MAT2 - homework 4

Urša Zrimšek

Nelder-Mead method

The first task in this homework was to implement Nelder-Mead (NM) method and compare it with GD, Polyak GD, Nesterov GD, AdaGrad GD, Newton, BFGS, already implemented in homework 2. We compared them on two functions,

$$f_1(x,y,z) = (x-z)^2 + (2y+z)^2 + (4x-2y+z)^2 + x + y$$

and

$$f_2(x,y,z) = (x-1)^2 + (y-1)^2 + 100(y-x^2)^2 + 100(z-y^2)^2.$$

In homework 2 we compared gradient methods based on their performance in limited number of steps and in limited time. For f_1 we took the results with starting point $(0,0,0)$ and for f_2 with starting point $(1.2, 1.2, 1.2)$. These were also the starting points for NM algorithm, from which we built the symplex in the shape of tetrahedron. We also compared different sizes of the tetrahedron, with edge sizes in $\{1, 2, 5, 10\}$.

Now if we compare previous results with NM results, we see that the distance from true minimum for f_1 is better or comparable already after 2 or 5 steps for all methods, except for Newton method. But the steps of Newton method are much more complicated and slower than NM. After 10 steps NM is closer to true minimum than all methods except for Newton and BFGS, and after 100 steps, similarly, only Nesterov GD is slightly better. For f_2, the results obtained with NM are better for any of previously mentioned number of steps.

The important thing that we must know is that NM steps are simpler than those of gradient methods. For more informative results we also compared the methods based on needed time. The lowest time limit we had in previous homework was $0.1s$ and NM method reached its best results in less time than that. But for f_1 we don't come as close to the minimum as with gradient methods - NM best error is 0.002 and gradient methods achieve error in range $1e-16$. But for f_2, most of the gradient methods failed. Only AdaGrad achieved comparable results, error of 0.343. With NM and edge length 2, the error after $0.03s$ is 0.033. We found that at least for these two function, the best results are achieved with edge length 2, and worst with length 5.

From these results we can conclude that NM can be good also where gradient methods fail, but if we can use gradient methods, we will probably achieve better accuracy with them. One important advantage of NM is also that we don't need to know the gradient of the function we are using. This advantage will be better seen in next task.

Black box optimization

In this task we were optimizing three functions, that we could only evaluate in each point. Their computation is time consuming and they output a real number. We want to minimize them.

Our first choice with this kind of problem is Nelder-Mead. We could also approximate the gradients and use a GD method, but that would lead to more function calls and it would make our computation slower. Partial derivatives in each of the three dimensions could be approximated by formula:

$$\frac{\partial f}{\partial x_i}(z) = \lim_{h \searrow 0} \frac{f(z + h \cdot e_i) - f(z)}{h}.$$

We can see that this computation needs 6 additional function calls (for normal GD, for others even more), 2 for each dimension. For NM we usually need just 2 function calls in a step (for reflection and expansion / contraction point). Others can be memorized from before. The worst case is when we shrink, as we need to call the function 3 additional times. This is still faster than with gradient approximation.

The results of NM method are seen in table I. We set the tolerance (of difference between function values at best and worst point) to $1e-10$ and maximum number of steps to 400.

Table I
Results of NM in black box optimization.

fn	min value	obtained in	steps
1	0.6965449309	[0.62867936 0.32932735 -0.09616035]	396
2	0.1442525034	[0.17471686 0.38391538 0.01517881]	291
3	0.1875664410	[0.19834739 0.55004277 0.15443822]	400

Local search study

For local search study we were implementing a method for minimum spanning tree (MST) problem. We know that this is a problem for which many efficient algorithms already exist, but here we will compare how local search would perform comparing to them.

One of the algorithms for finding MST is Kruskal's algorithm. It first constructs a forest where each vertex is a separate tree, and then goes through edges of the original graph, that are ordered by their weights. If the edge connects two different trees it adds it to the forest - it combines two trees into one. If we have a connected graph, the forest has a single component at the end and it forms a minimum spanning tree. If we assume that the Kruskal's algorithm correctly computes the MST, we see that it is uniquely defined. At each point of the algorithm we know exactly which edge we need to check (if the weight function is injective and we don't have two edges with same weight), and if we don't have a cycle, we will always use it. When we have only one component, we stop, and we don't need to check all edges with higher weight than those that already connect our MST. We know that if we would change any of those edges, the summed weight will be bigger. Since we are constructing a spanning tree, we know the number of edges that we need from the beginning, and therefore it can't happen that one edge that we didn't check would change two previous edges, and so it can't be better to use it.

In table II we can see the results of our implementation, with 4 different local search approaches - we can choose the edge with minimum weight in the tree to remove, or we can remove a random edge, and similarly we can add an edge with minimum weight that would connect the two components, or we can choose it randomly. We tried 3 different weight permutations, to see if our results are comparable.

From these results we can see that the best approach is to randomly delete an edge, and then chose the minimum possible edge for the alternative. This approach converges to minimum and in the first weight permutation it actually finds it. The second best approach is to choose both candidates randomly, but it needs twice as many iterations and more swaps. If we are

Table II
RESULTS OF NM IN BLACK BOX OPTIMIZATION.

true min	e_{out}/e_{in}	iterations	swaps	weight sum
86049	rand/rand	2787	336	90797
	max/rand	155	42	137857
	rand/min	**1594**	**254**	**86049**
	max/min	41	40	129030
84516	rand/rand	3190	373	88594
	max/rand	224	79	125272
	rand/min	**1428**	**248**	**84804**
	max/min	72	71	113264
85610	rand/rand	2460	340	92824
	max/rand	152	46	136540
	rand/min	**1293**	**250**	**87647**
	max/min	43	42	126901

only deleting maximum edge, we get stuck in a local minimum. If all edges that can exchange it have higher weights, we won't be able to continue with the algorithm and find better options for other edges.

We can't say that choosing random elements is the best approach generally. Here we were choosing between 399 edges for deletion, and that's why we quickly came to a good edge to change. In general, random choosing could mean that we would need too much time, especially in higher dimensions (curse of dimensionality).

If we would want to find MBST, we could compute MST and we know from the Kruskal's algorithm, that MST is also a MBST. If we would want to find it with simpler algorithm, we could just begin removing ordered edges from heaviest, and stop before splitting the graph into two components. Then any spanning tree we would take form this graph is MBST.