

# Constrained Optimisation's Homework

Course of "Numerical optimization for large scale problems and Stochastic Optimization"

January 2023

Savelli Claudio  
S317680  
Politecnico di Torino  
Data Science and Engineering

Spaccavento Bruno  
S314908  
Politecnico di Torino  
Data Science and Engineering

**Abstract**—The selected problem is the Projected Gradient Method. The objective of this paper is to implement the projected gradient method and then test its effectiveness on a selected problem. The initial part of the paper will describe the problem and all the mathematical tools required to reach the solution. In the second part, the results obtained comparing two methods that will be used to evaluate the gradient, the one with exact derivatives and the one approximated with finite differences, will be analysed.

**Index Terms**—Constrained Optimisation, Projected Gradient Method, Python, Numba

## I. CONSTRAINED OPTIMISATION PROBLEM

In constrained optimisation, an objective function that depends on real variables is minimised, with different restrictions placed on the variables as opposed to the unconstrained gradient method. The mathematical formulation is:

**Definition 1.1:** Let  $X \subset \mathbb{R}^n$  be a non empty, closed convex set and  $f : X \rightarrow \mathbb{R}$ , the problem:

$$\min_{x \in X} f(x) \quad (1)$$

where  $x \in \mathbb{R}^n$  is a real vector with  $n \geq 1$  components is called "Constrained optimisation problem".

**Definition 1.2:** A point  $x^*$  is a *global minimum* of  $f$  if  $f(x^*) \leq f(x)$  for all  $x \in \mathbb{R}^n$  (Or in general in the domain of  $f$ ).

**Definition 1.3:** A point  $x^*$  is a *local minimum* of  $f$  if there exists some  $\epsilon > 0$  such that  $f(x^*) \leq f(x)$  whenever  $|x - x^*| \leq \epsilon$ .

**Definition 1.4:** A point  $x^*$  is a *strict local minimum* of  $f$  if there exists some  $\epsilon > 0$  such that  $f(x^*) < f(x)$  whenever  $|x - x^*| < \epsilon$  with  $x \neq x^*$ .

When the function  $f$  is smooth there are more efficient and practical ways to identify local minima without having to examine all points near  $x^*$ . In particular, if  $f$  is twice continuously differentiable, is possible to tell that  $x$  is a local minimiser by examining just the gradient  $\nabla f(x^*)$  and the Hessian  $\nabla^2 f(x^*)$ .

**Theorem 1.5 (First-Order Necessary Conditions):** If  $x^*$  is a local minimiser and  $f$  is continuously differentiable in an open neighborhood of  $x^*$ , then  $\nabla f(x^*) = 0$ .

**Theorem 1.6 (Second-Order Necessary Conditions):** If  $x^*$  is a local minimiser and  $\nabla^2 f(x^*)$  exists and is continuous in an open neighborhood of  $x^*$ , then  $\nabla f(x^*) = 0$  and  $\nabla^2 f(x^*)$  is positive semidefinite.

**Theorem 1.7 (Second-Order Sufficient Conditions):** Suppose that  $\nabla^2 f(x^*)$  is continuous in an open neighborhood of  $x^*$  and that  $\nabla f(x^*) = 0$  and  $\nabla^2 f(x^*)$  is positive definite. Then  $x^*$  is a strict local minimiser of  $f$ .

**Theorem 1.8:** When  $f$  is convex, any local minimiser  $x^*$  is a global minimiser of  $f$ . If in addition  $f$  is differentiable, then any stationary point  $x^*$  is a global minimiser of  $f$ .

Now it is defined the concept of projection of a point onto a convex set,

**Definition 1.9:** Let  $X \subset \mathbb{R}^n$  be a non empty closed convex set. Then for each  $x \in \mathbb{R}^n$  exists and is unique  $\Pi_X(x) \in X$  such that  $\min_{y \in X} \|y - x\| = \|\Pi_X(x) - x\|$ . The point  $\Pi_X(x)$  is called the projection of  $x$  onto  $X$ .

It must be noted that  $X$  closed is fundamental for the existence of  $\Pi_X(x)$  and  $X$  convex ensures that  $\Pi_X(x)$  is unique.

## II. PROJECTED GRADIENT METHOD

### A. Introduction of the method

The projected gradient method is an iterative method used for minimising a constrained function in a defined set  $X$ . It is based on the steepest descent method, given that at each iteration, the 'path'  $p_k$  along which the function decreases locally is the negative gradient of the function at that point. Considering the principle behind this method, it's easy to understand that by following this method iteratively, a minimiser of the function  $x^*$  such that  $x^* = \arg \min_{x \in X} f(x)$  would be found, although not only it might not be a global minimum of the function, but not even a local minimum of the function, considering that the set  $X$  may not include any.

### B. Explanation of the method

As it is mentioned above, the idea behind the method is very easy to comprehend and it is characterised by the following steps at each iteration:

- 1) compute the steepest descent direction  $p_k = -\nabla f(x)$ ;

2) using the steepest descent method, compute

$$\bar{\mathbf{x}}_k = \mathbf{x}_k - \gamma_k \nabla f(\mathbf{x}_k), \quad (2)$$

where  $\gamma_k > 0$  is an arbitrary constant, and then check if the obtained point is in  $X$  or not by computing the projection of the former onto the latter:

$$\hat{\mathbf{x}}_k = \Pi_X(\bar{\mathbf{x}}_k) = \Pi_X(\mathbf{x}_k - \gamma_k \nabla f(\mathbf{x}_k)). \quad (3)$$

In theory the iteration could stop here considering that the point just reached is in  $X$ , in practice a few more steps are executed;

3) compute the feasible direction  $\boldsymbol{\pi}_k := \hat{\mathbf{x}}_k - \mathbf{x}_k$ ;

4) update the step with the formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \boldsymbol{\pi}_k, \quad (4)$$

where  $\alpha_k \in [0, 1]$  is the sequence of the step lenght factors, which initially are all equal to 1, but get refactored during the 'Armijo condition' phase.

The steepest descent method stops after checking two conditions: either a maximum number of iterations is done,  $k > k_{max}$ , or the euclidean norm of the gradient of the function at the current point is lower or equal than a constant value called *tolgrad*,  $\|\nabla f(\mathbf{x}_k)\| \leq \text{tolgrad}$ .

In the projected gradient method though, it is not ensured that the euclidean norm of the gradient would get closer to 0, so another condition must be added. This condition checks the euclidean distance between the current and the previous points reached, then the method stops if the former is lower or equal than a constant value called *tolx*,  $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| \leq \text{tolx}$

### C. Armijo Condition

With the Armijo condition, the goal is to find a "good" length that leads to a sufficient decrement of the function  $f$  at the new point.

Mathematically, the required condition is as follows:

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f(\mathbf{x}_k)^T \mathbf{p}_k, \quad (5)$$

taking  $c_1 \in (0, 1)$  constant.

The idea behind this method is quite simple. We denote the left-hand-side of (5) as  $\phi(\alpha)$  and the right-hand-side of the same, which is a linear function, as  $l(\alpha)$ . Such function has negative slope  $c_1 \nabla f(\mathbf{x}_k)^T \mathbf{p}_k$ , but due to  $c_1 \in (0, 1)$  ( $= 10^{-4}$  in our case), it lies above the graph of  $\phi(\cdot)$  for small positive values of  $\alpha$ . The sufficient decrease condition states that  $\alpha$  is acceptable only if  $\phi(\alpha) \leq l(\alpha)$ . A graphic example is available by looking at Fig. 1. Equation (5), however, is not sufficient on its own to ensure that the algorithm makes reasonable progress along the given search direction, but this can be compensated for by the algorithm, through the careful choice of step lengths.

After evaluating  $l(\alpha_k)$ , starting with a predefined  $\alpha_k$  (in this case  $= 1$ ), iteratively the latter is decreased by multiplying it by a constant value until the evaluation of  $\phi(\alpha_k)$  at that point falls into a valid range.

Therefore, the chosen strategy is a backtracking strategy in which, for  $0 \leq i \leq bt_{max}$ , if Armijo condition is satisfied, use  $\alpha_k^i$ , otherwise  $\alpha_k^{i+1} = \rho \alpha_k^i$  with  $\rho \in (0, 1)$  (in this case  $= 0.7$ ).

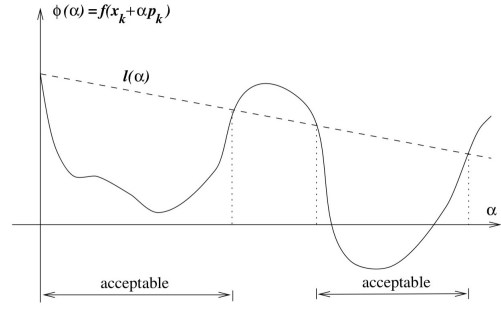


Fig. 1: Example of the Armijo condition.

## III. PROBLEM SETTING

### A. Test on the Test Problem

From the set of problems proposed within the document [3] the following function was selected:

1) Problem 15. Rosenbrock's function:

$$F(\mathbf{x}) = \sum_{k=1}^{n-1} [(x_k - 1)^2 + 100(x_{k+1} - x_k^2)^2], \quad (6)$$

$$x_0 = x_{n+1} = 0, \quad (7)$$

$$\bar{x}_k = 1, \quad k \geq 1 \quad (8)$$

In the tests performed for the following function the following parameters were chosen:  $k_{max} = 2500$ ,  $bt_{max} = 50$ ,  $\text{tolgrad} = \text{tolx} = 10^{-8}$ ,  $\rho = 0.7$ ,  $c = 10^{-4}$  and  $\gamma = 1$  for achieving convergence.

As requested in the delivery, an initial point of size  $n = 10^d$  (with  $d = 3, 4, 5$ ) were taken with the following function  $< \text{rng.randint}(-5, 5, n) >$ , which generates a row vector of length  $n$  whose elements were integers taken from the interval  $[-5, 5]$ , then with the *numpy* function  $< \text{numpy.ndarray.astype} >$ , the previous array was converted to a float array.

Since the problem is constrained, certain constraints were applied to the domain of the function. In particular, 3 different constraints were evaluated.

$$X_1 = [1, 5.12]^2, \quad (9)$$

$$X_2 = [-5.12, 5.12] \times [1, 5.12]^{n-1} \quad (10)$$

$$X_3 = [-5.12, 5.12]^{n/2} \times [1, 5.12]^{n/2} \quad (11)$$

A general method for the projection function  $\Pi_x$  has been created within the code, to use the same procedure for all constraints by simply passing them on each time. To give an idea, the projector function of  $X_1$  follows as an example.

$$\Pi_{X_1}(\mathbf{x})_i = \begin{cases} x_i, & \text{if } 1 \leq x_i \leq 5.12 \\ 5.12, & \text{if } x_i > 5.12 \\ 1, & \text{if } x_i < 1 \end{cases}$$

With regard to the finite derivative method, given the similarity of the results obtained and the time taken to obtain

the test results, the results visible in the table refer to the forward method only since it is the one that is most efficient since instead of evaluating the function twice it evaluates it once per iteration, although all three have been implemented in the code.

Following is described the equation representing the finite difference forward method:

$$\nabla f_i \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_i) - f(\mathbf{x}_0)}{h} \quad (12)$$

The constant  $h$  is defined as:

$$\begin{aligned} h &= 10^{-p} \|\mathbf{x}_k\|, \\ p &= 2, 4, 6, 8, 10, 12 \end{aligned} \quad (13)$$

where  $\mathbf{x}_k$  is the point at which the derivatives have to be approximated.

After generating the initial points, these were used to perform tests with the projected gradient method with exact derivatives and with finite derivatives, and the results were collected and put in Table I, Table II, and Table III.

In Fig. 2, it is possible to observe some comparisons of the results obtained and convergence, when the method,  $d$ ,  $p$ , and  $X$  vary. In fact, in the graphs it is possible to observe, as the number of iterations increases, the variation of  $f(\mathbf{x}_k)$ ,  $\|\nabla(f(\mathbf{x}_k))\|$  and  $\|\Delta(\mathbf{x}_k)\|$ .

### B. Numba

In addition to the results obtained, it is essential to emphasise the time required to obtain the various results below, in order to also be able to understand the reasons why it was impossible to collect certain results. It is also important to make this point in order to make a comparison between the two methods also from the point of view of time efficiency rather than calculation only.

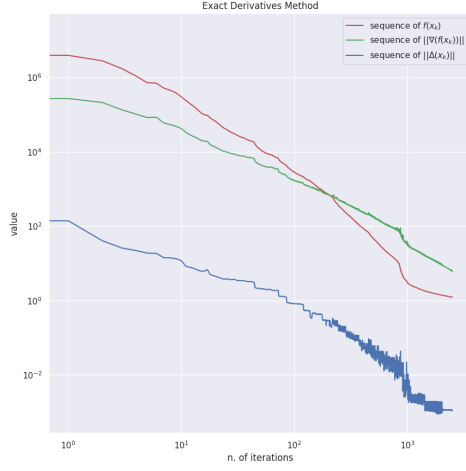
Initially, the overall time to achieve a result was too high, and as a mathematical solution that would speed up the code sufficiently was found, It has been speeded up from an IT point of view. This was possible through the Numba [4] library that was implemented, an open-source JIT compiler that translates a subset of Python and NumPy into fast machine code using LLVM. LLVM "is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimised with a variety of transformations over multiple passes" [5].

### C. Considerations regarding computational limits of the algorithm

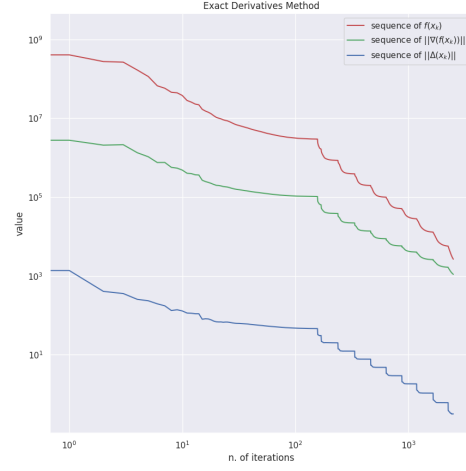
In both cases, as expected, the computation time increases as the  $d$  value increases. The chosen set  $X$  and the variable  $p$ , on the other hand, do not affect the computation time as expected. With  $d$  being equal, the exact derivative method proved to be extremely faster than the finite differences method. To make a comparison, assuming  $d = 4$  and  $p = 6$  (this only applies to the Finite Differences method), on the vector  $X_1$  the former method needs only 1,191 seconds to perform  $k = 2500$  iterations, whereas the latter needs a whopping 811,352 seconds, more than 800 times as long.

As  $d$  changes, for the exact derivative method, the time per iteration is approximately  $6.5 * 10^{-5}$  for  $d = 3$ ,  $2.4 * 10^{-3}$  for  $d = 4$  and  $1.4 * 10^{-2}$  for  $d = 5$ . For the finite difference method, since it is necessary to do more operations on the vector, the time per iteration increases much more dramatically, starting at 0.06 seconds for  $d = 3$ , passing through 0.33 seconds for  $d = 4$ , and even reaching about 69.94 seconds for  $d = 5$ .

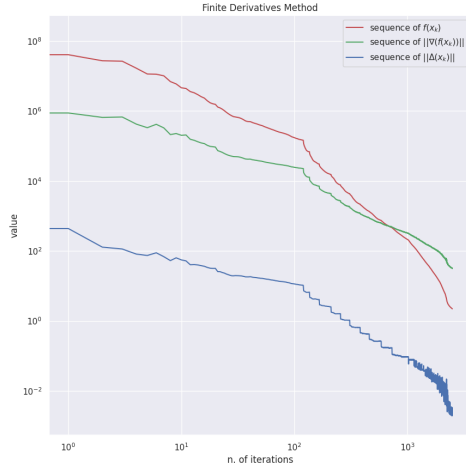
Therefore, apart from the values  $d = 5$ ,  $p = 2$  and  $X = X_1$ , where the iterations were only  $k = 3$ , for all other combinations containing  $d = 5$ , it was infeasible to estimate the values due to time constraints.



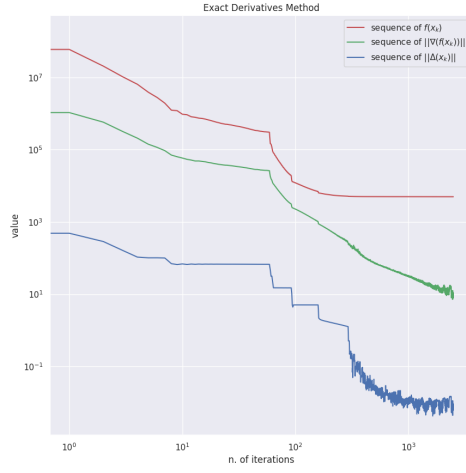
(a) Exact derivatives method,  $d = 3$ , done on  $X_1$ .



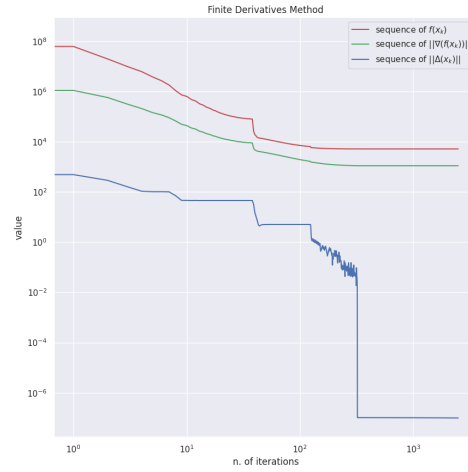
(b) Exact derivatives method,  $d = 5$ , done on  $X_1$ .



(c) Finite differences method,  $d = 4$ ,  $p = 6$ , done on  $X_1$ .



(d) Exact derivatives method,  $d = 4$ , done on  $X_3$ .



(e) Finite differences method,  $d = 4$ ,  $p = 4$ , done on  $X_3$ .

Fig. 2: Set of graphs generated for different combinations of Methods,  $d$ ,  $p$ , and  $X$ .

## APPENDIX

TABLE I: Results for  $X_1$ 

	<b>d</b>	<b>N. iterations</b>	$f(\mathbf{x}_{k_{max}})$
<i>Exact Derivatives</i>	$d = 3$	$k = k_{max} = 2500$	1.2323223993867778
	$d = 4$	$k = k_{max} = 2500$	12.567267477520394
	$d = 5$	$k = k_{max} = 2500$	2464.845454960328
$p = 2$	$d = 3$	$k = 34$	0.0
	$d = 4$	$k = 4$	0.0
	$d = 5$	$k = 3$	0.0
$p = 4$	$d = 3$	$k = 516$	0.0
	$d = 4$	$k = 1055$	0.0
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 6$	$d = 3$	$k = k_{max} = 2500$	1.0957717680199979
	$d = 4$	$k = k_{max} = 2500$	4.441641996119131
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 8$	$d = 3$	$k = k_{max} = 2500$	1.1636628256014978
	$d = 4$	$k = k_{max} = 2500$	2.415274269509432
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 10$	$d = 3$	$k = k_{max} = 2500$	1.1304521772716116
	$d = 4$	$k = k_{max} = 2500$	12.891551503338885
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 12$	$d = 3$	$k = k_{max} = 2500$	1.2647478967529633
	$d = 4$	$k = k_{max} = 2500$	17.930464484162016
	$d = 5$	[NO RESULT]	[NO RESULT]

TABLE II: Results for  $X_2$ 

	<b>d</b>	<b>N. iterations</b>	$f(\mathbf{x}_{k_{max}})$
<i>Exact Derivatives</i>	$d = 3$	$k = k_{max} = 2500$	5.247132221158945
	$d = 4$	$k = k_{max} = 2500$	15.250614433519466
	$d = 5$	$k = k_{max} = 2500$	2492.5172328230638
$p = 2$	$d = 3$	$k = k_{max} = 2500$	21.513598999623756
	$d = 4$	$k = k_{max} = 2500$	15.105125349897357
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 4$	$d = 3$	$k = k_{max} = 2500$	0.7048510957662084
	$d = 4$	$k = k_{max} = 2500$	2.411855040767321
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 6$	$d = 3$	$k = k_{max} = 2500$	1.1830274558791416
	$d = 4$	$k = k_{max} = 2500$	3.102213928338944
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 8$	$d = 3$	$k = k_{max} = 2500$	1.3240811121650649
	$d = 4$	$k = k_{max} = 2500$	6.1624106853537395
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 10$	$d = 3$	$k = k_{max} = 2500$	5.250741624571233
	$d = 4$	$k = k_{max} = 2500$	7.665529779150894
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 12$	$d = 3$	$k = k_{max} = 2500$	5.249046006389981
	$d = 4$	$k = k_{max} = 2500$	8.636845952934081
	$d = 5$	[NO RESULT]	[NO RESULT]

TABLE III: Results for  $X_3$ 

	<b>d</b>	<b>N. iterations</b>	$f(\mathbf{x}_{k_{max}})$
<i>Exact Derivatives</i>	$d = 3$	$k = k_{max} = 2500$	527.5634134467701
	$d = 4$	$k = k_{max} = 2500$	4984.124715705466
	$d = 5$	$k = k_{max} = 2500$	49532.85590418838
$p = 2$	$d = 3$	$k = k_{max} = 2500$	5438.79016743088
	$d = 4$	$k = k_{max} = 2500$	1537018.3962359375
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 4$	$d = 3$	$k = k_{max} = 2500$	554.7827767450482
	$d = 4$	$k = k_{max} = 2500$	5125.754141951335
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 6$	$d = 3$	$k = k_{max} = 2500$	527.7372197644975
	$d = 4$	$k = k_{max} = 2500$	4983.10153998026
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 8$	$d = 3$	$k = k_{max} = 2500$	527.9548773372485
	$d = 4$	$k = k_{max} = 2500$	4983.832060735148
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 10$	$d = 3$	$k = k_{max} = 2500$	527.3460953374043
	$d = 4$	$k = k_{max} = 2500$	4984.480378506935
	$d = 5$	[NO RESULT]	[NO RESULT]
$p = 12$	$d = 3$	$k = k_{max} = 2500$	527.4143032055298
	$d = 4$	$k = k_{max} = 2500$	4984.334074580872
	$d = 5$	[NO RESULT]	[NO RESULT]

## ACKNOWLEDGMENT

Given the results obtained, both methods lead to good results (indeed, sometimes the finite difference method achieves convergence with fewer iterations than the exact derivative method).

Undoubtedly it is important to emphasise that when the exact derivative can be evaluated because it is available, it offers an immense speed boost compared to the second method.

Another interesting consideration is related to whether or not the finite difference method approaches convergence as the  $p$ -value varies. In fact, by using a  $p$  that makes  $h$  close to the square root of the machine's precision ( $10^{-16}$  approximately), so for  $p = 8$ , the results obtained are closer to 0.

Finally, even though the elements seem very far from the point of minimum ( $f(\mathbf{x}) = 0$ ), one must remember how the function evaluated is a summation, so having a summation of  $10^4$  evaluated points tending to a number of about 500 is still a very good result.

To conclude, the results obtained are satisfactory, although there is certainly still room for improvement. One example is the more advanced use of the `Numba` library that was discovered in the course of this homework, of which only the basic functionality were used for code acceleration. But it is possible, through more advanced functionality, to exploit its parallelisation or the power of the GPU to perform vector and matrix calculations.

## REFERENCES

- [1] Thomas V. Mikosch, Sidney I. Resnick, Stephen M. Robinson, "Springer Series in Operations Research and Financial Engineering", Jorge Nocedal, Stephen J. Wright, Second Edition, Springer, 2006..
- [2] "Numerical optimization for large scale problems" slides, Pieraccini Sandra, Dipartimento di scienze Matematiche, Politecnico di Torino.
- [3] Xin-She Yang, "Test Problems in Optimization", Department of Engineering, University of Cambridge [https://www.researchgate.net/publication/45932888\\_Test\\_Problems\\_in\\_Optimization](https://www.researchgate.net/publication/45932888_Test_Problems_in_Optimization).
- [4] "Numba", <https://numba.pydata.org/>.
- [5] "LLVM", <https://en.wikipedia.org/wiki/LLVM>.

## CODE

Listing 1: Main

```
1 # %matplotlib ipympl
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from mpl_toolkits import mplot3d
5 from mpl_toolkits.axes_grid1 import make_axes_locatable
6 import seaborn as sns
7 import function as f
8 from function import *
9 import proj_gradient_method as pcg
10 from proj_gradient_method import *
11 from numba import typeof
12 from numba.core import types
13 from numba.typed import Dict
14 from sklearn.model_selection import ParameterGrid
15 from importlib import reload
16 import time
17 from varname.helpers import Wrapper
18
19 reload(f)
20 reload(pcg)
21
22 gamma = 1.0
23 kmax = 2500
24 tolgrad = 1e-8
25 tolx = 1e-8
26 c = 1e-4
27 rho = 0.7
28 btmax = 50
29 params = {'d': [3, 4, 5],
30           'k': [2, 4, 6, 8, 10, 12]}
31
32 """Exact derivatives"""
33
34 rng = np.random.RandomState(42)
35 for d in [3, 4, 5]:
36     n = 10*d
37     x = rng.randint(-5, 5, n)
38     x = x.astype('f8')
39
40     X1 = Dict.empty(
41         key_type=types.unicode_type,
42         value_type=types.float64[:]
43     )
44     X1['[0, n]'] = np.asarray([1, 5.12, 0, n], dtype='f8')
45     X1 = Wrapper(X1)
46
47     X2 = Dict.empty(
48         key_type=types.unicode_type,
49         value_type=types.float64[:]
50     )
51     X2['[0]'] = np.asarray([-5.12, 5.12, 0, 1], dtype='f8')
52     X2['[n/2, n-1]'] = np.asarray([1, 5.12, 1, n], dtype='f8')
53     X2 = Wrapper(X2)
54
55     X3 = Dict.empty(
56         key_type=types.unicode_type,
57         value_type=types.float64[:]
58     )
59     X3['[0, n/2-1]'] = np.asarray([-5.12, 5.12, 0, n/2], dtype='f8')
60     X3['[n/2, n-1]'] = np.asarray([1, 5.12, n/2, n], dtype='f8')
61     X3 = Wrapper(X3)
62
63     for X in [X1, X2, X3]:
64         start_time = time.time()
65         x_seq, f_seq, gradf_norm_seq, deltax_norm_seq, k, bt_seq = projected_gradient_bcktrck(x, X.value,
66                                                                                               gamma, kmax,
67                                                                                               tolgrad,
68                                                                                               tolx, c,
69                                                                                               rho, btmax,
70                                                                                               False,
```



```

68                                     'Empty', 0)
69     print("--- %s seconds ---" % (time.time() - start_time))
70     print(f"{i}) d = {d}, fin_diff = {False}, X = {X.name}")
71     print(f"k = {k}, fk = {f_seq[-1]}, gradfk_norm = {gradf_norm_seq[-1]}, deltax_norm = {
deltax_norm_seq[-1]}\n")
72     i += 1
73
74     sns.set()
75     fig, ax = plt.subplots(figsize=(10, 10))
76     plt.title("Exact Derivatives Method")
77     ax.plot(f_seq, '-r', label = 'sequence of $f(x_k)$')
78     ax.plot(gradf_norm_seq, '-g', label = 'sequence of $r$||\nabla(f(x_k))||$')
79     ax.plot(deltax_norm_seq, '-b', label = 'sequence of $r$||\Delta(x_k)||$')
80     ax.set_xscale('log', base=10)
81     ax.set_yscale('log', base=10)
82     plt.xlabel("n. of iterations")
83     plt.ylabel("value")
84     leg = ax.legend();
85     print(f"{i}) d = {d}, fin_diff = {False}, X = {X.name}")
86     print(f"k = {k}, fk = {f_seq[-1]}, gradfk_norm = {gradf_norm_seq[-1]}, deltax_norm = {deltax_norm_seq
[-1]}\n")
87
88     """Finite differences"""
89
90     rng = np.random.RandomState(42)
91     for param in ParameterGrid(params):
92         d = param['d']
93         n = 10*d
94         x = rng.randint(-5, 5, n)
95         x = x.astype('f8')
96
97         X1 = Dict.empty(
98             key_type=types.unicode_type,
99             value_type=types.float64[:]
100         )
101         X1['[0, n]'] = np.asarray([1, 5.12, 0, n], dtype='f8')
102         X1 = Wrapper(X1)
103
104         X2 = Dict.empty(
105             key_type=types.unicode_type,
106             value_type=types.float64[:]
107         )
108         X2['[0]'] = np.asarray([-5.12, 5.12, 0, 1], dtype='f8')
109         X2['[n/2, n-1]'] = np.asarray([1, 5.12, 1, n], dtype='f8')
110         X2 = Wrapper(X2)
111
112         X3 = Dict.empty(
113             key_type=types.unicode_type,
114             value_type=types.float64[:]
115         )
116         X3['[0, n/2-1]'] = np.asarray([-5.12, 5.12, 0, n/2], dtype='f8')
117         X3['[n/2, n-1]'] = np.asarray([1, 5.12, n/2, n], dtype='f8')
118         X3 = Wrapper(X3)
119
120         k = param['k']
121         h = 10*(-k) * np.linalg.norm(x)
122
123         for X in [X1, X2, X3]:
124             start_time = time.time()
125             x_seq2, f_seq2, gradf_norm_seq2, deltax_norm_seq2, k2, bt_seq2 = projected_gradient_bcktrck(x, X.
value, gamma, kmax,
126                                     tolgrad
127                                     , tolX, c,
128                                     rho,
129                                     btmax, True,
130                                     'fw', h
131
132             print("--- %s seconds ---" % (time.time() - start_time))
133             print(f"{i}) d = {d}, fin_diff = {True}, type = fw, p = {param['k']}, X = {X.name}")
134             print(f"k = {k2}, fk = {f_seq2[-1]}, gradfk_norm = {gradf_norm_seq2[-1]}, deltax_norm = {
deltax_norm_seq2[-1]}\n")
135             i += 1
136
137     sns.set()

```

```

135 fig, ax = plt.subplots(figsize=(10, 10))
136 plt.title("Finite Derivatives Method")
137 ax.plot(f_seq2, '-r', label = 'sequence of  $f(x_k)$ ')
138 ax.plot(gradf_norm_seq2, '-g', label = 'sequence of  $\| \nabla f(x_k) \|$ ')
139 ax.plot(deltax_norm_seq2, '-b', label = 'sequence of  $\| \Delta(x_k) \|$ ')
140 ax.set_xscale('log', base=10)
141 ax.set_yscale('log', base=10)
142 plt.xlabel("n. of iterations")
143 plt.ylabel("value")
144 leg = ax.legend();
145 print(f"{i}) d = {d}, fin_diff = {False}, X = {X.name}, p = {k}")
146 print(f"k = {k2}, fk = {f_seq2[-1]}, gradfk_norm = {gradf_norm_seq2[-1]}, deltax_norm = {deltax_norm_seq2[-1]}\n")

```

Listing 2: Projected Gradient Method

```

1 import numpy as np
2 from numba import jit
3 from numba.typed import Dict
4 from function import *
5
6
7 @jit(nopython=True)
8 def project(x, X):
9     '''the general definition of the projection function.
10
11     INPUTS:
12     x = points on which the projection is to be applied;
13     X = Constraints of the projection to be made and on the number of points;
14
15     OUTPUTS:
16     x = points of input x projected;'''
17     for v in X.values():
18         low = v[0]
19         upp = v[1]
20         start = int(v[2])
21         stop = int(v[3])
22         x[start:stop] = np.where(x[start:stop] > upp, upp, x[start:stop])
23         x[start:stop] = np.where(x[start:stop] < low, low, x[start:stop])
24     return x
25
26
27 @jit(nopython=True)
28 def projected_gradient_bcktrck(x0, box, gamma, kmax, tolgrad, tolX, c1, rho, btmax, fin_diff, fd_type, h):
29
30     ''' Function that performs the steepest descent optimization method for a given function.
31
32     INPUTS:
33     x0 = n-dimensional column vector;
34     box = the constraint box (in the report and elsewhere is the X)
35     gamma = fixed factor gamma > 0 that multiplies the descent direction before the (possible) projection
36           on delta(X), where X is the domain;
37     kmax = maximum number of iterations permitted;
38     tolgrad = value used as stopping criterion w.r.t. the norm of the gradient;
39     tolX = a real scalar value characterising the tol with respect to the norm of  $x_{(k+1)} - x_k$  to stop the
40           method;
41     c1 = factor of the Armijo condition that must be a scalar in (0,1);
42     rho = fixed factor, lesser than 1, used for reducing alpha0;
43     btmax = maximum number of steps for updating alpha during the backtracking strategy;
44     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
45     ;
46     fd_type = if fin_diff == True, choose between centered/forward/backword finite differences method;
47     h = the value of h previously evaluated to use in the evaluation of the gradient for finite difference
48           method;
49
50     OUTPUTS:
51     x_seq = n-by-k matrix where the columns are the xk computed during the iterations;
52     f_seq = sequence of values of f(xk) computed during the iterations;
53     gradf_norm_seq = sequence of norm of grad f(xk) computed during the iterations;
54     k = number of iterations;
55     bt_seq = k vector whose elements are the number of backtracking '''
56
57     x_seq = np.empty((kmax+1, x0.shape[0]))
58     x_seq[0, :] = x0

```

```

55 bt_seq = np.empty((kmax+1,))
56 f_seq = np.empty((kmax+1,))
57 gradf_norm_seq = np.empty((kmax+1,))
58 deltax_norm_seq = np.empty((kmax+1,))
59 xk = x0
60 fk = 0
61 k = 0
62 gradfk_norm = 0.0
63 deltaxk_norm = np.linalg.norm(x0, 2)
64 alphak = gamma
65
66 fk = rosenbrock(xk)
67 f_seq[0] = fk
68 gradfk_norm = np.linalg.norm(grad_rosenbrock(xk, fin_diff, fd_type, h), 2)
69 gradf_norm_seq[0] = gradfk_norm
70 deltax_norm_seq[0] = deltaxk_norm
71
72 while k < kmax and gradfk_norm > tolgrad and deltaxk_norm > tolx:
73     gradfk = grad_rosenbrock(xk, fin_diff, fd_type, h)
74     pk = -gradfk
75     xhat = project(xk + gamma*pk, box)
76     pik = xhat - xk
77     xnew = xk + alphak*pik
78     fnew = rosenbrock(xnew)
79
80     bt = 0
81     while bt < btmax and fnew > fk + c1*alphak*(gradfk @ pik):
82         # update alpha
83         alphak = rho*alphak
84         xnew = xk + alphak*pik
85         fnew = rosenbrock(xnew)
86         bt = bt + 1
87     alphak = gamma
88
89     deltaxk_norm = np.linalg.norm(xnew - xk, 2)
90     deltax_norm_seq[k+1] = deltaxk_norm
91     xk = xnew
92     fk = fnew
93     gradfk_norm = np.linalg.norm(grad_rosenbrock(xk, fin_diff, fd_type, h), 2)
94     x_seq[k+1, :] = xk
95     f_seq[k+1] = fk
96     gradf_norm_seq[k+1] = gradfk_norm
97     bt_seq[k] = bt
98     # print(k)
99     k = k + 1
100
101 return x_seq, f_seq, gradf_norm_seq, deltax_norm_seq, k, bt_seq

```

Listing 3: Function

```

1 import numpy as np
2 from numba import jit
3 from math import *
4
5
6 @jit(nopython=True)
7 def rosenbrock(x):
8     #Evaluation of the Rosenbrock function in the point x
9     n = x.shape[0] - 1
10    def fk(x: np.ndarray, k: int) -> float:
11        return 100 * (x[k+1] - x[k]**2)**2 + (x[k] - 1)**2
12    z = np.empty(n)
13    for k in range(0, n):
14        z[k] = fk(x, k)
15    return np.sum(z)
16
17
18 @jit(nopython=True)
19 def grad_rosenbrock(x, fin_diff, type, h):
20    '''
21    Compute the approximation of the gradient via finite differences or with the true gradient
22
23    INPUTS:
24    x = array of x-coordinates;

```

```

25     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
26     ;
27     type = if fin_diff == True, choose between centered/forward/backword finite differences method;
28     h = the value of h previously evaluated to use in the evaluation of the gradient for finite difference
29     method;
30
31     OUTPUTS:
32     gradfx=the appossimation of the gradient in x;
33     '''
34     num = x.shape[0]
35     grad = np.empty(num)
36     if fin_diff == True:
37         if (type == "fw" or type == "bw"):
38             fx = rosenbrock(x)
39             for i in range(0, num):
40                 y = np.copy(x)
41                 z = np.copy(x)
42                 if(type == "fw"):
43                     y[i] += h
44                     grad[i] = (rosenbrock(y) - fx) / h
45                     y[i] -= h
46                 elif(type == "bw"):
47                     y[i] -= h
48                     grad[i] = -(rosenbrock(y) - fx) / h
49                     y[i] += h
50                 else:
51                     y[i] += h
52                     z[i] -= h
53                     grad[i] = (rosenbrock(y) - rosenbrock(z)) / (2*h)
54                     y[i] -= h
55                     z[i] += h
56     else:
57         grad[0] = 400*x[0]**3 - 400*x[0]*x[1] + 2*x[0] - 2
58         for i in range(1, num-1):
59             grad[i] = 400*x[i]**3 - 400*x[i]*x[i+1] - 200*x[i-1]**2 + 202*x[i] - 2
60             grad[num-1] = 200*(x[num-1] - x[num-2]**2)
61     return grad

```