm found the global optimum ~ 95% of the time, further performance increases could likely be achieved by additional fine-tuning of the algorithm parameters. Increasing generation size and number of generations seems to help in general, but doing so also costs additional (and sometimes very significant) computation time.

Section V. Appendix

A listing of all MATLAB scripts and functions is given below in alphabetical order by filename:

Main Optimization Script: *genetic_multi_optimization.m*

```
clc
clear
close all

% Algorithm Tuning Params
N = 50;               % Generation size
n_gen = 20;           % Number of generations
p_crossover = 0.5;    % Probability that crossover occurs
p_mutation = 0.10;    % Probability that mutation occurs
roulette_exponent = 2;  % Fitness pressure (larger exponents give designs
                        % with greatest fitness a greater chance of being
                        % selected as parents)
beta = 1;             % dynamic mutation param

% Other Params
n_genes = 6;        % Number of genes (design variables) per chromosome

% ---------------------
% Chromosome Definition
% ---------------------
% A chromosome for this problem is defined by 6 integer design variables:

% Variable 1: Motor
%     Values: 1, 2, 3, or 4 corresponding to four different brushles DC
%             motors available from a manufacturer
%
% Variable 2: Propeller Blade
%     Values: 1, 2, 3, 4, 5, or 6 corresponding to six available propeller
%             blade profiles
%
% Variable 3: Number of Propeller Blades
%     Values: 1, or 2 corresponding to 2 or 3 propeller blades per motor
%
% Variable 4: Number of LiPo Battery Cells
%      Values: 1, 2, 3, 4, or 5 corresponding to 4, 6, 8, 10, or 12 lithium
%              polymer battery cells (each nominally 3.7 volts)
%
% Variable 5: Number of Motors
%     Values: 1, 2, or 3 corresponding to 4 (quadcopter), 6 (hexacopter),
%             or 8 (octocopter) motors and arms
%
% Variable 6: Battery Capacity
%     Values: 1, 2, 3, or 4 corresponding to 23, 22, 17, and 16 amp hour
%             (Ah) capacities available from a battery manufacurer
%
% -------------------------
% End Chromosome Definition
% -------------------------

% Memory Allocation
generation = zeros(N, n_genes);
```

```
avg_fitness = [];

% Randomly select the first generation
feasible = zeros(N,1);
for i = 1:N
    while ~feasible(i)
        for j = 1:n_genes
            switch j
                case 1
                    generation(i,j) = randi(4);
                case 2
                    generation(i,j) = randi(6);
                case 3
                    generation(i,j) = randi(2);
                case 4
                    generation(i,j) = randi(5);
                case 5
                    generation(i,j) = randi(3);
                case 6
                    generation(i,j) = randi(4);
                otherwise
                    disp("Error. Number of genes does not match number of cases.")
                    return
            end
        end

        % Check to see if the current chromosome is feasible
        check_feasible = get_amps(1, generation(i,:));

        if check_feasible == -1
            feasible(i) = 0;
            % disp("bad design")
        else
            feasible(i) = 1;
        end

    end
end

first_generation = generation;
% disp(first_generation)
avg_fit = compute_average_fitness(first_generation);
avg_fitness = [avg_fitness; avg_fit];


for i=1:n_gen

    % Allocate memory for the new generation
    new_generation = zeros(N, n_genes);

    spots_remaining = N;
    count = 0;

    % while there's room to add at least two more children to the new
    % generation...
    while spots_remaining >= 2

        count = count + 1;
```

```
        % SELECTION STEP:
        % Pick two designs from the current generation to become parents
        parents = roulette_selection(generation, roulette_exponent);


        % CROSSOVER STEP:
        children = blend_crossover(parents, p_crossover);
        %children = point_crossover(parents, p_crossover);


        % MUTATION STEP:
        for k=1:2
            children(k,:) = dynamic_mutation(children(k,:), p_mutation, i, n_gen, beta);
        end

        % add the children to the new generation
        new_generation(2*count-1,:) = children(1,:);
        new_generation(2*count,:) = children(2,:);

        spots_remaining = spots_remaining - 2;

    end

    % The new generation is full, now we perform elitism to ensure that the
    % new generation has superior fitness

    % ELITISM STEP:
    % Take the most fit chromosomes from the old generation and the new
    % generation to represent the next generation
    generation = elitism(generation, new_generation);
    avg_fit = compute_average_fitness(generation)
    avg_fitness = [avg_fitness; avg_fit];
end


% Get the best design from the final generation and this is your optimal
% design produced by the genetic algorithm:

optimal_design = get_best_of_generation(generation);
opt_fit = compute_fitness(optimal_design);


% This is the design that most-closely represents LLNL's current copter
% Flight time is ~ 25 minutes
starting_design = [4, 5, 1, 2, 3, 1];
fit0 = compute_fitness(starting_design);

diff = opt_fit - fit0;
percent_improvement = (diff/fit0) * 100;

disp('-------------------------------------------------->')
disp("Genetic Algorithm Finished.")
optimal_design

minutes = opt_fit * 60;
disp("Total flight time (minutes): " + num2str(minutes))
disp(newline)
```

```
disp("Objective increase from initial design: " + num2str(percent_improvement) + "%")
disp(newline)
parse_and_display_design(optimal_design);

% plots
figure(1), clf
generations = 1:n_gen + 1;
plot(generations, avg_fitness)
xlabel('Generation Number')
ylabel('Average Fitness')
title('Fitness vs Generation')
```

***blend_crossover.m*:**

```
function children = blend_crossover(parents, p_crossover)

[n_parents, n_genes] = size(parents);

% Pick a random value between 0 and 1
val = rand(1);

% Determine whether or not crossover takes place
if val <= p_crossover
    % Perform blend crossover
else
    % No crossover occurs and the children are clones of the parents
    children = parents;
    return
end

% Initialize matrix to hold children
children = zeros(n_parents, n_genes);

for i=1:n_genes

    % Pick a random value between 0 and 1
    r = rand(1);

    % Perform the blend crossover
    children(1,i) = r*parents(1,i) + (1-r)*parents(2,i);
    children(2,i) = (1-r)*parents(1,i) + r*parents(2,i);

    % Since each gene of the children chromosomes (designs) must be an
    % integer value in order to be valid, we round to the closest integer
    children(1,i) = round(children(1,i));
    children(2,i) = round(children(2,i));

    % And since each gene has a valid range of values, we saturate the
    % values to their max or min
    switch i
        case 1
            children(1,i) = saturate(children(1,i), 4, 1);
            children(2,i) = saturate(children(2,i), 4, 1);
        case 2
            children(1,i) = saturate(children(1,i), 6, 1);
            children(2,i) = saturate(children(2,i), 6, 1);
```

```
        case 3
            children(1,i) = saturate(children(1,i), 2, 1);
            children(2,i) = saturate(children(2,i), 2, 1);
        case 4
            children(1,i) = saturate(children(1,i), 5, 1);
            children(2,i) = saturate(children(2,i), 5, 1);
        case 5
            children(1,i) = saturate(children(1,i), 3, 1);
            children(2,i) = saturate(children(2,i), 3, 1);
        case 6
            children(1,i) = saturate(children(1,i), 4, 1);
            children(2,i) = saturate(children(2,i), 4, 1);
        otherwise
            disp("Error. Number of genes does not match number of cases.")
            return
    end

end

end

function rval = saturate(val, max, min)

if val > max
    rval = max;
elseif val < min
    rval = min;
else
    rval = val;
end


end
```

***compute_average_fitness.m*:**

```
function avg_fitness = compute_average_fitness(generation)

[rows, ~] = size(generation);

fit_vals = zeros(rows, 1);

for i=1:rows
    fit_vals(i) = compute_fitness(generation(i,:));
end

avg_fitness = sum(fit_vals)/rows;
end
```

compute_fitness.m:

```
function fitness = compute_fitness(design)

% Fitness will be predicted flight time
% Higher values are better

% Flight time will be computed as capacity of the battery in amp hours
% (Ah) divided by current (Amps) required to maintain hover. (h = Ah /
% Amps).

% INPUTS:
% design = [motor_code, prop_blade_code, n_blades_code, n_cells_code,
%           n_motors_code, bat_cap_code]

% Constants accross all designs:
mass_body = 5;  % mass of the central hub of the multirotor (kg)
delta = 0.05;  % clearance between adjacent propellers (meters)
mass_arm = 0.5;  % mass per meter of material used to construct arms (kg/m)

% first check to see if we even have motor data for the given design
check = get_amps(5, design);

if check == -1
    fitness = 0;
    return
end

% Get the mass of the motor for the given design
motor_code = design(1);

switch motor_code
    % mass_motor (kg)

    case 1
        mass_motor = 0.445;

    case 2
        mass_motor = 0.415;

    case 3
        mass_motor = 0.360;

    case 4
        mass_motor = 0.360;

    otherwise
        fitness = 0;
        return
end

% Get the propeller diameter for the given design
prop_blade_code = design(2);

switch prop_blade_code
    % D = propeller diameter (meters)
```

```
    case 1
        D = 30.5 * 0.0254;  % convert inches to meters

    case 2
        D = 27.5 * 0.0254;

    case 3
        D = 24.5 * 0.0254;

    case 4
        D = 21.5 * 0.0254;

    case 5
        D = 18.5 * 0.0254;

    case 6
        D = 15.5 * 0.0254;

    otherwise
        fitness = 0;
        return
end

% Get the number of battery cells for the given design
n_cells_code = design(4);

switch n_cells_code
    % (int) n_cells = number of LiPo cells

    case 1
        n_cells = 4;

    case 2
        n_cells = 6;

    case 3
        n_cells = 8;

    case 4
        n_cells = 10;

    case 5
        n_cells = 12;

    otherwise
        fitness = 0;
        return
end

% Get the number of motors for the given design
n_motors_code = design(5);

switch n_motors_code
    % (int) n = number of motors

    case 1
        n = 4;
```

```
    case 2
        n = 6;

    case 3
        n = 8;

    otherwise
        fitness = 0;
        return
end

% Get the battery capacity for the given design
bat_cap_code = design(6);

switch bat_cap_code

    % bat_cap = battery capacity in amp hours (Ah)
    case 1
        bat_cap = 23;

    case 2
        bat_cap = 22;

    case 3
        bat_cap = 17;

    case 4
        bat_cap = 16;

    otherwise
        fitness = 0;
        return
end

if bat_cap == 23
    cell_mass = 0.413;
elseif bat_cap == 22
    cell_mass = 0.422;
elseif bat_cap == 17
    cell_mass = 0.321;
elseif bat_cap == 16
    cell_mass = 0.310;
else
    fitness = 0;
    return
end

% compute battery mass
mass_battery = cell_mass * n_cells;

% compute total mass of multirotor design
mass_total = compute_mass(n, D, delta, mass_body, mass_motor, mass_battery, mass_arm);

% compute hover thrust (equal to total_weight)
thrust_total = mass_total * 9.81;

% compute required thrust per motor
thrust_motor = thrust_total / n;
```

```
% compute current draw (Amps) per motor
amps_motor = get_amps(thrust_motor, design);

% compute total current draw
amps = amps_motor * n;


% finally, compute fitness or total flight time (hours)
flight_time = bat_cap / amps;

fitness = flight_time;

end
```

***compute_fitness2.m*:**

```
function fitness = compute_fitness2(design)

% Fitness will be rotor disc area
% Lower values are better for wind resistance

% Rotor disc area will be computed as A = n * (pi * (D/2)^2).

% INPUTS:
% design = [motor_code, prop_blade_code, n_blades_code, n_cells_code,
%           n_motors_code, bat_cap_code]


% first check to see if we even have motor data for the given design
check = get_amps(5, design);

if check == -1
   fitness = 999;
   return
end

% Get the propeller diameter for the given design
prop_blade_code = design(2);

switch prop_blade_code
   % D = propeller diameter (meters)

   case 1
      D = 30.5 * 0.0254;  % convert inches to meters

   case 2
      D = 27.5 * 0.0254;

   case 3
      D = 24.5 * 0.0254;

   case 4
      D = 21.5 * 0.0254;

   case 5
```

```
        D = 18.5 * 0.0254;

    case 6
        D = 15.5 * 0.0254;

    otherwise
        fitness = 999;
        return
end

% Get the number of propeller blades
n_blade_code = design(3);

switch n_blade_code

    case 1
        n_blades = 2;

    case 2
        n_blades = 3;

    otherwise
        fitness = 999;
        return
end

% Get the number of motors for the given design
n_motors_code = design(5);

switch n_motors_code
    % (int) n = number of motors

    case 1
        n = 4;

    case 2
        n = 6;

    case 3
        n = 8;

    otherwise
        fitness = 999;
        return
end


% compute the rotor disc area
A = n * (pi * (D/2)^2);

if n_blades == 3
    A = A * 1.5;
end



% fitness
fitness = A;
```

```
end
```

***compute_mass.m*:**

```
function m_total = compute_mass(n, D, delta, mass_body, mass_motor, mass_battery, mass_arm)

% inputs
% n = number of motors/arms
% D = propeller Diameter (m)
% delta = desired clearance between adjacent spinning props (m)
% mass_body = mass of multirotor hub (center frame, autopilot, etc)(kg)
% mass_motor = mass of each individual motor (kg)
% mass_battery = battery mass (kg)
% mass_arm = mass per meter of the tube material used to make the arms (kg/m)

% Angle between arms
theta = 2*pi/n;

% Compute the required arm length
l_arm = (D + delta) / (2*sin(theta/2));

% Compute total mass
m_total = n*(l_arm * mass_arm) + mass_body + n*mass_motor + mass_battery;

end
```

***dynamic_mutation.m*:**

```
function new_chromosome = dynamic_mutation(chromosome, p_mutation, j, M, beta)

num_genes = length(chromosome);

% Compute alpha (eq. 6.9)
alpha = (1-((j-1)/M))^beta;

% minimum value for all genes is 1
x_min = 1;

for i=1:num_genes

    % pick a random number to decide if mutation occurs
    val = rand(1);

    if val <= p_mutation
        % mutate the current gene
    else
        % skip to the next gene without mutating
        continue
    end

    switch i
        case 1
            x_max = 4;
```

```
        case 2
            x_max = 6;
        case 3
            x_max = 2;
        case 4
            x_max = 5;
        case 5
            x_max = 3;
        case 6
            x_max = 4;
        otherwise
            disp("Error. Number of genes does not match number of cases.")
    end

    % pick a random number between the min and max allowed values for the
    % current gene
    r = random_number_between_two_values(x_min, x_max);

    % get the gene's current value
    x = chromosome(1, i);

    % eq 6.8
    if r <= x
        chromosome(1,i) = x_min + ((r - x_min)^alpha)*((x - x_min)^(1-alpha));
    else
        chromosome(1,i) = x_max - ((x_max - r)^alpha)*((x_max - x)^(1-alpha));
    end

    % round to the nearest integer value
    chromosome(1,i) = round(chromosome(1,i));

    % saturate just in case
    chromosome(1,i) = saturate(chromosome(1,i), x_max, x_min);
end

new_chromosome = chromosome;

end


function r = random_number_between_two_values(x_min, x_max)

r = x_min + (x_max - x_min) .* rand(1);

end

function rval = saturate(val, max, min)

if val > max
    rval = max;
elseif val < min
    rval = min;
else
    rval = val;
end

end
```

***elitism.m*:**

```
function next_generation = elitism(generation, new_generation)

[rows, cols] = size(generation);

% memory allocation
next_generation = zeros(rows, cols);

% the composite generation is the old and new_generation stacked on top of
% each other
comp_gen = vertcat(generation, new_generation);

% vector to hold the fitnesses of each chromosome
comp_fitness = zeros(2*rows, 1);

for i=1:2*rows
    % evaluate the fitness of each chromosome (design) and store it in our
    % vector
    comp_fitness(i) = compute_fitness(comp_gen(i,:));
end

% sort the comp_fitness vector in descending order (greatest ---> smallest)
% and get the indexes
[~, index] = sort(comp_fitness, 'descend');

% add the best chromosomes (designs) to the next generation
for i=1:rows

    % value of mIndex is the index from comp_gen corresponding to one of
    % the top 10 best (most fit) desings
    mIndex = index(i);
    % fill out the next_generation by grabbing the mIndex-th row from the
    % composite generation
    next_generation(i,:) = comp_gen(mIndex,:);

end



end
```

***get_amps.m*:**

```
function amps = get_amps(thrust, design)

% INPUTS:
% thrust: Required thrust in Newtons for each motor given the design
% design: Array of the design variables in integer representations

% design = [motor_code, prop_blade_code, n_blades_code, n_cells_code,
%           n_motors_code, bat_cap_code]

% motor codes:
% 1 = KDE7208XF-135
% 2 = KDE6213XF-185
```

```
% 3 = KDE5215XF-220
% 4 = KDE5215XF-330

% prop blade codes:
% 1 = KDE 30.5"
% 2 = KDE 27.5"
% 3 = KDE 24.5"
% 4 = KDE 21.5"
% 5 = KDE 18.5"
% 6 = KDE 15.5"

% n blades codes:
% 1 = dual-blade
% 2 = triple-blade

% n cells codes:
% 1 = 4S
% 2 = 6S
% 3 = 8S
% 4 = 10S
% 5 = 12S

% n motors codes:
% 1 = 4 (quad)
% 2 = 6 (hex)
% 3 = 8 (octo)

% bat_cap_codes
% 1 = 23,000 mAh
% 2 = 22,000 mAh
% 3 = 17,000 mAh
% 4 = 16,000 mAh

% get the first 4 elements of design that represent the power system
design = design(1:4);

% convert to a number code
str = sprintf('%d%d%d%d', design(1), design(2), design(3), design(4));
design_code = str2double(str);


switch design_code

    % Test data from KDE Direct
    % Thr = [...] are KDE's Thrust measurements in Newtons
    % Amp = [...] are KDE's current measurements in Amps

    % Cases for motor 1:
    case 1312
        Thr = [2.75, 5.69, 9.90, 14.81, 19.81, 25.6, 33.73];
        Amp = [0.5, 1.2, 2.4, 4.2, 6.4, 9.4, 13.1];

    case 1322
        Thr = [3.43, 6.96, 11.96, 17.75, 23.73, 30.11, 38.44];
        Amp = [0.6, 1.6, 3.2, 5.6, 8.7, 12.8, 17.5];

    case 1212
        Thr = [4.02, 8.63, 14.42, 20.99, 28.54, 36.48, 45.70];
```

```
    Amp = [0.6, 1.7, 3.7, 6.4, 10.0, 14.4, 19.2];

case 1222
    Thr = [4.81, 10.20, 16.97, 24.81, 32.56, 40.70, 49.43];
    Amp = [0.8, 2.3, 4.7, 8.5, 12.8, 18.1, 24.2];

case 1112
    Thr = [5.69, 11.38, 19.32, 26.67, 35.89, 44.42, 56.00];
    Amp = [0.8, 2.3, 5.2, 8.4, 13.6, 19.3, 26.8];

case 1313
    Thr = [4.81, 9.32, 16.28, 24.61, 33.15, 41.87, 53.25];
    Amp = [0.7, 1.9, 3.6, 6.8, 10.3, 14.8, 20.3];

case 1323
    Thr = [5.98, 11.57, 20.20, 28.34, 37.46, 46.68, 58.84];
    Amp = [0.8, 2.3, 5.2, 8.7, 13.9, 19.9, 27.0];

case 1213
    Thr = [6.86, 14.71, 23.54, 34.52, 45.31, 54.92, 63.37];
    Amp = [1.1, 2.9, 5.7, 10.1, 15.6, 22.4, 29.3];

case 1223
    Thr = [8.04, 17.16, 27.56, 38.64, 49.43, 59.43, 73.06];
    Amp = [1.3, 3.6, 7.5, 13.2, 19.5, 27.6, 35.8];

case 1314
    Thr = [7.45, 14.71, 25.11, 36.38, 45.40, 59.33, 74.04];
    Amp = [1.1, 2.6, 5.6, 9.6, 13.1, 20.8, 29.1];

case 1324
    Thr = [8.92, 17.55, 29.22, 41.48, 51.98, 64.23, 76.88];
    Amp = [1.3, 3.4, 7.6, 13.1, 19.8, 26.6, 40.6];

case 1214
    Thr = [10.40, 21.18, 33.73, 47.37, 59.43, 69.04, 80.22];
    Amp = [1.5, 4.0, 8.1, 14.5, 22.3, 32.9, 42.1];

case 1315
    Thr = [10.79, 19.61, 32.95, 45.31, 62.86, 78.85, 88.75];
    Amp = [1.5, 3.3, 6.9, 10.9, 19.4, 29.6, 40.1];

% Cases for motor 2:
case 2512
    Thr = [2.16, 4.22, 6.86, 10.20, 13.73, 17.65, 23.05];
    Amp = [0.6, 1.2, 2.1, 3.4, 5.1, 7.3, 9.9];

case 2522
    Thr = [2.55, 5.20, 8.43, 12.55, 16.87, 21.67, 28.24];
    Amp = [0.7, 1.5, 2.6, 4.4, 6.7, 9.7, 13.2];

case 2412
    Thr = [3.92, 7.45, 11.96, 17.36, 23.34, 30.50, 38.74];
    Amp = [0.9, 1.9, 3.6, 6.1, 9.5, 13.9, 18.2];

case 2422
    Thr = [4.61, 8.83, 14.42, 20.79, 27.46, 35.89, 45.90];
    Amp = [1.0, 2.2, 4.5, 7.9, 11.8, 16.8, 23.5];
```

```
case 2312
   Thr = [5.69, 10.30, 16.97, 25.01, 32.75, 41.78, 52.96];
   Amp = [1.2, 2.6, 5.2, 9.1, 13.4, 19.4, 25.9];

case 2322
   Thr = [6.77, 12.36, 20.20, 28.44, 37.36, 45.99, 57.47];
   Amp = [1.5, 3.3, 6.9, 11.4, 17.5, 24.7, 33.2];

case 2513
   Thr = [4.12, 7.26, 11.57, 16.48, 22.36, 30.11, 38.34];
   Amp = [0.9, 1.8, 3.1, 5.1, 7.9, 11.7, 15.5];

case 2523
   Thr = [4.61, 8.34, 13.73, 19.91, 26.97, 35.99, 46.09];
   Amp = [1.1, 2.1, 4.0, 6.5, 9.9, 14.7, 20.1];

case 2413
   Thr = [6.28, 11.87, 19.22, 28.24, 37.66, 47.37, 61.19];
   Amp = [1.2, 2.8, 5.6, 9.5, 14.2, 20.1, 28.3];

case 2423
   Thr = [7.55, 14.42, 22.65, 33.05, 43.15, 54.33, 70.51];
   Amp = [1.5, 3.5, 6.8, 11.8, 17.1, 24.2, 35.2];

case 2313
   Thr = [9.02, 16.77, 26.67, 39.32, 50.50, 64.63, 79.73];
   Amp = [1.8, 3.9, 7.7, 13.7, 19.7, 29.2, 38.8];

case 2323
   Thr = [10.89, 20.50, 32.17, 43.93, 56.68, 70.71, 85.71];
   Amp = [2.2, 5.4, 10.6, 17.0, 26.6, 39.0, 51.1];

case 2514
   Thr = [6.28, 10.49, 16.57, 23.54, 34.23, 43.44, 55.02];
   Amp = [1.2, 2.3, 4.3, 6.8, 11.2, 16.0, 21.4];

case 2524
   Thr = [7.06, 12.45, 19.91, 28.15, 40.99, 52.27, 65.41];
   Amp = [1.4, 2.9, 5.4, 8.6, 14.7, 21.7, 28.0];

case 2414
   Thr = [9.12, 17.36, 28.44, 40.01, 54.03, 67.47, 85.81];
   Amp = [1.6, 3.8, 7.6, 12.5, 19.4, 28.3, 39.2];

case 2424
   Thr = [10.89, 21.08, 33.54, 45.01, 60.80, 76.10, 96.50];
   Amp = [2.0, 5.0, 9.9, 15.1, 22.9, 36.2, 50.8];

case 2314
   Thr = [13.04, 25.11, 40.21, 53.45, 70.51, 89.34, 106.99];
   Amp = [2.3, 5.7, 11.3, 17.3, 27.0, 43.3, 57.1];

case 2324
   Thr = [15.69, 28.44, 43.54, 57.96, 74.53, 95.12, 113.86];
   Amp = [3.0, 7.1, 13.8, 21.5, 34.3, 50.0, 69.9];

case 2515
```

```
    Thr = [8.43, 14.22, 22.06, 34.03, 45.21, 58.74, 74.53];
    Amp = [1.6, 3.0, 5.3, 9.5, 14.4, 20.6, 28.3];

case 2525
    Thr = [9.81, 17.06, 26.87, 41.78, 54.52, 71.10, 86.00];
    Amp = [1.8, 3.7, 6.9, 12.8, 18.9, 28.5, 36.6];

case 2415
    Thr = [12.75, 23.93, 37.27, 52.56, 71.29, 91.40, 106.89];
    Amp = [2.1, 5.1, 9.6, 16.0, 25.8, 40.2, 51.4];

case 2425
    Thr = [14.91, 27.36, 42.07, 58.94, 79.63, 100.03, 116.99];
    Amp = [2.6, 6.1, 11.4, 19.1, 33.9, 45.1, 59.1];

case 2315
    Thr = [17.95, 33.24, 49.92, 69.82, 90.61, 107.28, 126.31];
    Amp = [3.1, 6.9, 13.0, 22.2, 37.8, 52.7, 76.2];

% Cases for motor 3:
case 3512
    Thr = [2.45, 4.41, 6.96, 10.30, 14.02, 18.83, 23.73];
    Amp = [0.7, 1.3, 2.2, 3.6, 5.4, 8.0, 10.4];

case 3522
    Thr = [2.75, 5.49, 8.63, 12.75, 17.26, 22.85, 28.73];
    Amp = [0.7, 1.6, 2.8, 4.6, 6.9, 10.0, 13.7];

case 3412
    Thr = [4.12, 7.55, 12.26, 18.04, 23.83, 30.40, 39.23];
    Amp = [0.9, 2.0, 3.8, 6.5, 9.7, 13.9, 18.7];

case 3422
    Thr = [4.81, 9.12, 14.91, 21.57, 28.54, 36.09, 45.60];
    Amp = [1.1, 2.6, 4.9, 8.3, 12.6, 18.0, 23.8];

case 3613
    Thr = [2.26, 4.41, 7.06, 10.10, 13.83, 18.53, 23.83];
    Amp = [0.6, 1.3, 2.1, 3.3, 4.8, 7.1, 9.6];

case 3623
    Thr = [2.75, 5.39, 8.63, 12.36, 16.57, 22.26, 28.54];
    Amp = [0.8, 1.6, 2.7, 4.1, 6.1, 9.1, 12.3];

case 3513
    Thr = [4.41, 7.55, 11.87, 17.06, 22.65, 30.20, 39.13];
    Amp = [1.0, 1.9, 3.3, 5.5, 8.1, 12.0, 16.2];

case 3523
    Thr = [5.20, 8.92, 14.51, 20.69, 27.26, 36.19, 46.09];
    Amp = [1.2, 2.2, 4.3, 6.9, 10.4, 15.5, 20.6];

case 3413
    Thr = [6.67, 12.26, 20.10, 28.05, 37.85, 47.46, 61.49];
    Amp = [1.4, 3.0, 5.8, 9.4, 14.7, 20.3, 28.4];

case 3423
    Thr = [7.94, 15.00, 24.61, 34.13, 44.52, 54.72, 68.94];
```

```
    Amp = [1.7, 3.9, 7.3, 12.5, 18.5, 25.8, 35.0];

case 3614
    Thr = [4.12, 6.67, 10.59, 15.69, 20.69, 26.87, 35.11];
    Amp = [1.0, 1.7, 2.9, 4.6, 6.8, 9.9, 13.4];

case 3624
    Thr = [4.71, 8.14, 12.85, 18.93, 25.01, 32.46, 41.68];
    Amp = [1.1, 2.0, 3.6, 5.6, 8.7, 13.0, 17.0];

case 3514
    Thr = [6.57, 11.18, 17.55, 24.91, 34.72, 43.93, 56.00];
    Amp = [1.3, 2.5, 4.5, 7.3, 11.6, 16.6, 22.2];

case 3524
    Thr = [8.14, 13.53, 20.89, 29.52, 40.89, 51.98, 65.41];
    Amp = [1.6, 3.1, 5.6, 9.4, 15.1, 21.8, 28.2];

case 3414
    Thr = [9.71, 18.14, 28.34, 40.80, 53.35, 67.47, 84.44];
    Amp = [1.8, 4.1, 7.9, 13.0, 20.1, 28.8, 37.9];

case 3424
    Thr = [11.57, 21.87, 34.23, 47.76, 62.37, 78.65, 97.58];
    Amp = [2.1, 5.3, 10.3, 16.0, 24.7, 38.2, 49.7];

case 3615
    Thr = [6.08, 9.51, 14.61, 21.28, 28.24, 36.87, 47.56];
    Amp = [1.3, 2.1, 3.7, 6.1, 9.1, 13.0, 17.4];

case 3625
    Thr = [6.67, 11.08, 17.06, 25.40, 34.42, 44.03, 55.70];
    Amp = [1.5, 2.6, 4.4, 7.6, 11.6, 16.6, 21.9];

case 3515
    Thr = [9.02, 15.20, 23.05, 34.91, 46.48, 59.13, 75.12];
    Amp = [1.7, 3.1, 5.6, 10.0, 15.1, 21.2, 28.7];

case 3525
    Thr = [10.30, 17.75, 27.85, 41.68, 54.82, 69.73, 86.30];
    Amp = [1.9, 3.9, 7.2, 13.0, 19.2, 27.8, 36.7];

case 3415
    Thr = [13.04, 24.91, 37.85, 53.74, 71.10, 90.91, 111.31];
    Amp = [2.3, 5.6, 10.0, 16.8, 26.0, 39.4, 51.3];

% Cases for motor 4:
case 4511
    Thr = [2.45, 5.20, 8.24, 11.96, 16.38, 21.48, 27.95];
    Amp = [1.1, 2.3, 4.1, 6.7, 10.3, 14.7, 20.3];

case 4521
    Thr = [3.33, 6.28, 10.0, 14.42, 19.71, 25.60, 33.05];
    Amp = [1.3, 2.8, 5.2, 8.4, 13.0, 19.1, 26.4];

case 4411
    Thr = [4.51, 8.43, 13.73, 20.01, 26.77, 35.30, 44.03];
    Amp = [1.6, 3.5, 6.9, 11.8, 18.2, 27.7, 36.4];
```

```
    case 4421
        Thr = [5.59, 10.30, 16.48, 23.54, 30.99, 39.42, 49.43];
        Amp = [2.0, 4.5, 9.0, 15.0, 23.2, 33.7, 45.4];

    case 4612
        Thr = [3.33, 6.28, 10.20, 14.81, 20.10, 27.85, 35.50];
        Amp = [1.3, 2.6, 4.6, 7.2, 10.8, 17.1, 22.5];

    case 4622
        Thr = [4.22, 7.75, 12.26, 17.75, 23.93, 32.46, 41.19];
        Amp = [1.6, 3.1, 5.7, 9.3, 14.2, 21.6, 29.0];

    case 4512
        Thr = [5.88, 10.40, 16.67, 24.12, 33.24, 43.74, 55.60];
        Amp = [2.0, 3.9, 7.3, 12.0, 19.0, 28.3, 37.9];

    case 4522
        Thr = [7.06, 12.45, 20.01, 28.93, 39.03, 50.99, 63.74];
        Amp = [2.3, 4.8, 9.3, 15.8, 24.1, 35.3, 48.2];

    case 4412
        Thr = [8.92, 17.16, 26.87, 39.32, 49.92, 63.84, 81.10];
        Amp = [2.8, 6.5, 12.5, 21.9, 32.0, 48.2, 65.5];

    case 4613
        Thr = [6.57, 10.59, 16.97, 25.11, 34.42, 44.33, 57.27];
        Amp = [2.2, 3.8, 6.8, 11.5, 17.9, 25.5, 35.4];

    case 4623
        Thr = [7.94, 12.85, 20.10, 29.32, 40.21, 52.07, 65.80];
        Amp = [2.5, 4.7, 8.4, 13.9, 22.6, 33.0, 44.5];

    case 4513
        Thr = [9.81, 17.75, 27.07, 38.54, 54.03, 71.10, 85.71];
        Amp = [3.0, 6.1, 11.1, 18.2, 30.0, 45.1, 57.3];

    case 4523
        Thr = [11.18, 20.59, 31.58, 44.52, 62.57, 79.34, 96.40];
        Amp = [3.2, 7.2, 13.5, 22.5, 38.9, 52.9, 73.1];

    otherwise
        Thr = [];
        Amp = [];
end

% If we have data for the given design code:
if ~isempty(Thr)

    Thr = Thr';
    Amp = Amp';

    % A matrix for LS polynomial curve fit
    A = [ones(length(Thr),1), Thr, Thr.^2, Thr.^3];

    % solve for coefficients of polynomial using pseudo-inverse
    x = A \ Amp;
```

```
    % using the polynomial curve fit, 'look up' the amps for the given thrust
    amps = x(1) + x(2)*thrust + x(3)*thrust^2 + x(4)*thrust^3;

else
    amps = -1;
end


end
```

***get_best_of_generation.m*:**

```
function best_design = get_best_of_generation(generation)

[rows, ~] = size(generation);


% vector to hold the fitnesses of each chromosome
fitnesses = zeros(rows, 1);

for i=1:rows
    % evaluate the fitness of each chromosome (design) and store it in our
    % vector
    fitnesses(i) = compute_fitness(generation(i,:));
end

% sort the fitnesses vector in descending order (greatest ---> smallest)
% and get the indexes
[~, index] = sort(fitnesses, 'descend');

% the best index is the first one since they're sorted in descending order
bestIndex = index(1);

best_design = generation(bestIndex,:);


end
```

***pareto.m*:**

```
clc
clear
close all

all_designs = [];
f1 = [];
f2 = [];

count = 0;

% Look at all possible designs by brute-force
for i=1:4
    for j=1:6
        for k=1:2
```

```
            for l=1:5
                for m=1:3
                    for n=1:4

                        % increment counter
                        count = count + 1;

                        % fill out the design
                        design = [i, j, k, l, m, n];

                        % check to see if design is feasible
                        check = get_amps(5, design);

                        if check ~= -1
                            % add design to our giant matrix
                            all_designs = [all_designs; design];

                            % add the fitness to our big fitness vector
                            f1 = [f1; 1/(compute_fitness(design))];
                            f2 = [f2; compute_fitness2(design)];
                        end

                    end
                end
            end
        end
    end
end


% compute the maximin_values
maximin_vals = zeros(length(f1), 1);

for i = 1:length(f1)
    mins = [];
    for j = 1:length(f1)
        if i ~= j
            mins = [mins; min(f1(i)-f1(j), f2(i)-f2(j))];
        end
    end
    maximin_vals(i) = max(mins);
end

% make an array of just the pareto designs
pareto_designs = [];
for i = 1:length(maximin_vals)
    if maximin_vals(i) < 0
        pareto_designs = [pareto_designs; all_designs(i,:)];
    end
end

[rows, ~] = size(pareto_designs);

f1_pareto = zeros(rows, 1);
f2_pareto = zeros(rows, 1);

for i = 1:rows
```

```
    f1_pareto(i,1) = 1/compute_fitness(pareto_designs(i,:));
    f2_pareto(i,1) = compute_fitness2(pareto_designs(i,:));

end

plot(f1, f2, 'r*')
xlabel('1 / Flight Time (1/hr)')
ylabel('Propeller Disc Area (m^2)')
hold on
plot(f1_pareto, f2_pareto, 'b-*')
% plot(f1_pareto, f2_pareto, 'b')
title('Objective Space and Pareto Front')

% select pareto_design(3,:) as the 'best design'
pareto_row = 3;

% do some fancy printing stuff
max_flight_time = compute_fitness(pareto_designs(1,:));
pareto3_flight_time = compute_fitness(pareto_designs(pareto_row,:));

diff_ft = max_flight_time - pareto3_flight_time;
percent_ft = (diff_ft/max_flight_time) * 100;
percent_ft_str = num2str(percent_ft);

pareto1_area = compute_fitness2(pareto_designs(1,:));
pareto3_area = compute_fitness2(pareto_designs(pareto_row,:));

diff_area = pareto1_area - pareto3_area;
percent_area = (diff_area/pareto1_area) * 100;
percent_area_str = num2str(percent_area);

s1 = "Results in only a ";
s2 = "% decrease in flight time but yields a ";
s3 = "% decrease in propeller disc area";

s = strcat(s1, percent_ft_str, s2, percent_area_str, s3);

disp('Selecting pareto design: ')
disp(pareto_designs(3,:))
disp(s)
disp(newline)

flight_time = num2str(pareto3_flight_time * 60);
s4 = "Total flight time: ";
s5 = " minutes";
s = strcat(s4, flight_time, s5);
disp(s)
disp(newline)

area = num2str(pareto3_area);
s6 = "Total disc area: ";
s7 = " m^2";
s = strcat(s6, area, s7);
disp(s)


plot(f1_pareto(pareto_row), f2_pareto(pareto_row), 'bx', 'MarkerSize', 12, 'LineWidth', 2)
legend('all feasible designs', 'pareto designs', '"best" design')
```

***point_crossover.m*:**

```
function children = point_crossover(parents, p_crossover)

[n_parents, n_genes] = size(parents);

% Pick a random value between 0 and 1
val = rand(1);

% Determine whether or not crossover takes place
if val <= p_crossover
    % Perform blend crossover
else
    % No crossover occurs and the children are clones of the parents
    children = parents;
    return
end

% Initialize children to be clones of the parents
children = parents;

% Choose the crossover point
point_cross = randi(n_genes);

if point_cross == n_genes
    % No tail to swap so children = parents
    return
else
    % Chop off the tails of the parent chromosomes at the crossover point
    parent_1_tail = parents(1, point_cross + 1:end);
    parent_2_tail = parents(2, point_cross + 1:end);

    % Swap the tails and now we have two children chromosomes
    children(1,point_cross + 1:end) = parent_2_tail;
    children(2,point_cross + 1:end) = parent_1_tail;

end

end
```

***roulette_selection.m*:**

```
function parents = roulette_selection(generation, roulette_exponent)

% Get the number of chromosomes (designs) in the generation
[gen_size, chromosome_size] = size(generation);

% Initialize a column vector to hold generation fitness data
fitnesses = zeros(gen_size, 1);

% Initialize a column vector to hold the widths on the roulette wheel that
% each design occupies
widths = zeros(gen_size, 1);
```

```
% Initialize matrix to hold parent chromosomes
parents = zeros(2, chromosome_size);


% Get the fitness for each chromosome
for i = 1:gen_size
   fitnesses(i) = compute_fitness(generation(i,:));
end

% Normalize all of the widths to sum to 1
for i = 1:gen_size
   widths(i) = fitnesses(i)^(roulette_exponent)/sum(fitnesses.^roulette_exponent);
end

% Here we make a sliding window defined by a left and right bound that
% represents the current segment or bin (whose width is widths(i)) of our
% roulette wheel.  If the random value we picked earlier falls into the
% current segment, then we select the design corresponding to this segment
% to be a parent chromosome. We do this twice to get two parents.

for n=1:2

   % Pick a random value between 0 and 1
   val = rand(1);

   % Initilize the bounds
   left = 0;
   right = 0;

   for i=1:gen_size

      % Define the left side of the current bin of width = widths(i)
      if i ~= 1
         left = left + widths(i-1);
      end

      % Define the right side of the current bin of width = widths(i)
      right = right + widths(i);

      if left <= val && val <= right
         break
      end

   end

   % Store the chosen parent chromosomes
   parents(n,:) = generation(i,:);

end

% fitnesses
% widths
end
```

End Appendix