# Homework 4 <RGA>

Find the minimized and maximized value of the function using RGA:

**Answer**:

Best answer found:

minimum\_f = -2.4280e+09

minimum\_x = -0.6959

minimum\_y = -1.4513

minimum\_z = 33.0092

**Results**:

Results for:

popu\_size=1000;

gene\_no=3;

range=[-40 -50 -60;

40 50 60];

fitfcn='GA\_fitfun';

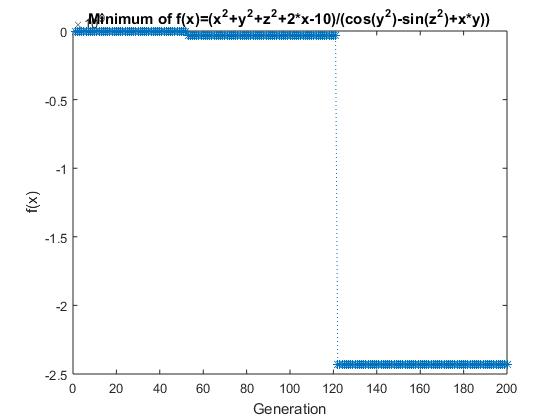
generation\_no=50;

crossover\_rate = 0.5;

mutate\_rate = 0.15;

elite = 1;

I set population size to a large number because I noticed that RGA takes much less time to execute in comparison to BGA. So, RGA with these initial parameters didn’t actually take much time to compute. I noticed that mutation rate wasn’t as effective as in BGA.



Answer was found in generation = 122

**RGA implementation**:

Implementation of RGA was very simple because there was no need to encode and decode schemes. Also because of that RGA was much more faster in computing large population for large number of generations in comparison to BGA when solving this task.

**New\_popu:**

function new\_popu = GA\_newpopu(popu, fitness, crossover\_rate, mutate\_rate, elite, range)

new\_popu = popu;

popu\_n = size(popu, 1);

gene\_number = size(popu, 2);

if elite == 1

tmp\_fitness = fitness;

[max1, index1] = max(tmp\_fitness);

tmp\_fitness(index1) = min(tmp\_fitness);

[max2, index2] = max(tmp\_fitness);

end

fitness\_rate = fitness/sum(fitness);

fitness\_rate\_cum = cumsum(fitness\_rate);

for i = 1: popu\_n/2

tmp = find(fitness\_rate\_cum > rand);

parent1 = popu(tmp(1), :);

tmp = find (fitness\_rate\_cum > rand);

parent2 = popu(tmp(1), :);

if rand < crossover\_rate

for J = 1 : gene\_number

new\_popu(i\*2 - 1, J) = parent1(J) + rand\*(parent2(J)-parent1(J));

new\_popu(i\*2, J) = parent2(J) + rand\*(parent1(J)-parent2(J));

end

end

end

for i = 1: popu\_n

for j = 1 : gene\_number

if rand < mutate\_rate

new\_popu(i,gene\_number) = range(1,j) + rand \* (range(2,j)-range(1,j));

end

end

end

if elite==1

new\_popu([1:2], :) = popu([index1 index2], :);

end

In this function I implemented:

* Adewuya crossover:

I decided to use Adewuya crossover, not arithmetic or Wright crossover, because it ensured that the offspring chromosomes will remain in the range of chromosomes.

* Uniform crossover:

Other functions needed only to remove decoding and encoding schemes.

Main function is very similar to main functions used in previous projects:

popu\_size=1000;

gene\_no=3;

range=[-40 -50 -60;

40 50 60];

fitfcn='GA\_fitfun';

generation\_no=200;

crossover\_rate = 0.5;

mutate\_rate = 0.15;

elite = 1;

global MIN\_offset

[popu, fitness, upper, average, lower, BEST\_popu]...

=GA\_genetic(popu\_size, gene\_no, range, fitfcn, ...

generation\_no, crossover\_rate, mutate\_rate, elite);

minfitness=MIN\_offset-upper;

[minimum\_f,generation]=min(minfitness)

minimum\_x=BEST\_popu(generation,1)

minimum\_y=BEST\_popu(generation,2)

minimum\_z=BEST\_popu(generation,3)

t=1:generation\_no;

plot(t,minfitness,'\*:')

title('Minimum of f(x)=(x^2+y^2+z^2+2\*x-10)/(cos(y^2)-sin(z^2)+x\*y))');

xlabel('Generation')

ylabel('f(x)')

The only difference is how GA\_genetic function is now called (it naturally lacks bit\_length parameter).

**GA\_genetic**:

function [popu, fitness, upper, average, lower, BEST\_popu]...

=GA\_genetic(popu\_size, gene\_no, range, fitfcn, ...

generation\_no, crossover\_rate, mutate\_rate, elite)

initpopu=GA\_initpopu(popu\_size, gene\_no, range);

popu=initpopu;

upper= zeros(generation\_no,1);

average= zeros(generation\_no,1);

lower= zeros(generation\_no,1);

BEST\_popu= zeros(generation\_no,gene\_no);

for nn = 1:generation\_no;

[fitness, popu] = GA\_fitpopu(popu, fitfcn);

[upper(nn),index]=max(fitness);

average(nn)=mean(fitness);

lower(nn)=min(fitness);

BEST\_popu(nn,1:gene\_no)=popu(index,:);

popu=GA\_newpopu(popu, fitness, crossover\_rate, mutate\_rate, elite, range);

end

This function similarly to main function differed only in parameters used to call functions (no bit\_lenght).

**Initpopu**:

function initpopu = GA\_initpopu(popu\_size, gene\_number, range)

initpopu=rand(popu\_size,gene\_number);

for i = 1:popu\_size

initpopu(i,:)=(range(2,:)-range(1,:)).\*initpopu(i,:)+range(1,:);

end

Initializing population was very simple in RGA – it was enough to fill the matrix with random values in range of chromosomes.

**Fitting population**:

function [fitness, popu] = GA\_fitpopu(popu, fitfcn)

popu\_size = size(popu, 1);

for I = 1 : popu\_size,

fitness(I)=GA\_fiteach(popu(I,:), fitfcn);

end

T=-fitness;

[T, index]=sort(T);

fitness=-T;

popu = popu(index,:);

Again, this function just differs from previous assignments in that it doesn’t need to convert chromosomes from binary to real.

**Fit\_each**:

function PI = GA\_fiteach(chro, fitfcn)

PI = feval(fitfcn, chro);

This function only calls fitting function. I left this function for the structure of the program to remain the same (but it is not needed and can be replaced just with PI=feval(…)).