Wydział Inżynierii Mechanicznej i Robotyki

Katedra Robotyki i Mechatroniki

Signal processing and identification in control of mechatronic devices

Signal processing and identification in monitoring of mechatronic devices

Topic: Genetic algorithm

Cel ćwiczenia: Implementation of a genetic algorithm in a task of optimization of a 2D objective function

Issues covered: Genetic algorithm, mutation, selection, crossover, exploration, exploitation, convergence curves

Implementation of a simple genetic algorithm

Actually, a simple genetic algoritm is already implemented. Last time we've prepared a 1+1 solution which is essentially a genetic algorithm with 2 individuals in population, elite succession and 1-best succession. Lets develop this code so it would enable usage of a typical genetic algorithm with **n-best** succession.

To this end we'll start our basic code (random sampling) to optimize `of_2D_oneminimum_2` function and prepare new metaparameters:

```
P_size = 20;        %  Population size
n = 10;             %  Parameter n for n best succession
Step = 0.1          %  Mutation range
```

We'll initialize population before starting a *while* loop:

```
for k = 1:P_size
    Population(k).OF = Inf;
    Population(k).Parameters(1) = InitialRangeX(1) + ...
    rand()*(InitialRangeX(2) - InitialRangeX(1));
    Population(k).Parameters(2) = InitialRangeY(1) + ...
    rand()*(InitialRangeY(2) - InitialRangeY(1));
end
```

Note, that genes (that is: X and Y coordinates of each individual) and a corresponding Objective Function (OF) values will be stored now in a matlab structure. Now, we'll modify the *while* loop. To make matters clear we'll show here the whole body of a loop, so right now we can just delete everything from inside of the *while* loop and fill it step-by-step with the code shown below.

We'll start from the iteration counting:

```
iter = iter + 1;
```

In each passing of a *while* loop, instead of assessment of fitness of just one current solution, we'll assess fitness of the whole population. We'll store results in *OF* field of the *Population* structure:

```
for k = 1:P_size
    Population(k).OF = FunctionForOptimization(Population(k).Parameters);
end
```

We'll want to use n-best individuals, thus we need to sort the population with respect to *OF* field of the structure. *Indices* will preserve information about order of the individuals (that is: *Indices(1)* stores number of the best individual, and so on:

```
[~,Indices] = sortrows([Population(:).OF]');
```

In order to see the best individual, we can simply use:

```
Population(Indices(1))
```

Now we'll show current state of the population by showing the objective function plot again and adding to it all the individuals from the population:

```
if(FunctionPlot == 1)
     figure(1);
     clf
     surf(X,Y,Val,'LineStyle','none');
     view(ViewVect)
     colormap(bone)
     hold on
else
end

if(PointPlot == 1)
    for k = 1:1:P_size
        plot3([Population(k).Parameters(1)],...
        [Population(k).Parameters(2)],[Population(k).OF],'.r'); hold on
    end
end
```

We'll store the best individual (current best solution) in the same way as before, but this time we'll also store the genome of the best individual:

```
BestHistory(iter) = Population(Indices(1)).OF;
CurrentHistory(iter) = Population(Indices(floor(P_size/2))).OF;
BestIndividualGenome(iter) = Population(Indices(1));
```

And now we can start building the offspring population. Until we have any space left in the offspring population, we'll draw two random parents individual from n best individuals stored in parent population, use one as a template for the offspring individual, add one parameter from another parent, then mutate offspring individual. Note the border condition that checks if any individual was created outside of search space and in such a situation moving it to the border.

```
for k = 1:1:P_size
    ind1 = randi(n);
    ind2 = randi(n);
    NewPopulation(k) = Population(Indices(ind1));
    NewPopulation(k).Parameters(1) = Population(Indices(ind2)).Parameters(1);
    NewPopulation(k).Parameters = NewPopulation(k).Parameters + ...
    Step*randn(size(NewPopulation(k).Parameters));
    NewPopulation(k).OF = Inf;
    NewPopulation(k).Parameters = ...
    min(MaxRangeX(2),max(NewPopulation(k).Parameters,MaxRangeX(1)));

end
```

Now we can replace parent population with offspring population:

```
Population = NewPopulation;
```

And we are ready to end the loop:

```
SimTime = toc;
 clc
 fprintf('\nCurrent best: %f',BestHistory(end));
 fprintf('\nIteration: %d',iter);
 fprintf('\nTime: %d',SimTime);
 if(iter > MaxSteps)
```

```
        EndingCondition = 1;
    else
    end
    pause(Delay);
```

If the result looks as in Fig. 1, than the algorithm probably works OK. Right now we don't have proper exploration and exploitation capabilities – but we'll work on that later. We can see initial starting points for population, most of them are quickly abandoned, only few promising are tested and developed, finally population looses much of its diversity and gradually exploits minimum from one approach. Convergence speed is pretty low right now.

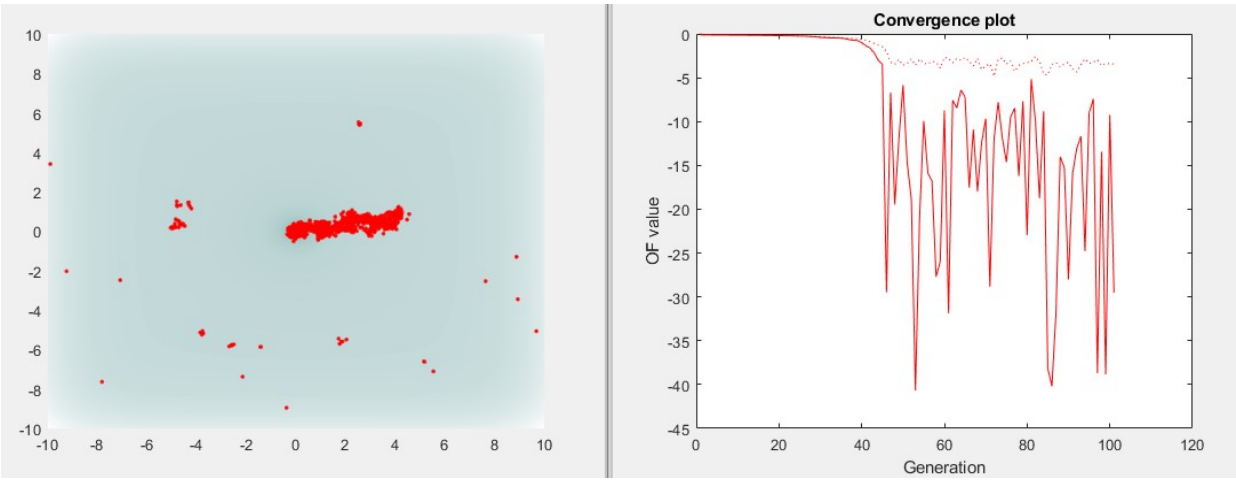


Fig 1 – example of operation of a simple genetic algorithm. "Best result" does deteriorate at least once.

Task 2.1: Please configure and test basic GA to optimize your **individual*** function of **2D, fewminima** type. Please store the code as a separate script – so it could be checked by the LA later.

Elite succession requires that the best individual (or a group of the best individuals) would be stored in the offspring population with no changes. Lets modify our code so the elite individual would be straightforwardly copied from parent population to offspring population:

```
NewPopulation(1) = Population(Indices(1));
```

Of course we'll now need to modify the beginning of the *for* loop that fills the offspring population, it should not start with index 1 but from index 2 (after all, 1st place in the offspring population is already taken by the elite individual). Implemented elite succession will allow the convergence curve to look similar to that in Fig 2.

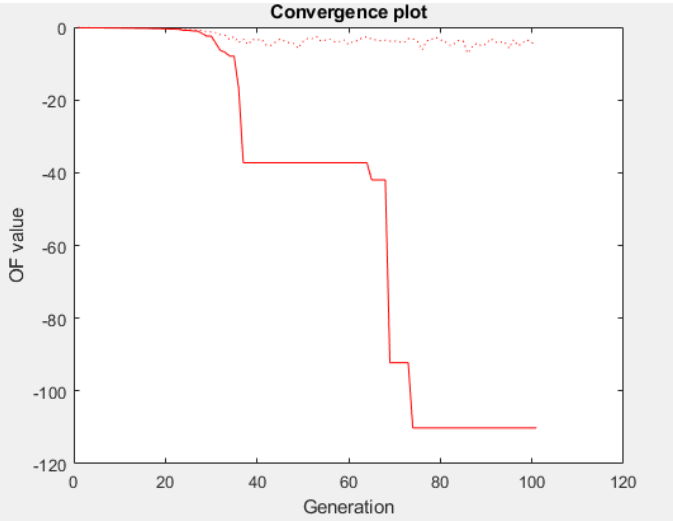


Fig 2 – Convergence curve of AG with elite succession implemented. The results do not deteriorate any more.

Note: we can compare two AGs with different configurations simply by running them one after another without closing plot window showing convergence curves. Consecutive runs will draw their CCs on previous results thus we can easily see where converence is faster or where it ends lower. In order to keep track of what is going on in the figure its good to change color of a CC after each run or after major change of metaparameters. Lack of figure closing we'll obtain by commenting out *close all* command in the code. Change of figure color is possible using `ConvergenceColor` variable.

Task 2.2: Compare operation of AG with and without elite succession in optimization of your **individual*** function of **2D <u>manyminima</u>** type. Show convergence curves from at least three runs of the GA without elite in red color and results of at least three runs of the GA with elite in blue color. Did elite presence allowed for improvement of the overall result? Did it allowed for faster convergence? Did it allowed for higher repeatability? Please store the code as a separate script – so it could be checked by the LA later.

Similarily as in the 1+1 algorithm case, exploration vs exploitation balance is crucial in configuration of GAs. Also similarily, proper values of metaparameters can be chosen on the basis of a convergence curves.

Lets now implement change of mutation step along reverse sigmoid curve. Similarily as before lets start with metaparameters:

```
InitialStep = 2;      % Exploration/exploitation balance parameters:
P1 = 2;
P2 = 10;
```

And calculate new value of a mutation step in each passing of a *while* loop as follows:

```
Step(iter) = InitialStep * (1/(1+exp((iter-(MaxSteps/P1))/P2)));
```

Of course during mutation we should no longer refer to *Step* but instead to particular value calculated for given iteration: *Step(iter)*. Finally, we can plot *Step* value as a function of iteration number:

```
figure(4);
plot(Step)
xlabel('iteration');
ylabel('mutation step value');
```

First of the parameters (InitialStep) influences initial range of mutation. It should be large enough so the population would "spread" in the whole search range but small enough so the border condition would not trigger frequently (we don't want situation when most of the individuals during first few iterations cover only border of a search space).

P1 determines x coordinate of a sigmoid curve (actually: x coordinate of a point in which second derivative of a sigmoid changes sign). Setting it at "2" (effectively: dividin population number by 2) would mean that this point is located at the middle iteration.

P2 determines how steep the sigmoid is (thus: how clear is division between exploration and exploitation phases).

These metaparameters should be set so in initial phase AG would maintain high diversity and cover all of the search space and then gradually focus on few or one selected local minimum. Final iterations should result in diversity approaching 0 (best and typical individuals on the CC should be almost identical)

Example of a CC for well-configured GA is shown in Fig. 3. Examples of bad configuration are provided in Fig. 4.

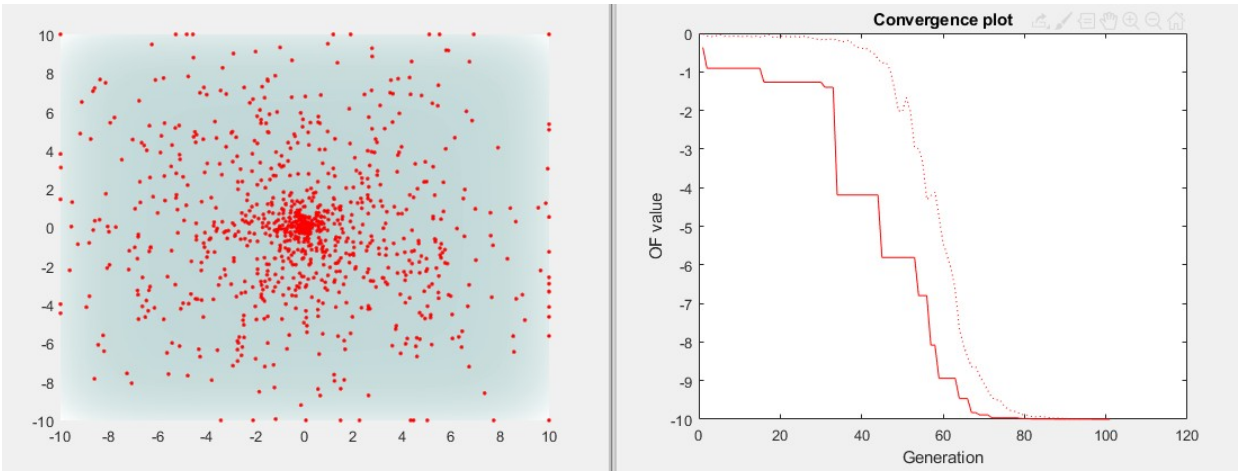
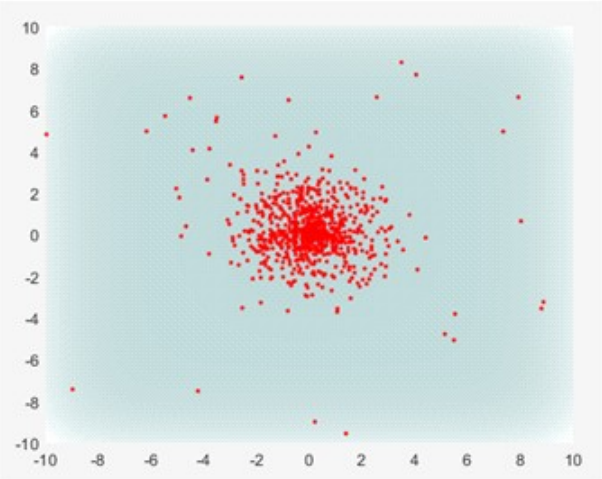
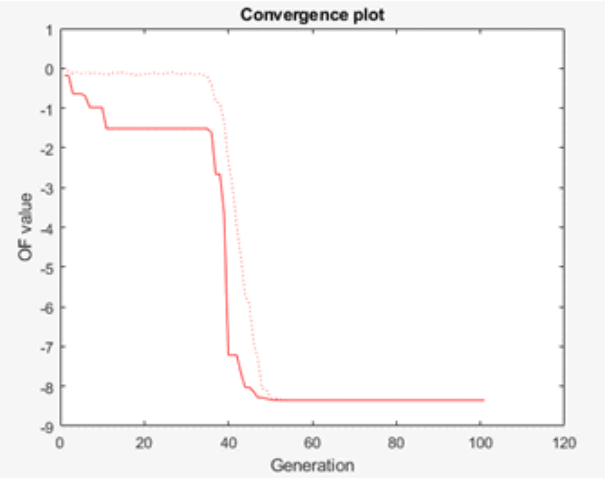


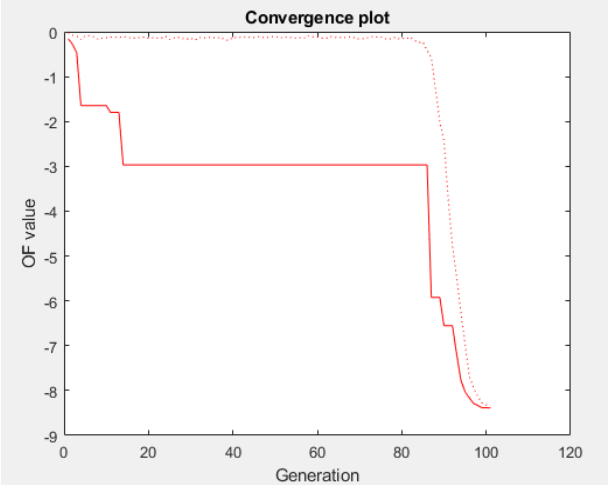
Fig 3 – Point map and convergence curve of a well-configured GA



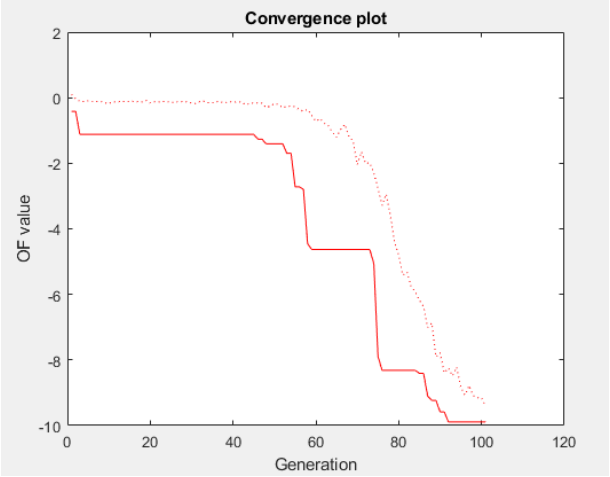
a) Poor exploration (points are not 'probing' the whole search space)



b) Too sharp change from exploration to exploitation and lots of 'dead iterations' – from the 80th iteration the algorithm is not doing anything



c) Too long exploration and too sharp change to exploitation (we are not sure if we've actually hit the bottom here)



d) Too weak exploitation – the population has very high diversity till the end. It means that we could have focused stronger on the neighborhood of the best minimum candidate

Fig 4 – Poor configuration of the GA parameters

Task 2.3: Set values of of metaparameters that govern exploration and exploitation (parameters of reverse sigmoid curve) so that you'd obtain a good-looking CC in optimization of your **individual*** task of **2D** manyminima type. Let your algorithm use roughly around 800 objective function checks. Repeat the task for **2D fewminima** function, this time use 400 objective function checs. Please store the code as a separate script – so it could be checked by the LA later.

Task 2.4: Compare:
- Multistart gradient algorithm in configuration from task 1.5
- 1+1 algorithm in configurations from tasks 1.6 and 1.7
- Genetic algorithm in configurations from task 2.3

in optimization of your **individual*** task of **2D manyminima** type and **2D fewminima** type. Calculate mean of the obtained results and standard deviation of the obtained results for 10 tries for each algorithm and each task (you may use those from instruction 1 if they were accepted by the LA). Store a representative example of convergence curve for each configuration of the algorithm and each function. You should have 6 images: 2 functions * 3 methods. Please store the code as a separate script – so it could be checked by the LA later.

In practice, configuration of the evolutionary algorithms is done usually according to the following procedure:

1) Pick a 'reasonable' set of metaparameter values, let the convergence curve look OK
2) Run the algorithm many times
3) Observe several CCs at the same time. Is the algorithm repeatable? Does it have good exploration? Is it sensitive to local minima? Does it have problems with exploitation?
4) Propose a change to parameters to address the identified problems, then return to point 2.
Repeat many times – until you see no improvement in consecutive tries.

We'll apply this process to configuration of the GA for multidimensional optimization problems. To this end we'll

1. Turn off all the visualizations except convergence curve plot and
2. Modify way of population generation. Instead of generating just two numbers for Parameters of each individual, we'll generate as many as is the number of dimensions of our problem. Note, that in mutation and in objective function value check we don't have to modify anything – the solution is already general enough and can handle any number of dimensions.

Task 2.5: Configure the GA to solve your **individual*** function of the **multidimensional** type. Use the default set of metaparameters. Use 1000 objective function checks:

```
P_size = 20;
n = 10;
InitialStep = 2;
P1 = 2;
P2 = 10;
```

Check repeatability of the GA with these meataparameters, identify possible problems, correct configuration and test it again. In case of poor repeatability test not only the change between exploration and exploitation, but also population size and selective pressure. Were you able to improve the results in a statistically-significant way? Repeat these steps several times (parameter change and test of its influence), save all the results (consecutive convergence curves and the obtained results). Be prepared to explain your actions.

Additional tasks:

Task 2.6: How non-deterministic objective function affects results provided by GA? Test your GA in optimization of a **rand** function, compare your results with well-configured 1+1 method. Check if you can improve the result by better choice of metaparameter values for GA algorithm

Task 2.7: In some cases optimization task changes during optimization process (so-called adaptive optimization problems). In our case we can try to solve this kind of problem by using

functions which name contain **_Adaptive** component. In addition to change of name, we also need to send to the function information about how many iterations are left (as a percent of time left), so instead of calling objective function like this:

```
Population(k).OF = FunctionForOptimization(Population(k).Parameters);
```

we'll do it like this:

```
Population(k).OF = FunctionForOptimization(Population(k).Parameters,iter/MaxSteps);
```

In this type of tasks it is essential to aim not for the final result of optimization (the final value of the *BestHistory*) but rather to keep the current best value as low as possible for the whole run of the code. One of the many possible modifications of the GA to work in this scenario is getting rid of the variable step and build the population using the following aproach:

1) Half of the offspring individuals is created using *small* value of mutation
2) The other half is created using *big* value of mutation.

The former half is responsible for exploitation, the latter for exploration. Your task will be to configure the method, so to check what values of *big* and *small* are the best in order to minimize the average value of *BestHistory.*

Task 2.8: Starting from the best result obtained in task 5 try improving it even further using a 1+1 solution. A starting point for your solution should be set of metaparameter values that you were not able to improve any further. Was 1+1 solution able to provide even better result? Was this improvement statistically significant?

Now change the problem to any other multidimensional function and again start from your set of metaparameter values optimized in task 5 - and improve them using 1+1 algorithm. Was the improvement better in this problem?

Task 2.9*: One of the very efficient method in evolutionary computation is a memetic algorithm. It consists of additional gradient optimization loop implemented within a main genetic loop, for the purpose of pre-optimization of each new individual. The procedure is as follows:

1) Evaluate all the individuals
2) Generate offspring population
3) For each offspring individual run *several* iterations of a gradient search. The gradient method step should be constant within this run, but depending on the current mutation step value.
4) Return to point (1)

Now configure this solution: Check how many gradient steps are necessary to provide the best performance. Then compare it with the best configuration of function from task 5 in optimization of your **individual** function of the **multidimensional** type. Is memetic algorithm statistically different? Remember, that you need to observe how many objective function checks each of the algorihms have. For instance:

10 individuals x 3 steps of gradient x (4+1) dimensions x 10 generations = 1500 objective function checks. You may increase number of available checks, but this needs to be done for both the algorithms (standard and memetic one).