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 $f : X \longrightarrow R$, the problem:

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

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

Definition 1.2: A point $\boldsymbol{x}^*$ is a *global minimum* of f if $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$ for all $\boldsymbol{x} \in X$).

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

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

In unconstrained optimisation the first basic necessary condition is:

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

In the constrained case the minimum may not have the derivative = 0. In fact, it may happen that the point $\boldsymbol{x}^*$ is not even a stationary point.

Theorem 1.6 (Necessary Condition): If $\boldsymbol{x}^*$ is a local minimiser of f over X, then $\nabla f(\boldsymbol{x}^*)^T(x - x^*) \geq 0$, $\forall x \in X$.

Theorem 1.7 (Sufficient Condition): If f is convex over X, then the necessary condition 1.6 is also sufficient.

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

Definition 1.8: Let $X \subset \mathbb{R}^n$ be a non empty closed convex set. Then for each $\boldsymbol{x} \in \mathbb{R}^n$ exists and is unique $\Pi_X(\boldsymbol{x}) \in X$ such that $\min_{\boldsymbol{y} \in X} ||\boldsymbol{y} - \boldsymbol{x}|| = ||\Pi_X(\boldsymbol{x}) - \boldsymbol{x}||$. The point $\Pi_X(\boldsymbol{x})$ is called the projection of $\boldsymbol{x}$ onto X.
It must be noted that X closed is fundamental for the existence of $\Pi_X(\boldsymbol{x})$ and X convex ensures that $\Pi_X(\boldsymbol{x})$ 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' $\boldsymbol{p}_k$ 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 $\boldsymbol{x}^*$ such that $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in X} f(\boldsymbol{x})$ 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 $\boldsymbol{p}_k = -\nabla f(\boldsymbol{x})$;
2) using the steepest descent method, compute

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

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:

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

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 $\boldsymbol{\pi}_k \coloneqq \hat{\boldsymbol{x}}_k - \boldsymbol{x}_k$;
4) update the step with the formula

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{\pi}_k, \tag{4}$$

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*, $||\nabla f(\boldsymbol{x}_k|| \leq tolgrad$.
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*, $||\boldsymbol{x}_k - \boldsymbol{x}_{k-1}|| \leq tolx$

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:

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k, \tag{5}$$

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 $c_1 \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k$, 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 α. The sufficient decrease condition states that α 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 α_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 α_k^i, otherwise $\alpha_k^{i+1} = \rho \alpha_k^i$ with $\rho \in (0, 1)$ (in this case $= 0.7$).

III. Problem setting

A. Test on the Test Problem

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

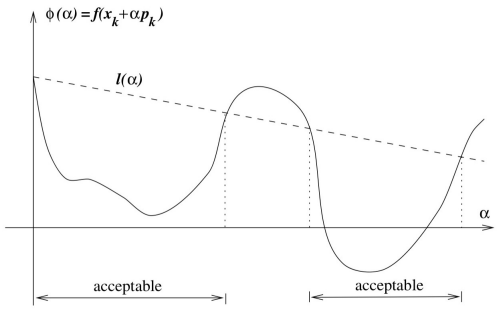


Fig. 1: Example of the Armijo condition.

1) Problem 15. Rosenbrock's function:

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

$$x_0 = x_{n+1} = 0, \tag{7}$$

$$\bar{\boldsymbol{x}}_k = 1, \quad k \geq 1 \tag{8}$$

In the tests performed for the following function the following parameters were chosen: $k_{max} = 2500$, $bt_{max} = 50$, $tolgrad = tolx = 10^{-8}$, $\rho = 0.7$, $c = 10^{-4}$ 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 $< rng.randint(-5, 5, n) >$, which generates a row vector of length n whose elements were integers taken from the interval $[-5, 5)$, then with the $numpy$ function $< numpy.ndarray.astype >$, 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.

$$X_1 = [1, 5.12]^2, \tag{9}$$

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

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

A general method for the projection function Π_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.

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

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:

$$\nabla f_i \approx \frac{f(\boldsymbol{x}_0 + he_i) - f(\boldsymbol{x}_0)}{h} \tag{12}$$

The constant h is defined as:

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

where $\boldsymbol{x}_k$ 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 $f(x_k)$, $||\nabla(f(x_k))||$ and $||\Delta(x_k)||$.

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.

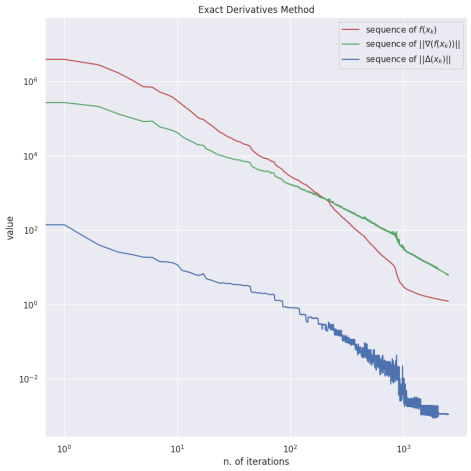
Initially, the overall time to achieve a result was too high, and as a mathematical solution that would speed up the code sufficiently was found, It has been speeded up from an IT point of view. This was possible through the `Numba` [4] library that was implemented, an open-source JIT compiler that translates 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

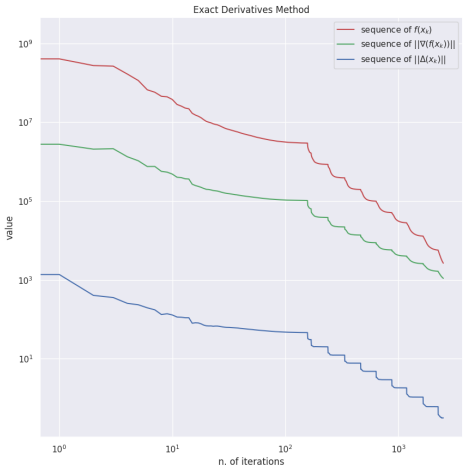
In both cases, as expected, the computation time increases as the d value increases. The chosen set X and the variable p, on the other hand, do not affect the computation time as expected. With d being equal, the exact derivative method proved to be extremely faster than the finite differences method. 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 $d = 5$.

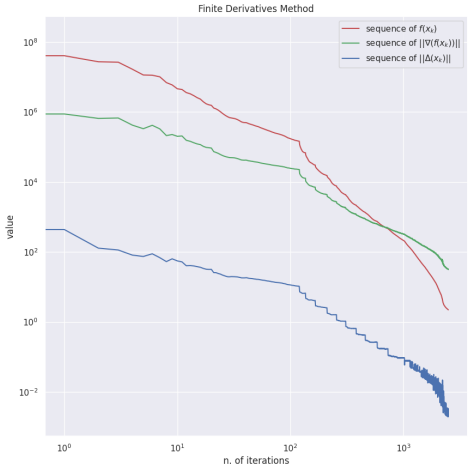
Therefore, apart from the values $d = 5$, $p = 2$ and $X = X_1$, where the iterations were only $k = 3$, for all other combinations containing $d = 5$, it was infeasible to estimate the values due to time constraints.



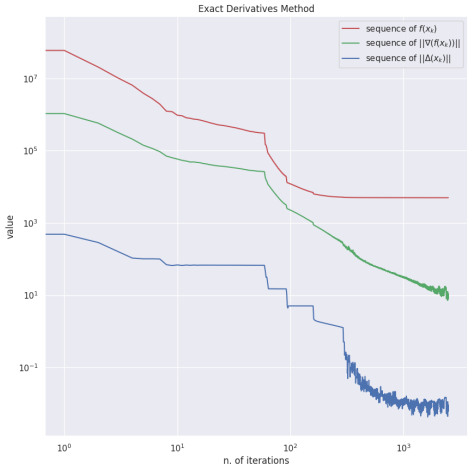
(a) Exact derivatives method, d = 3, done on X_1.



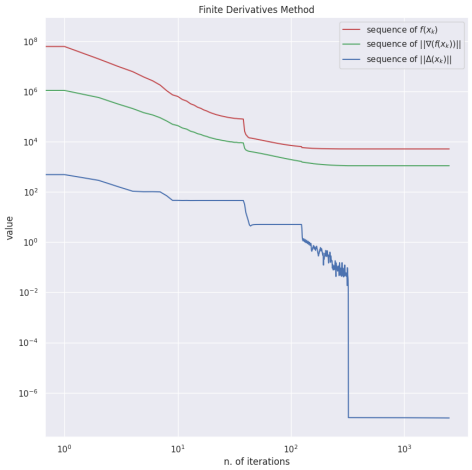
(b) Exact derivatives method, d = 5, done on X_1.



(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

	d	**N. iterations**	$f(\boldsymbol{x}_{k_{max}})$
Exact Derivatives	$d = 3$	$k = k_{max} = 2500$	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

	d	**N. iterations**	$f(\boldsymbol{x}_{k_{max}})$
Exact Derivatives	$d = 3$	$k = k_{max} = 2500$	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

	d	**N. iterations**	$f(\boldsymbol{x}_{k_{max}})$
Exact Derivatives	$d = 3$	$k = k_{max} = 2500$	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 ($f(\boldsymbol{x}) = 0$), 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.

[4] "`Numba`", https://numba.pydata.org/.

[5] "`LLVM`", https://en.wikipedia.org/wiki/LLVM.

CODE

Listing 1: Main

```
# %matplotlib ipympl
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from mpl_toolkits.axes_grid1 import make_axes_locatable
import seaborn as sns
import function as f
from function import *
import proj_gradient_method as pcg
from proj_gradient_method import *
from numba import typeof
from numba.core import types
from numba.typed import Dict
from sklearn.model_selection import ParameterGrid
from importlib import reload
import time
from  varname.helpers import Wrapper

reload(f)
reload(pcg)

gamma = 1.0
kmax = 2500
tolgrad = 1e-8
tolx = 1e-8
c = 1e-4
rho = 0.7
btmax = 50
params = {'d': [3, 4, 5],
          'k': [2, 4, 6, 8, 10, 12]}

"""Exact derivatives"""

rng = np.random.RandomState(42)
for d in [3, 4, 5]:
    n = 10**d
    x = rng.randint(-5, 5, n)
    x = x.astype('f8')

    X1 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X1['[0, n]'] = np.asarray([1, 5.12, 0, n], dtype='f8')
    X1 = Wrapper(X1)

    X2 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X2['[0]'] = np.asarray([-5.12, 5.12, 0, 1], dtype='f8')
    X2['[n/2, n-1]'] = np.asarray([1, 5.12, 1, n], dtype='f8')
    X2 = Wrapper(X2)

    X3 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X3['[0, n/2-1]'] = np.asarray([-5.12, 5.12, 0, n/2], dtype='f8')
    X3['[n/2, n-1]'] = np.asarray([1, 5.12, n/2, n], dtype='f8')
    X3 = Wrapper(X3)

    for X in [X1, X2, X3]:
        start_time = time.time()
        x_seq, f_seq, gradf_norm_seq, deltax_norm_seq, k, bt_seq = projected_gradient_bcktrck(x, X.value,
    gamma, kmax,
                                                                                              tolgrad,
    tolx, c,
                                                                                              rho, btmax,
     False,
```

```
                                                                                                                    'Empty', 0)
        print("--- %s seconds ---" % (time.time() - start_time))
        print(f"{i}) d = {d}, fin_diff = {False}, X = {X.name}")
        print(f"k = {k}, fk = {f_seq[-1]}, gradfk_norm = {gradf_norm_seq[-1]}, deltaxk_norm = {
    deltax_norm_seq[-1]}\n")
        i += 1

sns.set()
fig, ax = plt.subplots(figsize=(10, 10))
plt.title("Exact Derivatives Method")
ax.plot(f_seq, '-r', label = 'sequence of $f(x_k)$')
ax.plot(gradf_norm_seq, '-g', label = 'sequence of 'r'$||\nabla(f(x_k))||$')
ax.plot(deltax_norm_seq, '-b', label = 'sequence of 'r'$||\Delta(x_k)||$')
ax.set_xscale('log', base=10)
ax.set_yscale('log', base=10)
plt.xlabel("n. of iterations")
plt.ylabel("value")
leg = ax.legend();
print(f"{i}) d = {d}, fin_diff = {False}, X = {X.name}")
print(f"k = {k}, fk = {f_seq[-1]}, gradfk_norm = {gradf_norm_seq[-1]}, deltaxk_norm = {deltax_norm_seq
    [-1]}\n")

"""Finite differences"""

rng = np.random.RandomState(42)
for param in ParameterGrid(params):
    d = param['d']
    n = 10**d
    x = rng.randint(-5, 5, n)
    x = x.astype('f8')

    X1 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X1['[0, n]'] = np.asarray([1, 5.12, 0, n], dtype='f8')
    X1 = Wrapper(X1)

    X2 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X2['[0]'] = np.asarray([-5.12, 5.12, 0, 1], dtype='f8')
    X2['[n/2, n-1]'] = np.asarray([1, 5.12, 1, n], dtype='f8')
    X2 = Wrapper(X2)

    X3 = Dict.empty(
        key_type=types.unicode_type,
        value_type=types.float64[:]
    )
    X3['[0, n/2-1]'] = np.asarray([-5.12, 5.12, 0, n/2], dtype='f8')
    X3['[n/2, n-1]'] = np.asarray([1, 5.12, n/2, n], dtype='f8')
    X3 = Wrapper(X3)

    k = param['k']
    h = 10**(-k) * np.linalg.norm(x)

    for X in [X1, X2, X3]:
        start_time = time.time()
        x_seq2, f_seq2, gradf_norm_seq2, deltax_norm_seq2, k2, bt_seq2 = projected_gradient_bcktrck(x, X.
    value, gamma, kmax,
                                                                                                    tolgrad
    , tolx, c,
                                                                                                    rho,
    btmax, True,
                                                                                                    'fw', h
    )
        print("--- %s seconds ---" % (time.time() - start_time))
        print(f"{i}) d = {d}, fin_diff = {True}, type = fw, p = {param['k']}, X = {X.name}")
        print(f"k = {k2}, fk = {f_seq2[-1]}, gradfk_norm = {gradf_norm_seq2[-1]}, deltaxk_norm = {
    deltax_norm_seq2[-1]}\n")
        i += 1

sns.set()
```

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

Listing 2: Projected Gradient Method

```
import numpy as np
from numba import jit
from numba.typed import Dict
from function import *


@jit(nopython=True)
def project(x, X):
    '''the general definition of the projection function.

    INPUTS:
    x = points on which the projection is to be applied;
    X = Constraints of the projection to be made and on the number of points;

    OUTPUTS:
    x = points of input x projected;'''
    for v in X.values():
        low = v[0]
        upp = v[1]
        start = int(v[2])
        stop = int(v[3])
        x[start:stop] = np.where(x[start:stop] > upp, upp, x[start:stop])
        x[start:stop] = np.where(x[start:stop] < low, low, x[start:stop])
    return x


@jit(nopython=True)
def projected_gradient_bcktrck(x0, box, gamma, kmax, tolgrad, tolx, c1, rho, btmax, fin_diff, fd_type, h):

    ''' Function that performs the steepest descent optimization method for a given function.

    INPUTS:
    x0 = n-dimensional column vector;
    box =  the constraint box (in the report and elsewhere is the X)
    gamma = fixed factor gamma > 0 that multiplies the descent direction before the (possible) projection
    on delta(X), where X is the domain;
    kmax = maximum number of iterations permitted;
    tolgrad = value used as stopping criterion w.r.t. the norm of the gradient;
    tolx = a real scalar value characterising the tol with respect to the norm of x_(k+1) - x_k to stop the
     method;
    c1 = factor of the Armijo condition that must be a scalar in (0,1);
    rho = fixed factor, lesser than 1, used for reducing alpha0;
    btmax = maximum number of steps for updating alpha during the backtracking strategy;
    fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
    ;
    fd_type = if fin_diff == True, choose between centered/forward/backword finite differences method;
    h = the value of h previously evaluated to use in the evaluation of the gradient for finite difference
    method;

    OUTPUTS:
    x_seq = n-by-k matrix where the columns are the xk computed during the iterations;
    f_seq = sequence of values of f(xk) computed during the iterations;
    gradf_norm_seq = sequence of norm of grad f(xk) computed during the itarations;
    k = number of iterations;
    bt_seq = k vector whose elements are the number of backtracking '''

    x_seq = np.empty((kmax+1, x0.shape[0]))
    x_seq[0, :] = x0
```

```
    bt_seq = np.empty((kmax+1,))
    f_seq = np.empty((kmax+1,))
    gradf_norm_seq = np.empty((kmax+1,))
    deltax_norm_seq = np.empty((kmax+1,))
    xk = x0
    fk = 0
    k = 0
    gradfk_norm = 0.0
    deltaxk_norm = np.linalg.norm(x0, 2)
    alphak = gamma

    fk = rosenbrock(xk)
    f_seq[0] = fk
    gradfk_norm = np.linalg.norm(grad_rosenbrock(xk, fin_diff, fd_type, h), 2)
    gradf_norm_seq[0] = gradfk_norm
    deltax_norm_seq[0] = deltaxk_norm

    while k < kmax and gradfk_norm > tolgrad and deltaxk_norm > tolx:
        gradfk = grad_rosenbrock(xk, fin_diff, fd_type, h)
        pk = -gradfk
        xhat = project(xk + gamma*pk, box)
        pik = xhat - xk
        xnew = xk + alphak*pik
        fnew = rosenbrock(xnew)

        bt = 0
        while bt < btmax and fnew > fk + c1*alphak*(gradfk @ pik):
            # update alpha
            alphak = rho*alphak
            xnew = xk + alphak*pik
            fnew = rosenbrock(xnew)
            bt = bt + 1
        alphak = gamma

        deltaxk_norm = np.linalg.norm(xnew - xk, 2)
        deltax_norm_seq[k+1] = deltaxk_norm
        xk = xnew
        fk = fnew
        gradfk_norm = np.linalg.norm(grad_rosenbrock(xk, fin_diff, fd_type, h), 2)
        x_seq[k+1, :] = xk
        f_seq[k+1] = fk
        gradf_norm_seq[k+1] = gradfk_norm
        bt_seq[k] = bt
        # print(k)
        k = k + 1

    return x_seq, f_seq, gradf_norm_seq, deltax_norm_seq, k, bt_seq
```

Listing 3: Function

```
import numpy as np
from numba import jit
from math import *


@jit(nopython=True)
def rosenbrock(x):
    #Evaluation of the Rosenbrock function in the point x
    n = x.shape[0] - 1
    def fk(x: np.ndarray, k: int) -> float:
        return 100 * (x[k+1] - x[k]**2)**2 + (x[k] - 1)**2
    z = np.empty(n)
    for k in range(0, n):
        z[k] = fk(x, k)
    return np.sum(z)


@jit(nopython=True)
def grad_rosenbrock(x, fin_diff, type, h):
    '''
    Compute the appoximation of the gradient via finite differences or with the true gradient

    INPUTS:
    x = array of x-coordinates;
```

```
    fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
    ;
    type = if fin_diff == True, choose between centered/forward/backword finite differences method;
    h = the value of h previously evaluated to use in the evaluation of the gradient for finite difference
    method;

    OUTPUTS:
    gradfx=the appossimation of the gradient in x;
    '''
    num = x.shape[0]
    grad = np.empty(num)
    if fin_diff == True:
        if (type == "fw" or type == "bw"):
            fx = rosenbrock(x)
        for i in range(0, num):
            y = np.copy(x)
            z = np.copy(x)
            if(type == "fw"):
                y[i] += h
                grad[i] = (rosenbrock(y) - fx) / h
                y[i] -= h
            elif(type == "bw"):
                y[i] -= h
                grad[i] = -(rosenbrock(y) - fx) / h
                y[i] += h
            else:
                y[i] += h
                z[i] -= h
                grad[i] = (rosenbrock(y) - rosenbrock(z)) / (2*h)
                y[i] -= h
                z[i] += h
    else:
        grad[0] = 400*x[0]**3 - 400*x[0]*x[1] + 2*x[0] - 2
        for i in range(1, num-1):
            grad[i] = 400*x[i]**3 - 400*x[i]*x[i+1] - 200*x[i-1]**2 + 202*x[i] - 2
        grad[num-1] = 200*(x[num-1] - x[num-2]**2)
    return grad
```