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: Basic optimization algorithms

Cel ćwiczenia: Implementation of a selected basic optimization algorithms in a task of finding a minium of 2D objective function

Issues covered: Random algorithm, grid search algorithm, gradient descent algorithm, multistart gradient descent algorithm, 1+1 algorithm

Initial information
Applicable for all the laboratories supervised by PhD Eng. Ziemowit Dworakowski

A starting point for the exercises performed in this set of laboratories is a library of functions provided by the leading assistant (LA):

- *op_f_RandomSampling.m* (A function that optimizes a 2-parameter function using a random algorithm, with visualization of its operation and convergence curve)

- A family of functions located in a *FunctionsForOptimization* folder that have either one local minimum, few local minima or many local minima.

All of the programs during this set of laboratories should be prepared as separate scripts and stored for future classes - as often they will be used not only on a particular laboratory but also on future laboratories.

All of the adjustable parameters in all of the codes (e.g. all constants' values, number of iterations, step values, number of algorithms' starts, ranges for random number generations etc.) should be placed in initial part of a code and provide with clear comments.

A full theoretical background for the tasks solved during laboratories (knowledge sufficient to understand what is going on during instructions) is provided during lectures.

In the instructions there are tasks "For 3.0 mark" - marked in red, tasks "For 4.0" marked in orange, tasks "For 5.0" marked in green. Completing a set of tasks during laboratory results in a conditional mark (say, a student finished all the tasks marked in red and orange - he will receive a 4.0 mark provided that during next laboratory he will show and defend complete set of exercises including also those for 5.0.

If a student does not attend to classes once or does not finish at least "tasks for 3.0" once - this laboratory can be passed by solving all of the exercises for 3.0, 4.0 and 5.0 and one additional task (marked in blue) selected by LA - and then submit and defend a report of this scope.

In cases when there are more than 1 lack (either lack of attendance or lack of grade 3.0 or higher) student should contact with LA to get individual instruction on how to pass this part of a subject.

There will be initial tests before each laboratory. They would consist of several short questions regarding material that is covered during lectures. All of tests have to be passed (at least 3.0). For each test there will be one additional chance to obtain a pass (in case of lack of a pass there will be one more occasion to pass that material - usually during next laboratories). A second chance to pass a test will be more difficult than the initial test.

Scope of knowledge required during this exercise:

- Explaining optimization vocabulary and issues (optimization, objective function, local minima, global minimum, convergence curve, principle of operation of a random, grid search, gradient, multistart gradient and 1+1 algorithms)

Individual function

In most of the exercises you will be asked to solve problem defined by a particular objective function (OF) - usually it will be a different than functions used by your colleagues. This function will be referred to as **Individual function**. If LA will not say otherwise, its number is calculated as a sum of letters in your name and surname divided by 8. For instance *Jane Doe* will solve task no. 7

Initial task: running and testing basic code

Lets run a `of_f_randomSampling` code available in a library provided by LA (and also provided as attachment to this document). This code can be used to optimize a two-parameter objective function. This function is done here as "***nof_2D_oneminimum_2***" (objective-function (**of**), that has two parameters, i.e. is two-dimensional, (**2D**), has just one local minimum (**oneminimum)** and its number is 2.

After running the code we will see a function plot and "test points" in which a function was tested by a random algorithm for value.

Please read through the code focusing also on comments and try to understand its principle of operation. Right now a crucial issue is to get how coordinates of a test points are generated and what and why happens inside a ***while*** loop.
Successfull run of the code ends in showing a convergence curve showing a history of search for minimum: the curve consists of two components: value obtained in each iteration and historical minimum (best value obtained during a whole run of the algorithm)

Lets run and test the code for few chosen functions. Check how do various *Manyminima* and *Fewminima* functions look. Right now we won't need any funtion that has *adaptive* or *rand* in its name - let's leave them for later. We also won't need right now any function that has more than 2 parameters.

Basic optimization algorithms

Let's go back to optimization of a function with one local minimum (**'of_2D_oneminimum_2')** and based on that lets build a *grid search* algorithm.

The algorithm will need to check all the points in nodes of grid defined by values equally spaced on both parameter space axes. The best approach to do that would be to replace **while** loop with two nested **for** loops:

```
for NewX = MaxRangeX(1):step_x:MaxRangeX(2)
    for NewY = MaxRangeY(1):step_x:MaxRangeY(2)

            ...
    end
end
```

Here **step_x** and **step_y** are obviously distances between grid nodes, while size of a searched space is defined by values in **MaxRangeX** and **MaxRangeY.** Of course since we generate values of *NewX* and *NewY* in a for loop, we don't need to randomize them later.

If the obtained image looks similar to the one shown in Fig. 1, the algorithm is working well.

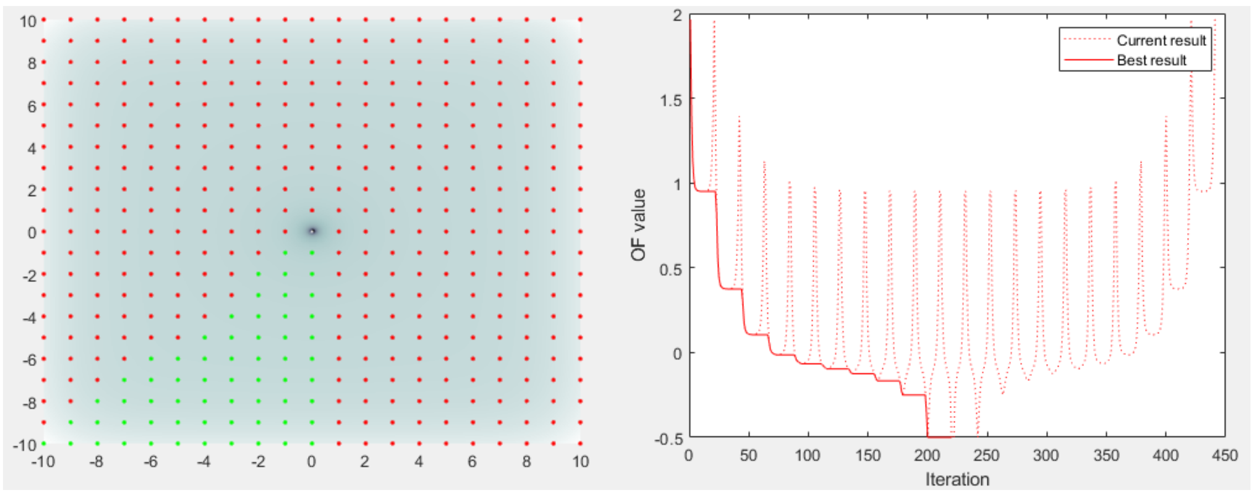


Fig 1 - example of operation of a Grid Search

Task 1.1: Let's configure and test *grid search* algorithm to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type) - after doing so please save the code as a separate script.

Next, we will implement a gradient-based optimization. Starting point would again be *of_f_randomSampling* script. This time coordinates of a next test point would not be generated randomly but instead will be picked at a direction of steepest gradient descent. Let's start optimization with a randomly chosen point, generated before start of a while loop, e.g. like this:

```
NewX = InitialRangeX(1) +  rand()*(InitialRangeX(2) - InitialRangeX(1));
NewY = InitialRangeY(1) +  rand()*(InitialRangeY(2) - InitialRangeY(1));
```

now we will go to a **while** loop and we will calculate gradient of OF around testing point and based on that select a new testing point (for next while loop iteration). In order to calculate a gradient lets calculate value of a OF in (*NewX,NewY*) point, and in points located at small (*g_step*) distance along X and Y axes:

```
CurrentValue =  FunctionForOptimization([NewX,NewY]);
CV_dx =  FunctionForOptimization([NewX+g_step,NewY]);
CV_dy =  FunctionForOptimization([NewX,NewY+g_step]);
```

Next we will choose another point for next loop passing by multiplying previous coordinates by a normalized gradient and Step constant:

```
CV = CurrentValue;  % So we'd fit on one page of the instruction ;)
NewX = NewX + Step*(CV-CV_dx)/sqrt((CV-CV_dx)^2 + (CV-CV_dy)^2)
NewY = NewY + Step*(CV-CV_dy)/sqrt((CV-CV_dx)^2 + (CV-CV_dy)^2)
```

I'd like to draw your attention to metaparameters in this code. We have here **step_g** and **Step**. The former defines how closely are located the points for gradient calculations. For our tasks usually a 0.01 would be a good choice. The latter defines how far away will each testing point be selected from its parent. Please check what would happens if Step has some different values from (0.1, 3) range. Note in particular how many steps are required to get to a minimum and how wide are the oscillations that can be observed in a minimum.

If a results look as in Fig. 2, the algorithm is working well.

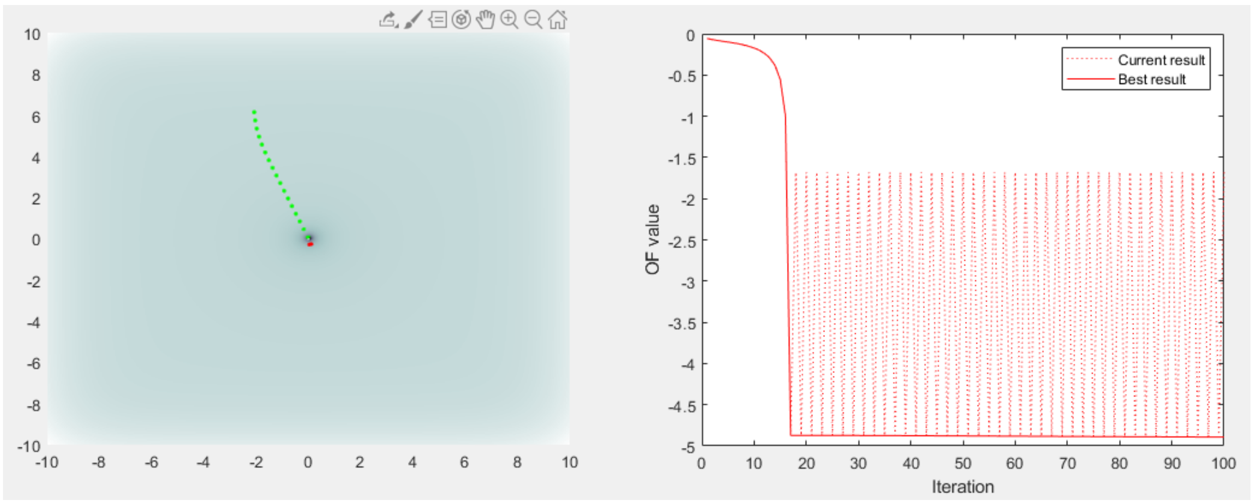


Fig 2 - Example of gradient algorithm in-operation

Task 1.2: Please prepare, configure and test gradient algorithm to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type) - after doing so please save the code as a separate script.

The last algorithm that we will implement today is a 1+1 method. Again, we will start from *of_f_randomSampling* function solving a **'of_2D_oneminimum_2'** task and again before start of a while loop we will need a randomly generated solution. We will also need a place to store currently best result corrdinates and value. Note, that we already have a suitable fragment in the code:

```
%% Storing of a best solution

    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;
```

And, similarily as in the case of basic random algorithm, we will check a value in a test point (**CurrentValue = ...**) and then check and possibly overwrite a best stored solution (a piece of code starting from **if(CurrentValue < CurrentMin)**). After that, if current solution was either preserved or discarded, we generate new point (to be tested in next iteration of a while loop) on a basis of our stored historical best solution, by adding to it a small random value:

NewX = ResultX+Step*randn();
NewY = ResultY+Step*randn();

Currently you have one metaparameter of a method: a **Step** value. Please check what would happen if its value would be changed in range from 0.1 to 4.

If the obtained results look similar to those provided in Fig. 3, the algorithm works OK.

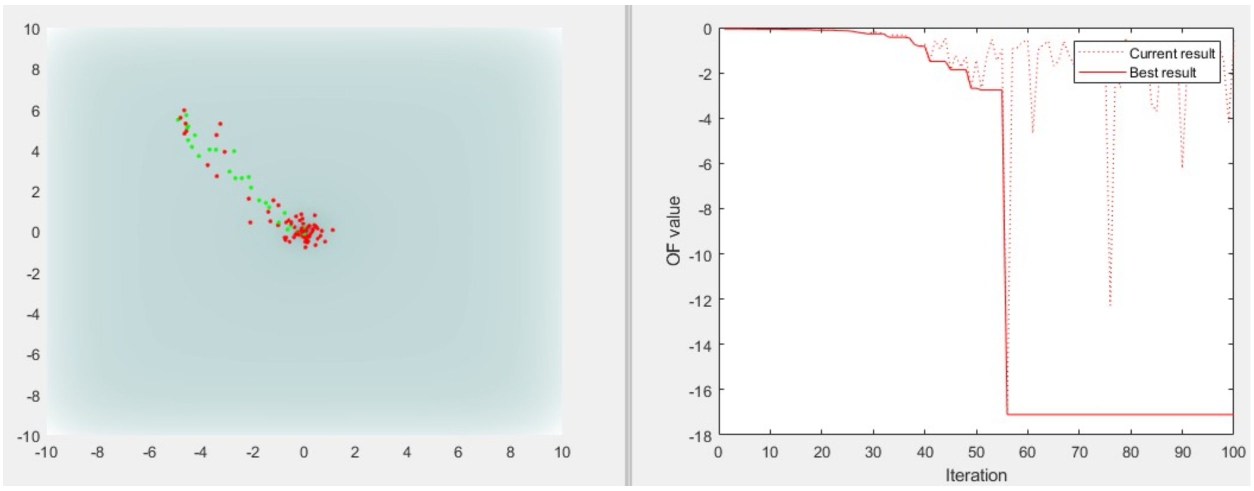


Fig 3 - Example of operation of a 1+1 method

Task 1.3: Please prepare, configure and test 1+1 to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type) - after doing so please save the code as a separate script.

Multistart algorithms

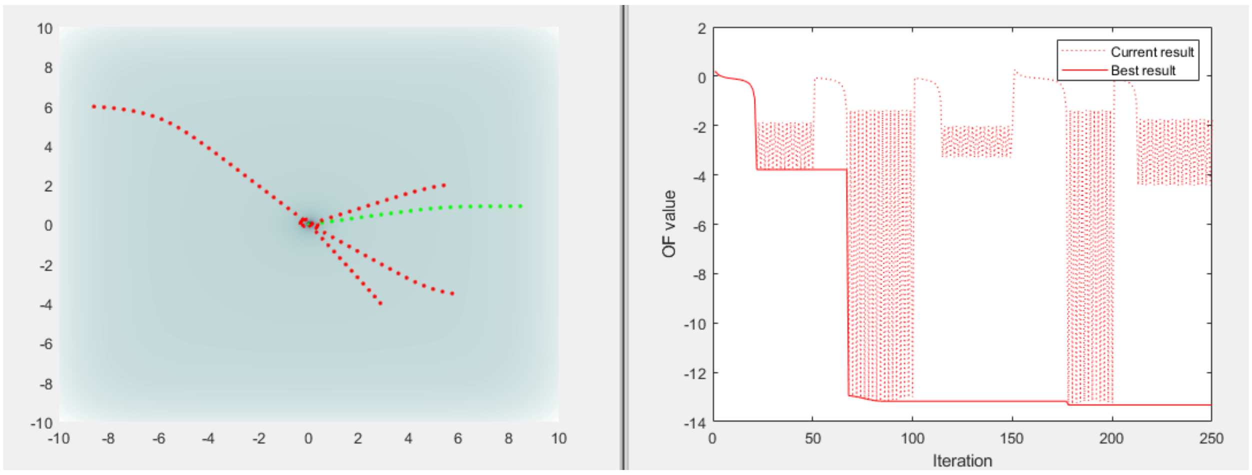
The algorithms that are sensitive to local minima require often many starts from random points, storing results of each run and, finally, picking a value found to be the best after all the starts performed. An example of the algorithm that strongly benefits from this approach is of course a gradient algorithm. We will include this functionality by storing our previous while loop inside a following part of the code:

```
for starts = 1:Starts

        ....

end
```

where **Starts** is a metaparameter defining how many starts will the method have. Note that initialization of a historical best solution should be done outside this loop (i.e. **ResultX, ResultY** and **CurrentMin** should not be initialized after each start - after all we want to save a value that was best among all tested in all starts). However, generation of a starting point should be done separately for each start. If a result looks similar to one provided in Fig. 4, the algorithm works correctly. Note, if you have trouble plotting convergence curves or if your algorithm suddenly stops after completing first start - before you had just one *iter* for both saving the convergence curves and checking stopping criterion for the method. Here you will need two: One for convergence curves (and it should keep track of total count of iterations of the method) and separate for dealing with iterations of one particular start of a method (and therefore for checking *ending condition*), and this should be reinitialized for each start.



Rys 4 - Example of operation of a multistart gradient descent algorithm

Task 1.4: Please prepare, configure and test a multistart gradient algoritm to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type). The algorithm should have 400 objective function checks (e.g. 5 starts x 26 iterations) - after doing so please save the code as a separate script.

Variable step size in optimization algorithms, exploration and exploitation

When the optimization aglorithm tests regions of OF that are far from minimum a large *Step* usually allows it to quickly correct its results. On the other hand, close to local minimum only small *Step* allows to improve the obtained result. As a result, Step parameter usually decrease in optimization algorithms with each iteration in such a way that its initial values allows are relatively big while final values approach zero. A good idea would be to use a sigmoid function to govern step value change (See Fig. 5)

In our case we could use the following piece of code inside a while loop:

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

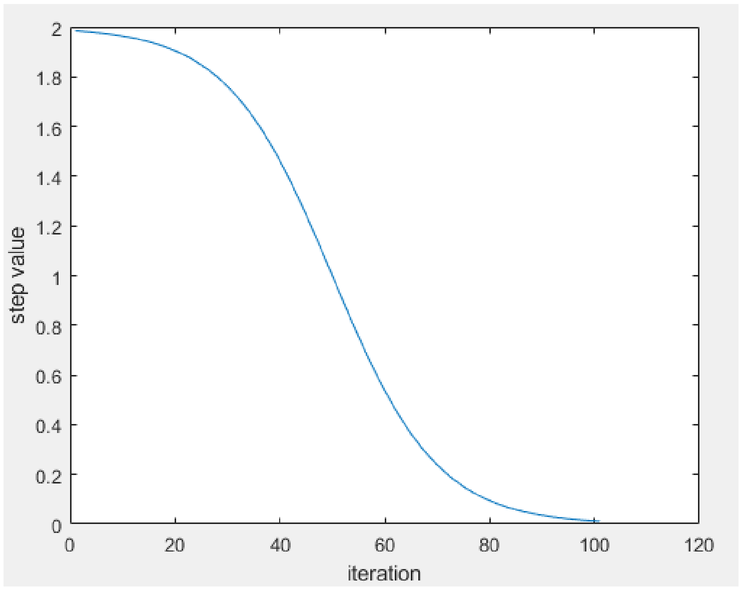
Using following metaparameters:

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

Finally, we can display the step value as in here:

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

Note! The "Step" is now a vector that stores historical values of step size. If you've used **`Step`** as a variable before in your code, now you need to use **`Step(iter)`**, so last value of step size instead.



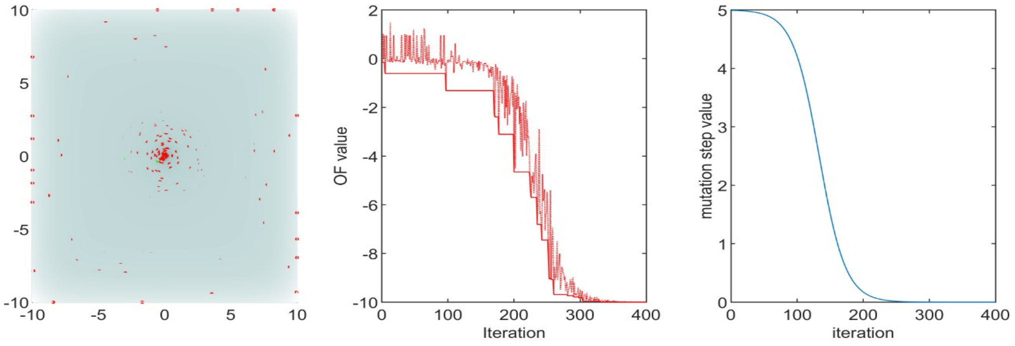
Rys 5 - Example of sigmoid curve governing size of step value for optimization algorithm

Fair and unbiased comparison of different methods

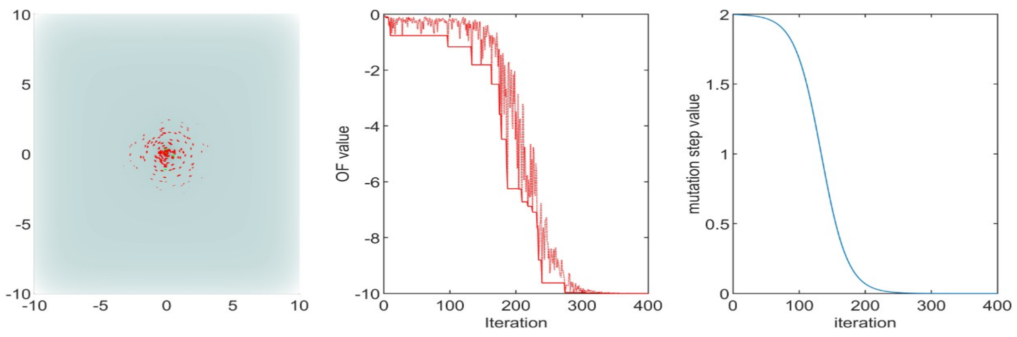
Comparison of different methods should always be fair and should not favor any of the solutions. For this reason all the optimization methods should always get the same number of objective function checks - because this is usually a limiting factor in computational terms. This means, that if gradient algorithm have 500 iterations, and multistart gradient algorithms starts 10 times, each of its starts should have only 50 iterations to be comparable to the former method. If, later, a 1+1 would be added to comparison, it should obtain 1500 iterations (provided that comparison is in 2D space: in 2D space for each step of a gradient method objective function needs to be calculated three times so for 1 iteration of gradient method 1+1 should be allowed three steps). If, later, a 20-individual genetic algorithm is used**,** it should get 75 generations (75 x 20 = 1500).

Task 1.5: Please provide your multistart gradient algorithms with variable step, configure metaparameters and test the solution in optimization of your **individual*** 2D function that has few local minima (**2D, fewminima** type). Please compare the method with basic predecessor (constant-step single-start version). The solution should be configured in such a way, so the method will use in total 400 objective function chekcs (e.g. 5 starts times 26 iterations). Please save the code as a separate script

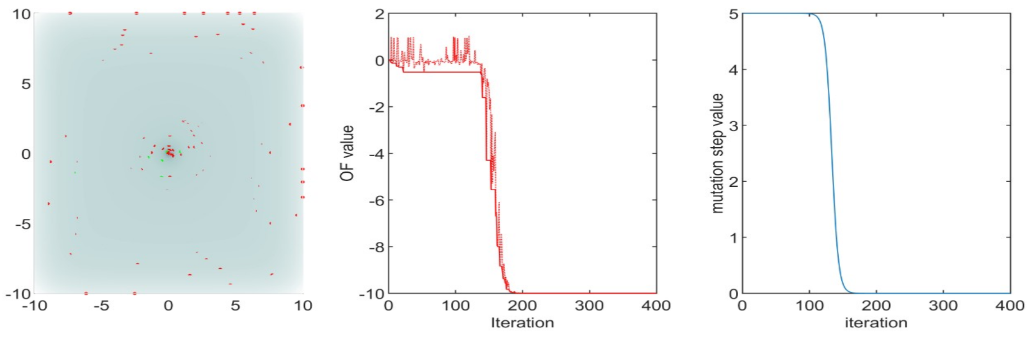
Note that multistart gradient algorithm should have step value changed in cycle, for each start (in other words: each start of a method should begin with a big step and end with a small one).



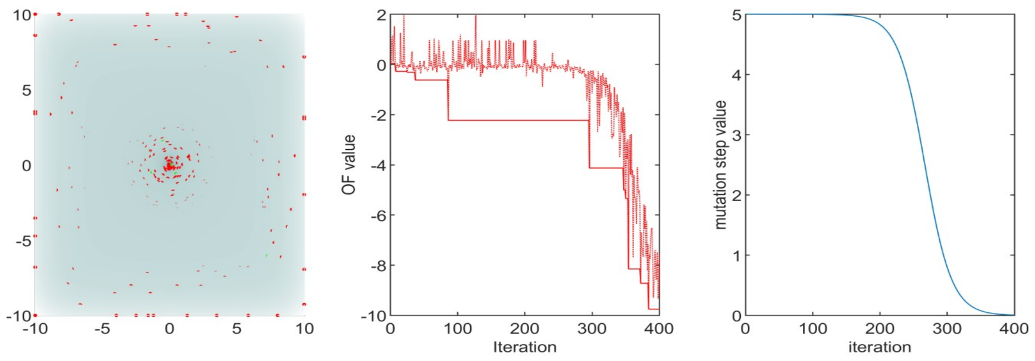
a) An example of a well-configured 1+1 solution



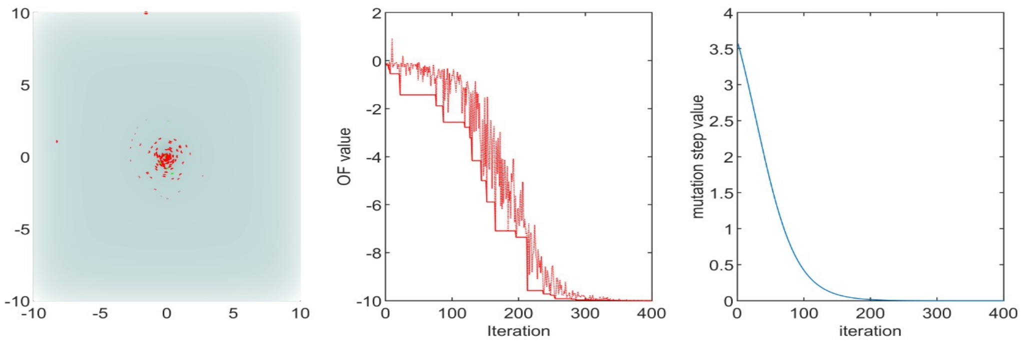
b) Initial step is too small. Exploration is nonexistent, for a manyminima function ending optimization in a local minimum is likely.



c) Too steep change from exploration to exploitation. Last 150 are "dead" - step value equal to 0 does not allow for changing of a final value . Sigmoid curve should be smoother.



d)Lack of exploitation. Until the very end result is changing rapidly and large variability of the result is present (large difference between "best" and "current" result. Sigmoid should be moved to the left or made more steep.



e) Too short exploration. The method imidiately starts exploiting a first-found local minimum and doesn't check the space for better candidates. Exploration phase should be longer (sigmoid should be moved to the right)

Fig. 6 - Examples of a good and poor configuration of a 1+1 solution

Task 1.6: Please provide your 1+1 algorithm with variable step, configure metaparameters and test it in optimization of your **individual*** 2D function that has few local minima (**2D, fewminima** type). The method should use 400 objective function checks (i.e. 400 iterations). In the initial part of its operation (exploration phase) test points should cover the whole observed area, the variability observed on convergence curve should be high. Towards the end (exploitation phase) points should be located at very small neighborhood of a minimum, while variability observed on convergence curve should approach 0. Please see Fig. 6 for reference. Please save the code as a separate script.

Task 1.7: Please configure a test your 1+1 algorithm for optimization of your **individual*** 2D function that has many local minima (**2D, manyminima** type). Use 800 checks of the objective function.

Repeatability check

It is worth remembering, that optimization algorithms usually work in a non-deterministic fashion: each run may provide a slightly different result. For this reason statistical approach to their assessment is required in order to check repeatability of a solution and likelihood of hitting a local minima. In order to get statistics from many runs fast, it is good to turn-off visualization of points and objective function. It is also good to encapsulate the main code with a for loop that runs for however-many repetitions we need for statistics. Note, that starting point generation and reinitialization of a best result should be placed inside this loop:

```
for repetition = 1:10

%% Reinitialization of a starting point and ending condition
    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;
    EndingCondition = 0;
    iter = 0;

    while(EndingCondition == 0);

       % Here we have a main optimization loop
       ...

    % Here we store best result from each run:
    Results(repetition) = BestHistory(end)
    end

end
% Here we store statistics from our run:
mean(Results)
std(Results)
```

Task 1.8: Please prepare a statistical repeatability check for algorithms configured in tasks 1.5, 1.6 and 1.7. Each of the methods should be run for at least 20 runs. Compare and discuss the results. Store the results as a matlab structure or as an excel file to be used later.

Additional tasks:

Task 1.9: Try to optimize metaparameters of the developed solutions in solving your **individual** task of a **2D fewminima** type. (Its a second-order optymalization task in which we optimize parameters of an optimization algorithm) - Can we use here grid search method? Czy do optymalizacji metaparametrów można zastosować metodę grid search?

A hint: tested algorithms are non-deterministic. It is an important factor here!

Task 1.10: Please generate a histogram of solutions for 1+1 algorithm in solving your **individual** task of a **2D manyminima** type. Try to configure a Step value on a basis of data provided by a histogram (one histogram for one decision regarding step value!)

Task 1.11: Provide statistical report on results of a well-configured 1+1 algorithm and well-configured multistart gradient algorithm in solving your **individual** task of a **2D fewminima** type. Please check the distribution of results in both cases and use this data to possibly improve metaparameter values. Finally, check whether this improvement is statistically significant.

Task 1.12: Implement a gradient descent with momentum algorithm. Please test it in solving your **individual** task of a **2D fewminima** type and also in solving **of_2D_fewminima_5** function. Does momentum help in terms of convergence speed? Does it help in accuracy?

Task 1.13: Modify a 1+1 and multistart gradient algorithms to work on 4-parameter (**4D)** functions. Note, that visualization of results will not be of much help here as we can only see a 2D cut through a parameter space. The only way of configuring metaparameters here is through the convergence curves or histograms of solutions.

Task 1.14: Please test gradient algorithm and 1+1 algorithm in a task where OF check has a noise added (the results of two consecutive tests in the same point can be different). Lets use here a **of_2D_oneminimum_1_rand** function **-** Does gradient algorithm work here? What should we do to allow it to try to compete with 1+1 method?

A hint: You already have all the metaparameters that should be changed here. Change of one of them should help.

Appendix: source codes:

of_f_randomSampling

```
% A simple random optimization algorithm. It tries new locations until it
% runs out of time. Delay serves as a way of slowing FunctionPlot.
% It requires a function for optimization (any function from folder
% "FunctionsForOptimization"

clear all
addpath FunctionsForOptimization


%% Optimization task:
FunctionForOptimization = str2func('of_2D_oneminimum_2');

%% Adjustable parameters:
MaxRangeX = [-10 10];          % Range of parameters for optimization
MaxRangeY = [-10 10];

MaxSteps = 100;     % How many iterations do we perform?
FunctionPlot = 1;   % change to 1 If you want to actually see the underlying function

ViewVect = [0,90];             % Initial viewpoint
Delay =0.001;                  % Inter-loop delay  - to slow down the visualization
FunctionPlotQuality = 0.05;    % Quality of function interpolation


%% Map initialization

close all

InitialRangeX = MaxRangeX;      % This is the range from which we can draw points.
InitialRangeY = MaxRangeY;


%% Map visualization (this code is not used for problem solving)

if(FunctionPlot == 1)
    figure(1);
        vectX = [MaxRangeX(1):FunctionPlotQuality:MaxRangeX(2)];
        vectY = [MaxRangeY(1):FunctionPlotQuality:MaxRangeY(2)];
        [X,Y] = meshgrid(vectX,vectY);
        indx = 1;  indy = 1;
        for x = vectX
            indy = 1;
            for y = vectY
                Val(indx,indy) = FunctionForOptimization([x,y]);
                indy = indy + 1;
            end
            indx = indx + 1;
        end
        mesh(X,Y,Val);
        surf(X,Y,Val,'LineStyle','none');
        view(ViewVect)

        colormap(bone)
        hold on
else end
```

```
%% Storing of a best solution

    CurrentMin = 50000;
    ResultX = 1;
    ResultY = 1;


%% The main optimization loop
    EndingCondition = 0;
    iter = 0;
    tic;

    while(EndingCondition == 0);
        iter = iter + 1;

        % Random selection of a candidate for optimum:
        NewX = InitialRangeX(1) +  rand()*(InitialRangeX(2) - InitialRangeX(1));
        NewY = InitialRangeY(1) +  rand()*(InitialRangeY(2) - InitialRangeY(1));
        % Check for constraints (they could be different than the range
        % from which we draw our solutions)
        NewX = min(MaxRangeX(2),max(NewX,MaxRangeX(1)));
        NewY = min(MaxRangeY(2),max(NewY,MaxRangeY(1)));

% If you'd like to provide function as a 2D image or use here any other objective
function, following line needs to be modified. The 0 passed to the function denotes
the fact that the function is constant in time.

        CurrentValue =  FunctionForOptimization([NewX,NewY]);

        if(CurrentValue < CurrentMin)
            CurrentMin = CurrentValue;
            ResultX = NewX;
            ResultY = NewY;
            % FunctionPlot (if we have a new minimum):
                figure(1)
                plot3(NewY, NewX, CurrentValue,'.g'); hold on

        else
            % FunctionPlot (if we don't have a new minimum):
                figure(1)
                plot3(NewY, NewX, CurrentValue,'.r'); hold on

        end

        SimTime = toc;
        clc
        fprintf('\nCurrent best:  %f',CurrentMin);
        fprintf('\nCurrent:       %f',CurrentValue);
        fprintf('\n\n\n');
        fprintf('\nIteration:     %d',iter);
        fprintf('\nTime:          %d',SimTime);

        BestHistory(iter) = CurrentMin;
        CurrentHistory(iter) = CurrentValue;

    if(iter >= MaxSteps)
        EndingCondition = 1;
    else

    end
        % If we'd like to slow down the simulation - this line is where it
        % is done
        pause(Delay);
    end
   figure(2);
    plot(CurrentHistory,':r'); hold on
    plot(BestHistory,'r'); hold on
    xlabel('Iteration');
    ylabel('OF value');
    legend('Current result','Best result');
```

Funkcje z 1 minimum lokalnym:

```
function [value] = of_2D_oneminimum_0(x,y,Time)
value = x^2+y^2;
end

function [value] = of_2D_oneminimum_1(x,y,Time)
value =  ((x)^8)/10000000 + ((y)^8)/10000000 - 1/(1+x^2) - 1/(1+y^2);
end

function [value] = of_2D_oneminimum_1_rand(x,y,Time)
value =  ((x)^8)/10000000 + ((y)^8)/10000000 - 1/(1+x^2) - 1/(1+y^2)
+0.1*randn();
end

function [value] = of_2D_oneminimum_2(x,y,Time)
value = (x^8)/100000000+(y^8)/100000000 - 1/(0.1+sqrt(4*x^2 + 4*y^2));
end

function [value] = of_2D_oneminimum_2_rand(x,y,Time)
value =- 1/sqrt(4*x^2 + 4*y^2)+0.1*randn();
end
```

Funkcje o kilku minimach lokalnych

```
function [value] = of_2D_fewminima_1(x,y,Time)
m1 = -2/(1+(sqrt(2*(x)^2 + 2*(y)^2)));
m2 = -1/(1+(sqrt(1*(x+7)^2 + 1*(y+1)^2)));
m3 = -1/(1+(sqrt(1*(x-6)^2 + 1*(y+7)^2)));
m4 = -1/(1+(sqrt(1*(x-6)^2 + 1*(y-7)^2)));

value = m1 + m2 + m3 + m4;
end

function [value] = of_2D_fewminima_2(x,y,Time)
m1 = -2/(1+(sqrt(19*(x)^2 + 12*(y)^2)));
m2 = -1/(1+(sqrt(2*(x+7)^2 + 1*(y+2)^2)));
m3 = -1/(1+(sqrt(1*(x+2)^2 + 7*(y+1)^2)));
m4 = -1/(1+(sqrt(2*(x+1)^2 + 3*(y-9)^2)));
m5 = -1/(1+(sqrt(5*(x-5)^2 + 6*(y+2)^2)));
m6 = -1/(1+(sqrt(3*(x-9)^2 + 5*(y-5)^2)));
```

```
m7 = -1/(1+(sqrt(4*(x+3)^2 + 3*(y-4)^2)));
m8 = -1/(1+(sqrt(1*(x+1)^2 + 2*(y+8)^2)));
m9 = -1/(1+(sqrt(6*(x)^2 + 3*(y+7)^2)));
m10 = -1/(1+(sqrt(2*(x-7)^2 + 12*(y+5)^2)));
m11 = -1/(1+(sqrt(10*(x+8)^2 + 1*(y-4)^2)));

value = m1 + m2 + m3 + m4 + m5 + m6 + m7+m8+m9+m10+m11;
end


function [value] = of_2D_fewminima_0_rand(x,y,Time)
m1 = -2/(1+(sqrt(2*(x)^2 + 2*(y)^2)));
m2 = -1/(1+(sqrt(1*(x+7)^2 + 1*(y+1)^2)));
m3 = -1/(1+(sqrt(1*(x-6)^2 + 1*(y+7)^2)));
m4 = -1/(1+(sqrt(1*(x-6)^2 + 1*(y-7)^2)));

value = m1 + m2 + m3 + m4+0.2*rand();
end
```

Funkcje o wielu minimach lokalnych

```
function [value] = of_2D_Manyminima_1(x,y,Time)
valx = 0.1*x.^2 + 2*sin(x) + 1*sin(4*x)+0.5*sin(9*x) + 0.2*sin(48*x) +
0.07*sin(70*x) + + 0.03*sin(543*x);
valy = 0.1*y.^2 + 2*sin(y) + 1*sin(4*y)+0.5*sin(9*y) + 0.2*sin(48*y) +
0.07*sin(70*y) + + 0.03*sin(543*y);
value = valx + valy;
end


function [value] = of_2D_manyminima_2(x,y,Time)
valx = 0.1*x.^2 + 2*sin(x) + 2.4*sin(4*x)+0.5*sin(9*x) + 1.9*sin(48*x)
+ 1.7*sin(70*x) +  1.2*sin(543*x);
valy = 0.1*y.^2 + 2*sin(y) + 2.4*sin(4*y)+0.5*sin(9*y) + 1.9*sin(48*y)
+ 1.7*sin(70*y) +  1.2*sin(543*y);
tempvalue = valx + valy;

value = tempvalue + 0.2*x.^2 + 0.2*y.^2 + 10*sin(x*0.2)*sin(y*0.4);
end

function [value] = of_2D_manyminima_3(x,y,Time)
valx = 0.1*x.^2 + 2*sin(x) + 2.4*sin(4*x)+0.5*sin(9*x) + 1.9*sin(48*x)
+ 1.7*sin(70*x) +  1.2*sin(543*x);
valy = 0.1*y.^2 + 2*sin(y) + 2.4*sin(4*y)+0.5*sin(9*y) + 1.9*sin(48*y)
+ 1.7*sin(70*y) +  1.2*sin(543*y);
value = valx + valy;
end
```

Funkcje wymagające optymalizacji adaptacyjnej (zmienne w czasie)

```
function [value] = of_2D_Adaptive1(x,y,TimePercent)
ind = 1;

Function1 = sin(0.3*(x+1))+sin(0.028*(x+5)) - cos(0.5*(y+2))-
cos(0.4*(y+15));
Function2 = sin(0.3*(x+7))+sin(0.028*(x-5)) - cos(0.5*(y+8))-
cos(0.4*(y+1));
Function3 = sin(0.3*(x-8))+sin(0.028*(x-5)) - cos(0.5*(y+12))-
cos(0.4*(y+3));
Function4 = sin(0.3*(x+9))+sin(0.028*(x+7)) - cos(0.5*(y-7))-
cos(0.4*(y+5));
Function5 = sin(0.3*(x+3))+sin(0.028*(x+2)) - cos(0.5*(y-3))-
cos(0.4*(y-5));

if((TimePercent >= 0 && TimePercent < 0.1) || (TimePercent >= 0.5 &&
TimePercent < 0.6))
value = Function1;
else end

if((TimePercent >= 0.1 && TimePercent < 0.2) || (TimePercent >= 0.6 &&
TimePercent < 0.7))
value = Function2;
else end

if((TimePercent >= 0.2 && TimePercent < 0.3) || (TimePercent >= 0.7 &&
TimePercent < 0.8))
value = Function3;
else end

if((TimePercent >= 0.3 && TimePercent < 0.4) || (TimePercent >= 0.8 &&
TimePercent < 0.9))
value = Function4;
else end

if((TimePercent >= 0.4 && TimePercent < 0.5) || (TimePercent >= 0.9 &&
TimePercent <= 1))
value = Function5;
else end
end



function [value] = of_2D_Adaptive2(x,y,TimePercent)
Function1 = sin(0.3*x)+sin(0.028*(x+5)) - cos(0.5*(y+2))-
cos(0.4*(y+5));
Function2 = -cos(0.3*x)-sin(x)*cos(0.29*(x+5)) -
sin(x)*cos(0.9*(y+2))-cos(0.89*(y-7));

value = Function1*(1-TimePercent) + Function2*TimePercent;
```

```
end
```