

Unconstrained Optimisation's Homework

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

December 2022

Savelli Claudio
S317680
Politecnico di Torino
Data Science and Engineering

Spaccavento Bruno
S314908
Politecnico di Torino
Data Science and Engineering

Abstract—The aim of the homework is to test two different methods for unconstrained nonlinear optimisation problems and compare them. In particular, the two methods taken as reference are the steepest descent and the nonlinear conjugate gradient method with Polak-Ribière. The initial part of the paper will describe the problem and all the mathematical tools required to reach the solutions and, from the data associated with them, analyse the two selected methods. These will be tested on a total of four different functions, which will then be described in detail.

Index Terms—Unconstrained Optimisation, Steepest descent, Nonlinear conjugate gradient method, Python

I. UNCONSTRAINED OPTIMISATION PROBLEM

In unconstrained optimisation, an objective function that depends on real variables is minimised, with no restrictions at all on the values of these variables. The mathematical formulation is:

Definition 1.1: Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a scalar function, the problem:

$$\min_x f(x) \quad (1)$$

where $x \in \mathbb{R}^n$ is a real vector with $n \geq 1$ components and $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is a smooth function, is called "Unconstrained optimisation problem".

What usually happens in large-scale numerical problems, however, is that the function is not available, but only know the values the function can take at certain points x and perhaps its derivatives. What is done in numerical optimisation is to exploit certain algorithms to choose these points, so that is possible to look for a valid solution without using too much time, energy and storage. In fact, what often happens in these cases is that the information from f cannot be extracted economically, so these algorithms are exploited to avoid unnecessarily calling too many of these evaluations.

Definition 1.2: A point x^* is a *global minimum* of f if $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}$ (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 $|x - x^*| < \epsilon$.

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 $|x - x^*| < \epsilon$.

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 .

II. THEORETICAL DESCRIPTION OF THE NUMERICAL METHODS USED

A. Overview

Suppose that the aim is to find the minimum of a function $f(x)$, $x \in \mathbb{R}^n$, and $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

The objective of both iterative methods under consideration is to find a sequence of points x_k such that a global minimum x^* is reached. An initial point, x_0 , is therefore chosen, from which the iteration follows the following principle:

$$x_{k+1} = x_k + \alpha_k p_k, \quad (2)$$

where $\alpha_k > 0$ is the step length, and $p_k \in \mathbb{R}^n$ is a descent direction vector.

The choice of the direction p_k is the main difference between the two methods.

Once the direction has been chosen, it is a little more complex to choose the length of the step; there are various techniques for solving the problem just mentioned, the one used for this homework, the *backtracking line search technique*, will be explained in detail in the next chapter.

B. Steepest descent method

1) *Introduction of the Method:* The steepest descent method is the simplest and historically best-known method of minimising a function. This method, which is still explained today in numerical optimisation courses, is helpful in order to better understand the operation of much more complex procedures that are based on the same principle but at the same time are extremely more functional (with a faster rate of convergence and more robust).

The idea of this method is quite straightforward, in fact, \mathbf{p}_k is simply the negative gradient, the 'path' along which the function decreases locally.

Considering the principle behind this method, it is easy to understand that, by following this method iteratively, a local minimum has been reached, which is not necessarily also the global minimum of a function, it all depends on the point from which the search starts.

2) *Explanation of the Method:* The idea behind the method, as written above, is very simple, just select at each iteration a new direction \mathbf{p}_k ($\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$) and a step (α_k). The iteration can then be written in the following way:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k), \quad (3)$$

The main advantage of this algorithm is the low memory requirement, that can be approximated as $O(n)$, where n is the function dimension. At each iteration, the main computational work lies in the computation of the gradient \mathbf{p}_k , which can be performed in various ways, and, although to a lesser extent, in the search for the step length α_k .

C. Nonlinear conjugate gradient method with Polak-Ribière

1) *Introduction of the Method:* As shown, the steepest descent method returns the *locally* steepest descent direction as the result. In contrast, the conjugate gradient method \mathbf{p}_k always considers the steepest descent direction, but this time remembering the previous iterations.

In fact, the main difference between the steepest descent method and the conjugate gradient method lies in the selection of directions. In fact, now the directions are selected to be A-conjugate to the previous ones. In this way a faster descent is ensured by moving along the directions that satisfy:

$$\mathbf{d}_k^T \mathbf{A} \mathbf{d}_{k+1} = 0 \quad (4)$$

Now let's consider the following corollaries:

Corollary 2.0.1 (conjugacy and independence): if vectors $\mathbf{d}_0, \mathbf{d}_1, \dots, \mathbf{d}_k$ form a A-conjugate set, they are also independent.

Corollary 2.0.2 (transitivity): if a vector \mathbf{a} is A-conjugate with respect to \mathbf{b} , and \mathbf{b} is A-conjugate with respect to \mathbf{c} , then \mathbf{a} will also be A-conjugate with respect to \mathbf{c} .

For 2.0.1, in theory, considering a function of dimension n , at most n different conjugate-vectors are sufficient to be able to reach the minimum, avoiding the repeated choice of suboptimal sub-solutions as may occur in steepest descent. However, in a real application, given machine-related errors, more than n iterations may be necessary in some cases.

2) *Explanation of the Method:* As mentioned above, the principle is the same as for steepest descent, only the way in which the direction vector \mathbf{p}_k is selected varies:

$$\mathbf{p}_{k+1} = -\nabla f(\mathbf{x}_{k+1}) + \beta_{k+1}^{pr} \mathbf{p}_k, \quad (5)$$

and in the particular case of Polak-Ribière method β_{k+1} is:

$$\beta_{k+1}^{pr} = \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\|\nabla f_k\|^2} \quad (6)$$

The cost of each iteration is similar to that of the steepest descent since it uses the same gradient several times, thus not making the following method computationally expensive.

On the other hand, from the point of view of the storage occupied this time it will be necessary to consider a memory space equal to $O(2n) = O(n)$. In fact, in the conjugate gradient method, it is necessary to save the current and previous vector at each iteration to test conjugacy. In fact, thanks to the property 2.0.2, it is possible to test conjugacy only with the last vector to ensure it is with all previously selected vectors.

Unlike the steepest descent method, however, this converges in fewer iterations due to the property and intelligent use of direction choice.

III. BACKTRACKING LINE SEARCH

As mentioned earlier, α_k is the length of our step, which must be chosen carefully. In fact, by going to select values of α_k that are too large, a $f(\mathbf{x}_{k+1}) > f(\mathbf{x}_k)$ going beyond the descending part of the function could be obtained. On the other hand, by choosing values of α_k that are too small the converge could be too slow.

Therefore it is necessary to choose for each iteration an α_k such that sufficient descent for f is guaranteed. A popular condition, also used in the following paper, is the following:

A. 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:

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f(\mathbf{x}_k)^T \mathbf{p}_k, \quad (7)$$

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

The idea behind this method is quite simple. We denote the left-hand-side of (7) 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 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 (7), 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 $\alpha_k = 1$, 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 $\rho \in (0, 1)$.

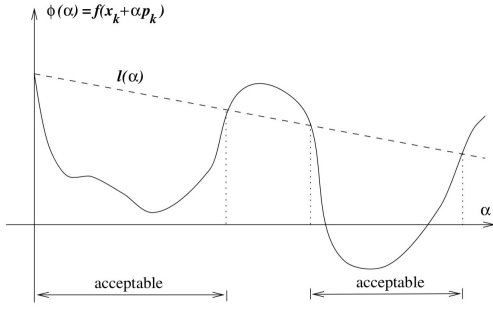


Fig. 1. Example of the Armijo condition.

IV. PROBLEM SETTING

A. Test on the Rosenbrock function

Consider the following function:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \quad (8)$$

starting from the following points:

$$x_0 = (1.2, 1.2), \quad x_0 = (-1.2, 1), \quad (9)$$

In the tests performed for the following function was decided, being the computation fast having a size $n = 2$, to have a $k_{max} = 50000$, a $bt_{max} = 50$ and a tolerance of 10^{-12} for achieving convergence.

k_{max} is the maximum number of iterations of the method, and bt_{max} is the maximum value of iterations for the steplength evaluation in the Armijo condition.

In addition, a tuning of the parameters c and ρ was performed, in order to see the best results obtainable for each of the two methods. In particular, the values considered are as follows:

```
1 c: [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
2 rho: [0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
```

The results for each best set of parameters for each of the two methods is described in the table I in the appendix.

For each of the initial points, the value of c and ρ that best approximates the minimum using the method was found. In particular, it is possible to see how in each case the conjugate gradient method achieves extremely more efficient and accurate results, coming closest to the function minimum in a much smaller number of iterations.

In Fig. 2, 3 and 4, the results obtained can be observed graphically. In particular is possible to see how the number of points through which the conjugate gradient method passes to reach the minimum is far less than the steepest descent, which simply chooses the direction with the lowest gradient each time.

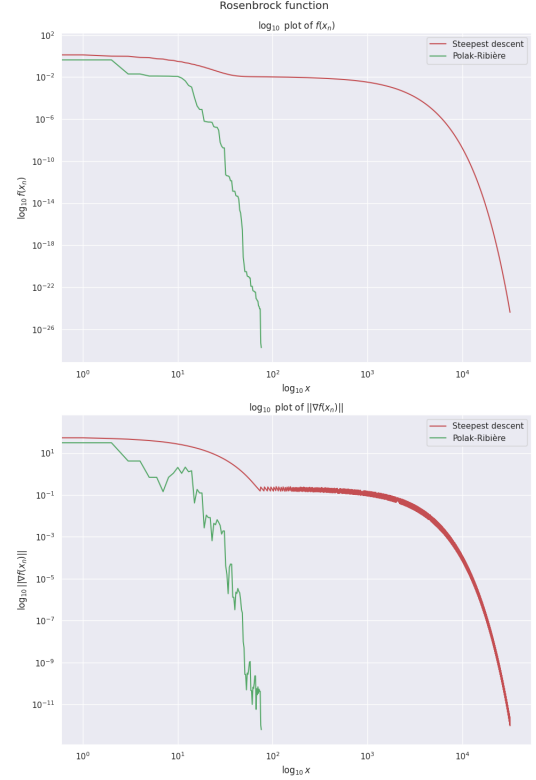


Fig. 2. Plot of convergence of $f(x_k)$ and $\|\nabla f(x_k)\|$ in Rosenbrock function starting from $x_0 = (1.2, 1.2)$.

B. Evaluation of the rate of convergence

For point $x_0 = (-1.2, 1)$ of the Rosenbrock function, also the rate of convergence was evaluated. Since the exact result, although known, was not present in the paper used, the method of calculating the rate of convergence was used assuming that the exact error (and thus the value sought by our solution) was not known.

The formula used when the exact solution is known is the following [2]:

$$p = \frac{\ln \frac{\|e_{k+1}\|}{\|e_k\|}}{\ln \frac{\|e_k\|}{\|e_{k-1}\|}}. \quad (10)$$

When the true error is not known, it is possible to surrogate it by $x_{k+1} - x_k$ obtaining the following [2]:

$$p = \frac{\ln \frac{\|x_{k+2} - x_{k+1}\|}{\|x_{k+1} - x_k\|}}{\ln \frac{\|x_{k+1} - x_k\|}{\|x_k - x_{k-1}\|}}. \quad (11)$$

Using the formula just described above, the results obtained were far from those expected. The reason is probably related

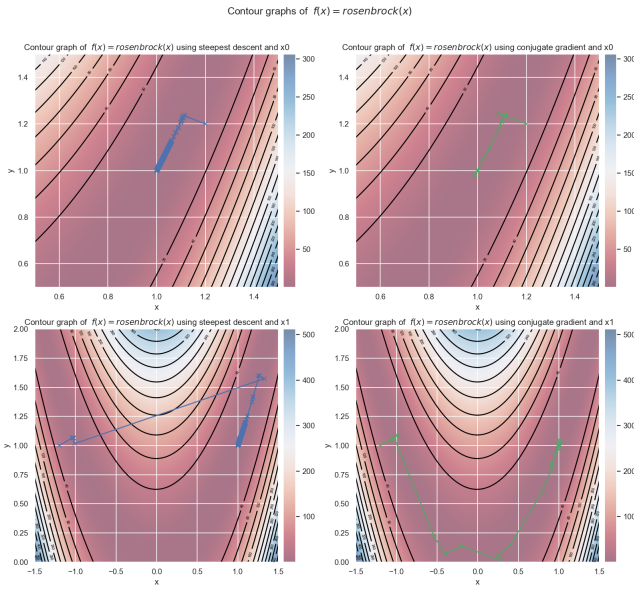


Fig. 3. Paths selected by the two methods on the contour plot.

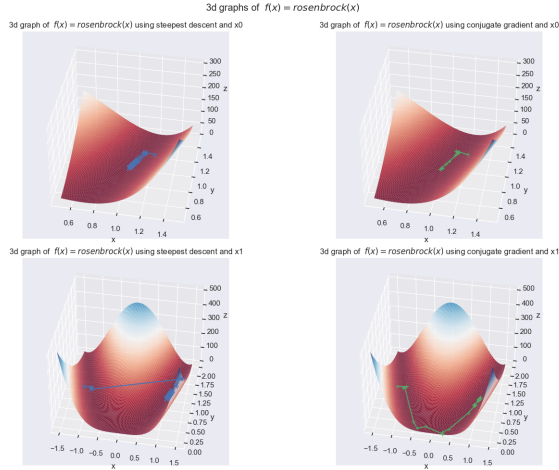


Fig. 4. Paths selected by the two methods on the 3d plot.

to the noise between iterations which led to steep changes in the value of x_k making p very variable in some iterations, especially when approaching the solution. It is possible to fully understand the problem by looking at the graph of solution attainment in Fig. 2.

For this reason, the evaluation of the rate of convergence was tested by taking into consideration only the first iterations, those that follow a more linear convergence. Advancing in this way actually led to good results, in fact considering only the first $k = 10$ iterations, the following results were obtained:

```
1 p of steepest descent method: 1.8619135773310265
2 p of conjugate gradient method: 1.0726581622394569
```

Since the result obtained with the steepest descent was not expected, considering that the value should be closer to 1 (linear convergence), the same test was performed with the

exact error.

```
1 p of steepest descent method using exact error
method: 1.1010415186256508
```

Using this test, it is possible to observe how using the formula with the exact error not only leads to a result closer to the real one, but this can be achieved by evaluating p on all points obtained in the steepest descent method, and not only in the first iterations.

C. Test on the Test Problems

From the set of problems proposed within the document [4] the following functions were selected:

1) *Problem 16. Banded trigonometric problem:*

$$F(\mathbf{x}) = \sum_{k=1}^n i(1 - \cos x_k + \sin x_{k-1} - \sin x_{k+1}), \quad (12)$$

$$x_0 = x_{n+1} = 0. \quad (13)$$

2) *Problem 25. Extended Rosenbrock function:*

$$F(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^n f_k^2(\mathbf{x}), \quad (14)$$

$$f_k(\mathbf{x}) = 10(x_k^2 - x_{k+1}), \quad \text{mod}(k, 2) = 1 \quad (15)$$

$$f_k(\mathbf{x}) = x_{k-1} - 1, \quad \text{mod}(k, 2) = 0. \quad (16)$$

3) *Problem 26. Extended Powell singular function:*

$$F(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^n f_k^2(\mathbf{x}), \quad (17)$$

$$f_k(\mathbf{x}) = x_k + x_{k+1}, \quad \text{mod}(k, 4) = 1 \quad (18)$$

$$f_k(\mathbf{x}) = \sqrt{5}(x_{k+1} - x_{k+2}), \quad \text{mod}(k, 4) = 2 \quad (19)$$

$$f_k(\mathbf{x}) = (x_{k-1} - 2x_k)^2, \quad \text{mod}(k, 4) = 3 \quad (20)$$

$$f_k(\mathbf{x}) = \sqrt{10}(x_{k-3} - x_k), \quad \text{mod}(k, 4) = 0. \quad (21)$$

The wording of this function has been slightly changed from the original [4] in order to solve problems with access to points that may not exist. In order to do this, the document reference [5] was used.

For the second set of problems, the values of tolgrad ($= 10^{-12}$), ρ ($= 0.5$), c ($= 10^{-4}$), and bt_{\max} ($= 50$) were kept unchanged. On the other hand, the k_{\max} value was varied from 50000 to 5000, as the tests were much longer to process, this time no longer having a function with dimension $n = 2$, but with $n = 10^5$.

As requested in the delivery, 10 different points of size $n = 10^5$ were taken with the following function " $x0_{\text{array}} = \text{np.random.randint}(1, 10, \text{size} = (10, 10000))$ ", which generates for each coordinate a random number between 1 and 9.

After generating the set of 10 points, these were used to perform tests with the steepest descent method and the conjugate gradient method, and the results were collected and put in Tables II, III, and IV.

From the tables, it can be seen that neither method achieves convergence considering the selected tolgrad (which is very low) for the first two functions, while for the third the conjugate gradient method is able to achieve the convergence.

On the other hand, even without achieving convergence, some interesting results are still obtained for analysis. In fact, as can also be seen in Fig. 5, 6 and 7, the conjugate gradient method achieves results that are much more promising with less iterations and better approximate the minimum than the steepest descent.

It is emphasised that the minimum $f(\mathbf{x}_{k_{max}})$ to be reached for the banded trigonometric function must be equal to $2m\pi$, with $m \in \mathbb{Z}$ (in our case for the mean of $f(\mathbf{x}_{k_{max}})$ is $m \approx -662$).

Next, in Fig. 5, 6 and 7, the plot of the convergence of $f(\mathbf{x}_{k_{max}})$ and $\|\nabla f(\mathbf{x}_k)\|$ considering the last point taken for each of the three functions (which is the same).

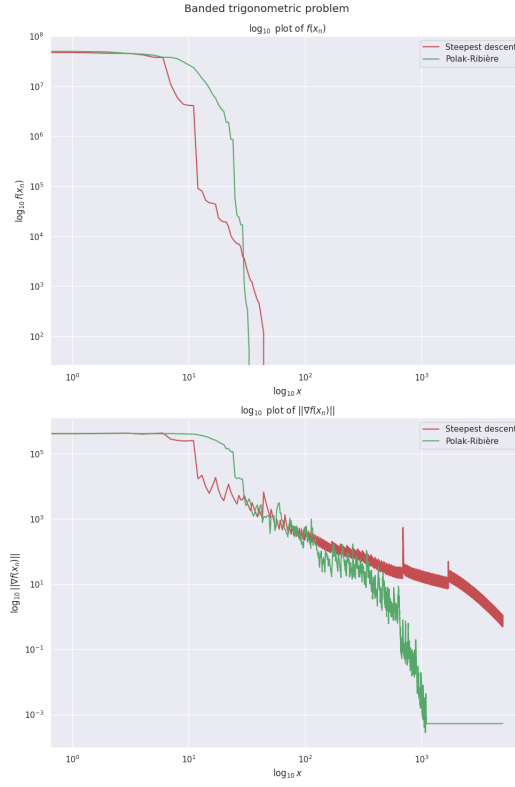


Fig. 5. Plot of convergence of $f(x_{k_{max}})$ and $\|\nabla f(x_k)\|$ in Banded trigonometric problem.

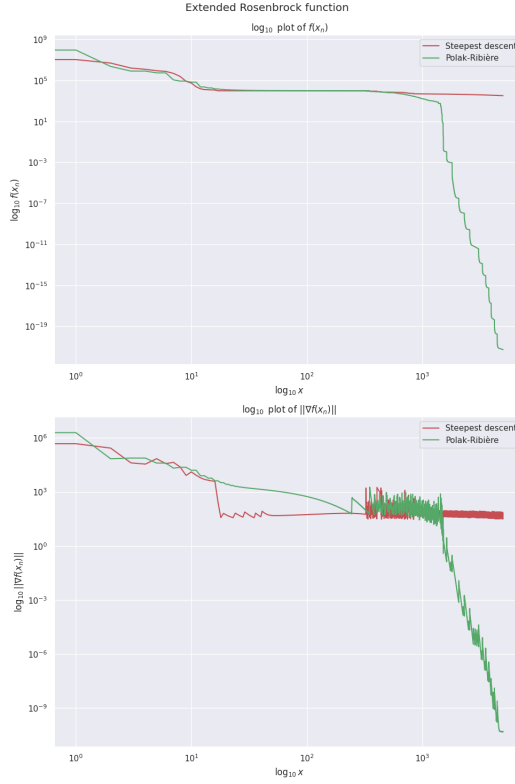


Fig. 6. Plot of convergence of $f(x_{k_{max}})$ and $\|\nabla f(x_k)\|$ in Extended Rosenbrock function.

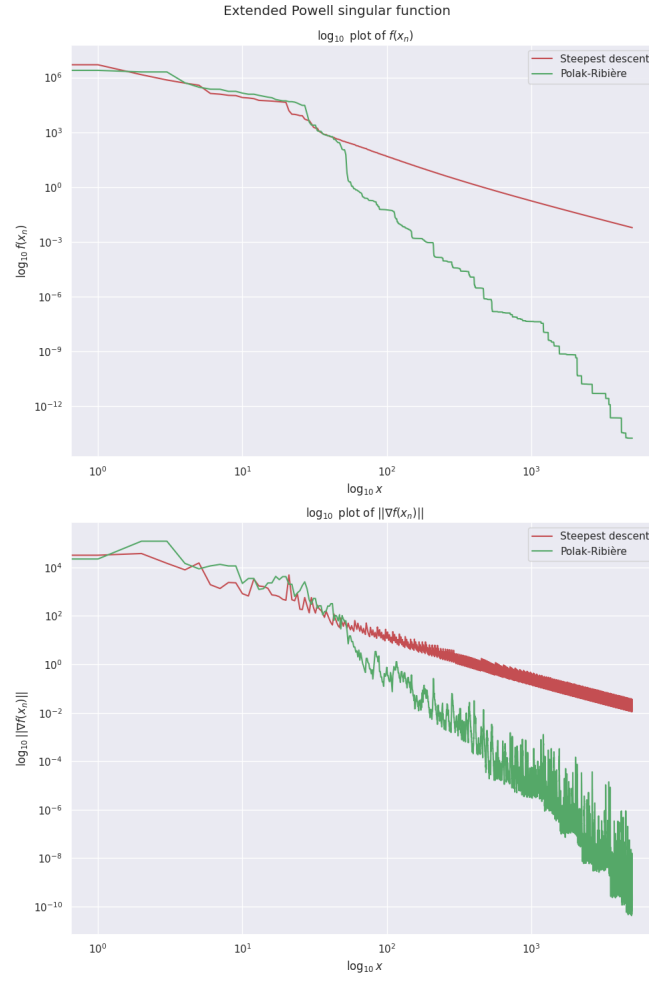


Fig. 7. Plot of convergence of $f(\mathbf{x}_{k_{max}})$ and $\|\nabla f(\mathbf{x}_k)\|$ in Extended Powell singular function.

APPENDIX

TABLE I
RESULTS OF ROSENBOCK FUNCTION.

\mathbf{x}_0	Method	c	ρ	$\mathbf{x}_{k_{max}}$	k	$f(\mathbf{x}_{k_{max}})$
[1.2, 1.2]	Steepest descent*	0.001	0.7	[1.0000000000006455, 1.0000000000012919]	31884	4.16728049830837e-25
[1.2, 1.2]	Polak-Ribière	0.001	0.7	[1.0000000000006075, 1.0000000000012192]	7752	3.708531750904866e-25
[1.2, 1.2]	Steepest descent	0.1	0.5	[1.0000000000008011, 1.0000000000016038]	27174	1.8148731200740903e-28
[1.2, 1.2]	Polak-Ribière*	0.1	0.5	[1.000000000000002, 1.000000000000053]	76	1.8148731200740903e-28
[-1.2, 1.0]	Steepest descent*	0.001	0.7	[1.0000000000006455, 1.0000000000012919]	35362	4.16728049830837e-25
[-1.2, 1.0]	Polak-Ribière	0.001	0.7	[1.0000000000002418, 1.000000000000486]	9121	5.906699565836136e-26
[-1.2, 1.0]	Steepest descent	0.01	0.5	[0.999999999992051, 0.999999999984085]	30571	6.321746405166527e-25
[-1.2, 1.0]	Polak-Ribière*	0.01	0.5	[1.0000000000000016, 1.000000000000003]	3835	2.4158865222393487e-30

*Best set of parameters for the following method at the selected point.

TABLE II
RESULTS OF BANDED TRIGONOMETRIC FUNCTION.

Method	k_{mean}	$\nabla(f(\mathbf{x}_{k_{max}}))$ (min, mean, max)	$f(\mathbf{x}_{k_{max}})$ (min, mean, max)
Steepest descent	5000.0	0.5666085576626245, 1.6250297170391028, 3.5193510223512705	-4159.9204812468615, -4159.006685527123, -1
Polak-Ribière	5000.0	7.858365941975501e-05, 0.00043952233670836065, 0.0008632744924607799	-4159.932447906154, -4159.932447902696, -1

TABLE III
RESULTS OF EXTENDED ROSENBOCK FUNCTION.

Method	k_{mean}	$\nabla(f(\mathbf{x}_{k_{max}}))$ (min, mean, max)	$f(\mathbf{x}_{k_{max}})$ (min, mean, max)
Steepest descent	5000.0	33.55722420022059, 47.50912801951978, 66.1652979565404	3204.5806992971775, 3805.470229207858, 4689.500499044189
Polak-Ribière	5000.0	1.9045891357765865e-12, 9.21666094858238e-09, 9.040049603455672e-08	1.8742656564413234e-25, 9.137386651613436e-17, 9.13710045174861e-16

TABLE IV
RESULTS OF EXTENDED POWELL FUNCTION.

Method	k_{mean}	$\nabla(f(\mathbf{x}_{k_{max}}))$ (min, mean, max)	$f(\mathbf{x}_{k_{max}})$ (min, mean, max)
Steepest descent	5000.0	0.011314446657241253, 0.015173309850936931, 0.022180116264650315	0.005939588885572014, 0.006007518319347166, 0.00605497879198791
Polak-Ribière	4888.7	9.64770409881263e-13, 3.4036168661708586e-08, 2.123968474566715e-07	9.522565612597491e-18, 3.775404270931218e-14, 2.496924078509818e-13

ACKNOWLEDGMENT

An in-depth analysis of the two methods showed how much more efficient the conjugate gradient method is in several respects than the steepest descent method.

In fact, as can be seen more clearly in the first part of the report, when the Rosenbrock function is analysed, the number of iterations required to reach convergence is much lower in the case of the conjugate gradient, reaching a decrease of up to the order of 0.002 compared to the other method.

Moreover, even once convergence is reached, the results achieved by the latter are more accurate by several orders of magnitude, both with the same number of parameters selected and, above all, by selecting the best parameters for each of the two methods.

Turning instead to the second part of the paper, at first glance, the differences between the two methods seem less obvious, since the conjugate gradient method fails to achieve convergence in two out of three cases (as opposed to three out of three for the steepest descent) in the set number of iterations.

On the other hand, however, it is essential to emphasise again how the result obtained at the last iteration approximates the minimum of the function much better than the steepest descent.

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] Juan C. Meza, "Steepest Descent", Lawrence Berkeley National Laboratory Berkeley, California 94720.
- [4] Ladislav Luksan, Jan Vlček, "Test Problems for Unconstrained Optimization", https://www.researchgate.net/publication/325314497_Test_Problems_for_Unconstrained_Optimization.
- [5] Moré, J.J., Garbow, B.S., Hillstöm, K.E., Testing Unconstrained Optimization Software, ACM Transactions on Mathematical Software, Vol. 7, pp. 17-41, 1981.

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 steepest_descent_bcktrck as sdb
8 from steepest_descent_bcktrck import *
9 import cgm_pol_rib as cgmpb
10 from cgm_pol_rib import *
11 import functions as funcs
12 from functions import *
13 from sklearn.model_selection import ParameterGrid
14 from importlib import reload
15
16 reload(funcs)
17 reload(sdb)
18 reload(cgmpb)
19
20 """Evaluate rate of convergence"""
21
22 def rate_of_convergence(x_seq: np.ndarray) -> float:
23     M = 10
24     m = 5
25     if (x_seq.shape[0] < m):
26         return None
27     if (x_seq.shape[0] <= M):
28         N = m
29     else:
30         N = M
31     p = np.empty((1,1))
32     for k in range(1, N+1):
33         if np.linalg.norm(x_seq[k+2, :] - x_seq[k+1, :], 2) != 0 and np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2) != 0 and \
34             np.linalg.norm(x_seq[k, :] - x_seq[k-1, :], 2) != 0 and \
35             log(np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2) / np.linalg.norm(x_seq[k, :] - x_seq[k-1, :], 2)) != 0:
36             pk = abs(log(np.linalg.norm(x_seq[k+2, :] - x_seq[k+1, :], 2) / np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2)) / \
37                 log(np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2) / np.linalg.norm(x_seq[k, :] - x_seq[k-1, :], 2)))
38             if k == 1:
39                 p[0] = pk
40             else:
41                 p = np.append(p, np.array([[pk]]))
42     return np.abs(p).mean()
43
44 """Evaluate exact rate of convergence"""
45
46 def exact_rate_of_convergence(x_seq: np.ndarray, x_star: np.ndarray) -> np.ndarray:
47     p = np.empty((1,1))
48     for k in range(1, x_seq.shape[0]-1):
49         if np.linalg.norm(x_seq[k+1, :] - x_star, 2) != 0 and np.linalg.norm(x_seq[k, :] - x_star, 2) != 0 \
50             and np.linalg.norm(x_seq[k-1, :] - x_star, 2) != 0 \
51             and log(np.linalg.norm(x_seq[k, :] - x_star, 2) / np.linalg.norm(x_seq[k-1, :] - x_star, 2)) != 0:
52             pk = log(np.linalg.norm(x_seq[k+1, :] - x_star, 2) / np.linalg.norm(x_seq[k, :] - x_star, 2)) / \
53                 log(np.linalg.norm(x_seq[k, :] - x_star, 2) / np.linalg.norm(x_seq[k-1, :] - x_star, 2))
54             if k == 1:
55                 p[0] = pk
56             else:
57                 p = np.append(p, np.array([[pk]]))
58     return p
59
60 """Initialisation of the data"""
61
62 x0 = np.array([1.2, 1.2])
63 x1 = np.array([-1.2, 1])

```

```

64 alpha0 = 1
65 tolgrad = 1e-12
66 rho = 0.5
67 c = 1e-4
68 kmax = 50000
69 btmax = 50
70 fin_diff = False
71 fd_type = 'centered'
72
73 params = {"c": [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6],
74           "rho": [0.2, 0.3, 0.4, 0.5, 0.6, 0.7]}
75
76 """Let's test the Steepest descent and conjugate gradient method with Backtrack tuning the parameters c and
77 rho using the Armijo condition.
78 Here our idea is quite simple. In order to make a complete comparison between the two methods the best set
79 of parameters for both methods is evaluated, once this is done a comparison is made with the best
80 result obtained in one of the two methods with one set and the same set is also used for the other
81 method (we will therefore have in all 2 saved evaluations for steepest descent, one with the best set
82 of parameters, and the other with the best set of parameters of the conjugate gradient method, and 2
83 saved evaluations for the conjugate gradient method)
84
85 """
86 #best result for steepest descent method is saved here
87 sd_best_fk_x0 = np.finfo(float).max
88 sd_best_param_x0 = 0
89 sd_best_k_x0 = 0
90 sd_best_gradfk_norm_x0 = 10
91 sd_best_xk_x0 = np.empty(2)
92 sd_best_x_seq_x0 = np.empty(2)
93 sd_best_f_seq_x0 = np.empty(2)
94
95 #best result for conjugate gradient method is saved here
96 cg_best_fk_x0 = np.finfo(float).max
97 cg_best_param_x0 = 0
98 cg_best_k_x0 = 0
99 cg_best_gradfk_norm_x0 = 10
100 cg_best_xk_x0 = np.empty(2)
101 cg_best_x_seq_x0 = np.empty(2)
102 cg_best_f_seq_x0 = np.empty(2)
103
104 #Other result of steepest descent method is saved here
105 sd_fk_x0 = np.finfo(float).max
106 sd_param_x0 = 0
107 sd_k_x0 = 0
108 sd_xk_x0 = np.empty(2)
109
110 #Other result of conjugate gradient method is saved here
111 cg_f_seq_x0 = np.finfo(float).max
112 cg_param_x0 = 0
113 cg_k_x0 = 0
114 cg_xk_x0 = np.empty(2)
115
116 for param in ParameterGrid(params):
117     # Method for the steepest descent
118     sd_x_seq, sd_f_seq, sd_gradf_norm_seq, sd_k, sd_bt_seq = steepest_descent_bcktrck(x0, 'Rosenbrock',
119     alpha0, kmax, tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)
120
121     # Method for the conjugate gradient method
122     cg_x_seq, cg_f_seq, cg_gradf_norm_seq, cg_k, cg_bt_seq = cgm_pol_rib(x0, 'Rosenbrock', alpha0, kmax,
123     tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)
124
125     #For saving the best result of the steepest descent, and the correspondent result of the conjugate
126     gradient method
127     if (sd_f_seq[-1] < sd_best_fk_x0):
128         sd_best_param_x0 = param
129
130         sd_best_xk_x0 = sd_x_seq[-1]
131         sd_best_k_x0 = sd_k
132         sd_best_fk_x0 = sd_f_seq[-1]
133         sd_best_gradfk_norm_x0 = sd_gradf_norm_seq[-1]
134         sd_best_x_seq_x0 = np.copy(sd_x_seq)
135         sd_best_f_seq_x0 = np.copy(sd_f_seq)

```

```

129     sd_best_gradf_norm_seq_x0 = np.copy(sd_gradf_norm_seq)
130
131     cg_xk_x0 = cg_x_seq[-1]
132     cg_k_x0 = cg_k
133     cg_fk_x0 = cg_f_seq[-1]
134
135     #For saving the best result of the conjugate gradient, and the correspondent result of the steepest
    descent method
136     if (cg_f_seq[-1] < cg_best_fk_x0):
137         cg_best_param_x0 = param
138
139         cg_best_xk_x0 = cg_x_seq[-1]
140         cg_best_k_x0 = cg_k
141         cg_best_fk_x0 = cg_f_seq[-1]
142         cg_best_gradfk_norm_x0 = cg_gradf_norm_seq[-1]
143         cg_best_x_seq_x0 = np.copy(cg_x_seq)
144         cg_best_f_seq_x0 = np.copy(cg_f_seq)
145         cg_best_gradf_norm_seq_x0 = np.copy(cg_gradf_norm_seq)
146
147         sd_xk_x0 = sd_x_seq[-1]
148         sd_k_x0 = sd_k
149         sd_fk_x0 = sd_f_seq[-1]
150
151     """Print the result obtained """
152
153     print("Best evaluation with steepest descent method:")
154     print("x0: ", x0[0], x0[1])
155     print("best set of parameters: ", sd_best_param_x0)
156     print("xk: ", sd_best_xk_x0[0], sd_best_xk_x0[1])
157     print("k: ", sd_best_k_x0)
158     print("fk: ", sd_best_fk_x0)
159
160     print("\nEvaluation of the conjugate gradient method using the best parameters of steepest descent: ")
161     print("x0: ", x0[0], x0[1])
162     print("set of parameters: ", sd_best_param_x0)
163     print("xk: ", cg_xk_x0[0], cg_xk_x0[1])
164     print("k: ", cg_k_x0)
165     print("fk: ", cg_fk_x0)
166
167     print("Best evaluation with conjugate gradient method:")
168     print("x0: ", x0[0], x0[1])
169     print("best set of parameters: ", cg_best_param_x0)
170     print("xk: ", cg_best_xk_x0[0], cg_best_xk_x0[1])
171     print("k: ", cg_best_k_x0)
172     print("fk: ", cg_best_fk_x0)
173
174     print("\nEvaluation of the steepest descent method using the best parameters of conjugate gradient: ")
175     print("x0: ", x0[0], x0[1])
176     print("set of parameters: ", cg_best_param_x0)
177     print("xk: ", sd_xk_x0[0], sd_xk_x0[1])
178     print("k: ", sd_k_x0)
179     print("fk: ", sd_fk_x0)
180
181     """Creation of plots in the report showing the values of f(\bm{x}) and gradf(\bm{x}) for both methods. """
182
183     sns.set()
184     fig_ep, ax_ep = plt.subplots(2, 1, figsize=(10, 15))
185     ax_ep[0].plot(sd_best_f_seq_x0, '-r', label='Steepest descent')
186     ax_ep[0].plot(cg_best_f_seq_x0, '-g', label='Polak-Ribière')
187
188     ax_ep[0].set_xscale('log', base=10)
189     ax_ep[0].set_yscale('log', base=10)
190     ax_ep[0].set_xlabel(r'$\log_{10}\{x\}$')
191     ax_ep[0].set_ylabel(r'$\log_{10}\{f(\{x_n\})\}$')
192     ax_ep[0].set_title(r'$\log_{10}$ plot of $f(\{x_n\})$')
193
194     ax_ep[0].legend()
195
196     ax_ep[1].plot(sd_best_gradf_norm_seq_x0, '-r', label='Steepest descent')
197     ax_ep[1].plot(cg_best_gradf_norm_seq_x0, '-g', label='Polak-Ribière')
198
199     ax_ep[1].set_xscale('log', base=10)
200     ax_ep[1].set_yscale('log', base=10)
201     ax_ep[1].set_xlabel(r'$\log_{10}\{x\}$')

```

```

202 ax_ep[1].set_ylabel(r'$\log_{10}(||\nabla f(\{x_n\})|)|$')
203 ax_ep[1].set_title(r'$\log_{10}$ plot of $||\nabla f(\{x_n\})||$')
204
205 ax_ep[1].legend()
206 fig_ep.tight_layout()
207 fig_ep.suptitle('Rosenbrock function')
208 fig_ep.subplots_adjust(top=0.94)
209
210 """Now let's do the same but using the other starting point (x1)
211
212 *From point x1 we will also try to evaluate the rate of convergence p
213 """
214
215 #best result for steepest descent method is saved here
216 sd_best_fk_x1 = np.finfo(float).max
217 sd_best_param_x1 = 0
218 sd_best_k_x1 = 0
219 sd_best_gradfk_norm_x1 = 10
220 sd_best_xk_x1 = np.empty(2)
221 sd_best_x_seq_x1 = np.empty(2)
222 sd_best_f_seq_x1 = np.empty(2)
223
224 #best result for conjugate gradient method is saved here
225 cg_best_fk_x1 = np.finfo(float).max
226 cg_best_param_x1 = 0
227 cg_best_k_x1 = 0
228 cg_best_gradfk_norm_x1 = 10
229 cg_best_xk_x1 = np.empty(2)
230 cg_best_x_seq_x1 = np.empty(2)
231 cg_best_f_seq_x1 = np.empty(2)
232
233 #Other result of steepest descent method is saved here
234 sd_fk_x1 = np.finfo(float).max
235 sd_param_x1 = 0
236 sd_k_x1 = 0
237 sd_xk_x1 = np.empty(2)
238
239 #Other result of conjugate gradient method is saved here
240 cg_fk_x1 = np.finfo(float).max
241 cg_param_x1 = 0
242 cg_k_x1 = 0
243 cg_xk_x1 = np.empty(2)
244
245 for param in ParameterGrid(params):
246     sd_x_seq, sd_f_seq, sd_gradf_norm_seq, sd_k, sd_bt_seq = steepest_descent_bcktrck(x1, 'Rosenbrock',
247     alpha0, kmax, tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)
248
249     cg_x_seq, cg_f_seq, cg_gradf_norm_seq_seq, cg_k, cg_bt_seq = cgm_pol_rib(x1, 'Rosenbrock', alpha0, kmax
250     , tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)
251
252     if (sd_f_seq[-1] < sd_best_fk_x1):
253         sd_best_param_x1 = param
254
255         sd_best_xk_x1 = sd_x_seq[-1]
256         sd_best_k_x1 = sd_k
257         sd_best_fk_x1 = sd_f_seq[-1]
258         sd_best_gradfk_norm_x1 = sd_gradf_norm_seq[-1]
259         sd_best_x_seq_x1 = np.copy(sd_x_seq)
260         sd_best_f_seq_x1 = np.copy(sd_f_seq)
261
262         cg_xk_x1 = cg_x_seq[-1]
263         cg_k_x1 = cg_k
264         cg_fk_x1 = cg_f_seq[-1]
265
266     if (cg_f_seq[-1] < cg_best_fk_x1):
267         cg_best_param_x1 = param
268
269         cg_best_xk_x1 = cg_x_seq[-1]
270         cg_best_k_x1 = cg_k
271         cg_best_fk_x1 = cg_f_seq[-1]
272         cg_best_gradfk_norm_x1 = cg_gradf_norm_seq_seq[-1]
273         cg_best_x_seq_x1 = np.copy(cg_x_seq)
274         cg_best_f_seq_x1 = np.copy(cg_f_seq)

```

```

274     sd_xk_x1 = sd_x_seq[-1]
275     sd_k_x1 = sd_k
276     sd_fk_x1 = sd_f_seq[-1]
277
278 print("Best evaluation with steepest descent method:")
279 print("x1: ", x1[0], x1[1])
280 print("best set of parameters: ", sd_best_param_x1)
281 print("xk: ", sd_best_xk_x1[0], sd_best_xk_x1[1])
282 print("k: ", sd_best_k_x1)
283 print("fk: ", sd_best_fk_x1)
284
285 print("\nEvaluation of the conjugate gradient method using the best parameters of steepest descent: ")
286 print("x1: ", x1[0], x1[1])
287 print("set of parameters: ", sd_best_param_x1)
288 print("xk: ", cg_xk_x1[0], cg_xk_x1[1])
289 print("k: ", cg_k_x1)
290 print("fk: ", cg_fk_x1)
291
292 print("Best evaluation with conjugate gradient method:")
293 print("x1: ", x1[0], x1[1])
294 print("best set of parameters: ", cg_best_param_x1)
295 print("xk: ", cg_best_xk_x1[0], cg_best_xk_x1[1])
296 print("k: ", cg_best_k_x1)
297 print("fk: ", cg_best_fk_x1)
298
299 print("\nEvaluation of the steepest descent method using the best parameters of conjugate gradient: ")
300 print("x1: ", x1[0], x1[1])
301 print("set of parameters: ", cg_best_param_x1)
302 print("xk: ", sd_xk_x1[0], sd_xk_x1[1])
303 print("k: ", sd_k_x1)
304 print("fk: ", sd_fk_x1)
305
306 """Evaluation of the rate of convergence p"""
307
308 if (sd_best_gradfk_norm_x1 <= tolgrad):
309     #It make sense to evaluate the rate of convergence if and only if I've reached the solution
310     \bm{p}_sd = rate_of_convergence(sd_best_x_seq_x1)
311     print("Evaluation of rate of convergence for steepest descent method:")
312     print(f"\bm{p}_sd = {\bm{p}_sd}")
313
314 """Evaluation of rate of convergence for steepest descent method:
315
316 \bm{p}_sd = 1.8619135773310265
317 """
318
319 if (cg_best_gradfk_norm_x1 <= tolgrad):
320     #It make sense to evaluate the rate of convergence if and only if I've reached the solution
321     \bm{p}_cg = rate_of_convergence(cg_best_x_seq_x1)
322     print("Evaluation of rate of convergence for conjugate gradient method:")
323     print(f"\bm{p}_cg = {\bm{p}_cg}")
324
325 """Evaluation of rate of convergence for conjugate gradient method:
326
327 \bm{p}_cg = 1.0726581622394569
328
329 Since we were not satisfied with the result obtained with the steepest descent, as we imagined a value
    closer to 1, we performed the same test but with the exact error.
330 """
331
332 x_star = np.array([1, 1]) #Exact solution
333
334 \bm{p}_sd_exact = exact_rate_of_convergence(sd_x_seq, x_star)
335
336 print("Evaluation of exact rate of convergence for steepest descent method:")
337 print(f"\bm{p}_sd_exact = {\bm{p}_sd_exact.mean()}")
338
339 """Evaluation of rate exact of convergence for steepest descent method:
340
341 \bm{p}_sd_exact: 1.1010415186256508
342
343 Contour graph for the Rosenbrock function that shows the path followed by both methods (The result is
    available in the report)
344 """
345

```

```

346 x = np.linspace(0.5, 1.5, 200)
347 y = np.linspace(0.5, 1.5, 200)
348 X, Y = np.meshgrid(\bm{x}, y)
349 Z = rosenbrock(\bm{x}, Y)
350
351 sns.set()
352 fig_rb, ax_rb = plt.subplots(2, 2, figsize=(13, 12))
353
354 divider = make_axes_locatable(ax_rb[0, 0])
355 cax = divider.append_axes('right', size='5%', pad=0.1)
356 contours = ax_rb[0, 0].contour(\bm{x}, Y, Z, 15, colors='black')
357 ax_rb[0, 0].clabel(contours, inline=True, fontsize=5)
358 im = ax_rb[0, 0].imshow(Z, extent=[0.5, 1.5, 0.5, 1.5], origin='lower', cmap='RdBu', alpha=0.5, aspect='
    auto')
359 ax_rb[0, 0].plot(sd_best_x_seq_x0[:, 0], sd_best_x_seq_x0[:, 1], '-xb')
360 ax_rb[0, 0].set_xlabel('x')
361 ax_rb[0, 0].set_ylabel('y')
362 ax_rb[0, 0].set_title(r'Contour graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using steepest descent and x0')
363 fig_rb.colorbar(im, cax=cax, orientation='vertical')
364
365 divider = make_axes_locatable(ax_rb[0, 1])
366 cax = divider.append_axes('right', size='5%', pad=0.1)
367 contours = ax_rb[0, 1].contour(\bm{x}, Y, Z, 15, colors='black')
368 ax_rb[0, 1].clabel(contours, inline=True, fontsize=5)
369 im = ax_rb[0, 1].imshow(Z, extent=[0.5, 1.5, 0.5, 1.5], origin='lower', cmap='RdBu', alpha=0.5, aspect='
    auto')
370 ax_rb[0, 1].plot(cg_best_x_seq_x0[:, 0], cg_best_x_seq_x0[:, 1], '-xg')
371 ax_rb[0, 1].set_xlabel('x')
372 ax_rb[0, 1].set_ylabel('y')
373 ax_rb[0, 1].set_title(r'Contour graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using conjugate gradient and x0')
374 fig_rb.colorbar(im, cax=cax, orientation='vertical')
375
376 x = np.linspace(-1.5, 1.5, 200)
377 y = np.linspace(0, 2, 200)
378 X, Y = np.meshgrid(\bm{x}, y)
379 Z = rosenbrock(\bm{x}, Y)
380
381 divider = make_axes_locatable(ax_rb[1, 0])
382 cax = divider.append_axes('right', size='5%', pad=0.1)
383 contours = ax_rb[1, 0].contour(\bm{x}, Y, Z, 15, colors='black')
384 ax_rb[1, 0].clabel(contours, inline=True, fontsize=5)
385 im = ax_rb[1, 0].imshow(Z, extent=[-1.5, 1.5, 0, 2], origin='lower', cmap='RdBu', alpha=0.5, aspect='auto')
386 ax_rb[1, 0].plot(sd_best_x_seq_x1[:, 0], sd_best_x_seq_x1[:, 1], '-xb')
387 ax_rb[1, 0].set_xlabel('x')
388 ax_rb[1, 0].set_ylabel('y')
389 ax_rb[1, 0].set_title(r'Contour graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using steepest descent and x1')
390 fig_rb.colorbar(im, cax=cax, orientation='vertical')
391
392 divider = make_axes_locatable(ax_rb[1, 1])
393 cax = divider.append_axes('right', size='5%', pad=0.1)
394 contours = ax_rb[1, 1].contour(\bm{x}, Y, Z, 15, colors='black')
395 ax_rb[1, 1].clabel(contours, inline=True, fontsize=5)
396 im = ax_rb[1, 1].imshow(Z, extent=[-1.5, 1.5, 0, 2], origin='lower', cmap='RdBu', alpha=0.5, aspect='auto')
397 ax_rb[1, 1].plot(cg_best_x_seq_x1[:, 0], cg_best_x_seq_x1[:, 1], '-xg')
398 ax_rb[1, 1].set_xlabel('x')
399 ax_rb[1, 1].set_ylabel('y')
400 ax_rb[1, 1].set_title(r'Contour graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using conjugate gradient and x1')
401 fig_rb.colorbar(im, cax=cax, orientation='vertical')
402
403 fig_rb.tight_layout()
404 fig_rb.suptitle(r'Contour graphs of $f(\bm{x})=\text{rosenbrock}(\bm{x})$')
405 fig_rb.subplots_adjust(top=0.9)
406
407 """3D graph of the Rosenbrock function that shows the path followed by both methods (The result is
    available in the report)"""
408
409 x = np.linspace(0.5, 1.5, 200)
410 y = np.linspace(0.5, 1.5, 200)
411 X, Y = np.meshgrid(\bm{x}, y)
412 Z = rosenbrock(\bm{x}, Y)
413
414 sns.set()
415 fig3d = plt.figure(figsize=(15, 10))
416

```

```

417 ax = fig3d.add_subplot(2, 2, 1, projection='3d')
418 ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
419 ax.plot(sd_best_x_seq_x0[:, 0], sd_best_x_seq_x0[:, 1], sd_best_f_seq_x0, '--xb')
420 ax.set_xlabel('x')
421 ax.set_ylabel('y')
422 ax.set_zlabel('z')
423 ax.set_title(r'3d graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using steepest descent and x0')
424 ax.view_init(40, -80)
425
426 ax = fig3d.add_subplot(2, 2, 2, projection='3d')
427 ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
428 ax.plot(cg_best_x_seq_x0[:, 0], cg_best_x_seq_x0[:, 1], cg_best_f_seq_x0, '--+g')
429 ax.set_xlabel('x')
430 ax.set_ylabel('y')
431 ax.set_zlabel('z')
432 ax.set_title(r'3d graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using conjugate gradient and x0')
433 ax.view_init(40, -80)
434
435 x = np.linspace(-1.5, 1.5, 200)
436 y = np.linspace(0, 2, 200)
437 X, Y = np.meshgrid(\bm{x}, y)
438 Z = rosenbrock(\bm{x}, Y)
439
440 ax = fig3d.add_subplot(2, 2, 3, projection='3d')
441 ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
442 ax.plot(sd_best_x_seq_x1[:, 0], sd_best_x_seq_x1[:, 1], sd_best_f_seq_x1, '--xb')
443 ax.set_xlabel('x')
444 ax.set_ylabel('y')
445 ax.set_zlabel('z')
446 ax.set_title(r'3d graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using steepest descent and x1')
447 ax.view_init(40, -80)
448
449 ax = fig3d.add_subplot(2, 2, 4, projection='3d')
450 ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
451 ax.plot(cg_best_x_seq_x1[:, 0], cg_best_x_seq_x1[:, 1], cg_best_f_seq_x1, '--+g')
452 ax.set_xlabel('x')
453 ax.set_ylabel('y')
454 ax.set_zlabel('z')
455 ax.set_title(r'3d graph of $f(\bm{x})=\text{rosenbrock}(\bm{x})$ using conjugate gradient and x1')
456 ax.view_init(40, -80)
457
458 fig3d.tight_layout()
459 fig3d.suptitle(r'3d graphs of $f(\bm{x})=\text{rosenbrock}(\bm{x})$')
460 fig3d.subplots_adjust(top=0.92)
461
462 """Now let's work with the other functions and collect the data.
463
464 1. Extended Powell
465 """
466
467 kmax = 5000 #Change the kmax from 50000 to 5000
468 x0_array = np.random.randint(1, 10, size=(10, 10000))
469
470 """Steepest descent method """
471
472 #set of variables to save the results
473 kmean = 0
474 grad_norm_mean = 0
475 grad_norm_min = np.finfo(float).max
476 grad_norm_max = -1
477 fx_mean = 0
478 fx_min = np.finfo(float).max
479 fx_max = -1
480
481 for point in x0_array:
482     sd_x_seq_ep, sd_f_seq_ep, sd_gradf_norm_seq_ep, sd_k_ep, sd_bt_seq_ep = steepest_descent_bcktrck(point,
483     'Extended Powell', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
484     print("Result of steepest descent method:")
485     print("x0: ", point, " (length: ", len(point), ")")
486     print("k: ", sd_k_ep)
487     print("fk: ", sd_f_seq_ep[-1])
488     print("gradfk: ", sd_gradf_norm_seq_ep[-1])
489     print("\n")

```



```

490 kmean += sd_k_ep
491 grad_norm_mean += sd_gradf_norm_seq_ep[-1]
492 if grad_norm_max < sd_gradf_norm_seq_ep[-1]:
493     grad_norm_max = sd_gradf_norm_seq_ep[-1]
494 if grad_norm_min > sd_gradf_norm_seq_ep[-1]:
495     grad_norm_min = sd_gradf_norm_seq_ep[-1]
496 fx_mean += sd_f_seq_ep[-1]
497 if fx_max < sd_f_seq_ep[-1]:
498     fx_max = sd_f_seq_ep[-1]
499 if fx_min > sd_f_seq_ep[-1]:
500     fx_min = sd_f_seq_ep[-1]
501
502 """Print of the results obtained"""
503
504 kmean = kmean / len(x0_array)
505 grad_norm_mean = grad_norm_mean / len(x0_array)
506 fx_mean = fx_mean / len(x0_array)
507
508 print("mean_of_k: ", kmean)
509
510 print("\n")
511 print("min_of_grad_norm: ", grad_norm_min)
512 print("mean_of_grad_norm: ", grad_norm_mean)
513 print("max_of_grad_norm: ", grad_norm_max)
514
515 print("\n")
516 print("min_of_fx: ", fx_min)
517 print("mean_of_fx: ", fx_mean)
518 print("max_of_fx: ", fx_max)
519
520 """Conjugate gradient method"""
521
522 kmean = 0
523 grad_norm_mean = 0
524 grad_norm_min = np.finfo(float).max
525 grad_norm_max = -1
526 fx_mean = 0
527 fx_min = np.finfo(float).max
528 fx_max = -1
529
530 for point in x0_array:
531     cg_x_seq_ep, cg_f_seq_ep, cg_gradf_norm_seq_ep, cg_k_ep, cg_bt_seq_ep = cgm_pol_rib(point, 'Extended
Powell', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
532     print("Result of conjugate gradient method:")
533     print("x0: ", point, " (length: ", len(point), ")")
534     print("k: ", cg_k_ep)
535     print("fk: ", cg_f_seq_ep[-1])
536     print("gradfk: ", cg_gradf_norm_seq_ep[-1])
537     print("\n")
538
539     kmean += cg_k_ep
540     grad_norm_mean += cg_gradf_norm_seq_ep[-1]
541     if grad_norm_max < cg_gradf_norm_seq_ep[-1]:
542         grad_norm_max = cg_gradf_norm_seq_ep[-1]
543     if grad_norm_min > cg_gradf_norm_seq_ep[-1]:
544         grad_norm_min = cg_gradf_norm_seq_ep[-1]
545     fx_mean += cg_f_seq_ep[-1]
546     if fx_max < cg_f_seq_ep[-1]:
547         fx_max = cg_f_seq_ep[-1]
548     if fx_min > cg_f_seq_ep[-1]:
549         fx_min = cg_f_seq_ep[-1]
550
551 """Print of the results obtained"""
552
553 kmean = kmean / len(x0_array)
554 grad_norm_mean = grad_norm_mean / len(x0_array)
555 fx_mean = fx_mean / len(x0_array)
556
557 print("mean_of_k: ", kmean)
558
559 print("\n")
560 print("min_of_grad_norm: ", grad_norm_min)
561 print("mean_of_grad_norm: ", grad_norm_mean)
562 print("max_of_grad_norm: ", grad_norm_max)

```

```

563
564 print("\n")
565 print("min_of_fx: ", fx_min)
566 print("mean_of_fx: ", fx_mean)
567 print("max_of_fx: ", fx_max)
568
569 """Creation of plots in the report showing the values of  $f(\mathbf{b}_m(\mathbf{x}))$  and  $\text{grad}f(\mathbf{b}_m(\mathbf{x}))$  for both methods. """
570
571 sns.set()
572 fig_ep, ax_ep = plt.subplots(2, 1, figsize=(10, 15))
573 ax_ep[0].plot(sd_f_seq_ep, '-r', label='Steepest descent')
574 ax_ep[0].plot(cg_f_seq_ep, '-g', label='Polak-Ribière')
575
576 ax_ep[0].set_xscale('log', base=10)
577 ax_ep[0].set_yscale('log', base=10)
578 ax_ep[0].set_xlabel(r' $\log_{10}\{x\}$ ')
579 ax_ep[0].set_ylabel(r' $\log_{10}\{f(\mathbf{x}_n)\}$ ')
580 ax_ep[0].set_title(r' $\log_{10}$  plot of  $f(\mathbf{x}_n)$ ')
581
582 ax_ep[0].legend()
583
584 ax_ep[1].plot(sd_gradf_norm_seq_ep, '-r', label='Steepest descent')
585 ax_ep[1].plot(cg_gradf_norm_seq_ep, '-g', label='Polak-Ribière')
586
587 ax_ep[1].set_xscale('log', base=10)
588 ax_ep[1].set_yscale('log', base=10)
589 ax_ep[1].set_xlabel(r' $\log_{10}\{x\}$ ')
590 ax_ep[1].set_ylabel(r' $\log_{10}\{|\nabla f(\mathbf{x}_n)|\}$ ')
591 ax_ep[1].set_title(r' $\log_{10}$  plot of  $|\nabla f(\mathbf{x}_n)|$ ')
592
593 ax_ep[1].legend()
594 fig_ep.tight_layout()
595 fig_ep.suptitle('Extended Powell singular function')
596 fig_ep.subplots_adjust(top=0.94)
597
598 """The exact same steps were followed for the other two functions
599
600 2. Extended Rosenbrock
601 """
602
603 kmean = 0
604 grad_norm_mean = 0
605 grad_norm_min = np.finfo(float).max
606 grad_norm_max = -1
607 fx_mean = 0
608 fx_min = np.finfo(float).max
609 fx_max = -1
610
611 for point in x0_array:
612     sd_x_seq_er, sd_f_seq_er, sd_gradf_norm_seq_er, sd_k_er, sd_bt_seq_er = steepest_descent_bcktrck(point,
613     'Extended Rosenbrock', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
614     print("Result of steepest descent method:")
615     print("x0: ", point, " (length: ", len(point), ")")
616     print("k: ", sd_k_er)
617     print("fk: ", sd_f_seq_er[-1])
618     print("gradfk: ", sd_gradf_norm_seq_er[-1])
619     print("\n")
620
621     kmean += sd_k_er
622     grad_norm_mean += sd_gradf_norm_seq_er[-1]
623     if grad_norm_max < sd_gradf_norm_seq_er[-1]:
624         grad_norm_max = sd_gradf_norm_seq_er[-1]
625     if grad_norm_min > sd_gradf_norm_seq_er[-1]:
626         grad_norm_min = sd_gradf_norm_seq_er[-1]
627     fx_mean += sd_f_seq_er[-1]
628     if fx_max < sd_f_seq_er[-1]:
629         fx_max = sd_f_seq_er[-1]
630     if fx_min > sd_f_seq_er[-1]:
631         fx_min = sd_f_seq_er[-1]
632
633 kmean = kmean / len(x0_array)
634 grad_norm_mean = grad_norm_mean / len(x0_array)
635 fx_mean = fx_mean / len(x0_array)
636

```

```

636 print("mean_of_k: ", kmean)
637
638 print("\n")
639 print("min_of_grad_norm: ", grad_norm_min)
640 print("mean_of_grad_norm: ", grad_norm_mean)
641 print("max_of_grad_norm: ", grad_norm_max)
642
643 print("\n")
644 print("min_of_fx: ", fx_min)
645 print("mean_of_fx: ", fx_mean)
646 print("max_of_fx: ", fx_max)
647
648 kmean = 0
649 grad_norm_mean = 0
650 grad_norm_min = np.finfo(float).max
651 grad_norm_max = -1
652 fx_mean = 0
653 fx_min = np.finfo(float).max
654 fx_max = -1
655
656 for point in x0_array:
657     cg_x_seq_er, cg_f_seq_er, cg_gradf_norm_seq_er, cg_k_er, cg_bt_seq_er = cgm_pol_rib(point, 'Extended
        Rosenbrock', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
658     print("Result of conjugate gradient method:")
659     print("x0: ", point, " (length: ", len(point), ")")
660     print("k: ", cg_k_er)
661     print("fk: ", cg_f_seq_er[-1])
662     print("gradfk: ", cg_gradf_norm_seq_er[-1])
663     print("\n")
664
665     kmean += cg_k_er
666     grad_norm_mean += cg_gradf_norm_seq_er[-1]
667     if grad_norm_max < cg_gradf_norm_seq_er[-1]:
668         grad_norm_max = cg_gradf_norm_seq_er[-1]
669     if grad_norm_min > cg_gradf_norm_seq_er[-1]:
670         grad_norm_min = cg_gradf_norm_seq_er[-1]
671     fx_mean += cg_f_seq_er[-1]
672     if fx_max < cg_f_seq_er[-1]:
673         fx_max = cg_f_seq_er[-1]
674     if fx_min > cg_f_seq_er[-1]:
675         fx_min = cg_f_seq_er[-1]
676
677 kmean = kmean / len(x0_array)
678 grad_norm_mean = grad_norm_mean / len(x0_array)
679 fx_mean = fx_mean / len(x0_array)
680
681 print("mean_of_k: ", kmean)
682
683 print("\n")
684 print("min_of_grad_norm: ", grad_norm_min)
685 print("mean_of_grad_norm: ", grad_norm_mean)
686 print("max_of_grad_norm: ", grad_norm_max)
687
688 print("\n")
689 print("min_of_fx: ", fx_min)
690 print("mean_of_fx: ", fx_mean)
691 print("max_of_fx: ", fx_max)
692
693 sns.set()
694 fig_er, ax_er = plt.subplots(2, 1, figsize=(10, 15))
695 ax_er[0].plot(sd_f_seq_er, '-r', label='Steepest descent')
696 ax_er[0].plot(cg_f_seq_er, '-g', label='Polak-Ribière')
697
698 ax_er[0].set_xscale('log', base=10)
699 ax_er[0].set_yscale('log', base=10)
700 ax_er[0].set_xlabel(r'$\log_{10}\{x\}$')
701 ax_er[0].set_ylabel(r'$\log_{10}\{f(\{x_n\})\}$')
702 ax_er[0].set_title(r'$\log_{10}$ plot of $f(\{x_n\})$')
703
704 ax_er[0].legend()
705
706 ax_er[1].plot(sd_gradf_norm_seq_er, '-r', label='Steepest descent')
707 ax_er[1].plot(cg_gradf_norm_seq_er, '-g', label='Polak-Ribière')
708

```

```

709 ax_er[1].set_xscale('log', base=10)
710 ax_er[1].set_yscale('log', base=10)
711 ax_er[1].set_xlabel(r'\log_{10}\{x\}')
712 ax_er[1].set_ylabel(r'\log_{10}\{|\nabla f(\{x_n\})|\}')
713 ax_er[1].set_title(r'\log_{10}\$ plot of \$|\nabla f(\{x_n\})|\$')
714
715 ax_er[1].legend()
716 fig_er.tight_layout()
717 fig_er.suptitle('Extended Rosenbrock function')
718 fig_er.subplots_adjust(top=0.94)
719
720 """3. Banded Trigonometric"""
721
722 kmean = 0
723 grad_norm_mean = 0
724 grad_norm_min = np.finfo(float).max
725 grad_norm_max = -1
726 fx_mean = 0
727 fx_min = np.finfo(float).max
728 fx_max = -1
729
730 for point in x0_array:
731     sd_x_seq_bt, sd_fk_bt, sd_gradf_norm_seq_bt, sd_k_bt, sd_bt_seq_bt = steepest_descent_bcktrck(point, '
Banded Trigonometric', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
732     print("Result of steepest descent method:")
733     print("x0: ", point, " (length: ", len(point), ")")
734     print("k: ", sd_k_bt)
735     print("fk: ", sd_fk_bt[-1])
736     print("gradfk: ", sd_gradf_norm_seq_bt[-1])
737     print("\n")
738
739     kmean += sd_k_bt
740     grad_norm_mean += sd_gradf_norm_seq_bt[-1]
741     if grad_norm_max < sd_gradf_norm_seq_bt[-1]:
742         grad_norm_max = sd_gradf_norm_seq_bt[-1]
743     if grad_norm_min > sd_gradf_norm_seq_bt[-1]:
744         grad_norm_min = sd_gradf_norm_seq_bt[-1]
745     fx_mean += sd_fk_bt[-1]
746     if fx_max < sd_fk_bt[-1]:
747         fx_max = sd_fk_bt[-1]
748     if fx_min > sd_fk_bt[-1]:
749         fx_min = sd_fk_bt[-1]
750
751 kmean = kmean / len(x0_array)
752 grad_norm_mean = grad_norm_mean / len(x0_array)
753 fx_mean = fx_mean / len(x0_array)
754
755 print("mean_of_k: ", kmean)
756
757 print("\n")
758 print("min_of_grad_norm: ", grad_norm_min)
759 print("mean_of_grad_norm: ", grad_norm_mean)
760 print("max_of_grad_norm: ", grad_norm_max)
761
762 print("\n")
763 print("min_of_fx: ", fx_min)
764 print("mean_of_fx: ", fx_mean)
765 print("max_of_fx: ", fx_max)
766
767 kmean = 0
768 grad_norm_mean = 0
769 grad_norm_min = np.finfo(float).max
770 grad_norm_max = -1
771 fx_mean = 0
772 fx_min = np.finfo(float).max
773 fx_max = -1
774
775 for point in x0_array:
776     cg_x_seq_bt, cg_f_seq_bt, cg_gradf_norm_seq_bt, cg_k_bt, cg_bt_seq_bt = cgm_pol_rib(point, 'Banded
Trigonometric', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
777     print("Result of steepest descent method:")
778     print("x0: ", point, " (length: ", len(point), ")")
779     print("k: ", cg_k_bt)
780     print("fk: ", cg_f_seq_bt[-1])

```

```

781 print("gradfk: ", cg_gradf_norm_seq_bt[-1])
782 print("\n")
783
784 kmean += cg_k_bt
785 grad_norm_mean += cg_gradf_norm_seq_bt[-1]
786 if grad_norm_max < cg_gradf_norm_seq_bt[-1]:
787     grad_norm_max = cg_gradf_norm_seq_bt[-1]
788 if grad_norm_min > cg_gradf_norm_seq_bt[-1]:
789     grad_norm_min = cg_gradf_norm_seq_bt[-1]
790 fx_mean += cg_f_seq_bt[-1]
791 if fx_max < cg_f_seq_bt[-1]:
792     fx_max = cg_f_seq_bt[-1]
793 if fx_min > cg_f_seq_bt[-1]:
794     fx_min = cg_f_seq_bt[-1]
795
796 kmean = kmean / len(x0_array)
797 grad_norm_mean = grad_norm_mean / len(x0_array)
798 fx_mean = fx_mean / len(x0_array)
799
800 print("mean_of_k: ", kmean)
801
802 print("\n")
803 print("min_of_grad_norm: ", grad_norm_min)
804 print("mean_of_grad_norm: ", grad_norm_mean)
805 print("max_of_grad_norm: ", grad_norm_max)
806
807 print("\n")
808 print("min_of_fx: ", fx_min)
809 print("mean_of_fx: ", fx_mean)
810 print("max_of_fx: ", fx_max)
811
812 sns.set()
813 fig_bt, ax_bt = plt.subplots(2, 1, figsize=(10, 15))
814 ax_bt[0].plot(sd_fk_bt, '-r', label='Steepest descent')
815 ax_bt[0].plot(cg_f_seq_bt, '-g', label='Polak-Ribière')
816
817 ax_bt[0].set_xscale('log', base=10)
818 ax_bt[0].set_yscale('log', base=10)
819 ax_bt[0].set_xlabel(r'$\log_{10}\{x\}$')
820 ax_bt[0].set_ylabel(r'$\log_{10}\{f(x_n)\}$')
821 ax_bt[0].set_title(r'$\log_{10}$ plot of $f(x_n)$')
822
823 ax_bt[0].legend()
824
825 ax_bt[1].plot(sd_gradf_norm_seq_bt, '-r', label='Steepest descent')
826 ax_bt[1].plot(cg_gradf_norm_seq_bt, '-g', label='Polak-Ribière')
827
828 ax_bt[1].set_xscale('log', base=10)
829 ax_bt[1].set_yscale('log', base=10)
830 ax_bt[1].set_xlabel(r'$\log_{10}\{x\}$')
831 ax_bt[1].set_ylabel(r'$\log_{10}\{|\nabla f(x_n)|\}$')
832 ax_bt[1].set_title(r'$\log_{10}$ plot of $|\nabla f(x_n)|$')
833
834 ax_bt[1].legend()
835 fig_bt.tight_layout()
836 fig_bt.suptitle('Banded trigonometric problem')
837 fig_bt.subplots_adjust(top=0.94)

```

Listing 2. Steepest Descent Method implementation

```

1 import numpy as np
2 from functions import *
3
4 def steepest_descent_bcktrck(x0: np.ndarray, f: str, alpha0: float, kmax: int, tolgrad: float, c1: float,
5   rho: float, btmax: int, fin_diff: bool, fd_type: str):
6
7     ''' Function that performs the conjugate gradient method with Polak-Ribière for a given function.
8
9     INPUTS:
10     x0 = starting point;
11     f = string that represent the function I want to use between the one stored there;
12     alpha0 = the initial factor that multiplies the descent direction at each iteration;
13     kmax = maximum number of iterations allowed;
14     tolgrad = value used as stopping criterion considering the norm of the gradient;
15     c1 = factor of the Armijo condition;
16     rho = fixed factor used for reducing alpha0;
17     btmax = maximum number of steps for updating alpha during the backtracking strategy.
18     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
19     fd_type = if fin_diff == True, choose between centered/forward/backward finite differences method
20
21     OUTPUTS:
22     x_seq = sequence of points xk computed during the iterations
23     f_seq = sequence of values f(xk) evaluated during the iterations
24     grad_norm_seq = sequence values of the norms of gradf(xk) computed during the iterations
25     k = index of the last iteration performed
26     bt_seq = sequence of the number of backtracking iterations done during the iterations '''
27
28     #Initialisation of the parameters
29     x_seq = x0.reshape(1, -1)
30     bt_seq = np.empty((1, 1))
31     f_seq = np.empty((1, 1))
32     gradf_norm_seq = np.empty((1, 1))
33     xk = x0
34     fk = 0
35     k = 0
36     alphak = alpha0
37     gradfk_norm = 0
38
39     if f == 'Rosenbrock': #The function that we are going to evaluate is the Rosenbrock one
40         fk = rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]))[0, 0] #evaluation of the function in xk
41         f_seq[0] = fk #add fk in the sequence of f
42         k = 0
43         gradfk_norm = np.linalg.norm(grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]])), fin_diff,
44           fd_type), 2) #Evaluate the norm of the gradient of fk
45         #linarg is just a numpy library that contains linear algebra functions like the norm one
46         gradf_norm_seq[0] = gradfk_norm #add the norm in the sequence of gradf_norm
47
48         while k < kmax and gradfk_norm > tolgrad:
49             gradfk = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]])), fin_diff, fd_type) #evaluate
50             the gradient of the function (the direction of the step)
51             pk = -gradfk
52             xnew = xk + alphak*pk #evaluate the new point following the direction pk
53             fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0] #evaluate the function in
54             the new point
55             bt = 0
56             alphak = alpha0
57
58             while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)): #backtracking using the Armijo
59             condition
60                 # update alpha
61                 alphak = rho*alphak
62                 xnew = xk + alphak*pk
63                 fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0]
64                 bt = bt + 1
65
66             #The next point is found, now what we do is evaluating all the important informations for doing
67             the analysis later and prepare the next iteration
68             xk = xnew
69             fk = fnew
70             gradfk_norm = np.linalg.norm(grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]])), fin_diff
71             , fd_type), 2)
72             x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
73             f_seq = np.append(f_seq, np.array([[fk]]))

```

```

67         gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
68         if k == 0:
69             bt_seq[0] = bt
70         else:
71             bt_seq = np.append(bt_seq, np.array([[bt]]))
72         k = k + 1
73
74     #The exact same steps were followed for the other three functions
75     elif f == 'Extended Powell': #The function that we are going to evaluate is the Extended Powell one
76         fk = extnd_powell(xk)
77         f_seq[0] = fk
78         k = 0
79         gradfk_norm = np.linalg.norm(grad_extnd_powell(xk, fin_diff, fd_type), 2)
80         gradf_norm_seq[0] = gradfk_norm
81
82         while k < kmax and gradfk_norm > tolgrad:
83             gradfk = grad_extnd_powell(xk, fin_diff, fd_type)
84             pk = -gradfk
85             xnew = xk + alphak*pk
86             fnew = extnd_powell(xnew)
87             bt = 0
88             alphak = alpha0
89
90             while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
91                 # update alpha
92                 alphak = rho*alphak
93                 xnew = xk + alphak*pk
94                 fnew = extnd_powell(xnew)
95                 bt = bt + 1
96
97             xk = xnew
98             fk = fnew
99             gradfk_norm = np.linalg.norm(grad_extnd_powell(xk, fin_diff, fd_type), 2)
100             x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
101             f_seq = np.append(f_seq, np.array([[fk]]))
102             gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
103             if k == 0:
104                 bt_seq[0] = bt
105             else:
106                 bt_seq = np.append(bt_seq, np.array([[bt]]))
107             #print(k)
108             k = k + 1
109
110     elif f == 'Extended Rosenbrock': #The function that we are going to evaluate is the Extended Rosenbrock
one
111         fk = extnd_rosenb(xk)
112         f_seq[0] = fk
113         k = 0
114         gradfk_norm = np.linalg.norm(grad_extnd_rosenb(xk, fin_diff, fd_type), 2)
115         gradf_norm_seq[0] = gradfk_norm
116
117         while k < kmax and gradfk_norm > tolgrad:
118             gradfk = grad_extnd_rosenb(xk, fin_diff, fd_type)
119             pk = -gradfk
120             xnew = xk + alphak*pk
121             fnew = extnd_rosenb(xnew)
122             bt = 0
123             alphak = alpha0
124
125             while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
126                 # update alpha
127                 alphak = rho*alphak
128                 xnew = xk + alphak*pk
129                 fnew = extnd_rosenb(xnew)
130                 bt = bt + 1
131
132             xk = xnew
133             fk = fnew
134             gradfk_norm = np.linalg.norm(grad_extnd_rosenb(xk, fin_diff, fd_type), 2)
135             x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
136             f_seq = np.append(f_seq, np.array([[fk]]))
137             gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
138             if k == 0:
139                 bt_seq[0] = bt

```

```

140         else:
141             bt_seq = np.append(bt_seq, np.array([[bt]]))
142             k = k + 1
143
144     elif 'Banded Trigonometric': #The function that we are going to evaluate is the Banded Trigonometric
one
145         fk = banded_trig(xk)
146         fk = banded_trig(xk)
147         f_seq[0] = fk
148         k = 0
149         gradfk_norm = np.linalg.norm(grad_banded_trig(xk, fin_diff, fd_type), 2)
150         gradf_norm_seq[0] = gradfk_norm
151
152         while k < kmax and gradfk_norm > tolgrad:
153             gradfk = grad_banded_trig(xk, fin_diff, fd_type)
154             pk = -gradfk
155             xnew = xk + alphak*pk
156             fnew = banded_trig(xnew)
157             bt = 0
158             alphak = alpha0
159
160             while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
161                 # update alpha
162                 alphak = rho*alphak
163                 xnew = xk + alphak*pk
164                 fnew = banded_trig(xnew)
165                 bt = bt + 1
166
167             xk = xnew
168             fk = fnew
169             gradfk_norm = np.linalg.norm(grad_banded_trig(xk, fin_diff, fd_type), 2)
170             x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
171             f_seq = np.append(f_seq, np.array([[fk]]))
172             gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
173             if k == 0:
174                 bt_seq[0] = bt
175             else:
176                 bt_seq = np.append(bt_seq, np.array([[bt]]))
177             k = k + 1
178
179     else:
180         print(f"No function called {f} exists.")
181
182     return x_seq, f_seq, gradf_norm_seq, k, bt_seq

```


Listing 3. Conjugate Gradient Method with Polak-Ribière implementation

```

1 import numpy as np
2 from functions import *
3
4 def cgm_pol_rib(x0: np.ndarray, f: str, alpha0: float, kmax: int, tolgrad: float, c1: float, rho: float,
5               btmax: int, fin_diff: bool, fd_type: str):
6
7     ''' Function that performs the conjugate gradient method with Polak-Ribière for a given function.
8
9     INPUTS:
10     x0 = starting point;
11     f = string that represent the function I want to use between the one stored there;
12     alpha0 = the initial factor that multiplies the descent direction at each iteration;
13     kmax = maximum number of iterations allowed;
14     tolgrad = value used as stopping criterion considering the norm of the gradient;
15     c1 = factor of the Armijo condition;
16     rho = fixed factor used for reducing alpha0;
17     btmax = maximum number of steps for updating alpha during the backtracking strategy.
18     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
19     fd_type = if fin_diff == True, choose between centered/forward/backward finite differences method
20
21     OUTPUTS:
22     x_seq = sequence of points xk computed during the iterations
23     f_seq = sequence of values f(xk) evaluated during the iterations
24     gradf_norm_seq = sequence values of the norms of gradf(xk) computed during the iterations
25     k = index of the last iteration performed
26     bt_seq = sequence of the number of backtracking iterations done during the iterations '''
27
28     #Initialisation of the parameters
29     x_seq = x0.reshape(1, -1)
30     bt_seq = np.empty((1, 1))
31     f_seq = np.empty((1, 1))
32     gradf_norm_seq = np.empty((1, 1))
33     xk = x0
34     fk = 0
35     betak = 0
36     k = 0
37     gradfk_norm = 0
38
39     if f == 'Rosenbrock': #The function that we are going to evaluate is the Rosenbrock one
40         fk = rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]))[0, 0] #evaluation of the function in xk
41         f_seq[0] = fk #add fk in the sequence of f
42         pk = -grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type) #evaluate the
43         gradient of the function (the direction of the first step)
44         gradfk_norm = np.linalg.norm(-pk, 2) #Evaluate the norm of the gradient of fk
45         #linalg is just a numpy library that contains linear algebra functions like the norm one
46         gradf_norm_seq[0] = gradfk_norm #add the norm in the sequence of gradf_norm
47
48         while k < kmax and gradfk_norm > tolgrad:
49             bt = 0
50             alphak = alpha0
51             xnew = xk + alphak*pk #evaluate the new point following the direction pk
52             fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0] #evaluate the function in
53             the new point
54             gradfk = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type) #evaluate
55             the gradient in the new point
56
57             while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)): #backtracking using the Armijo
58             condition
59                 # update alpha
60                 alphak = rho*alphak
61                 xnew = xk + alphak*pk
62                 fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0]
63                 bt = bt + 1
64
65             #The next point is found, now what we do is evaluating all the important informations for doing
66             the analysis later and prepare the next iteration
67             xk = xnew
68             fk = fnew
69             gradfnew = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type)
70             betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
71             pk = -gradfnew + betak*pk
72             gradfk_norm = np.linalg.norm(gradfnew, 2)
73             x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)

```

```

68     f_seq = np.append(f_seq, np.array([[fk]]))
69     gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
70     if k == 0:
71         bt_seq[0] = bt
72     else:
73         bt_seq = np.append(bt_seq, np.array([[bt]]))
74     k = k + 1
75
76 #The exact same steps were followed for the other three functions
77 elif f == 'Extended Powell': #The function that we are going to evaluate is the Extended Powell one
78     fk = extnd_powell(xk)
79     f_seq[0] = fk
80     pk = -grad_extnd_powell(xk, fin_diff, fd_type)
81     gradfk_norm = np.linalg.norm(-pk, 2)
82     gradf_norm_seq[0] = gradfk_norm
83
84     while k < kmax and gradfk_norm > tolgrad:
85         bt = 0
86         alphak = alpha0
87         xnew = xk + alphak*pk
88         fnew = extnd_powell(xnew)
89         gradfk = grad_extnd_powell(xk, fin_diff, fd_type)
90
91         while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
92             # update alpha
93             alphak = rho*alphak
94             xnew = xk + alphak*pk
95             fnew = extnd_powell(xnew)
96             bt = bt + 1
97
98         xk = xnew
99         fk = fnew
100        gradfnew = grad_extnd_powell(xnew, fin_diff, fd_type)
101        betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
102        pk = -gradfnew + betak*pk
103        gradfk_norm = np.linalg.norm(gradfnew, 2)
104        x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
105        f_seq = np.append(f_seq, np.array([[fk]]))
106        gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
107        if k == 0:
108            bt_seq[0] = bt
109        else:
110            bt_seq = np.append(bt_seq, np.array([[bt]]))
111        k = k + 1
112
113 elif f == 'Extended Rosenbrock': #The function that we are going to evaluate is the Extended Rosenbrock
    one
114     fk = extnd_rosenb(xk)
115     f_seq[0] = fk
116     pk = -grad_extnd_rosenb(xk, fin_diff, fd_type)
117     gradfk_norm = np.linalg.norm(-pk, 2)
118     gradf_norm_seq[0] = gradfk_norm
119
120     while k < kmax and gradfk_norm > tolgrad:
121         bt = 0
122         alphak = alpha0
123         xnew = xk + alphak*pk
124         fnew = extnd_rosenb(xnew)
125         gradfk = grad_extnd_rosenb(xk, fin_diff, fd_type)
126
127         while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
128             # update alpha
129             alphak = rho*alphak
130             xnew = xk + alphak*pk
131             fnew = extnd_rosenb(xnew)
132             bt = bt + 1
133
134         xk = xnew
135         fk = fnew
136         gradfnew = grad_extnd_rosenb(xnew, fin_diff, fd_type)
137         betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
138         pk = -gradfnew + betak*pk
139         gradfk_norm = np.linalg.norm(gradfnew, 2)
140         x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)

```

```

141     f_seq = np.append(f_seq, np.array([[fk]]))
142     gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
143     if k == 0:
144         bt_seq[0] = bt
145     else:
146         bt_seq = np.append(bt_seq, np.array([[bt]]))
147     k = k + 1
148
149 elif 'Banded Trigonometric': #The function that we are going to evaluate is the Banded Trigonometric
one
150     fk = banded_trig(xk)
151     f_seq[0] = fk
152     pk = -grad_banded_trig(xk, fin_diff, fd_type)
153     gradfk_norm = np.linalg.norm(-pk, 2)
154     gradf_norm_seq[0] = gradfk_norm
155
156     while k < kmax and gradfk_norm > tolgrad:
157         bt = 0
158         alphak = alpha0
159         xnew = xk + alphak*pk
160         fnew = banded_trig(xnew)
161         gradfk = grad_banded_trig(xk, fin_diff, fd_type)
162
163         while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)):
164             # update alpha
165             alphak = rho*alphak
166             xnew = xk + alphak*pk
167             fnew = banded_trig(xnew)
168             bt = bt + 1
169
170         xk = xnew
171         fk = fnew
172         gradfnew = grad_banded_trig(xnew, fin_diff, fd_type)
173         betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
174         pk = -gradfnew + betak*pk
175         gradfk_norm = np.linalg.norm(gradfnew, 2)
176         x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
177         f_seq = np.append(f_seq, np.array([[fk]]))
178         gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
179         if k == 0:
180             bt_seq[0] = bt
181         else:
182             bt_seq = np.append(bt_seq, np.array([[bt]]))
183         k = k + 1
184
185 else:
186     print(f"No function called {f} exists.")
187
188 return x_seq, f_seq, gradf_norm_seq, k, bt_seq

```

Listing 4. Functions and their Gradient evaluation

```

1 import numpy as np
2 from math import *
3 h = np.sqrt(np.finfo(float).eps/2)
4
5
6 def rosenbrock(\bm{x}: np.ndarray, y: np.ndarray) -> np.ndarray:
7     #Evaluation of the Rosenbrock function in the point(s) (\bm{x}, y)
8     return 100*(y - x**2)**2 + (1 - x)**2
9
10
11 def grad_rosenbrock(\bm{x}: np.ndarray, y: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
12     '''
13     Compute the approximation of the gradient via finite differences or with the true gradient
14
15     INPUTS:
16     x = array of x-coordinates, in a normal situation it is a single point, we have also implemented an
17     array of points to create the meshgrid for printing the 3D graph;
18     y = array of y-coordinates, in a normal situation it is a single point, we have also implemented an
19     array of points to create the meshgrid for printing the 3D graph;
20     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
21     ;
22     type = if fin_diff == True, choose between centered/forward/backward finite differences method;
23
24     OUTPUTS:
25     gradfx=the appossimation of the gradient in x;
26     '''
27
28     x_num = x.shape[1]
29     y_num = y.shape[0]
30
31     if (\bm{x}.size == 1 and y.size == 1): #There is only one point that was passed to the method
32         grad = np.empty(2)
33         if fin_diff == True: #Use the finite differences method for the evaluation of the gradient
34             if (type == "fw" or type == "bw"):
35                 fx = rosenbrock(\bm{x}, y)[0, 0]
36                 if (type == "fw"): #Use the forward finite difference method
37                     grad[0] = (rosenbrock(\bm{x}+h, y) - fx)[0, 0] / h
38                     grad[1] = (rosenbrock(\bm{x}, y+h) - fx)[0, 0] / h
39                 else: #Use the backward finite difference method
40                     grad[0] = -(rosenbrock(\bm{x}-h, y) - fx)[0, 0] / h
41                     grad[1] = -(rosenbrock(\bm{x}, y-h) - fx)[0, 0] / h
42             else: #Use the centered finite difference method
43                 grad[0] = (rosenbrock(\bm{x}+h, y) - rosenbrock(\bm{x}-h, y))[0, 0] / (2*h)
44                 grad[1] = (rosenbrock(\bm{x}, y+h) - rosenbrock(\bm{x}, y-h))[0, 0] / (2*h)
45             else: #Use the real gradient for evaluate the gradient in the point x
46                 grad[0] = (400*x**3 - 400*x*y + 2*x - 2)[0, 0]
47                 grad[1] = (200*(y - x**2))[0, 0]
48         else: #Do the same, but there are more than one point passed to the method
49             grad = np.empty((2, y_num, x_num))
50             if fin_diff == True:
51                 if (type == "fw" or type == "bw"):
52                     fx = rosenbrock(\bm{x}, y)
53                     if (type == "fw"):
54                         grad[0, :, :] = (rosenbrock(\bm{x}+h, y) - fx) / h
55                         grad[1, :, :] = (rosenbrock(\bm{x}, y+h) - fx) / h
56                     else:
57                         grad[0, :, :] = -(rosenbrock(\bm{x}-h, y) - fx) / h
58                         grad[1, :, :] = -(rosenbrock(\bm{x}, y-h) - fx) / h
59                 else:
60                     grad[0, :, :] = (rosenbrock(\bm{x}+h, y) - rosenbrock(\bm{x}-h, y)) / (2*h)
61                     grad[1, :, :] = (rosenbrock(\bm{x}, y+h) - rosenbrock(\bm{x}, y-h)) / (2*h)
62             else:
63                 grad[0, :, :] = 400*x**3 - 400*x*y + 2*x - 2
64                 grad[1, :, :] = 200*(y - x**2)
65         return grad
66
67 #The exact same principles were followed for the other three functions
68 def extnd_powell(\bm{x}: np.ndarray) -> float:
69     #Evaluation of the Extended Powell function in the point x
70     num = x.shape[0]
71     if num % 4 != 0:
72         raise Exception("Array length must be multiple of 4.")

```

```

71 def f(\bm{x}: np.ndarray, k: int) -> float:
72     match k % 4:
73         case 1:
74             k -= 1
75             return x[k] + 10*x[k+1]
76         case 2:
77             k -= 1
78             return sqrt(5)*(\bm{x}[k+1]-x[k+2])
79         case 3:
80             k -= 1
81             return (\bm{x}[k-1] - 2*x[k])**2
82         case 0:
83             k -= 1
84             return sqrt(10)*(\bm{x}[k-3] - x[k])**2
85
86     z = np.empty(num)
87     for k in range(0, num):
88         z[k] = f(\bm{x}, k+1)
89     return (0.5 * np.sum(z**2))
90
91
92 def grad_extnd_powell(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
93     '''
94     Compute the approximation of the gradient via finite differences or with the true gradient
95
96     INPUTS:
97     x: point in which I want to evaluate the gradient of the function;
98     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
99     ;
100     type = if fin_diff == True, choose between centered/forward/backward finite differences method;
101
102     OUTPUTS:
103     gradfx=the appossimation of the gradient in x;
104     '''
105     num = x.shape[0]
106     if num % 4 != 0:
107         raise Exception("Array length must be multiple of 4.")
108
109     def df(\bm{x}: np.ndarray, k: int) -> float:
110         xk = x[k-1]
111         match k % 4:
112             case 1:
113                 k -= 1
114                 return xk + 10*x[k+1] + 20*(xk - x[k+3])**3
115             case 2:
116                 k -= 1
117                 return 10*(\bm{x}[k-1]+10*xk) + 2*(xk-2*x[k+1])**3
118             case 3:
119                 k -= 1
120                 return 5*(xk - x[k+1]) + 4*(2*xk - x[k-1])**3
121             case 0:
122                 k -= 1
123                 return 5*(xk - x[k-1]) + 20*(xk - x[k-3])**3
124
125     grad = np.empty(num)
126     if fin_diff == True:
127         e = np.identity(num)
128         if (type == "fw" or type == "bw"):
129             fx = extnd_powell(\bm{x})
130             for i in range(0, num):
131                 if(type == "fw"):
132                     grad[i] = (extnd_powell(\bm{x}+h*e[i, :]) - fx) / h
133                 elif(type == "bw"):
134                     grad[i] = -(extnd_powell(\bm{x}-h*e[i, :]) - fx) / h
135                 else:
136                     grad[i] = (extnd_powell(\bm{x}+h*e[i, :]) - extnd_powell(\bm{x}-h*e[i, :])) / (2*h)
137     else:
138         for k in range(0, num):
139             grad[k] = df(\bm{x}, k+1)
140     return grad
141
142 def banded_trig(\bm{x}: np.ndarray) -> float:
143     #Evaluation of the Banded trigonometric problem in the point x

```

```

144 num = x.shape[0]
145 if num < 2:
146     raise Exception("Array length must be equal or higher than 2.")
147
148 z = np.empty(num)
149
150 #first iteration, different from the others
151 z[0] = 1 - cos(\bm{x}[0]) - sin(\bm{x}[1])
152
153 for k in range(1, num-1):
154     z[k] = (k + 1) * (1-cos(\bm{x}[k]) + sin(\bm{x}[k-1]) - sin(\bm{x}[k+1]))
155
156 #last iteration, different from the others
157 z[num-1] = num * (1 - cos(\bm{x}[num-1]) + sin(\bm{x}[num-2]))
158 return (np.sum(z))
159
160
161 def grad_banded_trig(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
162     """
163     Compute the approximation of the gradient via finite differences or with the true gradient
164
165     INPUTS:
166     x: point in which I want to evaluate the gradient of the function;
167     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
168     ;
169     type = if fin_diff == True, choose between centered/forward/backward finite differences method;
170
171     OUTPUTS:
172     gradfx=the appossimation of the gradient in x;
173     """
174     num = x.shape[0]
175     if num < 2:
176         raise Exception("Array length must be equal or higher than 2.")
177
178     grad = np.empty(num)
179     if fin_diff == True:
180         e = np.identity(num)
181         if (type == "fw" or type == "bw"):
182             fx = banded_trig(\bm{x})
183             for i in range(0, num):
184                 if(type == "fw"):
185                     grad[i] = (banded_trig(\bm{x}+h*e[i, :]) - fx) / h
186                 elif(type == "bw"):
187                     grad[i] = -(banded_trig(\bm{x}-h*e[i, :]) - fx) / h
188                 else:
189                     grad[i] = (banded_trig(\bm{x}+h*e[i, :]) - banded_trig(\bm{x}-h*e[i, :])) / (2*h)
190     else:
191         #first iteration, different from the others
192         grad[0] = (sin(\bm{x}[0]) + 2*cos(\bm{x}[0]))
193
194         for k in range(2, num):
195             grad[k-1] = -(k-1)*cos(\bm{x}[k-1]) + k*sin(\bm{x}[k-1]) + (k+1)*cos(\bm{x}[k-1])
196
197         #last iteration, different from the others
198         grad[num-1] = -(num-1)*cos(\bm{x}[num-1]) + num*sin(\bm{x}[num-1])
199     return grad
200
201 def extnd_rosenb(\bm{x}: np.ndarray) -> float:
202     #Evaluation of the Extended Rosenbrock function in the point x
203     num = x.shape[0]
204     if num % 2 != 0:
205         raise Exception("Array length must be multiple of 2.")
206
207     def f(\bm{x}: np.ndarray, k: int) -> float:
208         match k % 2:
209             case 1:
210                 k -= 1
211                 return 10*(\bm{x}[k]**2 - x[k+1])
212             case 0:
213                 k -= 1
214                 return x[k-1] - 1
215
216     z = np.empty(num)

```

```

217     for k in range(0, num):
218         z[k] = f(\bm{x}, k+1)
219     return (0.5 * np.sum(z**2))
220
221
222 def grad_extnd_rosenb(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
223     """
224     Compute the approximation of the gradient via finite differences or with the true gradient
225
226     INPUTS:
227     x: point in which I want to evaluate the gradient of the function;
228     fin_diff = choose between using the finite differences method for the evaluation of the gradient or not
229     ;
230     type = if fin_diff == True, choose between centered/forward/backward finite differences method;
231
232     OUTPUTS:
233     gradfx=the appossimation of the gradient in x;
234     """
235     num = x.shape[0]
236     if num % 2 != 0:
237         raise Exception("Array length must be multiple of 2.")
238
239     def df(\bm{x}: np.ndarray, k: int) -> float:
240         xk = x[k-1]
241         match k % 2:
242             case 1:
243                 k -= 1
244                 return 200*xk**3 - 200*xk*x[k+1] + xk - 1
245             case 0:
246                 k -= 1
247                 return 100*(xk - x[k-1]**2)
248
249     grad = np.empty(num)
250     if fin_diff == True:
251         e = np.identity(num)
252         if (type == "fw" or type == "bw"):
253             fx = extnd_rosenb(\bm{x})
254             for i in range(0, num):
255                 if(type == "fw"):
256                     grad[i] = (extnd_rosenb(\bm{x}+h*e[i, :]) - fx) / h
257                 elif(type == "bw"):
258                     grad[i] = -(extnd_rosenb(\bm{x}-h*e[i, :]) - fx) / h
259                 else:
260                     grad[i] = (extnd_rosenb(\bm{x}+h*e[i, :]) - extnd_rosenb(\bm{x}-h*e[i, :])) / (2*h)
261     else:
262         for k in range(0, num):
263             grad[k] = df(\bm{x}, k+1)
264     return grad

```