KAUNAS UNIVERSITY OF TECHNOLOGY

Faculty of Mathematics
and Natural Sciences

Optimization Methods

Laboratory work report
Lab 5

Names:Mohamed Abdelmonem,Alexandros Veremis
Group Number: 26
Submission date: 2020-05-22

(Erasmus- Spring 2020)

KAUNAS , 2020

REPORT FOR LABORATORY 5:

Part 1

Use standard Matlab optimization techniques from Global Optimization Toolbox for optimizing function by means of:
-Genetic Algorithms
-Simulated Annealing Algorithms
-Particle Swarm Optimization Algorithms
Optimize the 3 functions below:

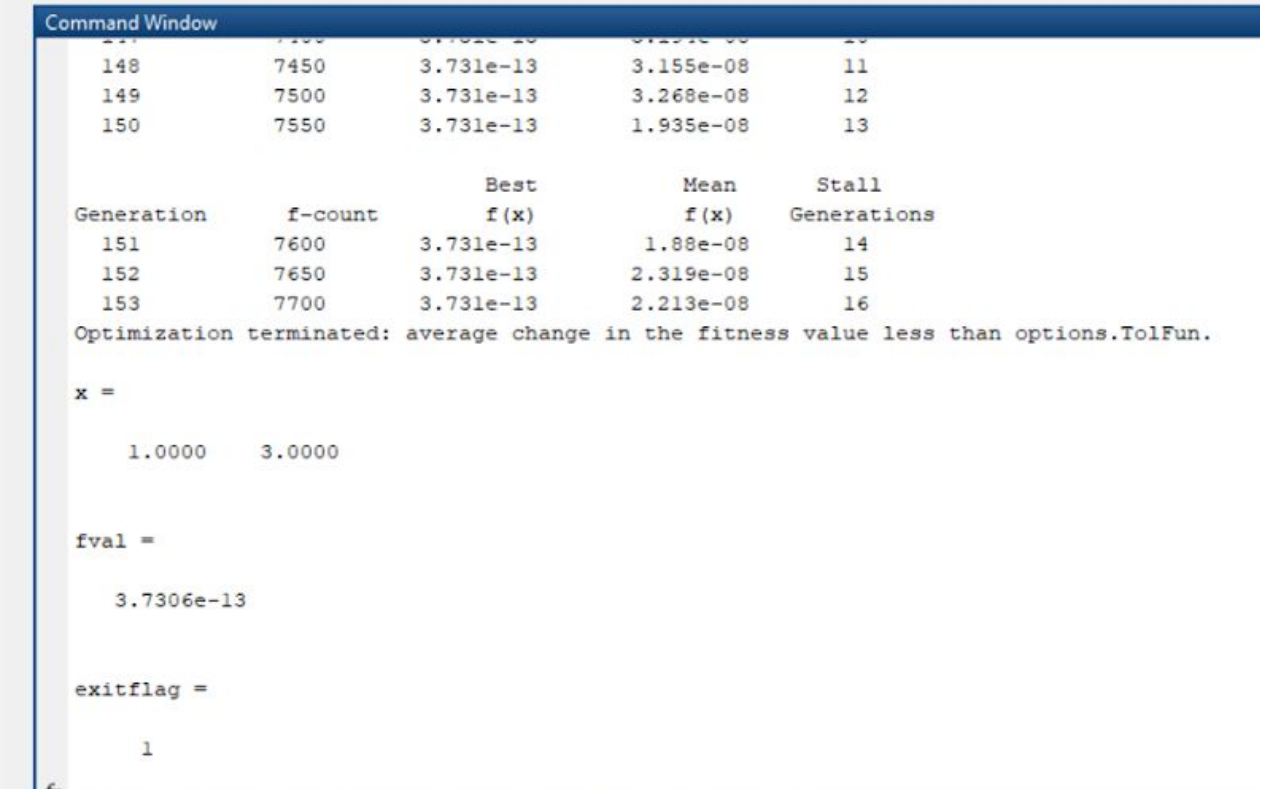
- *Booth Function: f(x,y)= (x+2*y-7).^2+(2*x+y-5).^2*
- *Matyas Function: f(x,y)=0.26*(x.^2+y.^2)- 0.48*x*y*
- *Three-hump Camel Function : f(x,y)= 2*x.^2-1.05*x.^4+(x.^6)/6 +x*y +y.^2*

Report the results, discuss the selection of parameters for different algorithms.

- For the First function - Booth Function: f(x,y)= (x+2*y-7).^2+(2*x+y-5).^2 we will use Genetic Algorithm in order to find the minimum. As a result we will get (xmin, ymin) = (1,3) which is correct and we will also get fmin=3.7306e-13 which is very close to 0. The correct answer is fmin to be zero and the deviation we got is typical and expected when we deal with minimums that are zeros. So, the algorithm works pretty well with this function.

About the parameters, we used:
i)options=optimset('Display','iter','TolFun', 1e-12);
because we can change the options, in order to observe the changes in every iteration and for the fmin to be less than 1e-12.
ii)lb = [-10; -10]; %lower bounds
ub = [10; 10]; %upper bounds
because -10<= x,y <=10
We did not use nonlinear constraints because we do not have any.
Moreover, there are not any linear equalities or inequalities to take care of. That is the reason why here we have empty square brackets in the places of A,b,Aeq,beq:
[x,fval,exitflag,output]=ga(@funBooth,2,[],[],[],[],lb,ub,[],options);
Finally, we have 2 dimensions and that is the reason why the number of variables (nvars) is 2.

Output:



- For the Second function - Matyas Function: f(x,y)=0.26*(x.^2+y.^2)- 0.48*x*y we will use Particle Swarm Optimization Algorithm to find the minimum of the Function. As a result we will get

(xmin, ymin) = (1.0e-05 *0.1909 , 1.0e-05 * 0.09631,3) which is very close to 0 . And we will additionally get fmin=3.0618e-13 which is also very close to 0. The correct answer for (xmin,ymin) is (0,0) and for fmin is also zero .As we mentioned above the deviation we got is typical and expected when we deal with minimums that are zeros. So, the algorithm works pretty well with this function.

About the parameters, we used:

i)options = optimoptions('particleswarm','SwarmSize',100);

in order for the Swarm size to be 100 ,which was recommended in Matlab documentation (100 for a swarm size seems to help the algorithm work fast and reliably).

ii)lb = [-10; -10]; %lower bounds

ub = [10; 10]; %upper bounds

because -10<= x,y <=10

Finally, we have 2 dimensions and that is the reason why the number of variables (nvars) is 2.

Output:

```
Optimization ended: relative change in the objective value
over the last OPTIONS.StallIterLimit iterations is less than OPTIONS.TolFun.

x =

   1.0e-05 *

    0.1909    0.0963


fval =

   3.0618e-13


exitflag =

     1
```

- For the Third function - Three-hump Camel Function :

f(x,y)= 2*x.^2-1.05*x.^4+(x.^6)/6 +x*y +y.^2, we will use Simulated Annealing Algorithm to find the minimum of this function. As a result we will get
(xmin, ymin) = (1.0e-04 *(-0.2036) ,1.0e-04 * 0.3995) which is very close to 0 . And we will additionally get fmin=1.6114e-09 which is also very close to 0. The correct answer for (xmin,ymin) is (0,0) and for fmin is also zero .As we mentioned above the deviation we got is typical and expected when we deal with minimums that are zeros. So, the algorithm works pretty well with this function.
About the parameters, we used:
i)options = optimoptions(@simulannealbnd,'TolFun',1e-08);
in order for the Termination tolerance on function value to be less than 1e-08 and we achieved it. Since the result came to be 1.6114e-09 < 1e-08 .
ii)lb = [-5; -5]; %lower bounds
ub = [5; 5]; %upper bounds
because -5<= x,y <=5
Finally, for our starting point x0 we randomly used (-1,1) as we could use anything in between the bounds. The optimal would be to use (0,0) as a starting point since we know that it is the minimum point, but we would not have put our algorithm into a test in this way.

Output:

```
Optimization terminated: change in best function value less than options.FunctionTolerance.

x =

   1.0e-04 *

   -0.2036    0.3995


fval =

   1.6114e-09


exitflag =

     1
```

Program Code for Part 1:

Part1.m

```
close all;
clear all;

%First function to be optimized:
%Booth Function
% f(x,y)= (x+2*y-7).^2+(2*x+y-5).^2
%Here we will use genetic algorithm to find the minimum of the Booth
%Function
options=optimset('Display','iter','TolFun', 1e-12);
lb = [-10; -10];
ub = [10; 10];
[x,fval,exitflag,output]=ga(@funBooth,2,[],[],[],[],lb,ub,[],options);
x
fval
exitflag

%Second function to be optimized:
%Matyas Function
%  f(x,y)=0.26*(x.^2+y.^2)- 0.48*x*y
%Here we will use Particle Swarm Optimization Algorithm to find the minimum
%of the Matyas Function
options = optimoptions('particleswarm','SwarmSize',100);
lb = [-10; -10];
ub = [10; 10];
[x,fval,exitflag,output]=particleswarm(@funMatyas,2,lb,ub,options);
x
fval
exitflag

%Third function to be  optimized:
%Three-hump Camel Function
```

```
% f(x,y)= 2*x.^2-1.05*x.^4+(x.^6)/6 +x*y +y.^2
%Here we will use Simulated Annealing Algorithm to find the minimum
%of the Three-hump Camel Function
options = optimoptions(@simulannealbnd,'TolFun',1e-08);
lb = [-5; -5];
ub = [5; 5];
x0=[-1 1];
[x,fval,exitflag,output]=simulannealbnd(@funCamel,x0,lb,ub,options);
x
fval
exitflag
```

funBooth.m

```
function f =funBooth(x)
        f =  (x(1)+2*x(2)-7).^2+(2*x(1)+x(2)-5).^2;
end
```

funMatyas.m

```
function f =funMatyas(x)
        f =  0.26*(x(1).^2+x(2).^2)- 0.48*x(1)*x(2);
end
```

funCamel.m

```
function f =funCamel(x)
        f =  2*x(1).^2-1.05*x(1).^4+(x(1).^6)/6 +x(1)*x(2) +x(2).^2;
end
```

Part 2

In Part 2 we must find the minimum of the given function. Since the provided code only calculates the maximum we need to do a small modification to our fitness function. Our fitness value must be high when the value of our function is low and iit needs to be low when the value of our function is high. So instead of setting our fitness value as:

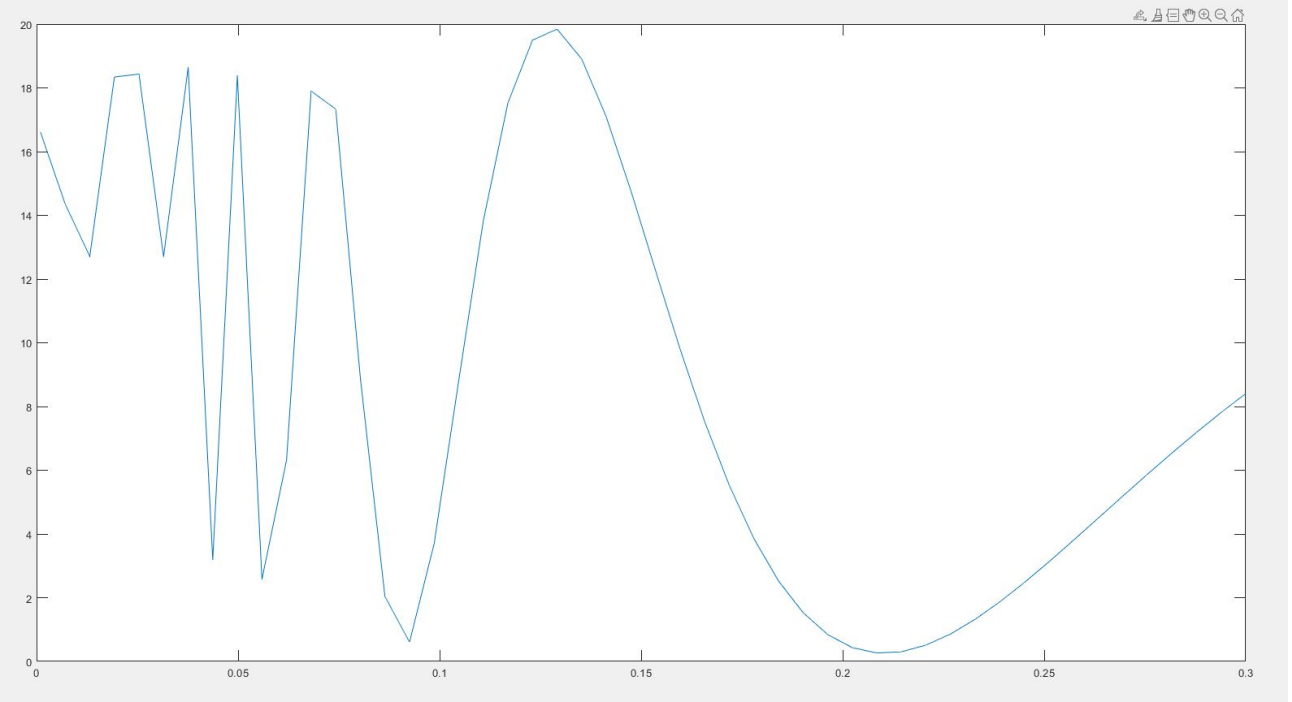
```
fit(k)=feval(kriter,v(k))
```

we will set it as:

```
fit(k)=feval(kriter,v(k)).^-1;
```

Because when we take the inverse, we can guarantee that high values become low and low values become high.

To be able to evaluate how well our genetic algorithm works let us plot the function and look at the minimum we aim to reach.



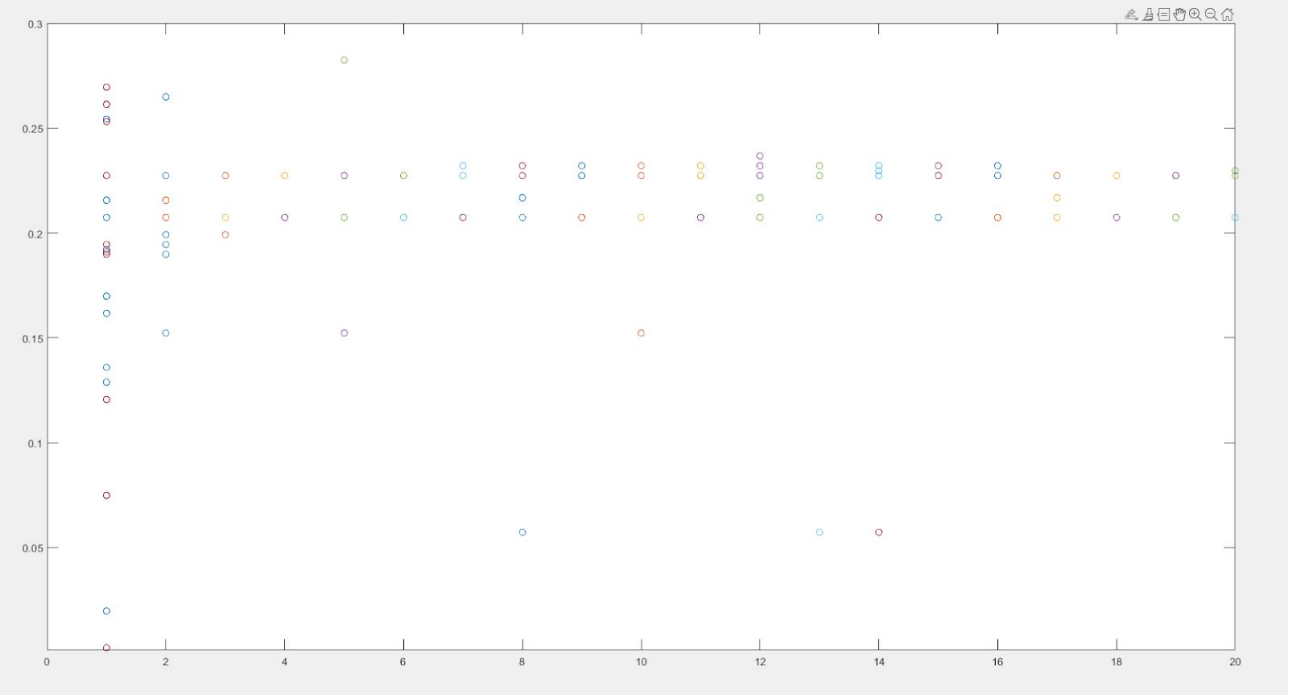
From this graph we can see that the minimum lies somewhere between 0.2 and 0.25 and if we compute it analytically we get 0.21027 as minimum with a value of 0.25566.

Lets see what our algorithm throws out when we use the same parameters as in the given example.

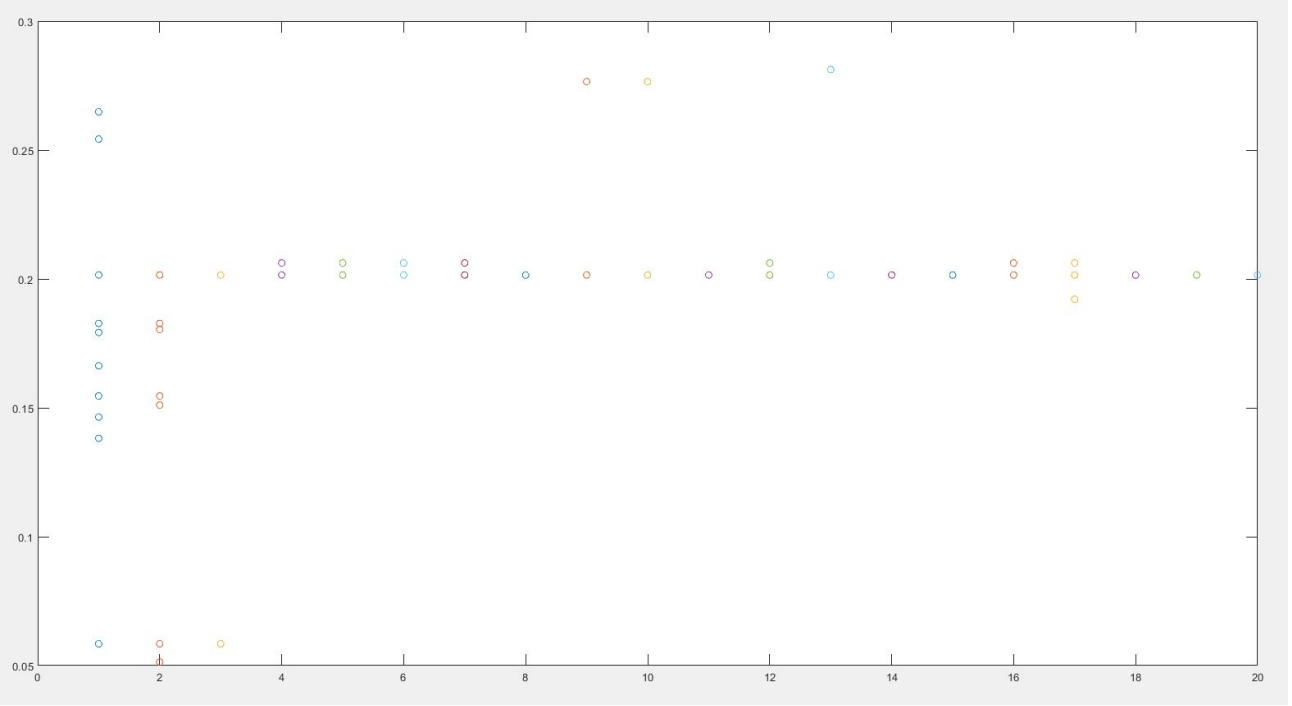
word length = 8;
population size = 10;
number of generations = 20;
mutation parameter = 0.005;
crossover degree = 0.6;

Result for minimum:
0.2273

with an error of only 0.01703, this result seems to be very good. Let us see how we got to that result:

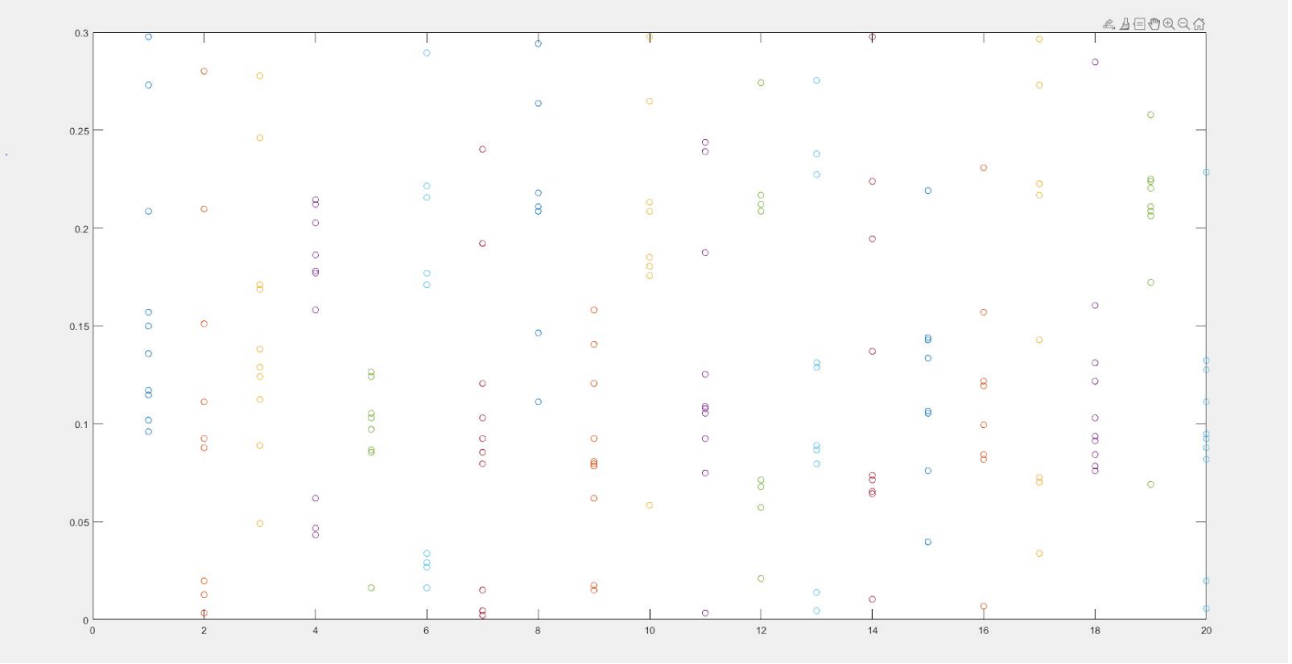


As we can see in the 20th generation, there are still chromosomes that are far away from each other. So if we want to achieve better results with only 20 generations, we need faster convergence. Let us try to get that by increasing the mutation parameter to 0.008

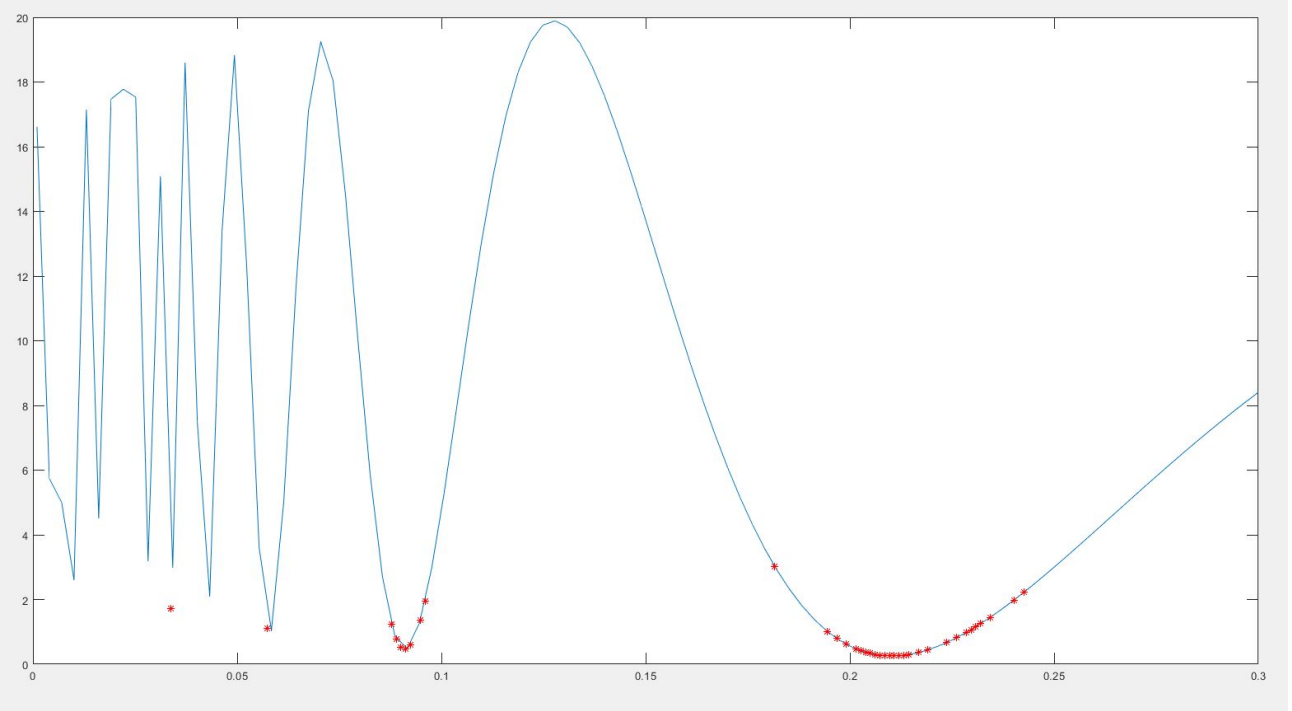


Now we got 0.2015 as minimum point or an error of 0.00877, which is even better than what we got before. We can also see that the values of our chromosomes have converged to one point.

But we need to be careful to not choose a mutation parameter that is to big, otherwise we will have huge diversity of results. If we choose the mutation parameter as 0.8 for example this will be the result.



If we run this algorithm 100 times and mark all the calculated minima in one graph, we get:



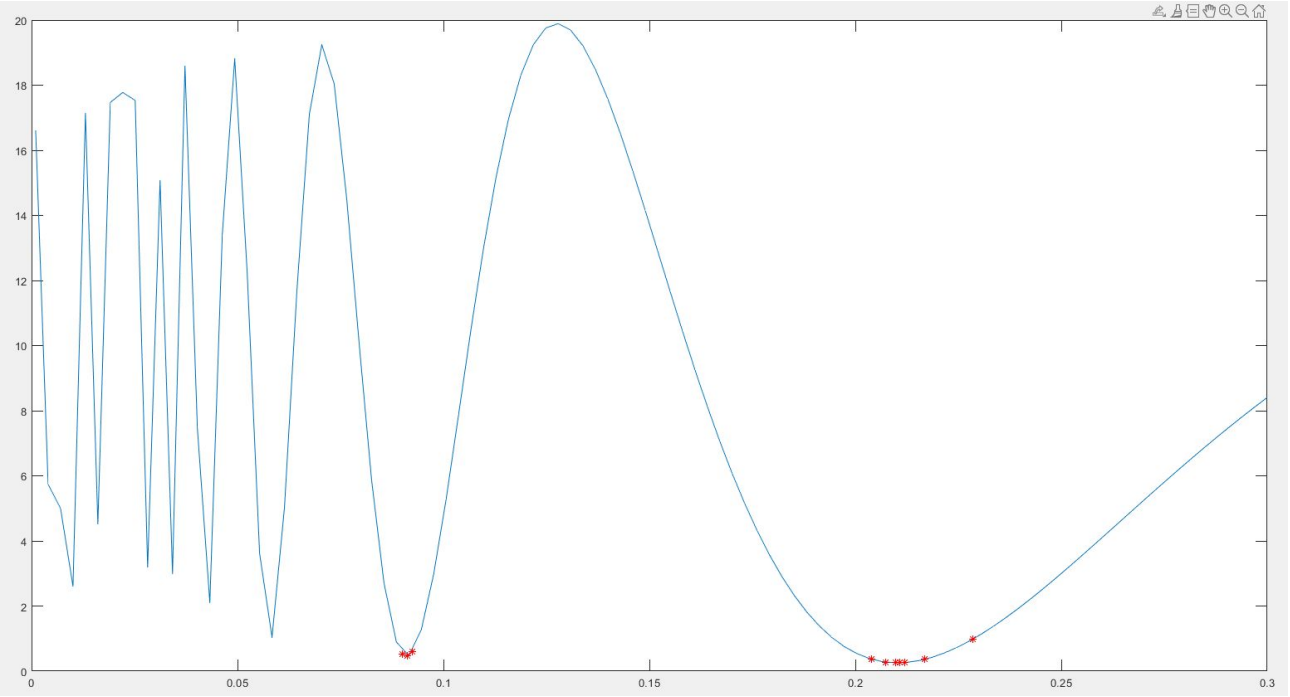
As we can see, in many cases the calculated minima is perfect, but in many cases as well it is far away from being good.

Let us see what happens if we reduce population. The problem I saw when I ran the code with half of the population, is that sometimes we land in local minimum instead of a global one. So I have created a loop to run the same code 15 times, to see how often we land in the local minimum and how often we land in the global minimum:

```
for i=1:15
   [x f] = optga('func',range,bits,pop,gens,mu,matenum);
   plot(x,f.^-1,'r*');
   hold on;
```

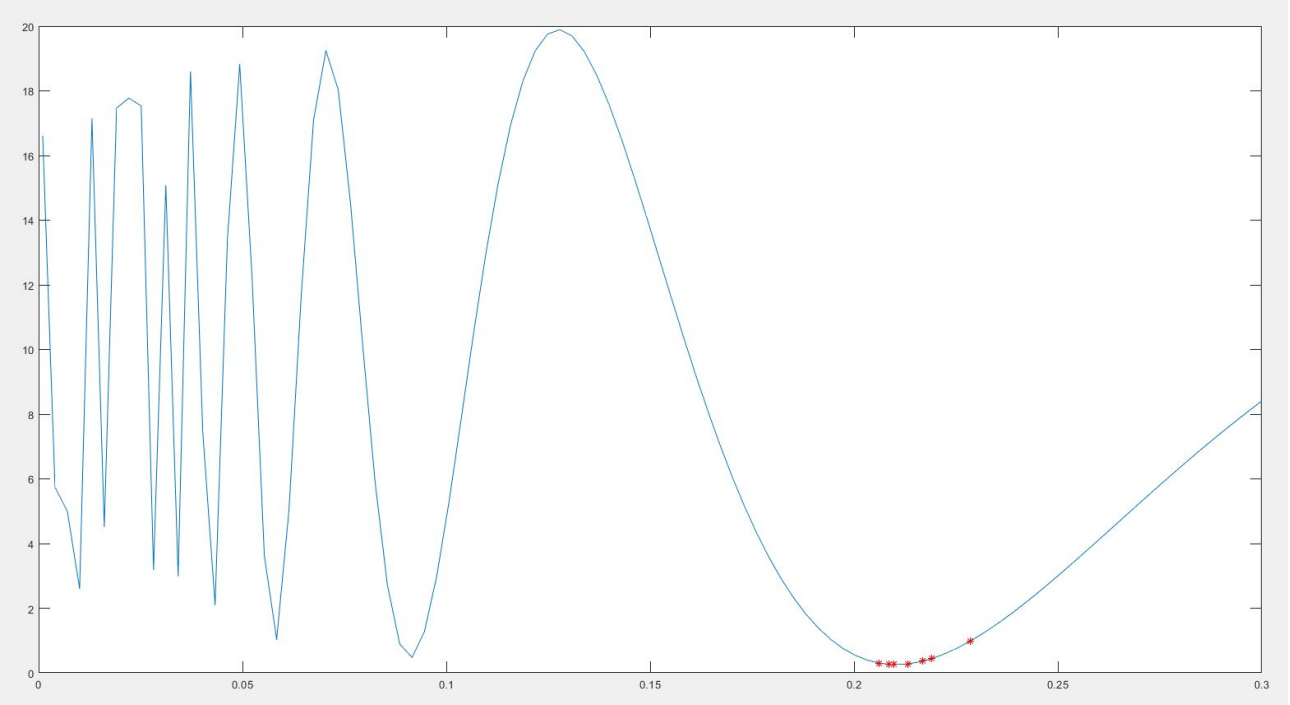
end

Output**:**



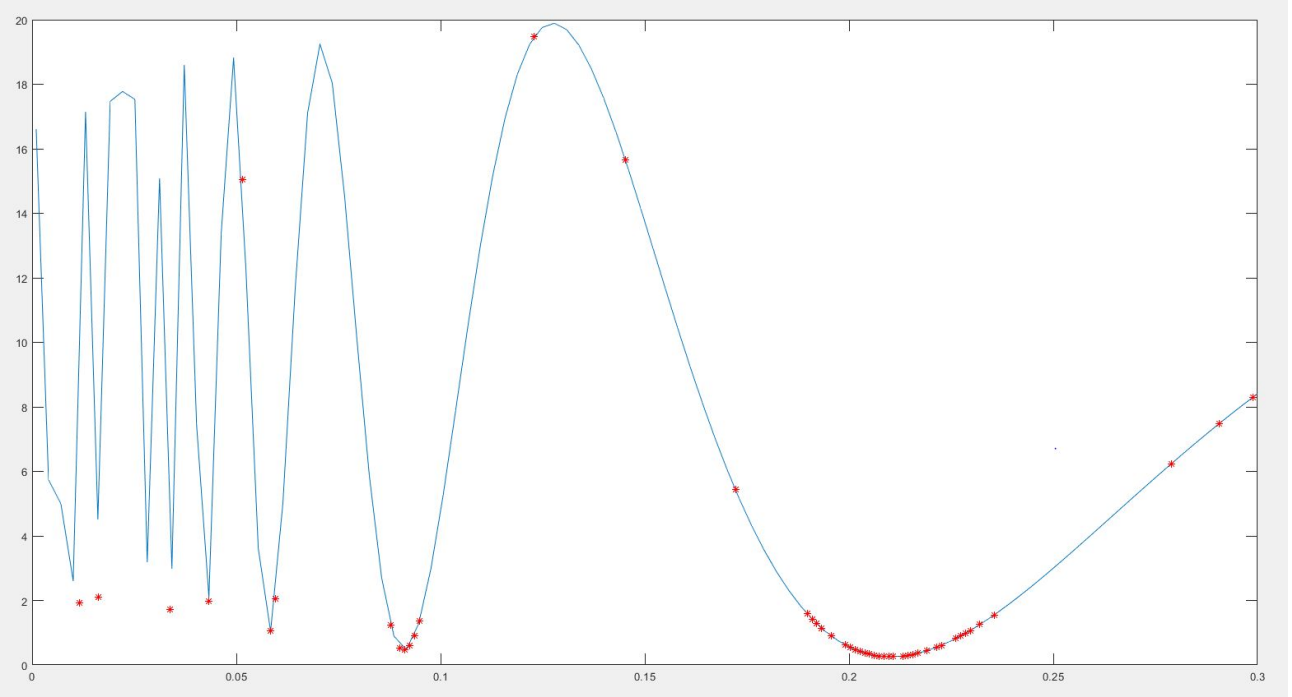
Here we can see that in 3 out of 15 cases we land in the local minimum instead of the desired global minimum but therefore we got a faster convergence.

When we doubled the population instead we only get global minima:



A similar phenomenon appears when play with the crossover degree. We will analyze with the same technique, but this time we will use 100 loops and more extreme values:

crossover degree: 0 (meaning, except fot mutation our next generation remains the same):



Something very interesting we see here, is that in the 100 times we ran the algorithm we found almost every local minima at least once. Which makes sense, since this algorithm doesn't change his population that much, meaning the final result highly depends on the initial population which is random.
For our function we still got results that are not that bad with this approach, because in our intervall a large proportion lies around the global minima.
If we would have a function where the global minima is very tight as it is in the third local minima in this function, this small crossover degree would be very bad. In this case we would need a large crossover degree or a large mutation parameter.

Program Code for Part 2:

```
range = [0.001 0.3];
bits = 8;
pop = 10;
gens = 20;
mu = 0.001; %mutation parame
matenum = 1; %crossover degree

[x f] = optga1G('func',range,bits,pop,gens,mu,matenum);

x = linspace(0.001, 0.3, 100);
fun = 10 + (((x-0.16).^2+0.1).^-1).*sin(x.^-1);
figure
plot(x, fun);
hold on

for i=1:100
   [x f] = optga('func',range,bits,pop,gens,mu,matenum);
```

```
   plot(x,f.^-1,'r*');
   hold on;
end
```

```
function [fit,fitot]=fitness(kriter,chrom,a,b)
% Fitness for the set of chromosomes in the range between a and b
% 'kriter' – the title of the external target function
[pop bitl]=size(chrom);
for k=1:pop
   v(k)=bin2real(chrom(k,:),a,b);
   fit(k)=feval(kriter,v(k)).^-1;
end
fitot=sum(fit);
```

```
function [xval,maxf]=optga(fun,range,bits,pop,gens,mu,matenum)
% GA optimization
% fun – the target function
% range[x1 x2] – the range of x: from x1 to x2
% bits – the number of bits in a word
% pop – the number of chromosomes in the population
% gens – the number of generations
% mu – the mutation parameter
newpop=[ ];
a=range(1); b=range(2);
newpop=genbin(bits,pop);
for i=1:gens
 selpop = selectga(fun,newpop,a,b);
 newgen = matesome(selpop,matenum);
 newgen1 = mutate(newgen,mu);
 newpop = newgen1;
end
[fit,fitot] = fitness(fun,newpop,a,b);
[maxf,mostfit] = max(fit);
xval = bin2real(newpop(mostfit,:),a,b);
```

```
function chromosome=genbin(bitl,numchrom)
% generation of a binary population
% word legth bitl
% size of the population numchrom
maxchros=2^bitl;
if numchrom >= maxchros
   numchrom = maxchros;
end
chromosome = round(rand(numchrom,bitl));
```

```
function newchrom=selectga(kriter,chrom,a,b)
% the title of the target function - ‘kriter’
[pop bitlength]=size(chrom);
fit=[];
% the assessment of chromosomes
[fit,fitot] = fitness(kriter,chrom,a,b);
for chromnum=1:pop
 sval(chromnum)=sum(fit(1,1:chromnum));
end
% roulette
parname=[];
for i=1:pop
 rval=floor(fitot*rand);
 if rval<sval(1)
 parname=[parname 1];
 else
 for j=1:pop-1
 sl=sval(j);
 su=sval(j)+fit(j+1);
 if (rval>=sl) & (rval<=su)
 parname=[parname j+1];
 end
 end
 end
end
newchrom(1:pop,:)=chrom(parname,:);


function deci=bin2real(chrom,a,b)
% binary chromosome -> decimal number in the range from a to b
[pop bitlength] = size(chrom);
maxchrom=2^bitlength-1;
% weights of binary digits
realel=chrom.*((2*ones(1,bitlength)).^fliplr([0:bitlength-1]));
% sumation
tot=sum(realel);
% range
deci=a+tot*(b-a)/maxchrom;



function chrom1=matesome(chrom,kiek)
% crossover procedure
% chrom – the initial set of genes
% chrom1 – genes after the crossover
% kiek – crossover degree from 0 to 1
mateind = [ ];
chrom1 = chrom;
[pop bitlength] = size(chrom);
ind = 1:pop;
```

```
u = floor(pop*kiek);
if floor(u/2) ~= u/2
 u = u-1;
end
% random crossover
while length(mateind)~=u
 i = round(rand*pop);
 if i == 0
 i = 1;
 end
 if ind(i) ~= -1
 mateind = [mateind i];
 i = -1;
 end
end
% crossover at a random point
for i = 1:2:u-1
    splitpos = floor(rand*bitlength);
 if splitpos == 0
 splitpos = 1;
 end
 i1 = mateind(i);
 i2 = mateind(i+1);
 tempgene = chrom(i1,splitpos+1:bitlength);
 chrom1(i1,splitpos+1:bitlength)=chrom(i2,splitpos+1:bitlength);
 chrom1(i2,splitpos+1:bitlength)=tempgene;
end
```