# 1．function nsga\_2(pop,gen)

is a multi-objective optimization function where the input arguments are pop - Population size gen - Total number of generations

This functions is based on evolutionary algorithm for finding the optimal solution for multiple objective i.e. pareto front for the objectives. Initially enter only the population size and the stoping criteria or the total number of generations after which the algorithm will automatically stopped.

You will be asked to enter the number of objective functions, the number of decision variables and the range space for the decision variables. Also you will have to define your own objective funciton by editing the evaluate\_objective() function. A sample objective function is described in evaluate\_objective.m. Kindly make sure that the objective function which you define match the number of objectives that you have entered as well as the number of decision variables that you have entered. The decision variable space is continuous for this function, but the objective space may or may not be continuous.

Original algorithm NSGA-II was developed by researchers in Kanpur Genetic Algorithm Labarotary and kindly visit their website for more information<http://www.iitk.ac.in/kangal/>

## Contents

* [Simple error checking](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#1)
* [Objective Function](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#2)
* [Initialize the population](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#3)
* [Sort the initialized population](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#4)
* [Start the evolution process](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#5)
* [Result](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#6)
* [Visualize](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\nsga_2.html#7)

## Simple error checking

Number of Arguments Check for the number of arguments. The two input arguments are necessary to run this function.

if nargin < 2

error('NSGA-II: Please enter the population size and number of generations as input arguments.');

end

% Both the input arguments need to of integer data type

if isnumeric(pop) == 0 || isnumeric(gen) == 0

error('Both input arguments pop and gen should be integer datatype');

end

% Minimum population size has to be 20 individuals

if pop < 20

error('Minimum population for running this function is 20');

end

if gen < 5

error('Minimum number of generations is 5');

end

% Make sure pop and gen are integers

pop = round(pop);

gen = round(gen);

## Objective Function

The objective function description contains information about the objective function. M is the dimension of the objective space, V is the dimension of decision variable space, min\_range and max\_range are the range for the variables in the decision variable space. User has to define the objective functions using the decision variables. Make sure to edit the function 'evaluate\_objective' to suit your needs.

[M, V, min\_range, max\_range] = objective\_description\_function();

## Initialize the population

Population is initialized with random values which are within the specified range. Each chromosome consists of the decision variables. Also the value of the objective functions, rank and crowding distance information is also added to the chromosome vector but only the elements of the vector which has the decision variables are operated upon to perform the genetic operations like corssover and mutation.

chromosome = initialize\_variables(pop, M, V, min\_range, max\_range);

## 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. At this stage the rank and the crowding distance for each chromosome is added to the chromosome vector for easy of computation.

chromosome = non\_domination\_sort\_mod(chromosome, M, V);

## Start the evolution process

The following are performed in each generation \* Select the parents which are fit for reproduction \* Perfrom crossover and Mutation operator on the selected parents \* Perform Selection from the parents and the offsprings \* Replace the unfit individuals with the fit individuals to maintain a constant population size.

for i = 1 : gen

% Select the parents

% Parents are selected for reproduction to generate offspring. 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;

% Selection process

% A binary tournament selection is employed in NSGA-II. In a binary

% tournament selection process two individuals are selected at random

% and their fitness is compared. The individual with better fitness is

% selcted as a parent. Tournament selection is carried out until the

% pool size is filled. Basically a pool size is the number of parents

% to be selected. The input arguments to the function

% tournament\_selection are chromosome, pool, tour. The function uses

% only the information from last two elements in the chromosome vector.

% The last element has the crowding distance information while the

% penultimate element has the rank information. Selection is based on

% rank and if individuals with same rank are encountered, crowding

% distance is compared. A lower rank and higher crowding distance is

% the selection criteria.

parent\_chromosome = tournament\_selection(chromosome, pool, tour);

% Perfrom crossover and Mutation operator

% The original NSGA-II algorithm uses Simulated Binary Crossover (SBX) and

% Polynomial mutation. 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, ...

M, V, mu, mum, min\_range, max\_range);

% Intermediate population

% Intermediate population is the combined population of parents and

% offsprings of the current generation. The population size is two

% times the initial population.

[main\_pop,temp] = size(chromosome);

[offspring\_pop,temp] = size(offspring\_chromosome);

% temp is a dummy variable.

clear temp

% intermediate\_chromosome is a concatenation of current population and

% the offspring population.

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, M, V);

% 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, M, V, pop);

if ~mod(i,100)

clc

fprintf('%d generations completed\n',i);

end

end

## Result

Save the result in ASCII text format.

save solution.txt chromosome -ASCII

## Visualize

The following is used to visualize the result if objective space dimension is visualizable.

if M == 2

plot(chromosome(:,V + 1),chromosome(:,V + 2),'\*');

elseif M ==3

plot3(chromosome(:,V + 1),chromosome(:,V + 2),chromosome(:,V + 3),'\*');

end

——————————————————————————————————

**2．Evaluate\_objective**

**Contents**

**function f = evaluate\_objective(x, M, V)**

Function to evaluate the objective functions for the given input vector x. x is an array of decision variables and f(1), f(2), etc are the objective functions. The algorithm always minimizes the objective function hence if you would like to maximize the function then multiply the function by negative one. M is the numebr of objective functions and V is the number of decision variables.

This functions is basically written by the user who defines his/her own objective function. Make sure that the M and V matches your initial user input. Make sure that the

An example objective function is given below. It has two six decision variables are two objective functions.

% f = [];

% %% Objective function one

% % Decision variables are used to form the objective function.

% 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 two

% f(2) = g\_x\*(1 - ((f(1))/(g\_x))^2);

**Kursawe proposed by Frank Kursawe.**

Take a look at the following reference A variant of evolution strategies for vector optimization. In H. P. Schwefel and R. Männer, editors, Parallel Problem Solving from Nature. 1st Workshop, PPSN I, volume 496 of Lecture Notes in Computer Science, pages 193-197, Berlin, Germany, oct 1991. Springer-Verlag.

Number of objective is two, while it can have arbirtarly many decision variables within the range -5 and 5. Common number of variables is 3.

f = [];

% Objective function one

sum = 0;

for i = 1 : V - 1

sum = sum - 10\*exp(-0.2\*sqrt((x(i))^2 + (x(i + 1))^2));

end

% Decision variables are used to form the objective function.

f(1) = sum;

% Objective function two

sum = 0;

for i = 1 : V

sum = sum + (abs(x(i))^0.8 + 5\*(sin(x(i)))^3);

end

% Decision variables are used to form the objective function.

f(2) = sum;

**Check for error**

if length(f) ~= M

error('The number of decision variables does not match you previous input. Kindly check your objective function');

end

——————————————————————————————————

# 3．Functionf=genetic\_operator(parent\_chromosome, M, V, mu, mum, l\_limit, u\_limit)

This function is utilized to produce offsprings from parent chromosomes. The genetic operators corssover and mutation which are carried out with slight modifications from the original design. For more information read the document enclosed.

parent\_chromosome - the set of selected chromosomes. M - number of objective functions V - number of decision varaiables mu - distribution index for crossover (read the enlcosed pdf file) mum - distribution index for mutation (read the enclosed pdf file) l\_limit - a vector of lower limit for the corresponding decsion variables u\_limit - a vector of upper limit for the corresponding decsion variables

The genetic operation is performed only on the decision variables, that is the first V elements in the chromosome vector.

[N,m] = size(parent\_chromosome);

clear m

p = 1;

% Flags used to set if crossover and mutation were actually performed.

was\_crossover = 0;

was\_mutation = 0;

for i = 1 : N

% With 90 % probability perform crossover

if rand(1) < 0.9

% Initialize the children to be null vector.

child\_1 = [];

child\_2 = [];

% Select the first parent

parent\_1 = round(N\*rand(1));

if parent\_1 < 1

parent\_1 = 1;

end

% Select the second parent

parent\_2 = round(N\*rand(1));

if parent\_2 < 1

parent\_2 = 1;

end

% Make sure both the parents are not the same.

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

% Get the chromosome information for each randomnly selected

% parents

parent\_1 = parent\_chromosome(parent\_1,:);

parent\_2 = parent\_chromosome(parent\_2,:);

% Perform corssover for each decision variable in the chromosome.

for j = 1 : V

% SBX (Simulated Binary Crossover).

% For more information about SBX refer the enclosed pdf file.

% 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

% Generate the jth element of first child

child\_1(j) = ...

0.5\*(((1 + bq(j))\*parent\_1(j)) + (1 - bq(j))\*parent\_2(j));

% Generate the jth element of second child

child\_2(j) = ...

0.5\*(((1 - bq(j))\*parent\_1(j)) + (1 + bq(j))\*parent\_2(j));

% Make sure that the generated element is within the specified

% decision space else set it to the appropriate extrema.

if child\_1(j) > u\_limit(j)

child\_1(j) = u\_limit(j);

elseif child\_1(j) < l\_limit(j)

child\_1(j) = l\_limit(j);

end

if child\_2(j) > u\_limit(j)

child\_2(j) = u\_limit(j);

elseif child\_2(j) < l\_limit(j)

child\_2(j) = l\_limit(j);

end

end

% Evaluate the objective function for the offsprings and as before

% concatenate the offspring chromosome with objective value.

child\_1(:,V + 1: M + V) = evaluate\_objective(child\_1, M, V);

child\_2(:,V + 1: M + V) = evaluate\_objective(child\_2, M, V);

% Set the crossover flag. When crossover is performed two children

% are generate, while when mutation is performed only only child is

% generated.

was\_crossover = 1;

was\_mutation = 0;

% With 10 % probability perform mutation. Mutation is based on

% polynomial mutation.

else

% Select at random the parent.

parent\_3 = round(N\*rand(1));

if parent\_3 < 1

parent\_3 = 1;

end

% Get the chromosome information for the randomnly selected parent.

child\_3 = parent\_chromosome(parent\_3,:);

% Perform mutation on eact element of the selected parent.

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

% Generate the corresponding child element.

child\_3(j) = child\_3(j) + delta(j);

% Make sure that the generated element is within the decision

% space.

if child\_3(j) > u\_limit(j)

child\_3(j) = u\_limit(j);

elseif child\_3(j) < l\_limit(j)

child\_3(j) = l\_limit(j);

end

end

% Evaluate the objective function for the offspring and as before

% concatenate the offspring chromosome with objective value.

child\_3(:,V + 1: M + V) = evaluate\_objective(child\_3, M, V);

% Set the mutation flag

was\_mutation = 1;

was\_crossover = 0;

end

% Keep proper count and appropriately fill the child variable with all

% the generated children for the particular generation.

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;

——————————————————————————————————————

## 4．Contents

* [function f = initialize\_variables(N, M, V, min\_tange, max\_range)](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\initialize_variables.html#1)
* [Initialize each chromosome](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\initialize_variables.html#2)

**function f = initialize\_variables(N, M, V, min\_tange, max\_range)**

This function initializes the chromosomes. Each chromosome has the following at this stage \* set of decision variables \* objective function values

where, N - Population size M - Number of objective functions V - Number of decision variables min\_range - A vector of decimal values which indicate the minimum value for each decision variable. max\_range - Vector of maximum possible values for decision variables.

min = min\_range;

max = max\_range;

% K is the total number of array elements. For ease of computation decision

% variables and objective functions are concatenated to form a single

% array. For crossover and mutation only the decision variables are used

% while for selection, only the objective variable are utilized.

K = M + V;

**Initialize each chromosome**

For each chromosome perform the following (N is the population size)

for i = 1 : N

% Initialize the decision variables based on the minimum and maximum

% possible values. V is the number of decision variable. A random

% number is picked between the minimum and maximum possible values for

% the each decision variable.

for j = 1 : V

f(i,j) = min(j) + (max(j) - min(j))\*rand(1);

end

% For ease of computation and handling data the chromosome also has the

% vlaue of the objective function concatenated at the end. The elements

% V + 1 to K has the objective function valued.

% The function evaluate\_objective takes one chromosome at a time,

% infact only the decision variables are passed to the function along

% with information about the number of objective functions which are

% processed and returns the value for the objective functions. These

% values are now stored at the end of the chromosome itself.

f(i,V + 1: K) = evaluate\_objective(f(i,:), M, V);

end

## 5．Contents

* [function f = non\_domination\_sort\_mod(x, M, V)](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\non_domination_sort_mod.html#1)
* [Non-Dominated sort.](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\non_domination_sort_mod.html#2)
* [Crowding distance](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\non_domination_sort_mod.html#3)
* [References](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\non_domination_sort_mod.html#4)

**function f = non\_domination\_sort\_mod(x, M, V)**

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.

[N, m] = size(x);

clear m

% Initialize the front number to 1.

front = 1;

% There is nothing to this assignment, used only to manipulate easily in

% MATLAB.

F(front).f = [];

individual = [];

**Non-Dominated sort.**

The initialized population is sorted based on non-domination. The fast sort algorithm [1] is described as below for each

% • for each individual p in main population P do the following

% – Initialize Sp = []. This set would contain all the individuals that is

% being dominated by p.

% – Initialize np = 0. This would be the number of individuals that domi-

% nate p.

% – for each individual q in P

% \* if p dominated q then

% · add q to the set Sp i.e. Sp = Sp ? {q}

% \* else if q dominates p then

% · increment the domination counter for p i.e. np = np + 1

% – if np = 0 i.e. no individuals dominate p then p belongs to the first

% front; Set rank of individual p to one i.e prank = 1. Update the first

% front set by adding p to front one i.e F1 = F1 ? {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 ith front is nonempty i.e. Fi != []

% – Q = []. The set for storing the individuals for (i + 1)th front.

% – for each individual p in front Fi

% \* for each individual q in Sp (Sp is the set of individuals

% dominated by p)

% · nq = nq?1, decrement the domination count for individual q.

% · if nq = 0 then none of the individuals in the subsequent

% fronts would dominate q. Hence set qrank = i + 1. Update

% the set Q with individual q i.e. Q = Q ? q.

% – Increment the front counter by one.

% – Now the set Q is the next front and hence Fi = Q.

%

% This algorithm is better than the original NSGA ([2]) since it utilize

% the informatoion about the set that an individual dominate (Sp) and

% number of individuals that dominate the individual (np).

%

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;

**Crowding distance**

%The crowing distance is calculated as below

% • For each front Fi, n is the number of individuals.

% – initialize the distance to be zero for all the individuals i.e. Fi(dj ) = 0,

% where j corresponds to the jth individual in front Fi.

% – for each objective function m

% \* Sort the individuals in front Fi based on objective m i.e. I =

% sort(Fi,m).

% \* Assign infinite distance to boundary values for each individual

% in Fi i.e. I(d1) = ? and I(dn) = ?

% \* for k = 2 to (n ? 1)

% · I(dk) = I(dk) + (I(k + 1).m ? I(k ? 1).m)/fmax(m) - fmin(m)

% · I(k).m is the value of the mth objective function of the kth

% individual in I

% 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();

**References**

[1] **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.

[2] **N. Srinivas and Kalyanmoy Deb**, Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms, Evolutionary Computation 2 (1994), no. 3, 221 ~ 248.

# 6．function [number\_of\_objectives, number\_of\_decision\_variables, min\_range\_of\_decesion\_variable, max\_range\_of\_decesion\_variable] = objective\_description\_function()

This function is used to completely describe the objective functions and the range for the decision variable space etc. The user is prompted for inputing the number of objectives, numebr of decision variables, the maximum and minimum range for each decision variable and finally the function waits for the user to modify the evaluate\_objective function to suit their need.

g = sprintf('Input the number of objective: ');

% Obtain the number of objective function

number\_of\_objectives = input(g);

g = sprintf('\nInput the number of decision variables: ');

% Obtain the number of decision variables

number\_of\_decision\_variables = input(g);

clc

for i = 1 : number\_of\_decision\_variables

clc

g = sprintf('\nInput the minimum value for decision variable %d : ', i);

% Obtain the minimum possible value for each decision variable

min\_range\_of\_decesion\_variable(i) = input(g);

g = sprintf('\nInput the maximum value for decision variable %d : ', i);

% Obtain the maximum possible value for each decision variable

max\_range\_of\_decesion\_variable(i) = input(g);

clc

end

g = sprintf('\n Now edit the function named "evaluate\_objective" appropriately to match your needs.\n Make sure that the number of objective functions and decision variables match your numerical input. \n Make each objective function as a corresponding array element. \n After editing do not forget to save. \n Press "c" and enter to continue... ');

% Prompt the user to edit the evaluate\_objective function and wait until

% 'c' is pressed.

x = input(g, 's');

if isempty(x)

x = 'x';

end

while x ~= 'c'

clc

x = input(g, 's');

if isempty(x)

x = 'x';

end

end

————————————————————————————————————————————

# 7．function f = replace\_chromosome(intermediate\_chromosome,pro,pop)

This function replaces the chromosomes based on rank and crowding distance. Initially until the population size is reached each front is added one by one until addition of a complete front which results in exceeding the population size. At this point the chromosomes in that front is added subsequently to the population based on crowding distance.

[N, m] = size(intermediate\_chromosome);

% Get the index for the population sort based on the rank

[temp,index] = sort(intermediate\_chromosome(:,M + V + 1));

clear temp m

% 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

% Get the index for current rank i.e the last the last element in the

% sorted\_chromosome with rank i.

current\_index = max(find(sorted\_chromosome(:,M + V + 1) == i));

% Check to see if the population is filled if all the individuals with

% rank i is added to the population.

if current\_index > pop

% If so then find the number of individuals with in with current

% rank i.

remaining = pop - previous\_index;

% Get information about the individuals in the current rank i.

temp\_pop = ...

sorted\_chromosome(previous\_index + 1 : current\_index, :);

% Sort the individuals with rank i in the descending order based on

% the crowding distance.

[temp\_sort,temp\_sort\_index] = ...

sort(temp\_pop(:, M + V + 2),'descend');

% Start filling individuals into the population in descending order

% until the population is filled.

for j = 1 : remaining

f(previous\_index + j,:) = temp\_pop(temp\_sort\_index(j),:);

end

return;

elseif current\_index < pop

% Add all the individuals with rank i into the population.

f(previous\_index + 1 : current\_index, :) = ...

sorted\_chromosome(previous\_index + 1 : current\_index, :);

else

% Add all the individuals with rank i into the population.

f(previous\_index + 1 : current\_index, :) = ...

sorted\_chromosome(previous\_index + 1 : current\_index, :);

return;

end

% Get the index for the last added individual.

previous\_index = current\_index;

end

————————————————————————————————————————————

# 8．function tournament\_selection(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. But for NSGA-II the tour\_size is fixed to two, but the user may feel free to experiment with different tournament size. Also it has been observed that a tournament size of more than five has no significant meaning.

## Contents

* [Tournament selection process](file:///F:\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2-%E7%94%B0%E4%B8%80\%E4%B8%AA%E4%BA%BA%E8%B5%84%E6%96%99-%E7%94%B0-%E6%9C%AA%E7%BB%8F%E5%85%81%E8%AE%B8%E8%AF%B7%E5%8B%BF%E5%8A%A8-%E8%B0%A2%E8%B0%A2\Matlab%E8%B5%84%E6%96%99\MATLAB%E5%A4%9A%E7%9B%AE%E6%A0%87%E8%BF%9B%E5%8C%96%E7%AE%97%E6%B3%95%E7%A8%8B%E5%BA%8F\NSGA-II%E5%A4%9A%E7%9B%AE%E6%A0%87%E4%BC%98%E5%8C%96%E7%A8%8B%E5%BA%8F\NSGA-II\html\tournament_selection.html#1)

## Tournament selection process

In a tournament selection process n individuals are selected at random, where n is equal to tour\_size. From these individuals only one is selected and is added to the mating pool, where size of the mating pool is pool\_size. Selection is performed based on two criteria. First and foremost is the rank or the front in which the solutions reside. Individuals with lower rank are selected. Secondly if the rank of two individuals are the same then, the crowding distance is compared. Individuals with greater crowding distance is selcted.

% Get the size of chromosome. The number of chromosome is not important

% while the number of elements in chromosome are important.

[pop, variables] = size(chromosome);

% The peunltimate element contains the information about rank.

rank = variables - 1;

% The last element contains information about crowding distance.

distance = variables;

% Until the mating pool is filled, perform tournament selection

for i = 1 : pool\_size

% Select n individuals at random, where n = tour\_size

for j = 1 : tour\_size

% Select an individual at random

candidate(j) = round(pop\*rand(1));

% Make sure that the array starts from one.

if candidate(j) == 0

candidate(j) = 1;

end

if j > 1

% Make sure that same candidate is not choosen.

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

% Collect information about the selected candidates.

for j = 1 : tour\_size

c\_obj\_rank(j) = chromosome(candidate(j),rank);

c\_obj\_distance(j) = chromosome(candidate(j),distance);

end

% Find the candidate with the least rank

min\_candidate = ...

find(c\_obj\_rank == min(c\_obj\_rank));

% If more than one candiate have the least rank then find the candidate

% within that group having the maximum crowding distance.

if length(min\_candidate) ~= 1

max\_candidate = ...

find(c\_obj\_distance(min\_candidate) == max(c\_obj\_distance(min\_candidate)));

% If a few individuals have the least rank and have maximum crowding

% distance, select only one individual (not at random).

if length(max\_candidate) ~= 1

max\_candidate = max\_candidate(1);

end

% Add the selected individual to the mating pool

f(i,:) = chromosome(candidate(min\_candidate(max\_candidate)),:);

else

% Add the selected individual to the mating pool

f(i,:) = chromosome(candidate(min\_candidate(1)),:);

end

end