MULTI-OBJECTIVE OPTIMIZATION USING EVOLUTIONARY ALGORITHMS (MOEA)

ARAVIND SESHADRI

1. Multi-Objective Optimization Using NSGA-II

NSGA (Non-Dominated Sorting in Genetic Algorithms [5]) is a popular non-domination based genetic algorithm for multi-objective optimization. It is a very effective algorithm but has been generally criticized for its computational complexity, lack of elitism and for choosing the optimal parameter value for sharing parameter σ_{share}. A modified version, NSGA-II ([3]) was developed, which has a better sorting algorithm , incorporates elitism and no sharing parameter needs to be chosen *a priori*. NSGA-II is discussed in detail in this report and two sample test functions are optimized using it.

2. General Description of NSGA-II

The population is initialized as usual. Once the population in initialized the population is sorted based on non-domination into each front. The first front being completely non-dominant set in the current population and the second front being dominated by the individuals in the first front only and the front goes so on. Each individual in the each front are assigned rank (fitness) values or based on front in which they belong to. Individuals in first front are given a fitness value of 1 and individuals in second are assigned fitness value as 2 and so on.

In addition to fitness value a new parameter called ***crowding distance*** is calculated for each individual. The crowding distance is a measure of how close an individual is to its neighbors. Large average crowding distance will result in better diversity in the population.

Parents are selected from the population by using binary tournament selection based on the rank and crowding distance. An individual is selected in the rank is lesser than the other or if crowding distance is greater than the other [1]. The selected population generates offsprings from crossover and mutation operators, which will be discussed in detail in a later section.

The population with the current population and current offsprings is sorted again based on non-domination and only the best N individuals are selected, where N is the population size. The selection is based on rank and the on crowding distance on the last front.

3. Detailed Description of NSGA-II

3.1. **Population Initialization.** The population is initialized based on the problem range and constraints if any.

[1]Crowding distance is compared only if the rank for both individuals are same

3.2. **Non-Dominated sort.** The initialized population is sorted based on non-domination [2]. The fast sort algorithm [3] is described as below for each

- for each individual p in main population P do the following
  - Initialize $S_p = \emptyset$. This set would contain all the individuals that is being dominated by p.
  - Initialize $n_p = 0$. This would be the number of individuals that dominate p.
  - for each individual q in P
    - if p dominated q then
      - add q to the set S_p i.e. $S_p = S_p \cup \{q\}$
    - else if q dominates p then
      - increment the domination counter for p i.e. $n_p = n_p + 1$
  - if $n_p = 0$ i.e. no individuals dominate p then p belongs to the first front; Set rank of individual p to one i.e $p_{rank} = 1$. Update the first front set by adding p to front one i.e $F_1 = F_1 \cup \{p\}$
- This is carried out for all the individuals in main population P.
- Initialize the front counter to one. $i = 1$
- following is carried out while the i^{th} front is nonempty i.e. $F_i \neq \emptyset$
  - $Q = \emptyset$. The set for storing the individuals for $(i+1)^{th}$ front.
  - for each individual p in front F_i
    - for each individual q in S_p (S_p is the set of individuals dominated by p)
      - $n_q = n_q - 1$, decrement the domination count for individual q.
      - if $n_q = 0$ then none of the individuals in the subsequent fronts would dominate q. Hence set $q_{rank} = i + 1$. Update the set Q with individual q i.e. $Q = Q \cup q$.
  - Increment the front counter by one.
  - Now the set Q is the next front and hence $F_i = Q$.

This algorithm is better than the original NSGA ([5]) since it utilize the information about the set that an individual dominate (S_p) and number of individuals that dominate the individual (n_p).

3.3. **Crowding Distance.** Once the non-dominated sort is complete the crowding distance is assigned. Since the individuals are selected based on rank and crowding distance all the individuals in the population are assigned a crowding distance value. Crowding distance is assigned front wise and comparing the crowding distance between two individuals in different front is meaning less. The crowing distance is calculated as below

- For each front F_i, n is the number of individuals.
  - initialize the distance to be zero for all the individuals i.e. $F_i(d_j) = 0$, where j corresponds to the j^{th} individual in front F_i.
  - for each objective function m
    - Sort the individuals in front F_i based on objective m i.e. $I = sort(F_i, m)$.

[2]An individual is said to dominate another if the objective functions of it is no worse than the other and at least in one of its objective functions it is better than the other

- $*$ Assign infinite distance to boundary values for each individual in F_i i.e. $I(d_1) = \infty$ and $I(d_n) = \infty$
- $*$ for $k = 2$ to $(n-1)$
  - $\cdot$ $I(d_k) = I(d_k) + \dfrac{I(k+1).m - I(k-1).m}{f_m^{max} - f_m^{min}}$
  - $\cdot$ $I(k).m$ is the value of the m^{th} objective function of the k^{th} individual in I

The basic idea behind the crowing distance is finding the euclidian distance between each individual in a front based on their m objectives in the m dimensional hyper space. The individuals in the boundary are always selected since they have infinite distance assignment.

3.4. **Selection.** Once the individuals are sorted based on non-domination and with crowding distance assigned, the selection is carried out using a ***crowded-comparison-operator*** $(\prec_n)$. The comparison is carried out as below based on

(1) non-domination rank p_{rank} i.e. individuals in front F_i will have their rank as $p_{rank} = i$.
(2) crowding distance $F_i(d_j)$

- $p \prec_n q$ if
  - $p_{rank} < q_{rank}$
  - or if p and q belong to the same front F_i then $F_i(d_p) > F_i(d_q)$ i.e. the crowing distance should be more.

The individuals are selected by using a binary tournament selection with crowed-comparison-operator.

3.5. **Genetic Operators.** Real-coded GA's use ***Simulated Binary Crossover (SBX)*** [2], [1] operator for crossover and ***polynomial mutation*** [2], [4].

3.5.1. *Simulated Binary Crossover.* Simulated binary crossover simulates the binary crossover observed in nature and is give as below.

$$c_{1,k} = \frac{1}{2}[(1-\beta_k)p_{1,k} + (1+\beta_k)p_{2,k}]$$
$$c_{2,k} = \frac{1}{2}[(1+\beta_k)p_{1,k} + (1-\beta_k)p_{2,k}]$$

where $c_{i,k}$ is the i^{th} child with k^{th} component, $p_{i,k}$ is the selected parent and β_k (≥ 0) is a sample from a random number generated having the density

$$p(\beta) = \frac{1}{2}(\eta_c + 1)\beta^{\eta_c}, \quad \text{if } 0 \leq \beta \leq 1$$
$$p(\beta) = \frac{1}{2}(\eta_c + 1)\frac{1}{\beta^{\eta_c + 2}}, \quad \text{if } \beta > 1.$$

This distribution can be obtained from a uniformly sampled random number u between $(0,1)$. η_c is the distribution index for crossover[3]. That is

$$\beta(u) = (2u)^{\frac{1}{(\eta+1)}}$$
$$\beta(u) = \frac{1}{[2(1-u)]^{\frac{1}{(\eta+1)}}}$$

[3]This determine how well spread the children will be from their parents.

Population	Generations	Pool Size	Tour Size	η_c	η_m
200	500	50	2	20	20

TABLE 1. MOP1- Parameters for NSGA-II

3.5.2. *Polynomial Mutation.*

$$c_k = p_k + (p_k^u - p_k^l)\delta_k$$

where c_k is the child and p_k is the parent with p_k^u being the upper bound[4] on the parent component, p_k^l is the lower bound and δ_k is small variation which is calculated from a polynomial distribution by using

$$\begin{aligned}\delta_k =&(2r_k)^{\overline{\frac{1}{\eta_m + 1}}} - 1, \quad \text{if } r_k < 0.5\\ \delta_k =&1 - [2(1 - r_k)]^{\overline{\frac{1}{\eta_m + 1}}} \quad \text{if } r_k \geq 0.5\end{aligned}$$

r_k is an uniformly sampled random number between $(0, 1)$ and η_m is mutation distribution index.

3.6. **Recombination and Selection.** The offspring population is combined with the current generation population and selection is performed to set the individuals of the next generation. Since all the previous and current best individuals are added in the population, elitism is ensured. Population is now sorted based on non-domination. The new generation is filled by each front subsequently until the population size exceeds the current population size. If by adding all the individuals in front F_j the population exceeds N then individuals in front F_j are selected based on their crowding distance in the descending order until the population size is N. And hence the process repeats to generate the subsequent generations.

4. MOP1 (Example Problem - 1)

This problem is to find the global pareto front for a discontinuous function given by objective function as in [3]

$$\begin{aligned}f_1(x) =&1 - e^{-4x_1}\sin^6(6\pi x_1)\\ f_2(x) =&g(x)(1 - (\frac{f_1(x)}{g(x)})^2)\end{aligned}$$

where,

$$g(x) =1 + 9(\sum_{i=1}^{6}\frac{x_i}{4})^{0.25}$$

subject to $0 \leq x_i \leq 1, \;\; i = 1, \ldots, 6$. This set of objective function resulted in the following solution as shown in Figure 1.

[4]The decision space upper bound and lower bound for that particular component.

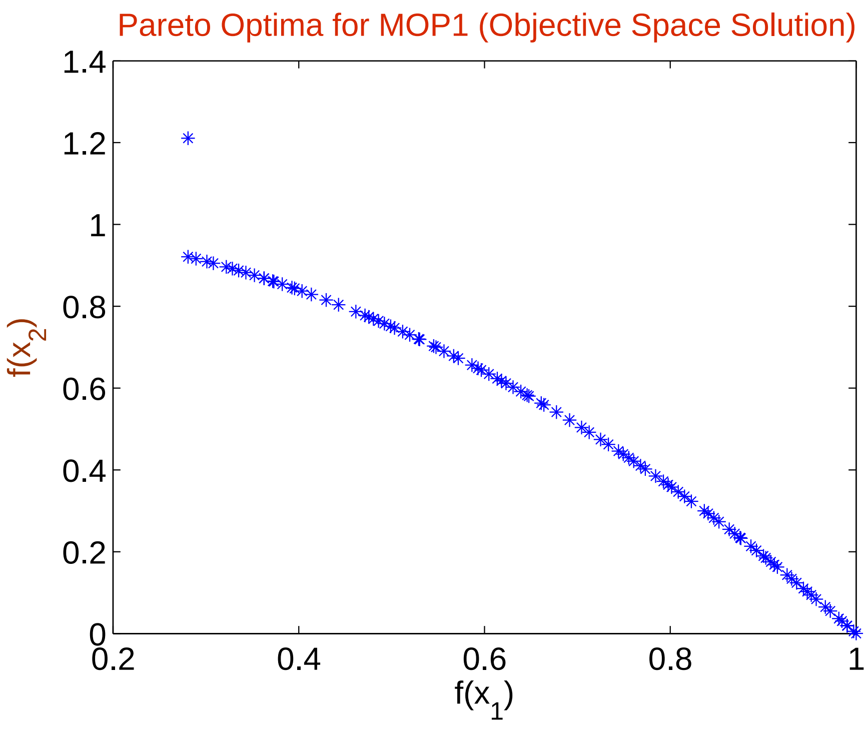

FIGURE 1. MOP1

Population	Generations	Pool Size	Tour Size	η_c	η_m
200	1000	100	2	20	20

TABLE 2. MOP2- Parameters for NSGA-II (Figures 2, 3,4)

5. MOP2 (EXAMPLE PROBLEM - 2)

This problem is to find the global pareto front for the 3 dimensional objective space for the following function.

$$f_1(x) = (1 + g(x)) \cos(0.5\pi x_1) \cos(0.5\pi x_2)$$
$$f_2(x) = (1 + g(x)) \cos(0.5\pi x_1) \sin(0.5\pi x_2)$$

$$f_3(x) = (1 + g(x)) \sin(0.5\pi x_1)$$

$$g_x(x) = \sum_{i=1}^{12} (x_i - 0.5)^2$$

subject to $0 \leq x_i \leq 1, \quad i = 1, \ldots, 12$. The simulation results are shown in Figures 2 to 9.

A modification in the objective function resulted in similar result as shown in [6].

$$g(x) = \sum_{i=3}^{12} (x_i - 0.5)^2$$

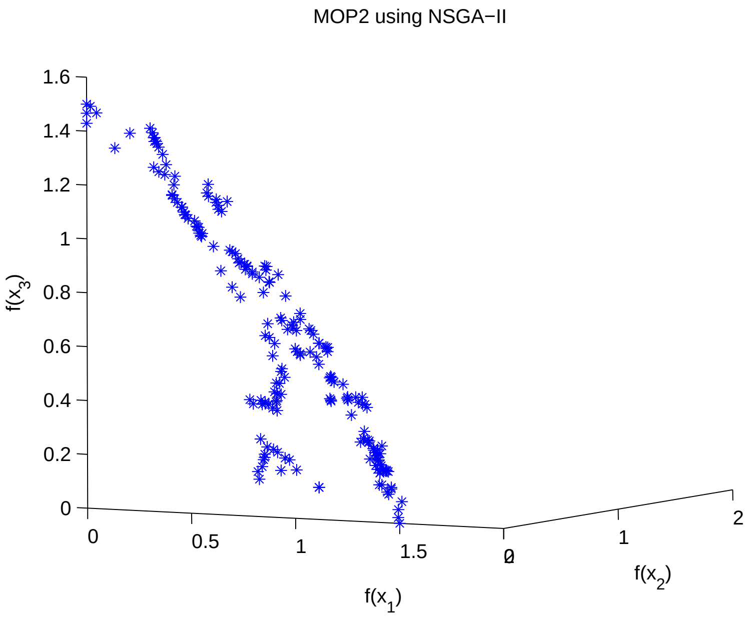

FIGURE 2. MOP2: One of the solutions after 1000 generations (view 1)

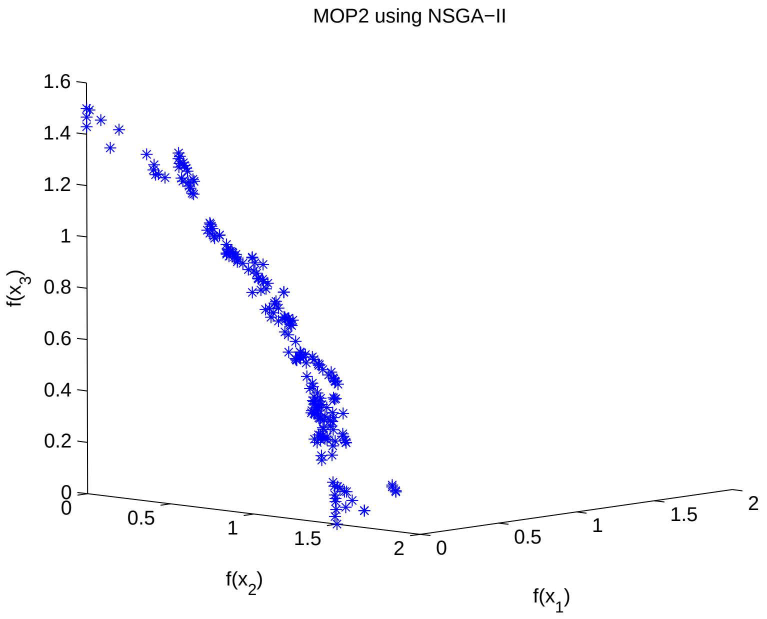

FIGURE 3. MOP2: One of the solutions after 1000 generations (view 2)

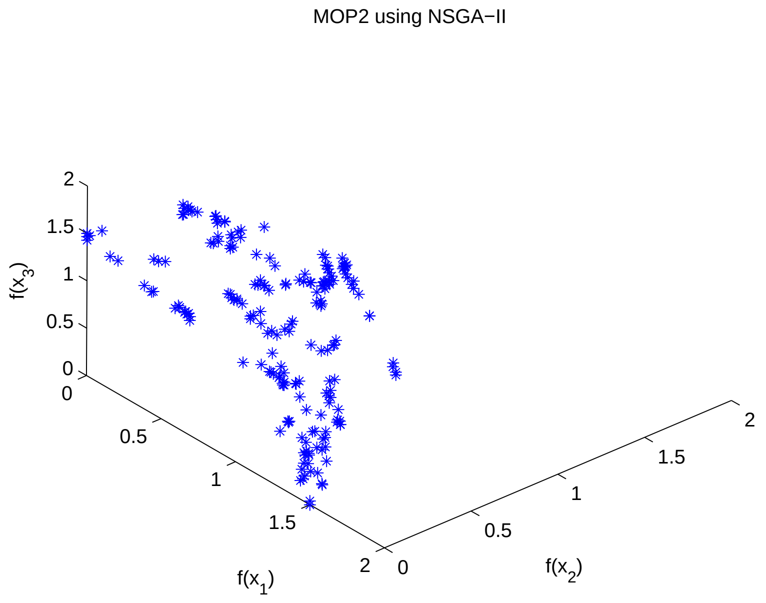

FIGURE 4. MOP2: One of the solutions after 1000 generations (view 3)

Population	Generations	Pool Size	Tour Size	η_c	η_m
200	2000	100	2	20	20

TABLE 3. MOP2- Parameters for NSGA-II (Figures 5 , 6, 7, 8)

Population	Generations	Pool Size	Tour Size	η_c	η_m
200	1000	100	2	20	20

TABLE 4. MOP2- Parameters for NSGA-II (Figures 9)

Population	Generations	Pool Size	Tour Size	η_c	η_m
1000	1000	500	2	20	20

TABLE 5. MOP2- Parameters for NSGA-II Modified (Figures 10,11,12,13

The resulting pareto front is shown in figures 10 to 13.

6. Observations

- Definitely there needs to be a better way of implementing the crowding distance operator since as seen from MOP2 the population is not well distributed.
- The algorithm is fast when compared to NSGA [5].

[5]No simulation was done to verify

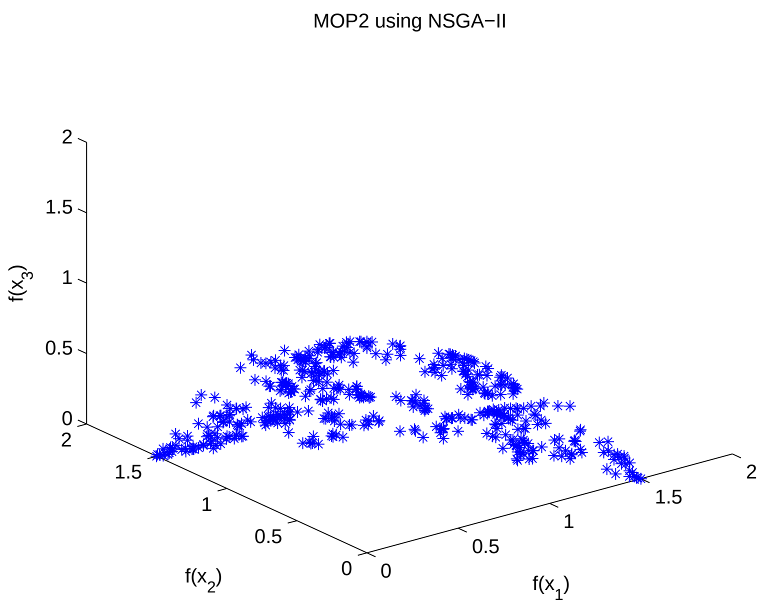

FIGURE 5. MOP2: Second set of the solutions after 2000 generations (view 1)

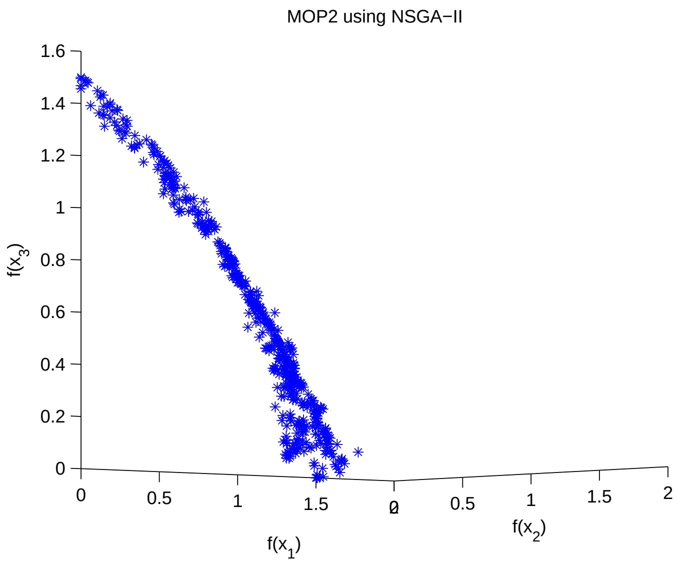

FIGURE 6. MOP2: Second of the solutions after 2000 generations (view 2)

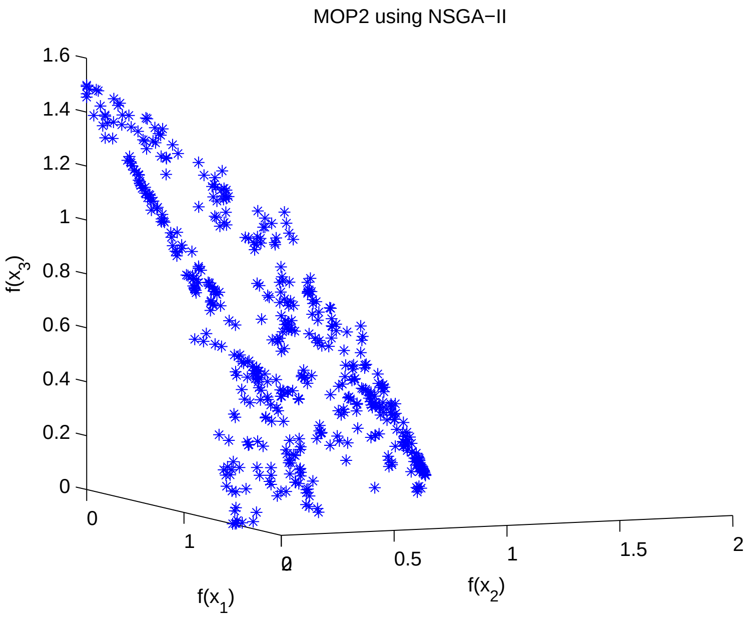

FIGURE 7. MOP2: Second of the solutions after 2000 generations (view 3)

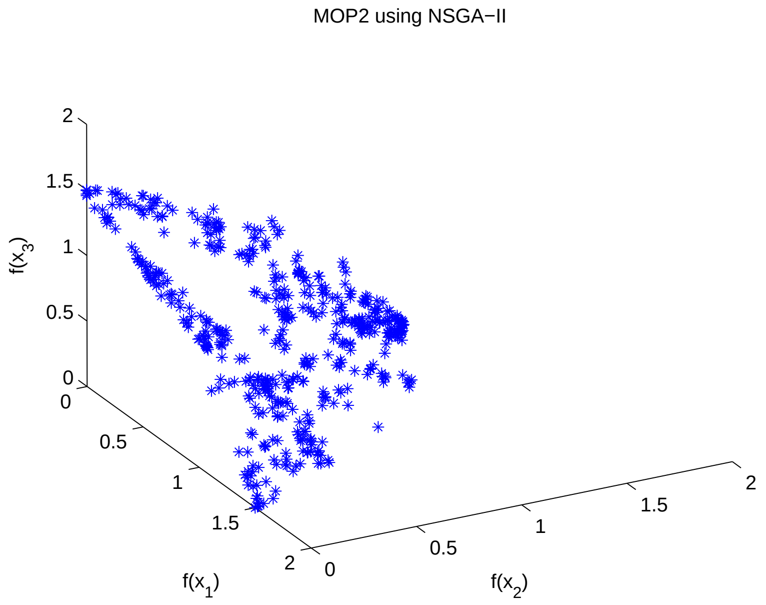

FIGURE 8. MOP2: Second of the solutions after 2000 generations (view 4)

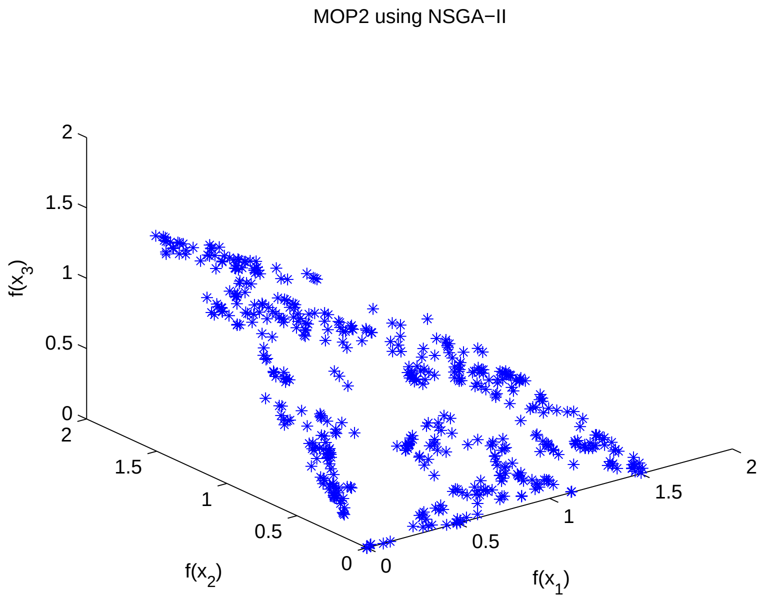

FIGURE 9. MOP2: Another set of the solutions after 1000 generations (view 1))

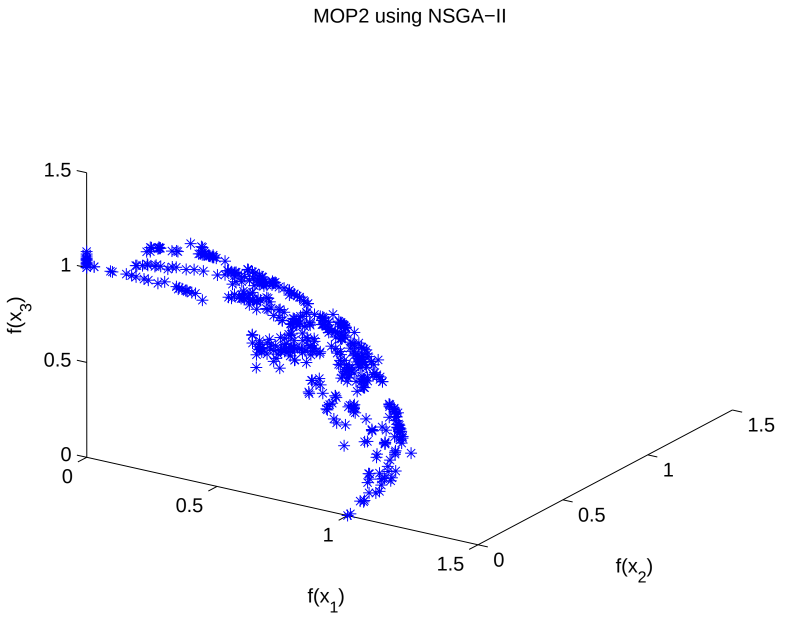

FIGURE 10. MOP2: Modified solutions after 1000 generations (view 1)

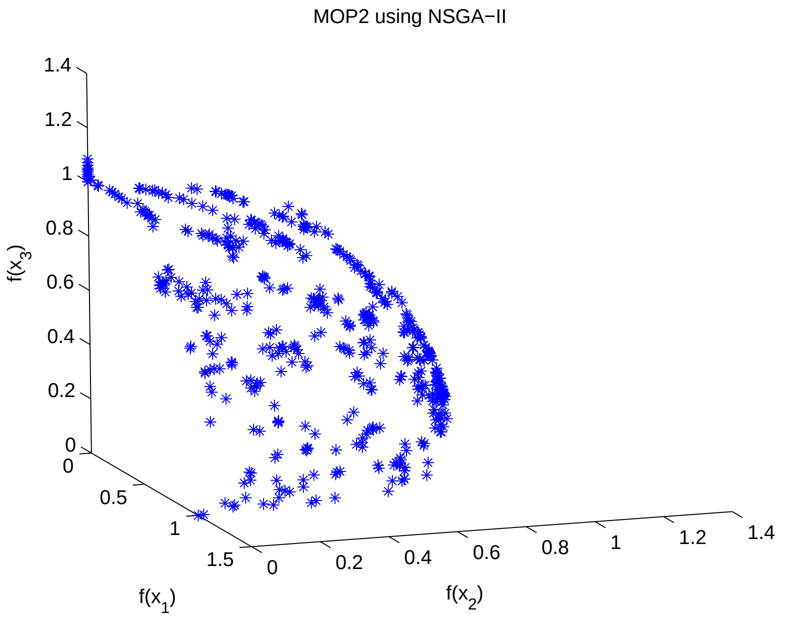

FIGURE 11. MOP2: Modified solutions after 1000 generations (view 2)

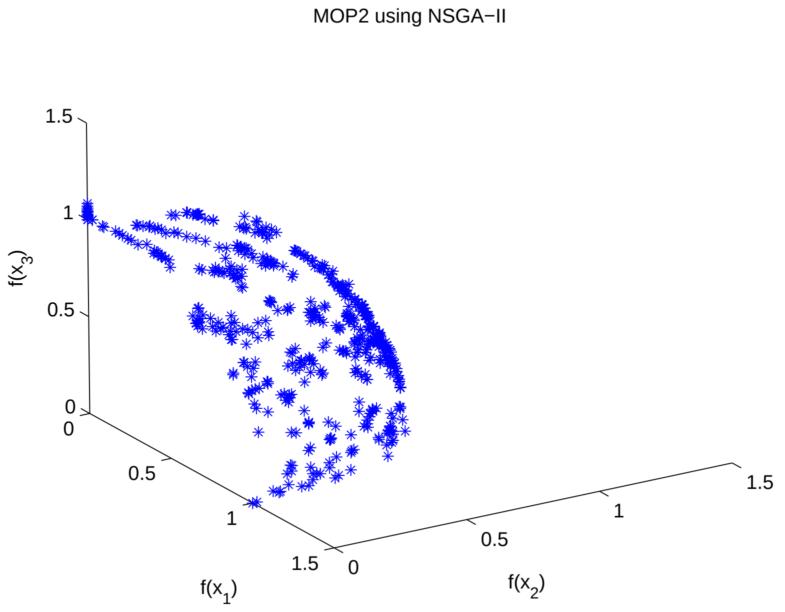

FIGURE 12. MOP2: Modified solutions after 1000 generations (view 3)

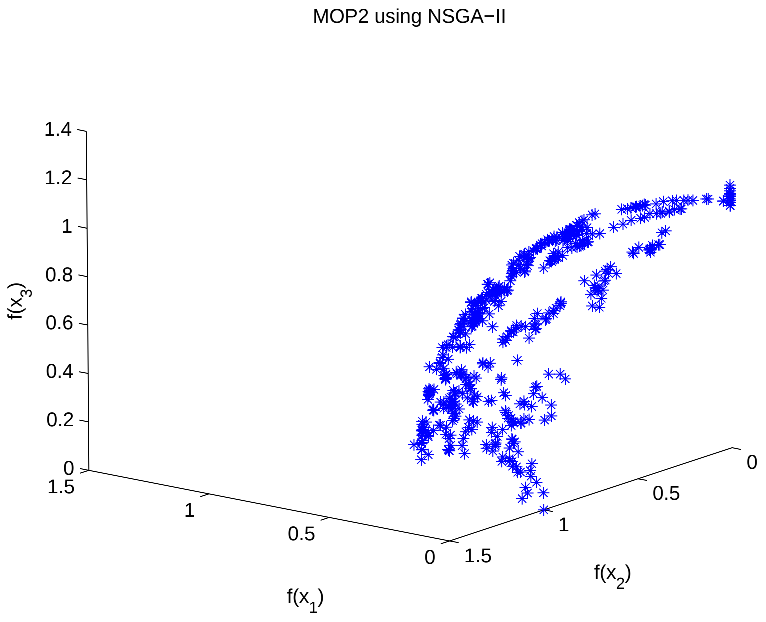

FIGURE 13. MOP2: Modified solutions after 1000 generations (view 4)

- Occasionally the crossover operator did not result in a child within the range of parents.
- For any set of population the number of individuals within any front with maximum distance did not go beyond 5. I assumed that it should be 6. But I am not able to visualize if the objective space dimension go beyond 3. Even though this does not have any relevance to the ability of the algorithm, while debugging I had to find out if my algorithm was working properly or not and it took me some time to visualize and figure this out.

7. Potential Improvements

- The crowding distance assignment has to be modified so that better spread of solution is possible. Sometimes in my opinion a better spread may be achieved if we do not assign infinite distance to the boundary solutions. This is just my intuition and I have no simulation result to substantiate it.
- In my opinion a better genetic operators can be designed specifically for NSGA-II algorithm so the crowding operator will result in a better diversity of individuals.

8. Source Code

8.1. **Function for Initialization.**

```
% Initialize population
%
% function f = initialize_variables(N,problem)
% N - Population size
% problem - takes integer values 1 and 2 where,
%           '1' for MOP1
%           '2' for MOP2
```

```
%
% This function initializes the population with N individuals and each
% individual having M decision variables based on the selected problem.
% M = 6 for problem MOP1 and M = 12 for problem MOP2. The objective space
% for MOP1 is 2 dimensional while for MOP2 is 3 dimensional.
function f = initialize_variables(N,problem)
% Both the MOP's given in Homework # 5 has 0 to 1 as its range for all the
% decision variables.
min = 0; max = 1; switch problem
    case 1
        M = 6;
        K = 8;
    case 2
        M = 12;
        K = 15;
end for i = 1 : N
    % Initialize the decision variables
    for j = 1 : M
        f(i,j) = rand(1); % i.e f(i,j) = min + (max - min)*rand(1);
    end
    % Evaluate the objective function
    f(i,M + 1: K) = evaluate_objective(f(i,:),problem);
end

}
```

8.2. **Non-Dominated Sort and Crowding Distance Calculation.**

```
%% Non-Donimation Sort
% This function sort the current popultion based on non-domination. All the
% individuals in the first front are given a rank of 1, the second front
% individuals are assigned rank 2 and so on. After assigning the rank the
% crowding in each front is calculated.

function f = non_domination_sort_mod(x,problem) [N,M] = size(x);
switch problem
    case 1
        M = 2;
        V = 6;
    case 2
        M = 3;
        V = 12;
end front = 1;

% There is nothing to this assignment, used only to manipulate easily in
% MATLAB.
F(front).f = []; individual = []; for i = 1 : N
    % Number of individuals that dominate this individual
    individual(i).n = 0;
    % Individuals which this individual dominate
    individual(i).p = [];
    for j = 1 : N
        dom_less = 0;
        dom_equal = 0;
        dom_more = 0;
        for k = 1 : M
            if (x(i,V + k) < x(j,V + k))
                dom_less = dom_less + 1;
            elseif (x(i,V + k) == x(j,V + k))
                dom_equal = dom_equal + 1;
            else
                dom_more = dom_more + 1;
            end
        end
        if dom_less == 0 & dom_equal ~= M
            individual(i).n = individual(i).n + 1;
        elseif dom_more == 0 & dom_equal ~= M
            individual(i).p = [individual(i).p j];
        end
    end
    if individual(i).n == 0
        x(i,M + V + 1) = 1;
        F(front).f = [F(front).f i];
    end
end
% Find the subsequent fronts
while ~isempty(F(front).f)
   Q = [];
   for i = 1 : length(F(front).f)
       if ~isempty(individual(F(front).f(i)).p)
            for j = 1 : length(individual(F(front).f(i)).p)
                individual(individual(F(front).f(i)).p(j)).n = ...
                    individual(individual(F(front).f(i)).p(j)).n - 1;
                if individual(individual(F(front).f(i)).p(j)).n == 0
                    x(individual(F(front).f(i)).p(j),M + V + 1) = ...
                        front + 1;
                    Q = [Q individual(F(front).f(i)).p(j)];
                end
            end
       end
   end
   front =  front + 1;
   F(front).f = Q;
end [temp,index_of_fronts] = sort(x(:,M + V + 1)); for i = 1 :
length(index_of_fronts)
    sorted_based_on_front(i,:) = x(index_of_fronts(i),:);
end current_index = 0;
% Find the crowding distance for each individual in each front
for front = 1 : (length(F) - 1)
    objective = [];
    distance = 0;
    y = [];
    previous_index = current_index + 1;
    for i = 1 : length(F(front).f)
        y(i,:) = sorted_based_on_front(current_index + i,:);
    end
    current_index = current_index + i;
    % Sort each individual based on the objective
    sorted_based_on_objective = [];
    for i = 1 : M
        [sorted_based_on_objective, index_of_objectives] = ...
            sort(y(:,V + i));
        sorted_based_on_objective = [];
        for j = 1 : length(index_of_objectives)
            sorted_based_on_objective(j,:) = y(index_of_objectives(j),:);
        end
        f_max = ...
            sorted_based_on_objective(length(index_of_objectives), V + i);
        f_min = sorted_based_on_objective(1, V + i);
        y(index_of_objectives(length(index_of_objectives)),M + V + 1 + i)...
            = Inf;
        y(index_of_objectives(1),M + V + 1 + i) = Inf;
         for j = 2 : length(index_of_objectives) - 1
            next_obj  = sorted_based_on_objective(j + 1,V + i);
            previous_obj  = sorted_based_on_objective(j - 1,V + i);
            if (f_max - f_min == 0)
                y(index_of_objectives(j),M + V + 1 + i) = Inf;
            else
                y(index_of_objectives(j),M + V + 1 + i) = ...
                     (next_obj - previous_obj)/(f_max - f_min);
            end
         end
    end
    distance = [];
    distance(:,1) = zeros(length(F(front).f),1);
    for i = 1 : M
        distance(:,1) = distance(:,1) + y(:,M + V + 1 + i);
    end
    y(:,M + V + 2) = distance;
    y = y(:,1 : M + V + 2);
    z(previous_index:current_index,:) = y;
end
f = z();
```

}

8.3. **Tournament Selection.**

```
function f = selection_individuals(chromosome,pool_size,tour_size)

% function selection_individuals(chromosome,pool_size,tour_size)
% function selection_individuals(chromosome,pool_size,tour_size) is the
% selection policy for selecting the individuals for the mating pool. The
% selection is based on tournament selection. Argument 'chromosome' is the
% current generation population from which the individuals are selected to
% form a mating pool of size 'pool_size' after performing tournament
% selection, with size of the tournament being 'tour_size'. By varying the
% tournament size the selection pressure can be adjusted.

[pop,variables] = size(chromosome); rank = variables - 1; distance
= variables;

for i = 1 : pool_size
    for j = 1 : tour_size
        candidate(j) = round(pop*rand(1));
        if candidate(j) == 0
            candidate(j) = 1;
        end
        if j > 1
            while ~isempty(find(candidate(1 : j - 1) == candidate(j)))
                candidate(j) = round(pop*rand(1));
                if candidate(j) == 0
                    candidate(j) = 1;
                end
            end
        end
    end
    for j = 1 : tour_size
        c_obj_rank(j) = chromosome(candidate(j),rank);
        c_obj_distance(j) = chromosome(candidate(j),distance);
    end
    min_candidate = ...
        find(c_obj_rank == min(c_obj_rank));
    if length(min_candidate) ~= 1
        max_candidate = ...
        find(c_obj_distance(min_candidate) == max(c_obj_distance(min_candidate)));
        if length(max_candidate) ~= 1
            max_candidate = max_candidate(1);
        end
        f(i,:) = chromosome(candidate(min_candidate(max_candidate)),:);
    else
        f(i,:) = chromosome(candidate(min_candidate(1)),:);
    end
end

}
```

8.4. **Genetic Operator.**

```
function f  = genetic_operator(parent_chromosome,pro,mu,mum);

[N,M] = size(parent_chromosome); switch pro
    case 1
        M = 2;
        V = 6;
    case 2
        M = 3;
        V = 12;
end p = 1; was_crossover = 0; was_mutation = 0; l_limit = 0;
u_limit = 1; for i = 1 : N
    if rand(1) < 0.9
        child_1 = [];
        child_2 = [];
        parent_1 = round(N*rand(1));
        if parent_1 < 1
            parent_1 = 1;
        end
        parent_2 = round(N*rand(1));
        if parent_2 < 1
            parent_2 = 1;
        end
        while isequal(parent_chromosome(parent_1,:),parent_chromosome(parent_2,:))
            parent_2 = round(N*rand(1));
            if parent_2 < 1
                parent_2 = 1;
            end
        end
        parent_1 = parent_chromosome(parent_1,:);
        parent_2 = parent_chromosome(parent_2,:);
        for j = 1 : V
            %% SBX (Simulated Binary Crossover)
            % Generate a random number
            u(j) = rand(1);
            if u(j) <= 0.5
                bq(j) = (2*u(j))^(1/(mu+1));
            else
                bq(j) = (1/(2*(1 - u(j))))^(1/(mu+1));
            end
            child_1(j) = ...
                0.5*(((1 + bq(j))*parent_1(j)) + (1 - bq(j))*parent_2(j));
            child_2(j) = ...
                0.5*(((1 - bq(j))*parent_1(j)) + (1 + bq(j))*parent_2(j));
            if child_1(j) > u_limit
                child_1(j) = u_limit;
            elseif child_1(j) < l_limit
                child_1(j) = l_limit;
            end
            if child_2(j) > u_limit
                child_2(j) = u_limit;
            elseif child_2(j) < l_limit
                child_2(j) = l_limit;
            end
        end
        child_1(:,V + 1: M + V) = evaluate_objective(child_1,pro);
        child_2(:,V + 1: M + V) = evaluate_objective(child_2,pro);
        was_crossover = 1;
        was_mutation = 0;
    else
        parent_3 = round(N*rand(1));
        if parent_3 < 1
            parent_3 = 1;
        end
        % Make sure that the mutation does not result in variables out of
        % the search space. For both the MOP's the range for decision space
        % is [0,1]. In case different variables have different decision
        % space each variable can be assigned a range.
        child_3 = parent_chromosome(parent_3,:);
        for j = 1 : V
           r(j) = rand(1);
           if r(j) < 0.5
               delta(j) = (2*r(j))^(1/(mum+1)) - 1;
           else
               delta(j) = 1 - (2*(1 - r(j)))^(1/(mum+1));
           end
           child_3(j) = child_3(j) + delta(j);
           if child_3(j) > u_limit
               child_3(j) = u_limit;
           elseif child_3(j) < l_limit
               child_3(j) = l_limit;
           end
        end
        child_3(:,V + 1: M + V) = evaluate_objective(child_3,pro);
        was_mutation = 1;
        was_crossover = 0;
    end
    if was_crossover
        child(p,:) = child_1;
        child(p+1,:) = child_2;
        was_cossover = 0;
        p = p + 2;
    elseif was_mutation
        child(p,:) = child_3(1,1 : M + V);
        was_mutation = 0;
        p = p + 1;
    end
end f = child;

}
```

8.5. **Objective Function Evaluation.**

```
function f = evaluate_objective(x,problem)

% Function to evaluate the objective functions for the given input vector
% x. x has the decision variables

switch problem
    case 1
        f = [];
        %% Objective function one
        f(1) = 1 - exp(-4*x(1))*(sin(6*pi*x(1)))^6;
        sum = 0;
        for i = 2 : 6
            sum = sum + x(i)/4;
        end
        %% Intermediate function
        g_x = 1 + 9*(sum)^(0.25);
        %% Objective function one
        f(2) = g_x*(1 - ((f(1))/(g_x))^2);
    case 2
        f = [];
        %% Intermediate function
        g_x = 0;
        for i = 3 : 12
            g_x = g_x + (x(i) - 0.5)^2;
        end
        %% Objective function one
        f(1) = (1 + g_x)*cos(0.5*pi*x(1))*cos(0.5*pi*x(2));
        %% Objective function two
        f(2) = (1 + g_x)*cos(0.5*pi*x(1))*sin(0.5*pi*x(2));
        %% Objective function three
        f(3) = (1 + g_x)*sin(0.5*pi*x(1));
end

}
```

8.6. **Selection Operator.**

```
function f  = replace_chromosome(intermediate_chromosome,pro,pop)

[N,V] = size(intermediate_chromosome); switch pro
        case 1
        M = 2;
        V = 6;
    case 2
        M = 3;
        V = 12;
end
% Get the index for the population sort based on the rank
[temp,index] = sort(intermediate_chromosome(:,M + V + 1));
% Now sort the individuals based on the index
for i = 1 : N
    sorted_chromosome(i,:) = intermediate_chromosome(index(i),:);
end
% Find the maximum rank in the current population
max_rank = max(intermediate_chromosome(:,M + V + 1));
% Start adding each front based on rank and crowing distance until the
% whole population is filled.
previous_index = 0; for i = 1 : max_rank
    current_index = max(find(sorted_chromosome(:,M + V + 1) == i));
    if current_index > pop
        remaining = pop - previous_index;
        temp_pop = ...
            sorted_chromosome(previous_index + 1 : current_index, :);
        [temp_sort,temp_sort_index] = ...
            sort(temp_pop(:, M + V + 2),'descend');
        for j = 1 : remaining
            f(previous_index + j,:) = temp_pop(temp_sort_index(j),:);
        end
        return;
    elseif current_index < pop
        f(previous_index + 1 : current_index, :) = ...
            sorted_chromosome(previous_index + 1 : current_index, :);
    else
        f(previous_index + 1 : current_index, :) = ...
            sorted_chromosome(previous_index + 1 : current_index, :);
        return;
    end
    previous_index = current_index;
end
```

}

8.7. **Main Function NSGA-II.**

```
%% Main Script
% Main program to run the NSGA-II MOEA.
% initialize_variables has two arguments; First being the population size
% and the second the problem number. '1' corresponds to MOP1 and '2'
% corresponds to MOP2.

%% Initialize the variables
% Declare the variables and initialize their values
% pop - population
% gen - generations
% pro - problem number
function nsga_2(); pop = 500; gen = 1000; pro = 2;

switch pro
        case 1
        M = 2;
        V = 6;
    case 2
        M = 3;
        V = 12;
end

chromosome = initialize_variables(pop,pro);


%% Sort the initialized population
% Sort the population using non-domination-sort. This returns two columns
% for each individual which are the rank and the crowding distance
% corresponding to their position in the front they belong.
chromosome = non_domination_sort_mod(chromosome,pro); for i = 1 :
gen
    %% Select the parents
    % Parents are selected for reproduction to generate offspringd. The
    % original NSGA-II uses a binary tournament selection based on the
    % crowded-comparision operator. The arguments are
    % pool - size of the mating pool. It is common to have this to be half the
    %        population size.
    % tour - Tournament size. Original NSGA-II uses a binary tournament
    %        selection, but to see the effect of tournament size this is kept
    %        arbitary, to be choosen by the user.
    pool = round(pop/2);
    tour = 2;
    parent_chromosome = tournament_selection(chromosome,pool,tour);

    %% Perfrom crossover and Mutation operator
    % The original NSGA-II algorithm uses Simulated Binary Crossover (SBX) and
    % Polynomial crossover. Crossover probability pc = 0.9 and mutation
    % probability is pm = 1/n, where n is the number of decision variables.
    % Both real-coded GA and binary-coded GA are implemented in the original
    % algorithm, while in this program only the real-coded GA is considered.
    % The distribution indeices for crossover and mutation operators as mu = 20
    % and mum = 20 respectively.
    mu = 20;
    mum = 20;
    offspring_chromosome = genetic_operator(parent_chromosome,pro,mu,mum);

    %% Intermediate population
    % Intermediate population is the combined population of parents and
    % offsprings of the current generation. The population size is almost 1 and
    % half times the initial population.
    [main_pop,temp] = size(chromosome);
    [offspring_pop,temp] = size(offspring_chromosome);
    intermediate_chromosome(1:main_pop,:) = chromosome;
    intermediate_chromosome(main_pop + 1 : main_pop + offspring_pop,1 : M+V) = ...
        offspring_chromosome;

    %% Non-domination-sort of intermediate population
    % The intermediate population is sorted again based on non-domination sort
    % before the replacement operator is performed on the intermediate
    % population.
    intermediate_chromosome = ...
        non_domination_sort_mod(intermediate_chromosome,pro);
    %% Perform Selection
    % Once the intermediate population is sorted only the best solution is
    % selected based on it rank and crowding distance. Each front is filled in
    % ascending order until the addition of population size is reached. The
    % last front is included in the population based on the individuals with
    % least crowding distance
    chromosome = replace_chromosome(intermediate_chromosome,pro,pop);
    if ~mod(i,10)
        fprintf('%d\n',i);
    end
end save solution.txt chromosome -ASCII switch pro
    case 1
        plot(chromosome(:,V + 1),chromosome(:,V + 2),'*');
        title('MOP1 using NSGA-II');
        xlabel('f(x_1)');
        ylabel('f(x_2)');
    case 2
        plot3(chromosome(:,V + 1),chromosome(:,V + 2),chromosome(:,V + 3),'*');
        title('MOP2 using NSGA-II');
        xlabel('f(x_1)');
        ylabel('f(x_2)');
        zlabel('f(x_3)');
end


}
```

References

1. Hans-Georg Beyer and Kalyanmoy Deb, *On Self-Adaptive Features in Real-Parameter Evolutionary Algorithm*, IEEE Trabsactions on Evolutionary Computation **5** (2001), no. 3, 250 – 270.
2. Kalyanmoy Deb and R. B. Agarwal, *Simulated Binary Crossover for Continuous Search Space*, Complex Systems **9** (1995), 115 – 148.
3. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan, *A Fast Elitist Multiobjective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation **6** (2002), no. 2, 182 – 197.
4. M. M. Raghuwanshi and O. G. Kakde, *Survey on multiobjective evolutionary and real coded genetic algorithms*, Proceedings of the 8th Asia Pacific Symposium on Intelligent and Evolutionasy Systems, 2004, pp. 150 – 161.
5. N. Srinivas and Kalyanmoy Deb, *Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms*, Evolutionary Computation **2** (1994), no. 3, 221 – 248.
6. Gary G. Yen and Haiming Lu, *Dynamic multiobjective evolutionary algorithm: adaptive cell-based rank and density estimation*, Evolutionary Computation, IEEE Transactions on **7** (2003), no. 3, 253 – 274.

E-mail address: `aravind.seshadri@okstate.edu`