Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Basics of AI and Deep Learning

Course for Mechatronic Engineering with English as instruction language

Instruction 1:

<u>Course introduction</u>
and
<u>Basic Optimization methods</u>

You will learn: How to implement from scratch and configure basic optimization algorithms: 1+1, gradient and grid search. We will cover metaparameter setup including variable step changes and number of starts for multistart gradient method. These algorithms will serve as a cornerstone for further, more advanced solutions that will be implemented later this semester (including neural networks)

Additional materials:

- Course lecture 1 [*obligatory*] [link]
- [Additional supporting materials for further study]

Learning outcomes supported by this instruction:
[Here a list of learning outcomes' codes]

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Ziemowit Dworakowski, zdw@agh.edu.pl

0. Initial information regarding course laboratories

Laboratory structure

The laboratories are prepared using a *Reversed Classroom* method. It means that the intended way of doing them is to first do part of them at home and then finalize work during classes. The more you do at home, the more knowledge you'll gain and (hopefully) the easier the entire course will be for you.

The general flow of each instruction includes explanation of steps that should be taken to solve a problem [in white background] and then a series of tasks to do by you. These tasks are divided into three categories: "For 3.0 mark" - marked in red, "For 4.0" marked in orange, and "For 5.0" marked in green. Completing a set of tasks during laboratory results in a conditional mark (say, you finished all the tasks marked in red and orange - you will receive a 4.0 mark provided that during next laboratory you will show and defend complete set of exercises including also those for 5.0. So in other words: the more tasks you do during classes, the higher your grade will be – but you will have to do all of them either way.

Colors represent also intended level of difficulty of the tasks. The red ones tend to be simple and doable by just following initial instructions. Orange ones usually require some coding or decision-making while green ones often require solving actual problems and often require feedback from teacher. Because of that the recommended approach is to finish red ones at home prior to classes and try to solve orange ones, leaving greens for the actual laboratory.

Correcting absence and laboratory fails

The intended way of getting a laboratory pass can be disturbed in three ways:

- You can be absent (then you won't have a chance to get a 'conditional grade')
- You can fail to do tasks "for 3.0" during the laboratory
- You can fail to defend the conditional grade during the next laboratory

In all of these cases you will be required to prepare a laboratory report including all the necessary tasks (required for 3.0, 4.0 and 5.0) plus an additional one selected by the teacher from the additional tasks pool available at the end of each laboratory. You will be required to defend this report. In order to make report preparation easier, we've prepared a set of instructions regarding report requirements which is available here [link]

Note, that you can pass the laboratory in such way only three times during the entire course. If you have more laboratories to correct, you will need consultations with the course coordinator who will individually assess the situation and will either determine the scope of work required to get a pass or direct you for taking the course again.

Tests and test corrections

Some laboratories will start with a lecture test. The schedule for tests is available in notes from the first lecture. For each test there will be two correction chances at the end of the course. If, however, you fail more than two tests during the course, the correction will involve entire lecture material.

Introduction to optimization tasks

A starting point for the exercises performed in this set of laboratories is a library of functions provided by the teacher:

- *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.

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 remainder of a sum of letters in your name and surname divided by 8. For instance *Jane Doe* will solve task no. 7 and *John Smith* will solve the task no. 1.

Initial task: running and testing basic code

Let's run a *of_f_randomSampling* code available in a course library. This code can be used to optimize a two-parameter objective function. This function is named here as "***of_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.
Successful 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 = Range(1,1):step_x:Range(1,2)
        for NewY = Range(2,1):step_y:Range(2,2)

        iter = iter + 1;
        Point = [NewX,NewY];

          ... % Here comes the rest of what was previously inside while loop
    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 x and y parameter of *Point* 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.

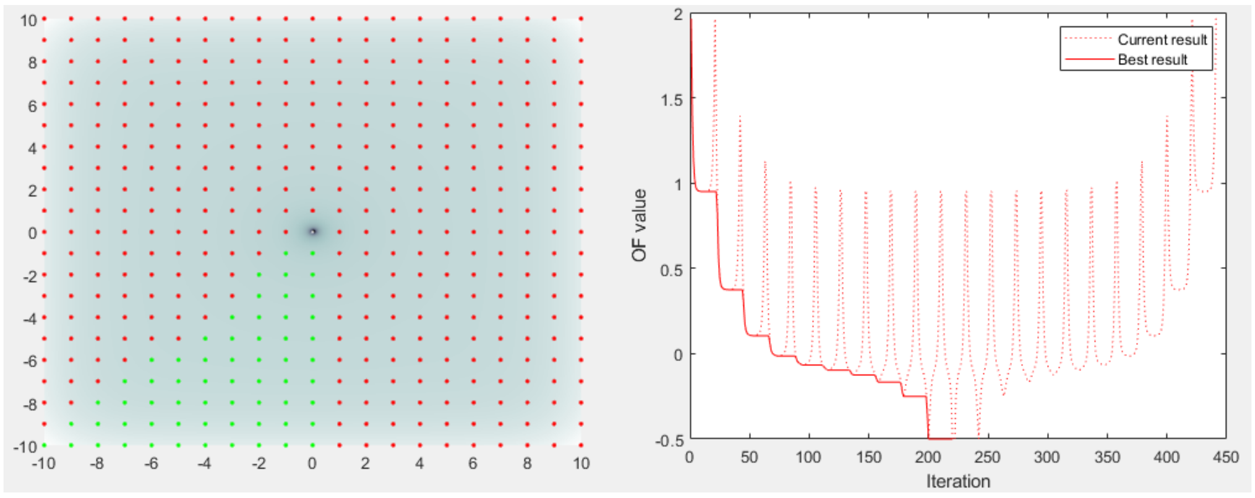


Fig 1 - Example of operation of a Grid Search method

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). Run the method 3 times, filling results in Table 1. 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:

```
Point = Range(:,1)' + rand(1,dimensions).*(Range(:,2)-Range(:,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 our *Point*, and in points located at small (*g_step*) distances along all the axes. Note the *dimensions* metaparameter which governs in how many directions we want to calculate our partial derivatives of the OF. In our case *dimensions* should be set to 2.

```
CurrentValue = FunctionForOptimization(Point);
for d = 1:dimensions   % "dimensions" says how long is our parameter vector
    TestPoint = Point;
    TestPoint(d) = TestPoint(d) + g_step;
    CV_d(d) = FunctionForOptimization(TestPoint);
end

Grad = CV_d - CurrentValue;

% If we want to use just direction of the gradient - this is the way to go:
if(max(abs(Grad))==0)
 Grad = 0
else
 Grad = Grad/max(abs(Grad));
end
```

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

```
Point = Point - Step*Grad;
```

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 0.001 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.

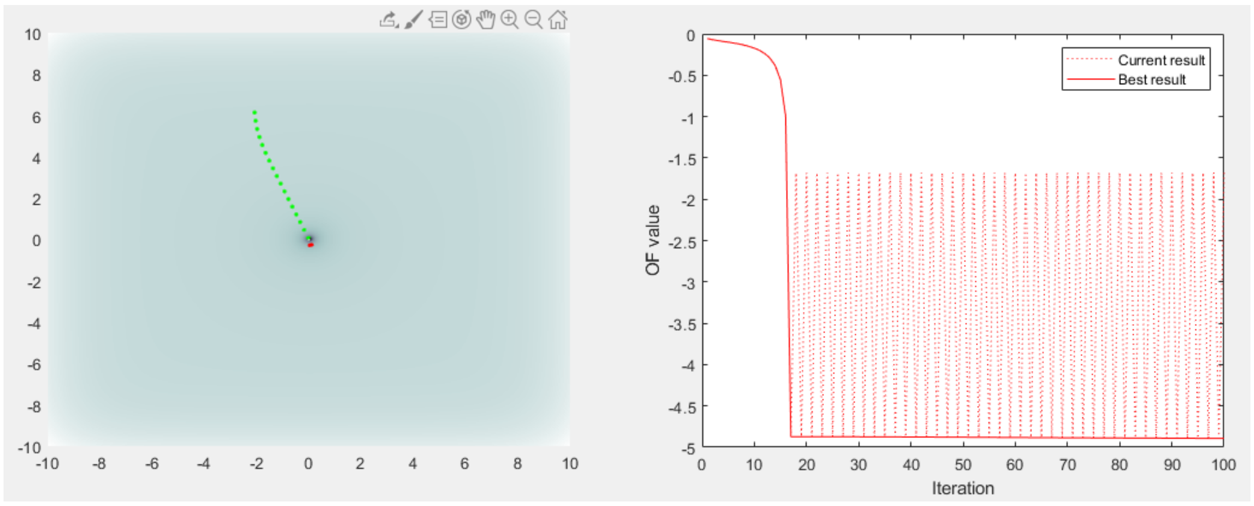


Fig 2 - Example of gradient algorithm in-operation

Task 1.2: Please prepare, configure (by picking a reasonable Step size) and test gradient algorithm to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type). Run the method 3 times, filling results in Table 1. 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:

```
    CurrentMin = Inf;
    Result = [10,10];
```

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:

```
   Point = Result + Step*randn(size(Point));
```

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.

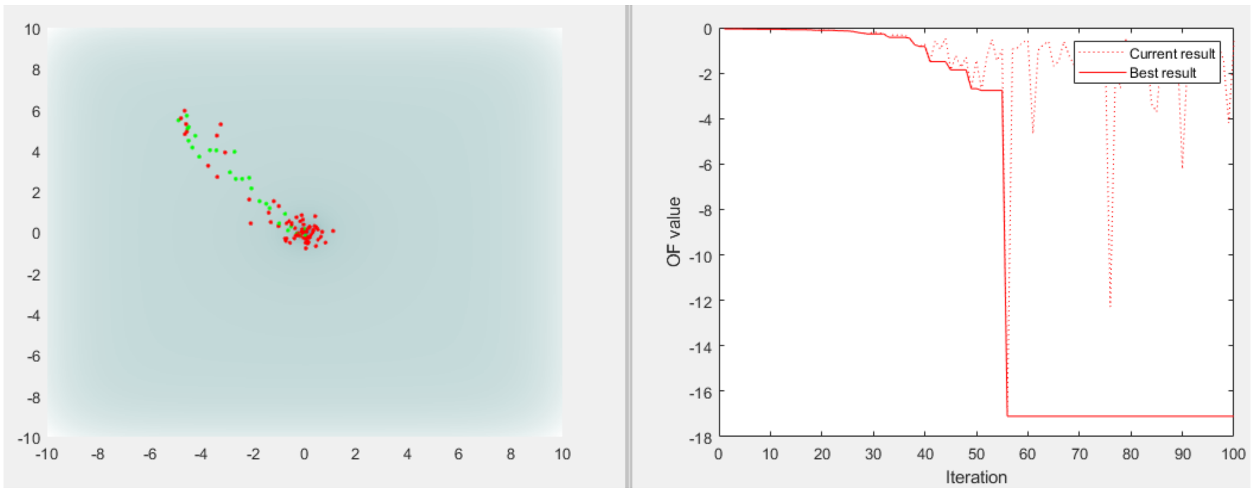


Fig 3 - Example of operation of a 1+1 method

Task 1.3: Please prepare, configure (by picking a reasonable Step size) and test 1+1 to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type). Run the method 3 times, filling results in Table 1. After doing so please save the code as a separate script.

Multistart gradient algorithm

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 of a gradient algorithm 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. **Result** 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, setting an initial step and resetting *EndingCondition* 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 one (e.g. *iter2*) for dealing with iterations of one particular start of a method (and therefore for checking *EndingCondition*), and this should be reinitialized for each start.

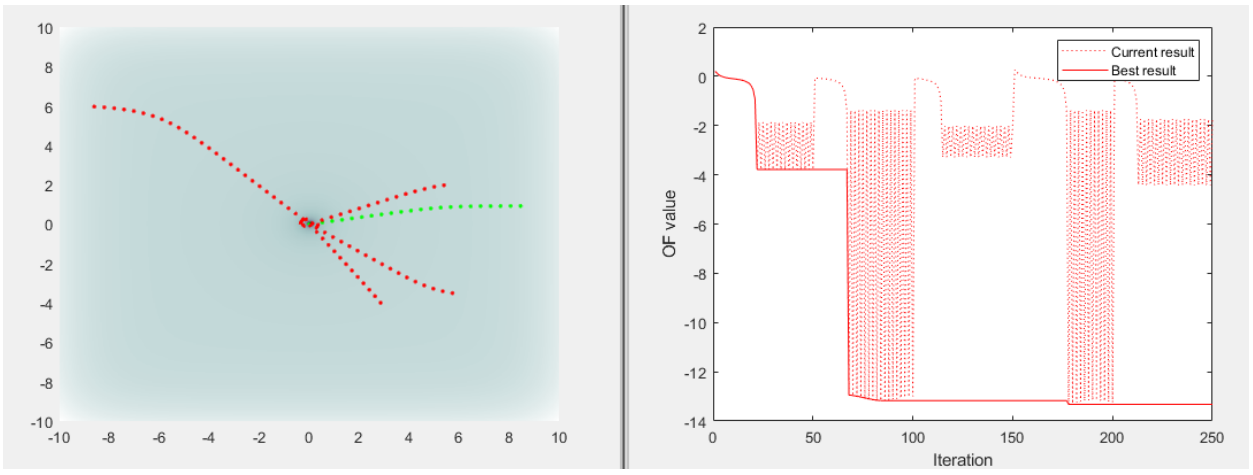


Fig 4 - Example of operation of a multistart gradient descent algorithm

Task 1.4: Please prepare, configure (by picking a reasonable Step size) and test a multistart gradient algoritm to optimize your **individual*** 2D function that has few local minima (**2D, fewminima** type). The algorithm should have roughly 400 objective function checks (e.g. 5 starts x 26 iterations). Run the method 3 times, filling results in Table 1. After doing so please save the code as a separate script.

Note: 5 starts x 26 iterations actually makes 390 objective function checks. Can you explain why?

Adaptive step size in optimization algorithms

If an optimization step is too large, the algorithm tends to improve very slowly or not improve at all. For this reason it is good to reduce the step size if we see no improvement over a number of generations. Lets count failed iterations of our gradient solution:

```
if(CurrentValue < CurrentMin)
     % ...
     % Here is our piece of code for storing best results, etc…
     % ...
     NoImprove = 0;
else
     % FunctionPlot (red, if we don't have a new minimum):
     if(PointPlot == 1)
          figure(1);   plot3(Point(1),Point(2), CurrentValue,'.r'); hold on
     end
     NoImprove = NoImprove + 1;

end
```

We of course need to initialize our *NoImprove* counter before the start of the loop. But now we can also reduce Step value if we see that we failed e.g. 3 times in a row:

```
        if(NoImprove > 2)  % The '2' is the adaptation trigger here
            NoImprove = 0;
            Step = Step * 0.6;  % The '6' is the reduction factor here
        end
```

Notice the highlighted values. For now we just leave them in the code like that – but they are metaparameters and we'll need to address their values later. It would also be good to track what actually happens to our Step value:

```
StepHistory(iter) = Step;
```

And such history we can later display and see how algorithm adapted to the optimized function:

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

After running this code you should see something like in Fig. 5. Note that by having adaptive step we can easily start from a higher initial step than before and we won't hit problem with oscillations close to local minimum!

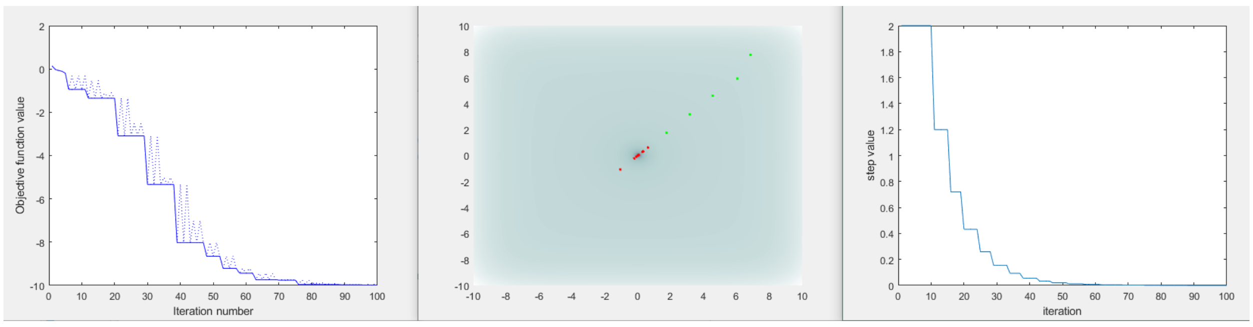


Fig 5 – Convergence curve, point plot and history of changes of step size

Task 1.5: Please provide your multistart gradient algorithm and vanilla gradient algorithm with adaptive step, and test them in optimization of your **individual*** 2D function that has few local minima (**2D, fewminima** type). Test the solutions 3 times each and fill respective rows the Table 1. Please save the code as a separate script.

After that, prepare the adaptive solution for 1+1 method as well – and check its performance

Note 1: If you see that your 1+1 solution "stops too early", don't panic. It is expected at this stage

Note 2: If you see that your gradient solution does not reduce step size as you expected it to, don't panic either – but try to explain why it actually happens and what could be done to prevent it?

Configuration of metaparameters for adaptation

While doing task 1.5 you probably encountered a significant problems with both methods: the optimization of 1+1 stopped too early and in gradient-based solution it did not actually reduce step enough. It is caused by the fact, that "fails" should be treated differently depending on what algorithm we are using and what optimization problem it solves. Remember the metaparameters highlighted in previous code? We assumed that 3 fails is enough to trigger step reduction – but gradient solution is expected to improve in each and every iteration while 1+1 solution is expected to have 50% success rate in the best case scenario. The more dimensions we have and the closer we are to the minimum, the less likely we are to improve from iteration to iteration. For this reason the trigger needs to be higher for 1+1 solution and lower for gradient solution. If I set trigger to 9 for 1+1, the method is consistently hitting optimum for my function (See Fig 6). If I set trigger to > 0 for gradient-based solution, it should also work consistently better.

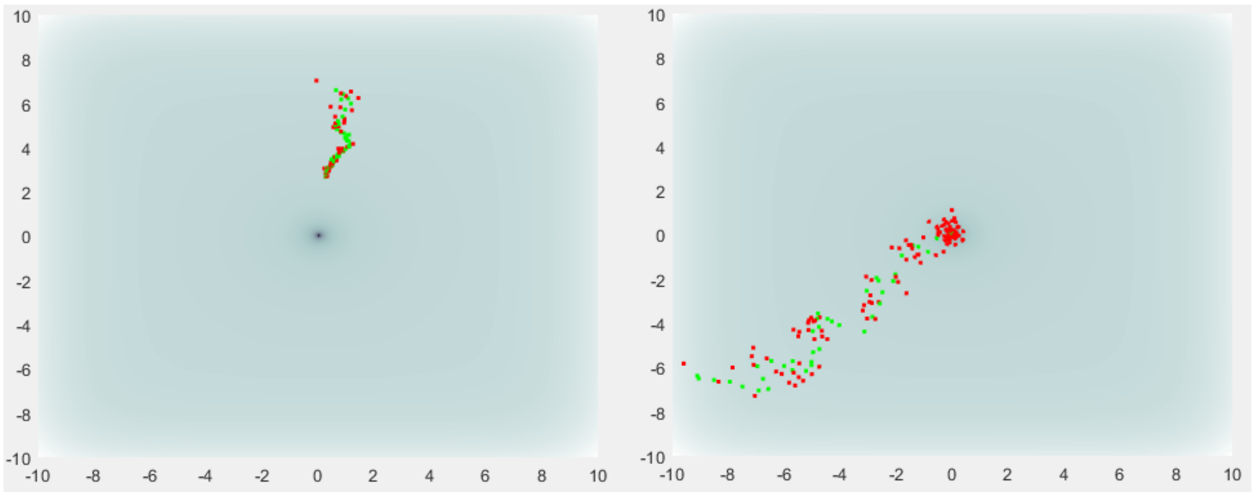


Fig 6 – Results of too small trigger for step reduction (left, NoImprove > 2) and proper setup (right, NoImprove > 8)

Another problem, that we need to address is the reduction coefficient. This again will need to be setup for particular method and particular problem. The goal is to reduce the step slowly enough to allow for gradual transition from exploration to exploitation, but quickly enough to allow for good exploitation at the end. Note that we can also work with initial step values – determining from what level we start our adaptation.

The goal to which we aim should look as in Fig. 7: Step is large enough to allow for good exploration (points initially spread in the whole search range), transition from exploration to exploitation is gradual and smooth, step and thus solutions' variability approaches 0 at the end.

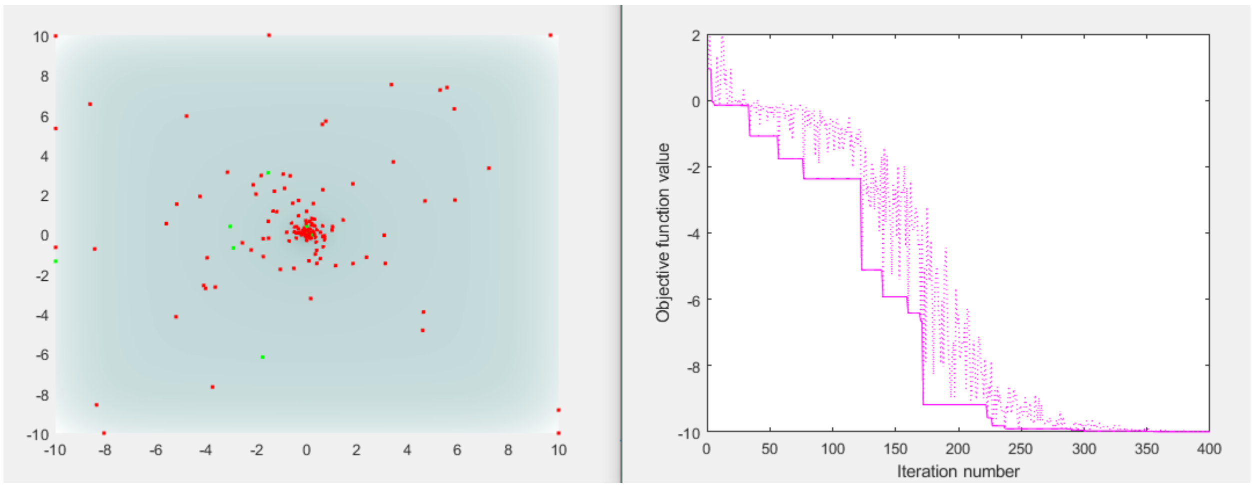
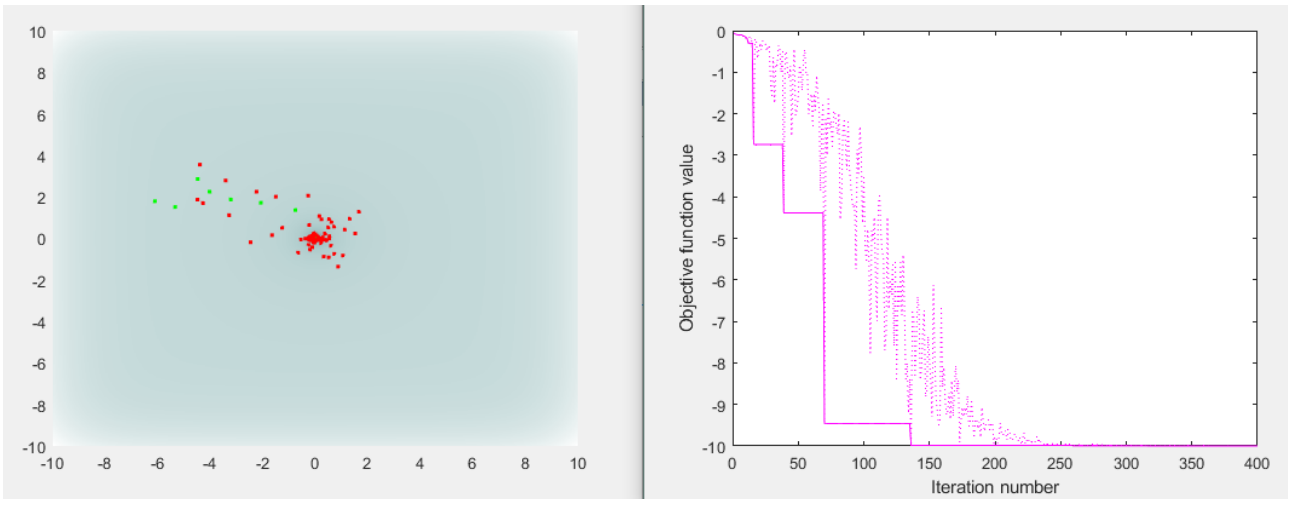


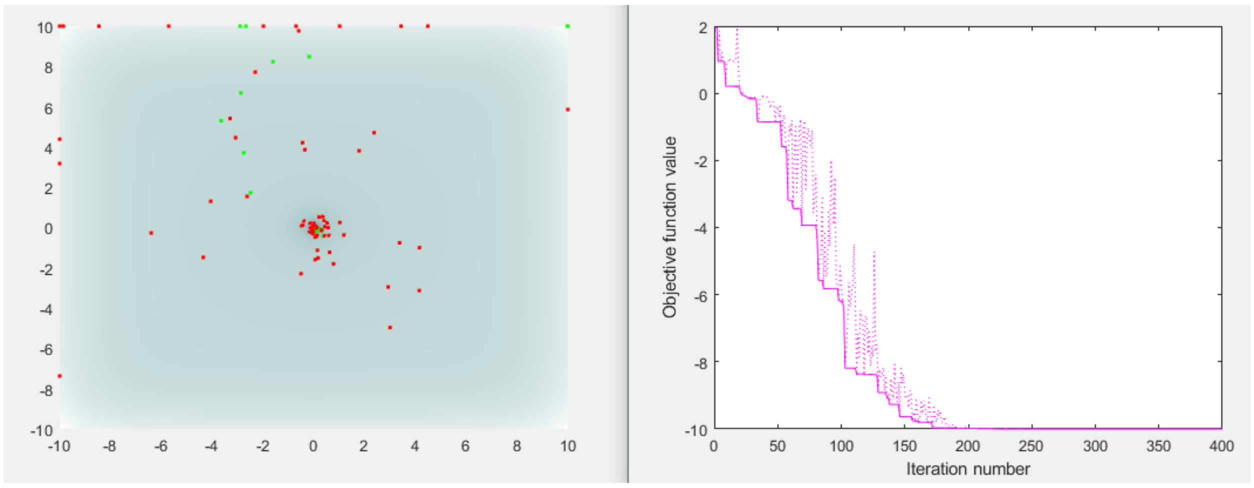
Fig 7 – Good configuration of a 1+1 solution

Examples of problems we can run into are shown in Figure 8.

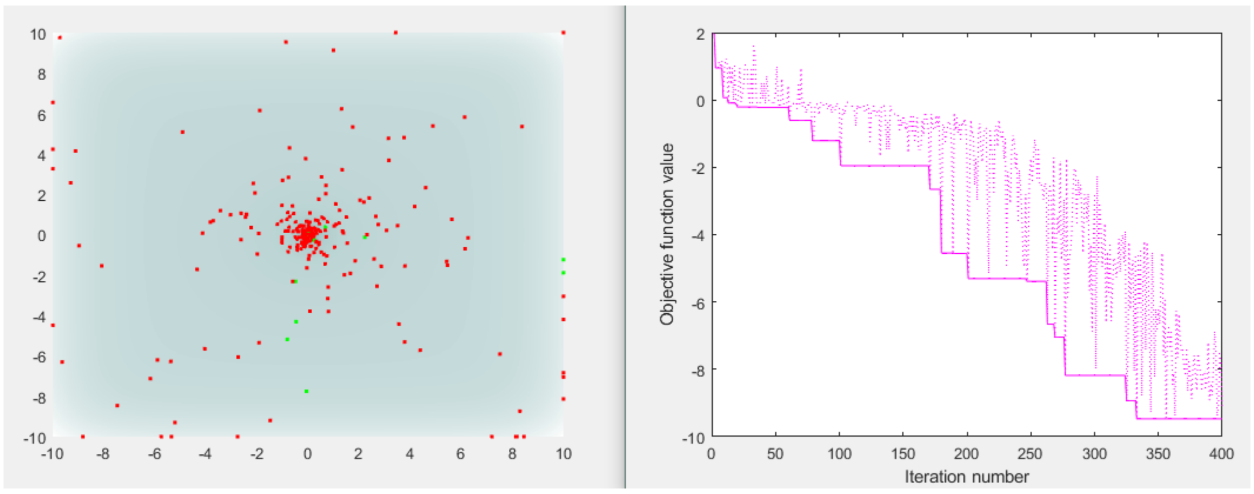
Task 1.6: Please provide your 1+1 algorithm with adaptive step, configure metaparameters (Initial Step, Adaptation trigger and Reduction factor) for optimization of your **individual*** 2D function that has few local minima (**2D, fewminima** type). The method should use 400 objective function checks (i.e. 400 iterations). Consult Fig 8 for common mistakes. Run the configured solution 3 times and save the results in Table 1. Please save the code as a separate script.



(a) – Too small initial step – only a small part of search range is tested.



(b) – Too high reduction factor – the transition from exploration to exploitation is too sharp, there are 'dead iterations' at the end (the algorithm is not doing anything after 200 iteration)



(c) – Too low reduction factor – the transition from exploration to exploitation is too slow, there algorithm is not exploiting the minimum well (the variability at the end is too high)

Fig 8 – Examples of mistakes in configuration of a 1+1 solution

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).

Repeatability check

Optimization algorithms usually work in a non-deterministic fashion: each run may provide a slightly different result. For this reason statistical check of method's repeatability and likelihood of hitting a local minima is needed. 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. 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 = Inf;
    Result = [0,0];
    EndingCondition = 0;
    iter = 0;

    while(EndingCondition == 0);
       % Here we have a main optimization loop
       ...


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

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

Task 1.7: Please configure and test your 1+1 algorithm with adaptive step, single-start and multistart gradient algorithms with adaptive step for optimization of your **individual*** 2D function that has **<u>many</u>** local minima (**2D, manyminima** type). Configure adaptation parameters in such a way as to allow the methods to consistently exploit the minimum it was attracted to. Use 800 checks of the objective function for both methods. Test the methods statistically by averaging 20 runs of each method, store the means in table 1. Save all the codes for further use

Multidimensional extension

In practical situations we rarely optimize problems that have just 2 parameters to set-up. Now it is time to extend our solutions to multidimensional problems. Fortunately, the methods are mostly prepared for multidimensional problems already. You just need to turn off map visualization (*FunctionPlot = 0, PointPlot = 0*) as showing just a slice of our multidimensional space does not make much sense, adjust *dimensions* number and *Range* matrix accordingly and generate starting point for your method of a proper size.

Task 1.8: Please configure and test your:
- 1+1 algorithm with adaptive step,
- single-start gradient algorithm with adaptive step
- multistart gradient algorithm with adaptive step

in optimization of your **individual* multidimensional** function. Use 1200 checks of the objective function for all three methods. Test the methods statistically by averaging 20 runs of each method, store the means in table 1. Save all the codes for further use.

Table 1: Aggregated numerical results of the instruction

	2D Fewminima			**2D Manyminima,** 20 runs statistics		**Multidimensional,** 20 runs statistics	
	Test1	**Test2**	**Test3**	**mean**	**std**	**mean**	**std**
Grid search				-	-	-	-
1+1				-	-	-	-
Gradient				-	-	-	-
Multistart gradient				-	-	-	-
Gradient adaptive							
Multistart gradient adaptive							
1+1 adaptive							

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 optimization task in which we optimize parameters of an optimization algorithm) - Can we use here grid search method?

A hint*: tested algorithms are non-deterministic. It is an important factor here! If you aim to use any optimization method that you know now, each "OF check" will give you the value from some statistical distribution. You should average many tests to gain better knowledge about "quality" that should be attributed to particular values of metaparameters.*

Task 1.10: 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 1.11: 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 1.12: 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, you don't actually need to modify the code. Change of one of them should help.