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 $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a scalar function, the problem:

$$\min_x f(\boldsymbol{x}) \tag{1}$$

where $x \in \mathbb{R}^n$ is a real vector with $n \geq 1$ components and $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ 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 $\boldsymbol{x}^*$ is a *global minimum* of f if $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$ for all $x \in \mathbb{R}$ (Or in general in the domain of f).

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 $|\boldsymbol{x} - \boldsymbol{x}^*| < \epsilon$.

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 $|\boldsymbol{x} - \boldsymbol{x}^*| < \epsilon$.

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

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

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

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

Theorem 1.8: When f is convex, any local minimiser $\boldsymbol{x}^*$ is a global minimiser of f. If in addition f is differentiable, then any stationary point $\boldsymbol{x}^*$ 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 $f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^n$, and $f : \mathbb{R}^n \longrightarrow \mathbb{R}$.

The objective of both iterative methods under consideration is to find a sequence of points $\boldsymbol{x}_k$ such that a global minimum $\boldsymbol{x}^*$ is reached. An initial point, $\boldsymbol{x}_0$, is therefore chosen, from which the iteration follows the following principle:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \tag{2}$$

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

The choice of the direction $\boldsymbol{p}_k$ 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, $\boldsymbol{p}_k$ 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 is 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 $\boldsymbol{p}_k$ ($\boldsymbol{p}_k = -\nabla f(\boldsymbol{x}_k)$) and a step ($\alpha_k$). The iteration can then be written in the following way:

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

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 $\boldsymbol{p}_k$, 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 $\boldsymbol{p}_k$ 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:

$$d_k^T A d_{k+1} = 0 \tag{4}$$

Now let's consider the following corollaries:

Corollary 2.0.1 (conjugacy and independence): if vectors $d_0, d_1, ..., d_k$ form a A-conjugate set, they are also independent.

Corollary 2.0.2 (transitivity): if a vector *a* is A-conjugate with respect to *b*, and *b* is A-conjugate with respect to *c*, then *a* will also be A-conjugate with respect to *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 $\boldsymbol{p}_k$ is selected varies:

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

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

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

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 $f(\boldsymbol{x}_{k+1}) > f(\boldsymbol{x}_k)$ 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:

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha_k \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k, \tag{7}$$

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 $c_1 \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p}_k$, 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 (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 $\rho \in (0, 1)$.

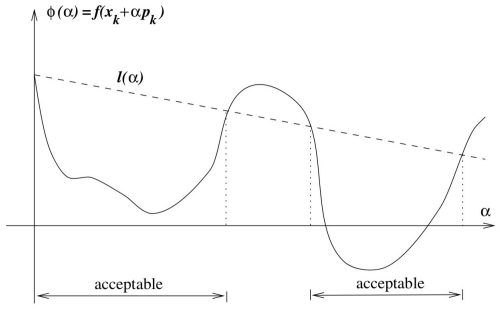

Fig. 1. Example of the Armijo condition.

IV. Problem Setting

A. Test on the Rosenbrook function

Consider the following function:

$$f(\boldsymbol{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \tag{8}$$

starting from the following points:

$$\boldsymbol{x}_0 = (1.2, 1.2),\ \boldsymbol{x}_0 = (-1.2, 1), \tag{9}$$

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 steplenght 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:

```
c: [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6]
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.

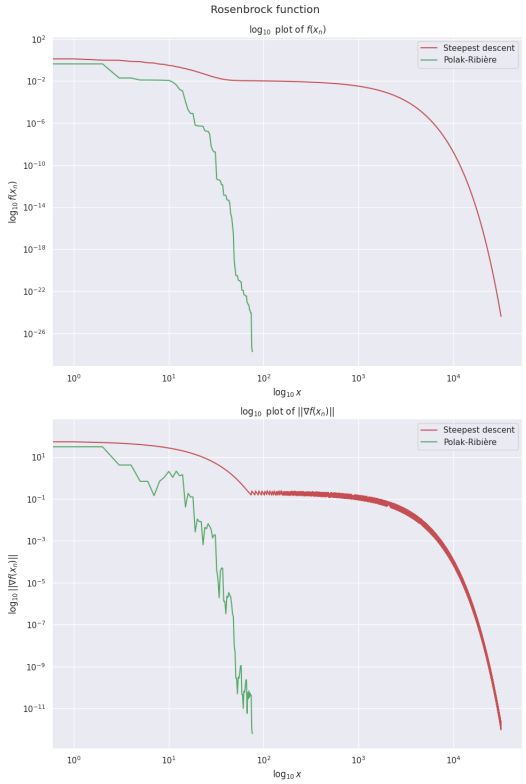

Fig. 2. Plot of convergence of $f(\boldsymbol{x}_k)$ and $||\nabla f(\boldsymbol{x}_k)||$ in Rosenbrock function starting from $\boldsymbol{x}_0 = (1.2, 1.2)$.

B. Evaluation of the rate of convergence

For point $\boldsymbol{x}_0 = (-1.2, 1)$ 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]:

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

When the true error is not known, it is possible to surrogate it by $\boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ obtaining the following [2]:

$$p = \frac{\ln \frac{||\boldsymbol{x}_{k+2} - \boldsymbol{x}_{k+1}||}{||\boldsymbol{x}_{k+1} - \boldsymbol{x}_k||}}{\ln \frac{||\boldsymbol{x}_{k+1} - \boldsymbol{x}_k||}{||\boldsymbol{x}_k - \boldsymbol{x}_{k-1}||}}. \tag{11}$$

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

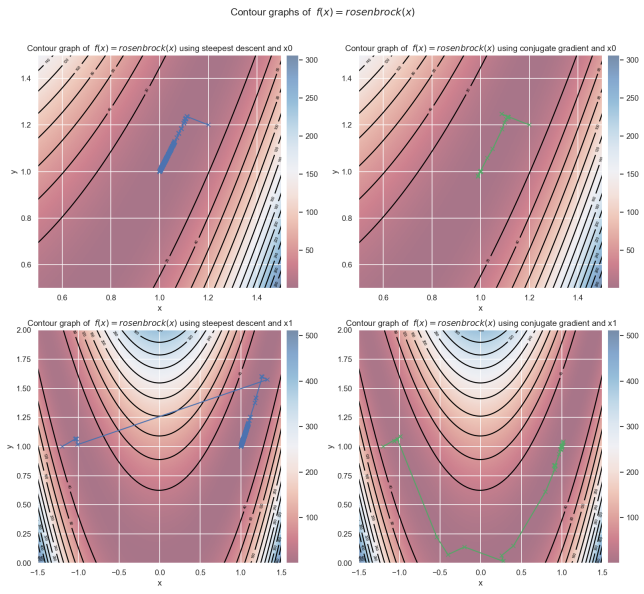

Fig. 3. Paths selected by the two methods on the contour plot.

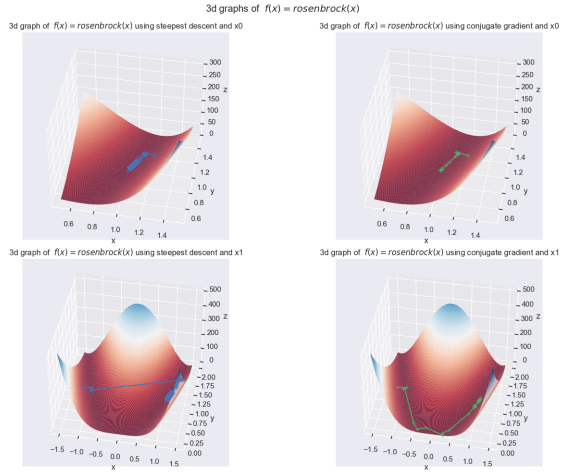

Fig. 4. Paths selected by the two methods on the 3d plot.

to the noise between iterations which led to steep changes in the value of $\boldsymbol{x}_k$ 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:

```
p of steepest descent method: 1.8619135773310265
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.

```
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:

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

$$x_0 = x_{n+1} = 0. \tag{13}$$

2) Problem 25. Extended Rosenbrock function:

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

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

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

3) Problem 26. Extended Powell singular function:

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

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

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

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

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

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 "$x0_{array} = np.random.randint(1, 10, size = (10, 10000))$", 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 $f(\boldsymbol{x}_{k_{max}})$ 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 $f(\boldsymbol{x}_{k_{max}})$ is $m \approx -662$).

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

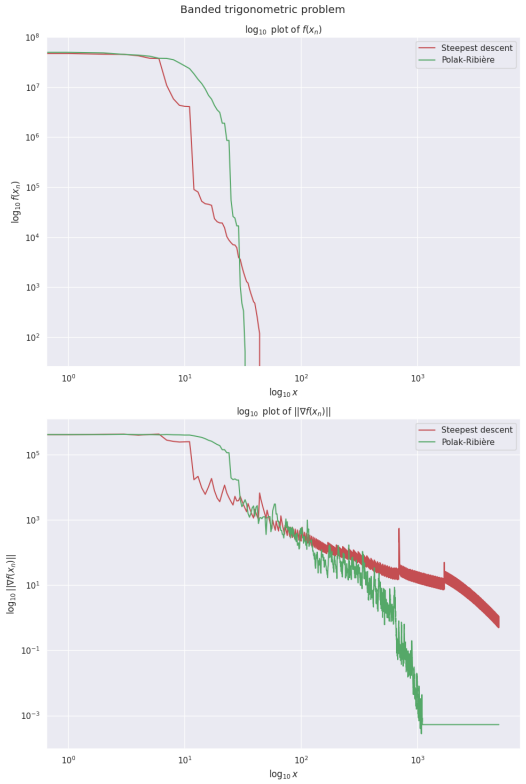

Fig. 5. Plot of convergence of $f(\boldsymbol{x}_{k_{max}})$ and $||\nabla f(\boldsymbol{x}_k)||$ in Banded trigonometric problem.

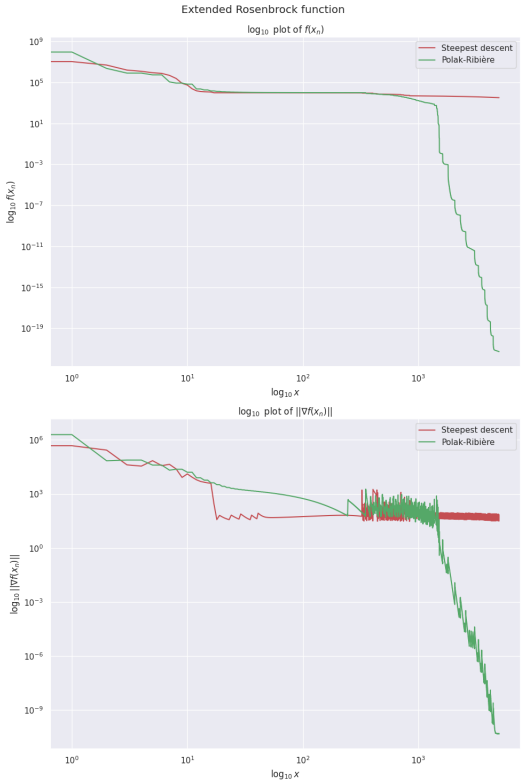

Fig. 6. Plot of convergence of $f(\boldsymbol{x}_{k_{max}})$ and $||\nabla f(\boldsymbol{x}_k)||$ in Extended Rosenbrock function.

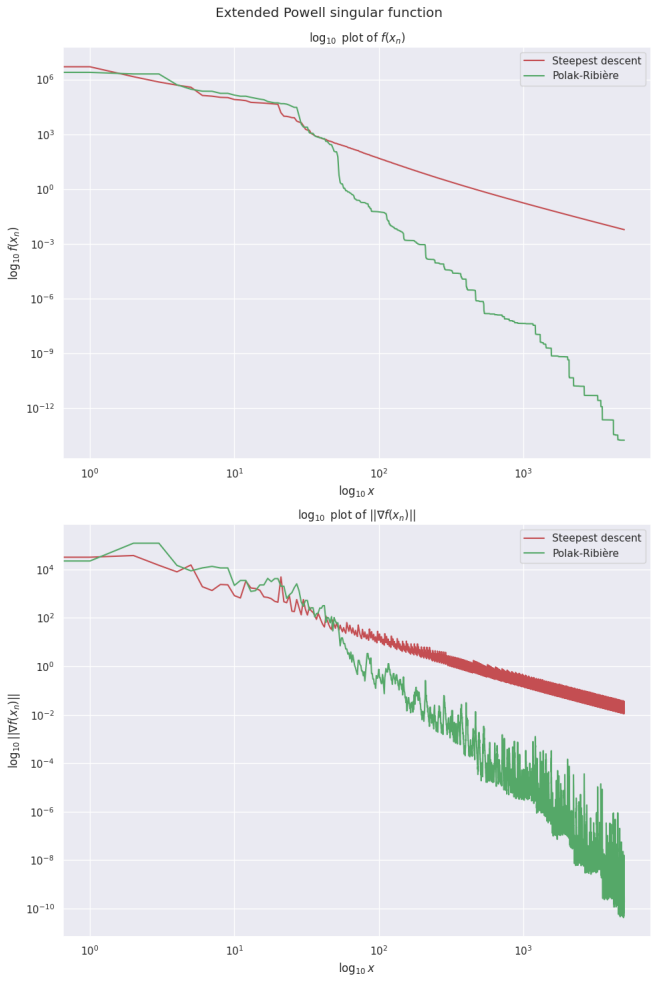

Fig. 7. Plot of convergence of $f(\boldsymbol{x}_{k_{max}})$ and $||\nabla f(\boldsymbol{x}_k)||$ in Extended Powell singular function.

APPENDIX

TABLE I
RESULTS OF ROSENBROCK FUNCTION.

$\boldsymbol{x}_0$	Method	c	ρ	$\boldsymbol{x}_{kmax}$	k	$f(\boldsymbol{x}_{k_{max}})$
[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, 1.0000000000000053]	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	[0.9999999999992051, 0.9999999999984085]	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}	$\nabla(f(\boldsymbol{x}_{k_{max}}))$ (min, mean, max)	$f(\boldsymbol{x}_{k_{max}})$ (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 ROSENBROCK FUNCTION.

Method	k_{mean}	$\nabla(f(\boldsymbol{x}_{k_{max}}))$ (min, mean, max)	$f(\boldsymbol{x}_{k_{max}})$ (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}	$\nabla(f(\boldsymbol{x}_{k_{max}}))$ (min, mean, max)	$f(\boldsymbol{x}_{k_{max}})$ (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., Hillström, K.E., Testing Unconstrained Optimization Software, ACM Transactions on Mathematical Software, Vol. 7, pp. 17-41, 1981.

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 steepest_descent_bcktrck as sdb
from steepest_descent_bcktrck import *
import cgm_pol_rib as cgmpb
from cgm_pol_rib import *
import functions as funcs
from functions import *
from sklearn.model_selection import ParameterGrid
from importlib import reload

reload(funcs)
reload(sdb)
reload(cgmpb)

"""Evaluate rate of convergence"""

def rate_of_convergence(x_seq: np.ndarray) -> float:
    M = 10
    m = 5
    if (x_seq.shape[0] < m):
        return None
    if (x_seq.shape[0] <= M):
        N = m
    else:
        N = M
    p = np.empty((1,1))
    for k in range(1, N+1):
        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 \
           np.linalg.norm(x_seq[k, :] - x_seq[k-1, :], 2) != 0 and \
           log(np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2) / np.linalg.norm(x_seq[k, :] - x_seq[k-1, :],
     2)) != 0:
            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)) / \
                log(np.linalg.norm(x_seq[k+1, :] - x_seq[k, :], 2) / np.linalg.norm(x_seq[k, :] - x_seq[k
    -1, :], 2)))
            if k == 1:
                p[0] = pk
            else:
                p = np.append(p, np.array([[pk]]))
    return np.abs(p).mean()

"""Evaluate exact rate of convergence"""

def exact_rate_of_convergence(x_seq: np.ndarray, x_star: np.ndarray) -> np.ndarray:
    p = np.empty((1,1))
    for k in range(1,  x_seq.shape[0]-1):
        if np.linalg.norm(x_seq[k+1, :] - x_star, 2) != 0 and np.linalg.norm(x_seq[k, :] - x_star, 2) != 0
    \
           and np.linalg.norm(x_seq[k-1, :] - x_star, 2) != 0 \
           and log(np.linalg.norm(x_seq[k, :] - x_star, 2) / np.linalg.norm(x_seq[k-1, :] - x_star, 2)) !=
    0:
            pk = log(np.linalg.norm(x_seq[k+1, :] - x_star, 2) / np.linalg.norm(x_seq[k, :] - x_star, 2)) /
     \
                log(np.linalg.norm(x_seq[k, :] - x_star, 2) / np.linalg.norm(x_seq[k-1, :] - x_star, 2))
            if k == 1:
                p[0] = pk
            else:
                p = np.append(p, np.array([[pk]]))
    return p

"""Initialisation of the data"""

x0 = np.array([1.2, 1.2])
x1 = np.array([-1.2, 1])
```

```
alpha0 = 1
tolgrad = 1e-12
rho = 0.5
c = 1e-4
kmax = 50000
btmax = 50
fin_diff = False
fd_type = 'centered'

params = {"c": [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6],
              "rho": [0.2, 0.3, 0.4, 0.5, 0.6, 0.7]}

"""Let's test the Steepest descent and conjugate gradient method with Backtrack tuning the parameters c and
     rho using the Armijo condition.

Here our idea is quite simple. In order to make a complete comparison between the two methods the best set
    of paramenters for both methods is evaluated, once this is done a comparison is made with the best
    result obtained in one of the two methods with one set and the same set is also used for the other
    method (we will therefore have in all 2 saved evaluations for steepest descent, one with the best set
    of parameters, and the other with the best set of parameters of the conjugate gradient method, and 2
    saved evaluations for the conjugate gradient method)

"""

#best result for steepest descent method is saved here
sd_best_fk_x0 = np.finfo(float).max
sd_best_param_x0 = 0
sd_best_k_x0 = 0
sd_best_gradfk_norm_x0 = 10
sd_best_xk_x0 = np.empty(2)
sd_best_x_seq_x0 = np.empty(2)
sd_best_f_seq_x0 = np.empty(2)

#best result for conjugate gradient method is saved here
cg_best_fk_x0 = np.finfo(float).max
cg_best_param_x0 = 0
cg_best_k_x0 = 0
cg_best_gradfk_norm_x0 = 10
cg_best_xk_x0 = np.empty(2)
cg_best_x_seq_x0 = np.empty(2)
cg_best_f_seq_x0 = np.empty(2)

#Other result of steepest descent method is saved here
sd_fk_x0 = np.finfo(float).max
sd_param_x0 = 0
sd_k_x0 = 0
sd_xk_x0 = np.empty(2)

#Other result of conjugate gradient method is saved here
cg_f_seq_x0 = np.finfo(float).max
cg_param_x0 = 0
cg_k_x0 = 0
cg_xk_x0 = np.empty(2)

for param in ParameterGrid(params):
    # Method for the steepest descent
    sd_x_seq, sd_f_seq, sd_gradf_norm_seq, sd_k, sd_bt_seq = steepest_descent_bcktrck(x0, 'Rosenbrock',
    alpha0, kmax, tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)

    # Method for the conjugate gradient method
    cg_x_seq, cg_f_seq, cg_gradf_norm_seq, cg_k, cg_bt_seq = cgm_pol_rib(x0, 'Rosenbrock', alpha0, kmax,
    tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)

    #For saving the best result of the steepest descent, and the correspondent result of the conjugate
    gradient method
    if (sd_f_seq[-1] < sd_best_fk_x0):
        sd_best_param_x0 = param

        sd_best_xk_x0 = sd_x_seq[-1]
        sd_best_k_x0 = sd_k
        sd_best_fk_x0 = sd_f_seq[-1]
        sd_best_gradfk_norm_x0 = sd_gradf_norm_seq[-1]
        sd_best_x_seq_x0 = np.copy(sd_x_seq)
        sd_best_f_seq_x0 = np.copy(sd_f_seq)
```

```
        sd_best_gradf_norm_seq_x0 = np.copy(sd_gradf_norm_seq)

        cg_xk_x0 = cg_x_seq[-1]
        cg_k_x0 = cg_k
        cg_fk_x0 = cg_f_seq[-1]

    #For saving the best result of the conjugate gradient, and the correspondent result of the steepest descent method
    if (cg_f_seq[-1] < cg_best_fk_x0):
        cg_best_param_x0 = param

        cg_best_xk_x0 = cg_x_seq[-1]
        cg_best_k_x0 = cg_k
        cg_best_fk_x0 = cg_f_seq[-1]
        cg_best_gradfk_norm_x0 = cg_gradf_norm_seq[-1]
        cg_best_x_seq_x0 = np.copy(cg_x_seq)
        cg_best_f_seq_x0 = np.copy(cg_f_seq)
        cg_best_gradf_norm_seq_x0 = np.copy(cg_gradf_norm_seq)

        sd_xk_x0 = sd_x_seq[-1]
        sd_k_x0 = sd_k
        sd_fk_x0 = sd_f_seq[-1]

"""Print the result obtained """

print("Best evaluation with steepest descent method:")
print("x0: ", x0[0], x0[1])
print("best set of parameters: ", sd_best_param_x0)
print("xk: ", sd_best_xk_x0[0], sd_best_xk_x0[1])
print("k: ", sd_best_k_x0)
print("fk: ", sd_best_fk_x0)

print("\nEvaluation of the conjugate gradient method using the best parameters of steepest descent: ")
print("x0: ", x0[0], x0[1])
print("set of parameters: ", sd_best_param_x0)
print("xk: ", cg_xk_x0[0], cg_xk_x0[1])
print("k: ", cg_k_x0)
print("fk: ", cg_fk_x0)

print("Best evaluation with conjugate gradient method:")
print("x0: ", x0[0], x0[1])
print("best set of parameters: ", cg_best_param_x0)
print("xk: ", cg_best_xk_x0[0], cg_best_xk_x0[1])
print("k: ", cg_best_k_x0)
print("fk: ", cg_best_fk_x0)

print("\nEvaluation of the steepest descent method using the best parameters of conjugate gradient: ")
print("x0: ", x0[0], x0[1])
print("set of parameters: ", cg_best_param_x0)
print("xk: ", sd_xk_x0[0], sd_xk_x0[1])
print("k: ", sd_k_x0)
print("fk: ", sd_fk_x0)

"""Creation of plots in the report showing the values of f(\bm{x}) and gradf(\bm{x}) for both methods. """

sns.set()
fig_ep, ax_ep = plt.subplots(2, 1, figsize=(10, 15))
ax_ep[0].plot(sd_best_f_seq_x0, '-r', label='Steepest descent')
ax_ep[0].plot(cg_best_f_seq_x0, '-g', label='Polak-Ribière')

ax_ep[0].set_xscale('log', base=10)
ax_ep[0].set_yscale('log', base=10)
ax_ep[0].set_xlabel(r'$\log_{10}{x}$')
ax_ep[0].set_ylabel(r'$\log_{10}{f({x_n})}$')
ax_ep[0].set_title(r'$\log_{10}$ plot of $f({x_n})$')

ax_ep[0].legend()

ax_ep[1].plot(sd_best_gradf_norm_seq_x0, '-r', label='Steepest descent')
ax_ep[1].plot(cg_best_gradf_norm_seq_x0, '-g', label='Polak-Ribière')

ax_ep[1].set_xscale('log', base=10)
ax_ep[1].set_yscale('log', base=10)
ax_ep[1].set_xlabel(r'$\log_{10}{x}$')
```

```
ax_ep[1].set_ylabel(r'$\log_{10}{||\nabla f({x_n})||}$')
ax_ep[1].set_title(r'$\log_{10}$ plot of $||\nabla f({x_n})||$')

ax_ep[1].legend()
fig_ep.tight_layout()
fig_ep.suptitle('Rosenbrock function')
fig_ep.subplots_adjust(top=0.94)

"""Now let's do the same but using the other starting point (x1)

*From point x1 we will also try to evaluate the rate of convergence p
"""

#best result for steepest descent method is saved here
sd_best_fk_x1 = np.finfo(float).max
sd_best_param_x1 = 0
sd_best_k_x1 = 0
sd_best_gradfk_norm_x1 = 10
sd_best_xk_x1 = np.empty(2)
sd_best_x_seq_x1 = np.empty(2)
sd_best_f_seq_x1 = np.empty(2)

#best result for conjugate gradient method is saved here
cg_best_fk_x1 = np.finfo(float).max
cg_best_param_x1 = 0
cg_best_k_x1 = 0
cg_best_gradfk_norm_x1 = 10
cg_best_xk_x1 = np.empty(2)
cg_best_x_seq_x1 = np.empty(2)
cg_best_f_seq_x1 = np.empty(2)

#Other result of steepest descent method is saved here
sd_fk_x1 = np.finfo(float).max
sd_param_x1 = 0
sd_k_x1 = 0
sd_xk_x1 = np.empty(2)

#Other result of conjugate gradient method is saved here
cg_fk_x1 = np.finfo(float).max
cg_param_x1 = 0
cg_k_x1 = 0
cg_xk_x1 = np.empty(2)

for param in ParameterGrid(params):
    sd_x_seq, sd_f_seq, sd_gradf_norm_seq, sd_k, sd_bt_seq = steepest_descent_bcktrck(x1, 'Rosenbrock',
    alpha0, kmax, tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)

    cg_x_seq, cg_f_seq, cg_gradf_norm_seq_seq, cg_k, cg_bt_seq = cgm_pol_rib(x1, 'Rosenbrock', alpha0, kmax
    , tolgrad, param["c"], param["rho"], btmax, fin_diff, fd_type)

    if (sd_f_seq[-1] < sd_best_fk_x1):
        sd_best_param_x1 = param

        sd_best_xk_x1 = sd_x_seq[-1]
        sd_best_k_x1 = sd_k
        sd_best_fk_x1 = sd_f_seq[-1]
        sd_best_gradfk_norm_x1 = sd_gradf_norm_seq[-1]
        sd_best_x_seq_x1 = np.copy(sd_x_seq)
        sd_best_f_seq_x1 = np.copy(sd_f_seq)

        cg_xk_x1 = cg_x_seq[-1]
        cg_k_x1 = cg_k
        cg_fk_x1 = cg_f_seq[-1]

    if (cg_f_seq[-1] < cg_best_fk_x1):
        cg_best_param_x1 = param

        cg_best_xk_x1 = cg_x_seq[-1]
        cg_best_k_x1 = cg_k
        cg_best_fk_x1 = cg_f_seq[-1]
        cg_best_gradfk_norm_x1 = cg_gradf_norm_seq_seq[-1]
        cg_best_x_seq_x1 = np.copy(cg_x_seq)
        cg_best_f_seq_x1 = np.copy(cg_f_seq)

```

```
        sd_xk_x1 = sd_x_seq[-1]
        sd_k_x1 = sd_k
        sd_fk_x1 = sd_f_seq[-1]

print("Best evaluation with steepest descent method:")
print("x1: ", x1[0], x1[1])
print("best set of parameters: ", sd_best_param_x1)
print("xk: ", sd_best_xk_x1[0], sd_best_xk_x1[1])
print("k: ", sd_best_k_x1)
print("fk: ", sd_best_fk_x1)

print("\nEvaluation of the conjugate gradient method using the best parameters of steepest descent: ")
print("x1: ", x1[0], x1[1])
print("set of parameters: ", sd_best_param_x1)
print("xk: ", cg_xk_x1[0], cg_xk_x1[1])
print("k: ", cg_k_x1)
print("fk: ", cg_fk_x1)

print("Best evaluation with conjugate gradient method:")
print("x1: ", x1[0], x1[1])
print("best set of parameters: ", cg_best_param_x1)
print("xk: ", cg_best_xk_x1[0], cg_best_xk_x1[1])
print("k: ", cg_best_k_x1)
print("fk: ", cg_best_fk_x1)

print("\nEvaluation of the steepest descent method using the best parameters of conjugate gradient: ")
print("x1: ", x1[0], x1[1])
print("set of parameters: ", cg_best_param_x1)
print("xk: ", sd_xk_x1[0], sd_xk_x1[1])
print("k: ", sd_k_x1)
print("fk: ", sd_fk_x1)

"""Evaluation of the rate of convergence p"""

if (sd_best_gradfk_norm_x1 <= tolgrad):
    #It make sense to evaluate the rate of convergence if and only if I've reached the solution
    \bm{p}_sd = rate_of_convergence(sd_best_x_seq_x1)
    print("Evaluation of rate of convergence for steepest descent method:")
    print(f"\bm{p}_sd = {\bm{p}_sd}")

"""Evaluation of rate of convergence for steepest descent method:

\bm{p}_sd = 1.8619135773310265
"""

if (cg_best_gradfk_norm_x1 <= tolgrad):
    #It make sense to evaluate the rate of convergence if and only if I've reached the solution
    \bm{p}_cg = rate_of_convergence(cg_best_x_seq_x1)
    print("Evaluation of rate of convergence for conjugate gradient method:")
    print(f"\bm{p}_cg = {\bm{p}_cg}")

"""Evaluation of rate of convergence for conjugate gradient method:

\bm{p}_cg = 1.0726581622394569

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.
"""

x_star = np.array([1, 1]) #Exact solution

\bm{p}_sd_exact = exact_rate_of_convergence(sd_x_seq, x_star)

print("Evaluation of exact rate of convergence for steepest descent method:")
print(f"\bm{p}_sd_exact = {\bm{p}_sd_exact.mean()}")

"""Evaluation of rate exact of convergence for steepest descent method:

\bm{p}_sd_exact: 1.1010415186256508

Contour graph for the Rosenbrock function that shows the path followed by both methods (The result is
    available in the report)
"""

```

```
x = np.linspace(0.5, 1.5, 200)
y = np.linspace(0.5, 1.5, 200)
X, Y = np.meshgrid(\bm{x}, y)
Z = rosenbrock(\bm{x}, Y)

sns.set()
fig_rb, ax_rb = plt.subplots(2, 2, figsize=(13, 12))

divider = make_axes_locatable(ax_rb[0, 0])
cax = divider.append_axes('right', size='5%', pad=0.1)
contours = ax_rb[0, 0].contour(\bm{x}, Y, Z, 15, colors='black')
ax_rb[0, 0].clabel(contours, inline=True, fontsize=5)
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')
ax_rb[0, 0].plot(sd_best_x_seq_x0[:, 0], sd_best_x_seq_x0[:, 1], '-xb')
ax_rb[0, 0].set_xlabel('x')
ax_rb[0, 0].set_ylabel('y')
ax_rb[0, 0].set_title(r'Contour graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using steepest descent and x0')
fig_rb.colorbar(im, cax=cax,  orientation='vertical')

divider = make_axes_locatable(ax_rb[0, 1])
cax = divider.append_axes('right', size='5%', pad=0.1)
contours = ax_rb[0, 1].contour(\bm{x}, Y, Z, 15, colors='black')
ax_rb[0, 1].clabel(contours, inline=True, fontsize=5)
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')
ax_rb[0, 1].plot(cg_best_x_seq_x0[:, 0], cg_best_x_seq_x0[:, 1], '-xg')
ax_rb[0, 1].set_xlabel('x')
ax_rb[0, 1].set_ylabel('y')
ax_rb[0, 1].set_title(r'Contour graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using conjugate gradient and x0')
fig_rb.colorbar(im, cax=cax,  orientation='vertical')

x = np.linspace(-1.5, 1.5, 200)
y = np.linspace(0, 2, 200)
X, Y = np.meshgrid(\bm{x}, y)
Z = rosenbrock(\bm{x}, Y)

divider = make_axes_locatable(ax_rb[1, 0])
cax = divider.append_axes('right', size='5%', pad=0.1)
contours = ax_rb[1, 0].contour(\bm{x}, Y, Z, 15, colors='black')
ax_rb[1, 0].clabel(contours, inline=True, fontsize=5)
im = ax_rb[1, 0].imshow(Z, extent=[-1.5, 1.5, 0, 2], origin='lower', cmap='RdBu', alpha=0.5, aspect='auto')
ax_rb[1, 0].plot(sd_best_x_seq_x1[:, 0], sd_best_x_seq_x1[:, 1], '-xb')
ax_rb[1, 0].set_xlabel('x')
ax_rb[1, 0].set_ylabel('y')
ax_rb[1, 0].set_title(r'Contour graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using steepest descent and x1')
fig_rb.colorbar(im, cax=cax,  orientation='vertical')

divider = make_axes_locatable(ax_rb[1, 1])
cax = divider.append_axes('right', size='5%', pad=0.1)
contours = ax_rb[1, 1].contour(\bm{x}, Y, Z, 15, colors='black')
ax_rb[1, 1].clabel(contours, inline=True, fontsize=5)
im = ax_rb[1, 1].imshow(Z, extent=[-1.5, 1.5, 0, 2], origin='lower', cmap='RdBu', alpha=0.5, aspect='auto')
ax_rb[1, 1].plot(cg_best_x_seq_x1[:, 0], cg_best_x_seq_x1[:, 1], '-xg')
ax_rb[1, 1].set_xlabel('x')
ax_rb[1, 1].set_ylabel('y')
ax_rb[1, 1].set_title(r'Contour graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using conjugate gradient and x1')
fig_rb.colorbar(im, cax=cax,  orientation='vertical')

fig_rb.tight_layout()
fig_rb.suptitle(r'Contour graphs of  $f(\bm{x})=rosenbrock(\bm{x})$')
fig_rb.subplots_adjust(top=0.9)

"""3D graph of the Rosenbrock function that shows the path followed by both methods (The result is
    available in the report)"""

x = np.linspace(0.5, 1.5, 200)
y = np.linspace(0.5, 1.5, 200)
X, Y = np.meshgrid(\bm{x}, y)
Z = rosenbrock(\bm{x}, Y)

sns.set()
fig3d = plt.figure(figsize=(15, 10))

```

```
ax = fig3d.add_subplot(2, 2, 1, projection='3d')
ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
ax.plot(sd_best_x_seq_x0[:, 0], sd_best_x_seq_x0[:, 1], sd_best_f_seq_x0, '-xb')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title(r'3d graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using steepest descent and x0')
ax.view_init(40, -80)

ax = fig3d.add_subplot(2, 2, 2, projection='3d')
ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
ax.plot(cg_best_x_seq_x0[:, 0], cg_best_x_seq_x0[:, 1], cg_best_f_seq_x0, '-+g')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title(r'3d graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using conjugate gradient and x0')
ax.view_init(40, -80)

x = np.linspace(-1.5, 1.5, 200)
y = np.linspace(0, 2, 200)
X, Y = np.meshgrid(\bm{x}, y)
Z = rosenbrock(\bm{x}, Y)

ax = fig3d.add_subplot(2, 2, 3, projection='3d')
ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
ax.plot(sd_best_x_seq_x1[:, 0], sd_best_x_seq_x1[:, 1], sd_best_f_seq_x1, '-xb')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title(r'3d graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using steepest descent and x1')
ax.view_init(40, -80)

ax = fig3d.add_subplot(2, 2, 4, projection='3d')
ax.plot_surface(\bm{x}, Y, Z, rstride=1, cstride=1, cmap='RdBu', edgecolor='none')
ax.plot(cg_best_x_seq_x1[:, 0], cg_best_x_seq_x1[:, 1], cg_best_f_seq_x1, '-+g')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title(r'3d graph of  $f(\bm{x})=rosenbrock(\bm{x})$ using conjugate gradient and x1')
ax.view_init(40, -80)

fig3d.tight_layout()
fig3d.suptitle(r'3d graphs of  $f(\bm{x})=rosenbrock(\bm{x})$')
fig3d.subplots_adjust(top=0.92)

"""Now let's work with the other functions and collect the data.

1. Extended Powell
"""

kmax = 5000 #Change the kmax from 50000 to 5000
x0_array = np.random.randint(1, 10, size=(10, 10000))

"""Steepest descent method """

#set of variables to save the results
kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    sd_x_seq_ep, sd_f_seq_ep, sd_gradf_norm_seq_ep, sd_k_ep, sd_bt_seq_ep = steepest_descent_bcktrck(point,
     'Extended Powell', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
    print("Result of steepest descent method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", sd_k_ep)
    print("fk: ", sd_f_seq_ep[-1])
    print("gradfk: ", sd_gradf_norm_seq_ep[-1])
    print("\n")

```

```
    kmean += sd_k_ep
    grad_norm_mean += sd_gradf_norm_seq_ep[-1]
    if grad_norm_max < sd_gradf_norm_seq_ep[-1]:
        grad_norm_max = sd_gradf_norm_seq_ep[-1]
    if grad_norm_min > sd_gradf_norm_seq_ep[-1]:
        grad_norm_min = sd_gradf_norm_seq_ep[-1]
    fx_mean += sd_f_seq_ep[-1]
    if fx_max < sd_f_seq_ep[-1]:
        fx_max = sd_f_seq_ep[-1]
    if fx_min > sd_f_seq_ep[-1]:
        fx_min = sd_f_seq_ep[-1]

"""Print of the results obtained"""

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

"""Conjugate gradient method"""

kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    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)
    print("Result of conjugate gradient method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", cg_k_ep)
    print("fk: ", cg_f_seq_ep[-1])
    print("gradfk: ", cg_gradf_norm_seq_ep[-1])
    print("\n")

    kmean += cg_k_ep
    grad_norm_mean += cg_gradf_norm_seq_ep[-1]
    if grad_norm_max < cg_gradf_norm_seq_ep[-1]:
        grad_norm_max = cg_gradf_norm_seq_ep[-1]
    if grad_norm_min > cg_gradf_norm_seq_ep[-1]:
        grad_norm_min = cg_gradf_norm_seq_ep[-1]
    fx_mean += cg_f_seq_ep[-1]
    if fx_max < cg_f_seq_ep[-1]:
        fx_max = cg_f_seq_ep[-1]
    if fx_min > cg_f_seq_ep[-1]:
        fx_min = cg_f_seq_ep[-1]

"""Print of the results obtained"""

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)
```

```

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

"""Creation of plots in the report showing the values of f(\bm{x}) and gradf(\bm{x}) for both methods. """

sns.set()
fig_ep, ax_ep = plt.subplots(2, 1, figsize=(10, 15))
ax_ep[0].plot(sd_f_seq_ep, '-r', label='Steepest descent')
ax_ep[0].plot(cg_f_seq_ep, '-g', label='Polak-Ribière')

ax_ep[0].set_xscale('log', base=10)
ax_ep[0].set_yscale('log', base=10)
ax_ep[0].set_xlabel(r'$\log_{10}{x}$')
ax_ep[0].set_ylabel(r'$\log_{10}{f({x_n})}$')
ax_ep[0].set_title(r'$\log_{10}$ plot of $f({x_n})$')

ax_ep[0].legend()

ax_ep[1].plot(sd_gradf_norm_seq_ep, '-r', label='Steepest descent')
ax_ep[1].plot(cg_gradf_norm_seq_ep, '-g', label='Polak-Ribière')

ax_ep[1].set_xscale('log', base=10)
ax_ep[1].set_yscale('log', base=10)
ax_ep[1].set_xlabel(r'$\log_{10}{x}$')
ax_ep[1].set_ylabel(r'$\log_{10}{||\nabla f({x_n})||}$')
ax_ep[1].set_title(r'$\log_{10}$ plot of $||\nabla f({x_n})||$')

ax_ep[1].legend()
fig_ep.tight_layout()
fig_ep.suptitle('Extended Powell singular function')
fig_ep.subplots_adjust(top=0.94)

"""The exact same steps were followed for the other two functions

2. Extended Rosenbrock
"""

kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    sd_x_seq_er, sd_f_seq_er, sd_gradf_norm_seq_er, sd_k_er, sd_bt_seq_er = steepest_descent_bcktrck(point,
     'Extended Rosenbrock', alpha0, kmax, tolgrad, c, rho, btmax, fin_diff, fd_type)
    print("Result of steepest descent method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", sd_k_er)
    print("fk: ", sd_f_seq_er[-1])
    print("gradfk: ", sd_gradf_norm_seq_er[-1])
    print("\n")

    kmean += sd_k_er
    grad_norm_mean += sd_gradf_norm_seq_er[-1]
    if grad_norm_max < sd_gradf_norm_seq_er[-1]:
        grad_norm_max = sd_gradf_norm_seq_er[-1]
    if grad_norm_min > sd_gradf_norm_seq_er[-1]:
        grad_norm_min = sd_gradf_norm_seq_er[-1]
    fx_mean += sd_f_seq_er[-1]
    if fx_max < sd_f_seq_er[-1]:
        fx_max = sd_f_seq_er[-1]
    if fx_min > sd_f_seq_er[-1]:
        fx_min = sd_f_seq_er[-1]

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

```

```
print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    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)
    print("Result of conjugate gradient method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", cg_k_er)
    print("fk: ", cg_f_seq_er[-1])
    print("gradfk: ", cg_gradf_norm_seq_er[-1])
    print("\n")

    kmean += cg_k_er
    grad_norm_mean += cg_gradf_norm_seq_er[-1]
    if grad_norm_max < cg_gradf_norm_seq_er[-1]:
        grad_norm_max = cg_gradf_norm_seq_er[-1]
    if grad_norm_min > cg_gradf_norm_seq_er[-1]:
        grad_norm_min = cg_gradf_norm_seq_er[-1]
    fx_mean += cg_f_seq_er[-1]
    if fx_max < cg_f_seq_er[-1]:
        fx_max = cg_f_seq_er[-1]
    if fx_min > cg_f_seq_er[-1]:
        fx_min = cg_f_seq_er[-1]

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

sns.set()
fig_er, ax_er = plt.subplots(2, 1, figsize=(10, 15))
ax_er[0].plot(sd_f_seq_er, '-r', label='Steepest descent')
ax_er[0].plot(cg_f_seq_er, '-g', label='Polak-Ribière')

ax_er[0].set_xscale('log', base=10)
ax_er[0].set_yscale('log', base=10)
ax_er[0].set_xlabel(r'$\log_{10}{x}$')
ax_er[0].set_ylabel(r'$\log_{10}{f({x_n})}$')
ax_er[0].set_title(r'$\log_{10}$ plot of $f({x_n})$')

ax_er[0].legend()

ax_er[1].plot(sd_gradf_norm_seq_er, '-r', label='Steepest descent')
ax_er[1].plot(cg_gradf_norm_seq_er, '-g', label='Polak-Ribière')

```

```
ax_er[1].set_xscale('log', base=10)
ax_er[1].set_yscale('log', base=10)
ax_er[1].set_xlabel(r'$\log_{10}{x}$')
ax_er[1].set_ylabel(r'$\log_{10}{||\nabla f({x_n})||}$')
ax_er[1].set_title(r'$\log_{10}$ plot of $||\nabla f({x_n})||$')

ax_er[1].legend()
fig_er.tight_layout()
fig_er.suptitle('Extended Rosenbrock function')
fig_er.subplots_adjust(top=0.94)

"""3. Banded Trigonometric"""

kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    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)
    print("Result of steepest descent method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", sd_k_bt)
    print("fk: ", sd_fk_bt[-1])
    print("gradfk: ", sd_gradf_norm_seq_bt[-1])
    print("\n")

    kmean += sd_k_bt
    grad_norm_mean += sd_gradf_norm_seq_bt[-1]
    if grad_norm_max < sd_gradf_norm_seq_bt[-1]:
        grad_norm_max = sd_gradf_norm_seq_bt[-1]
    if grad_norm_min > sd_gradf_norm_seq_bt[-1]:
        grad_norm_min = sd_gradf_norm_seq_bt[-1]
    fx_mean += sd_fk_bt[-1]
    if fx_max < sd_fk_bt[-1]:
        fx_max = sd_fk_bt[-1]
    if fx_min > sd_fk_bt[-1]:
        fx_min = sd_fk_bt[-1]

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

kmean = 0
grad_norm_mean = 0
grad_norm_min = np.finfo(float).max
grad_norm_max = -1
fx_mean = 0
fx_min = np.finfo(float).max
fx_max = -1

for point in x0_array:
    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)
    print("Result of steepest descent method:")
    print("x0: ", point, " (length: ", len(point), ")")
    print("k: ", cg_k_bt)
    print("fk: ", cg_f_seq_bt[-1])
```

```
    print("gradfk: ", cg_gradf_norm_seq_bt[-1])
    print("\n")

    kmean += cg_k_bt
    grad_norm_mean += cg_gradf_norm_seq_bt[-1]
    if grad_norm_max < cg_gradf_norm_seq_bt[-1]:
        grad_norm_max = cg_gradf_norm_seq_bt[-1]
    if grad_norm_min > cg_gradf_norm_seq_bt[-1]:
        grad_norm_min = cg_gradf_norm_seq_bt[-1]
    fx_mean += cg_f_seq_bt[-1]
    if fx_max < cg_f_seq_bt[-1]:
        fx_max = cg_f_seq_bt[-1]
    if fx_min > cg_f_seq_bt[-1]:
        fx_min = cg_f_seq_bt[-1]

kmean = kmean / len(x0_array)
grad_norm_mean = grad_norm_mean / len(x0_array)
fx_mean = fx_mean / len(x0_array)

print("mean_of_k: ", kmean)

print("\n")
print("min_of_grad_norm: ", grad_norm_min)
print("mean_of_grad_norm: ", grad_norm_mean)
print("max_of_grad_norm: ", grad_norm_max)

print("\n")
print("min_of_fx: ", fx_min)
print("mean_of_fx: ", fx_mean)
print("max_of_fx: ", fx_max)

sns.set()
fig_bt, ax_bt = plt.subplots(2, 1, figsize=(10, 15))
ax_bt[0].plot(sd_fk_bt, '-r', label='Steepest descent')
ax_bt[0].plot(cg_f_seq_bt, '-g', label='Polak-Ribière')

ax_bt[0].set_xscale('log', base=10)
ax_bt[0].set_yscale('log', base=10)
ax_bt[0].set_xlabel(r'$\log_{10}{x}$')
ax_bt[0].set_ylabel(r'$\log_{10}{f({x_n})}$')
ax_bt[0].set_title(r'$\log_{10}$ plot of $f({x_n})$')

ax_bt[0].legend()

ax_bt[1].plot(sd_gradf_norm_seq_bt, '-r', label='Steepest descent')
ax_bt[1].plot(cg_gradf_norm_seq_bt, '-g', label='Polak-Ribière')

ax_bt[1].set_xscale('log', base=10)
ax_bt[1].set_yscale('log', base=10)
ax_bt[1].set_xlabel(r'$\log_{10}{x}$')
ax_bt[1].set_ylabel(r'$\log_{10}{||\nabla f({x_n})||}$')
ax_bt[1].set_title(r'$\log_{10}$ plot of $||\nabla f({x_n})||$')

ax_bt[1].legend()
fig_bt.tight_layout()
fig_bt.suptitle('Banded trigonometric problem')
fig_bt.subplots_adjust(top=0.94)
```

Listing 2. Steepest Descent Method implementation

```python
import numpy as np
from functions import *

def steepest_descent_bcktrck(x0: np.ndarray, f: str, alpha0: float, kmax: int, tolgrad: float, c1: float, rho: float, btmax: int, fin_diff: bool, fd_type: str):

    ''' Function that performs the conjugate gradient method with Polak-Ribière for a given function.

    INPUTS:
    x0 = starting point;
    f = string that represent the function I want to use between the one stored there;
    alpha0 = the initial factor that multiplies the descent direction at each iteration;
    kmax = maximum number of iterations allowed;
    tolgrad = value used as stopping criterion considering the norm of the gradient;
    c1 = factor of the Armijo condition;
    rho = fixed factor 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

    OUTPUTS:
    x_seq = sequence of points xk computed during the iterations
    f_seq = sequence of values f(xk) evaluated during the iterations
    grad_norm_seq = sequence values of the norms of gradf(xk) computed during the iterations
    k = index of the last iteration performed
    bt_seq = sequence of the number of backtracking iterations done during the iterations '''

    #Initialisation of the parameters
    x_seq = x0.reshape(1, -1)
    bt_seq = np.empty((1, 1))
    f_seq = np.empty((1, 1))
    gradf_norm_seq = np.empty((1, 1))
    xk = x0
    fk = 0
    k = 0
    alphak = alpha0
    gradfk_norm = 0

    if f == 'Rosenbrock': #The function that we are going to evaluate is the Rosenbrock one
        fk = rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]))[0, 0]  #evaluation of the function in xk
        f_seq[0] = fk #add fk in the sequence of f
        k = 0
        gradfk_norm = np.linalg.norm(grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type), 2) #Evaluate the norm of the gradient of fk
        #linarg is just a numpy library that contains linear algebra functions like the norm one
        gradf_norm_seq[0] = gradfk_norm #add the norm in the sequence of gradf_norm

        while k < kmax and gradfk_norm > tolgrad:
            gradfk = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type) #evaluate the gradient of the function (the direction of the step)
            pk = -gradfk
            xnew = xk + alphak*pk #evaluate the new point following the direction pk
            fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0] #evaluate the function in the new point
            bt = 0
            alphak = alpha0

            while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)): #backtracking using the Armijo condition
                # update alpha
                alphak = rho*alphak
                xnew = xk + alphak*pk
                fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0]
                bt = bt + 1

            #The next point is found, now what we do is evaluating all the important informations for doing the analysis later and prepare the next iteration
            xk = xnew
            fk = fnew
            gradfk_norm = np.linalg.norm(grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type), 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
```

```
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    #The exact same steps were followed for the other three functions
    elif f == 'Extended Powell': #The function that we are going to evaluate is the Extended Powell one
        fk = extnd_powell(xk)
        f_seq[0] = fk
        k = 0
        gradfk_norm = np.linalg.norm(grad_extnd_powell(xk, fin_diff, fd_type), 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            gradfk = grad_extnd_powell(xk, fin_diff, fd_type)
            pk = -gradfk
            xnew = xk + alphak*pk
            fnew = extnd_powell(xnew)
            bt = 0
            alphak = alpha0

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

            xk = xnew
            fk = fnew
            gradfk_norm = np.linalg.norm(grad_extnd_powell(xk, fin_diff, fd_type), 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            #print(k)
            k = k + 1

    elif f == 'Extended Rosenbrock': #The function that we are going to evaluate is the Extended Rosenbrock
     one
        fk = extnd_rosenb(xk)
        f_seq[0] = fk
        k = 0
        gradfk_norm = np.linalg.norm(grad_extnd_rosenb(xk, fin_diff, fd_type), 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            gradfk = grad_extnd_rosenb(xk, fin_diff, fd_type)
            pk = -gradfk
            xnew = xk + alphak*pk
            fnew = extnd_rosenb(xnew)
            bt = 0
            alphak = alpha0

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

            xk = xnew
            fk = fnew
            gradfk_norm = np.linalg.norm(grad_extnd_rosenb(xk, fin_diff, fd_type), 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
```

```
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    elif 'Banded Trigonometric': #The function that we are going to evaluate is the Banded Trigonometric
    one
        fk = banded_trig(xk)
        fk = banded_trig(xk)
        f_seq[0] = fk
        k = 0
        gradfk_norm = np.linalg.norm(grad_banded_trig(xk, fin_diff, fd_type), 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            gradfk = grad_banded_trig(xk, fin_diff, fd_type)
            pk = -gradfk
            xnew = xk + alphak*pk
            fnew = banded_trig(xnew)
            bt = 0
            alphak = alpha0

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

            xk = xnew
            fk = fnew
            gradfk_norm = np.linalg.norm(grad_banded_trig(xk, fin_diff, fd_type), 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    else:
        print(f"No function called {f} exists.")

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

Listing 3. Conjugate Gradient Method wit Polak-Ribière implementation

```python
import numpy as np
from functions import *

def cgm_pol_rib(x0: np.ndarray, f: str, alpha0: float, kmax: int, tolgrad: float, c1: float, rho: float, btmax: int, fin_diff: bool, fd_type: str):

    ''' Function that performs the conjugate gradient method with Polak-Ribière for a given function.

    INPUTS:
    x0 = starting point;
    f = string that represent the function I want to use between the one stored there;
    alpha0 = the initial factor that multiplies the descent direction at each iteration;
    kmax = maximum number of iterations allowed;
    tolgrad = value used as stopping criterion considering the norm of the gradient;
    c1 = factor of the Armijo condition;
    rho = fixed factor 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

    OUTPUTS:
    x_seq = sequence of points xk computed during the iterations
    f_seq = sequence of values f(xk) evaluated during the iterations
    grad_norm_seq = sequence values of the norms of gradf(xk) computed during the iterations
    k = index of the last iteration performed
    bt_seq = sequence of the number of backtracking iterations done during the iterations '''

    #Initialisation of the parameters
    x_seq = x0.reshape(1, -1)
    bt_seq = np.empty((1, 1))
    f_seq = np.empty((1, 1))
    gradf_norm_seq = np.empty((1, 1))
    xk = x0
    fk = 0
    betak = 0
    k = 0
    gradfk_norm = 0

    if f == 'Rosenbrock': #The function that we are going to evaluate is the Rosenbrock one
        fk = rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]))[0, 0] #evaluation of the function in xk
        f_seq[0] = fk #add fk in the sequence of f
        pk = -grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type) #evaluate the gradient of the function (the direction of the first step)
        gradfk_norm = np.linalg.norm(-pk, 2) #Evaluate the norm of the gradient of fk
        #linarg is just a numpy library that contains linear algebra functions like the norm one
        gradf_norm_seq[0] = gradfk_norm #add the norm in the sequence of gradf_norm

        while k < kmax and gradfk_norm > tolgrad:
            bt = 0
            alphak = alpha0
            xnew = xk + alphak*pk #evaluate the new point following the direction pk
            fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0] #evaluate the function in the new point
            gradfk = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type) #evaluate the gradient in the new point

            while (bt < btmax) and (fnew > fk + c1*alphak*(gradfk @ pk)): #backtracking using the Armijo condition
                # update alpha
                alphak = rho*alphak
                xnew = xk + alphak*pk
                fnew = rosenbrock(np.array([[xnew[0]]]), np.array([[xnew[1]]]))[0, 0]
                bt = bt + 1

            #The next point is found, now what we do is evaluating all the important informations for doing the analysis later and prepare the next iteration
            xk = xnew
            fk = fnew
            gradfnew = grad_rosenbrock(np.array([[xk[0]]]), np.array([[xk[1]]]), fin_diff, fd_type)
            betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
            pk = -gradfnew + betak*pk
            gradfk_norm = np.linalg.norm(gradfnew, 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
```

```
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    #The exact same steps were followed for the other three functions
    elif f == 'Extended Powell': #The function that we are going to evaluate is the Extended Powell one
        fk = extnd_powell(xk)
        f_seq[0] = fk
        pk = -grad_extnd_powell(xk, fin_diff, fd_type)
        gradfk_norm = np.linalg.norm(-pk, 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            bt = 0
            alphak = alpha0
            xnew = xk + alphak*pk
            fnew = extnd_powell(xnew)
            gradfk = grad_extnd_powell(xk, fin_diff, fd_type)

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

            xk = xnew
            fk = fnew
            gradfnew = grad_extnd_powell(xnew, fin_diff, fd_type)
            betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
            pk = -gradfnew + betak*pk
            gradfk_norm = np.linalg.norm(gradfnew, 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    elif f == 'Extended Rosenbrock': #The function that we are going to evaluate is the Extended Rosenbrock
     one
        fk = extnd_rosenb(xk)
        f_seq[0] = fk
        pk = -grad_extnd_rosenb(xk, fin_diff, fd_type)
        gradfk_norm = np.linalg.norm(-pk, 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            bt = 0
            alphak = alpha0
            xnew = xk + alphak*pk
            fnew = extnd_rosenb(xnew)
            gradfk = grad_extnd_rosenb(xk, fin_diff, fd_type)

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

            xk = xnew
            fk = fnew
            gradfnew = grad_extnd_rosenb(xnew, fin_diff, fd_type)
            betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
            pk = -gradfnew + betak*pk
            gradfk_norm = np.linalg.norm(gradfnew, 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
```

```
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    elif 'Banded Trigonometric': #The function that we are going to evaluate is the Banded Trigonometric
    one
        fk = banded_trig(xk)
        f_seq[0] = fk
        pk = -grad_banded_trig(xk, fin_diff, fd_type)
        gradfk_norm = np.linalg.norm(-pk, 2)
        gradf_norm_seq[0] = gradfk_norm

        while k < kmax and gradfk_norm > tolgrad:
            bt = 0
            alphak = alpha0
            xnew = xk + alphak*pk
            fnew = banded_trig(xnew)
            gradfk = grad_banded_trig(xk, fin_diff, fd_type)

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

            xk = xnew
            fk = fnew
            gradfnew = grad_banded_trig(xnew, fin_diff, fd_type)
            betak = (gradfnew @ (gradfnew - gradfk)) / gradfk_norm**2
            pk = -gradfnew + betak*pk
            gradfk_norm = np.linalg.norm(gradfnew, 2)
            x_seq = np.append(x_seq, xk.reshape(1, -1), axis=0)
            f_seq = np.append(f_seq, np.array([[fk]]))
            gradf_norm_seq = np.append(gradf_norm_seq, np.array([[gradfk_norm]]))
            if k == 0:
                bt_seq[0] = bt
            else:
                bt_seq = np.append(bt_seq, np.array([[bt]]))
            k = k + 1

    else:
        print(f"No function called {f} exists.")

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

Listing 4. Functions and their Gradient evaluation

```
import numpy as np
from math import *
h = np.sqrt(np.finfo(float).eps/2)


def rosenbrock(\bm{x}: np.ndarray, y: np.ndarray) -> np.ndarray:
    #Evaluation of the Rosenbrock function in the point(s) (\bm{x}, y)
    return 100*(y - x**2)**2 + (1 - x)**2


def grad_rosenbrock(\bm{x}: np.ndarray, y: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
    '''
    Compute the appoximation of the gradient via finite differences or with the true gradient

    INPUTS:
    x = array of x-coordinates, in a normal situation it is a single point, we have also implemented an
    array of points to create the meshgrid for printing the 3D graph;
    y = array of y-coordinates, in a normal situation it is a single point, we have also implemented an
    array of points to create the meshgrid for printing the 3D graph;
    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;

    OUTPUTS:
    gradfx=the appossimation of the gradient in x;
    '''

    x_num = x.shape[1]
    y_num = y.shape[0]

    if (\bm{x}.size == 1 and y.size == 1): #There is only one point that was passed to the method
        grad = np.empty(2)
        if fin_diff == True: #Use the finite differences method for the evaluation of the gradient
            if (type == "fw" or type == "bw"):
                fx = rosenbrock(\bm{x}, y)[0, 0]
                if(type == "fw"): #Use the forward finite difference method
                    grad[0] = (rosenbrock(\bm{x}+h, y) - fx)[0, 0] / h
                    grad[1] = (rosenbrock(\bm{x}, y+h) - fx)[0, 0] / h
                else: #Use the backward finite difference method
                    grad[0] = -(rosenbrock(\bm{x}-h, y) - fx)[0, 0] / h
                    grad[1] = -(rosenbrock(\bm{x}, y-h) - fx)[0, 0] / h
            else: #Use the centered finite difference method
                grad[0] = (rosenbrock(\bm{x}+h, y) - rosenbrock(\bm{x}-h, y))[0, 0] / (2*h)
                grad[1] = (rosenbrock(\bm{x}, y+h) - rosenbrock(\bm{x}, y-h))[0, 0] / (2*h)
        else: #Use the real gradient for evaluate the gradient in the point x
            grad[0] = (400*x**3 - 400*x*y + 2*x - 2)[0, 0]
            grad[1] = (200*(y - x**2))[0, 0]
    else: #Do the same, but there are more than one point passed to the method
        grad = np.empty((2, y_num, x_num))
        if fin_diff == True:
            if (type == "fw" or type == "bw"):
                fx = rosenbrock(\bm{x}, y)
                if(type == "fw"):
                    grad[0, :, :] = (rosenbrock(\bm{x}+h, y) - fx) / h
                    grad[1, :, :] = (rosenbrock(\bm{x}, y+h) - fx) / h
                else:
                    grad[0, :, :] = -(rosenbrock(\bm{x}-h, y) - fx) / h
                    grad[1, :, :] = -(rosenbrock(\bm{x}, y-h) - fx) / h
            else:
                grad[0, :, :] = (rosenbrock(\bm{x}+h, y) - rosenbrock(\bm{x}-h, y)) / (2*h)
                grad[1, :, :] = (rosenbrock(\bm{x}, y+h) - rosenbrock(\bm{x}, y-h)) / (2*h)
        else:
            grad[0, :, :] = 400*x**3 - 400*x*y + 2*x - 2
            grad[1, :, :] = 200*(y - x**2)
    return grad

#The exact same principles were followed for the other three functions
def extnd_powell(\bm{x}: np.ndarray) -> float:
    #Evaluation of the Extended Powell function in the point x
    num = x.shape[0]
    if num % 4 != 0:
        raise Exception("Array length must be multiple of 4.")

```

```
    def f(\bm{x}: np.ndarray, k: int) -> float:
        match k % 4:
            case 1:
                k -= 1
                return x[k] + 10*x[k+1]
            case 2:
                k -= 1
                return sqrt(5)*(\bm{x}[k+1]-x[k+2])
            case 3:
                k -= 1
                return (\bm{x}[k-1] - 2*x[k])**2
            case 0:
                k -= 1
                return sqrt(10)*(\bm{x}[k-3] - x[k])**2

    z = np.empty(num)
    for k in range(0, num):
        z[k] = f(\bm{x}, k+1)
    return (0.5 * np.sum(z**2))


def grad_extnd_powell(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
    '''
    Compute the appoximation of the gradient via finite differences or with the true gradient

    INPUTS:
    x: point in which I want to evaluate the gradient of the function;
    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;

    OUTPUTS:
    gradfx=the appossimation of the gradient in x;
    '''
    num = x.shape[0]
    if num % 4 != 0:
        raise Exception("Array length must be multiple of 4.")

    def df(\bm{x}: np.ndarray, k: int) -> float:
        xk = x[k-1]
        match k % 4:
            case 1:
                k -= 1
                return xk + 10*x[k+1] + 20*(xk - x[k+3])**3
            case 2:
                k -= 1
                return 10*(\bm{x}[k-1]+10*xk) + 2*(xk-2*x[k+1])**3
            case 3:
                k -= 1
                return 5*(xk - x[k+1]) + 4*(2*xk - x[k-1])**3
            case 0:
                k -= 1
                return 5*(xk - x[k-1]) + 20*(xk - x[k-3])**3

    grad = np.empty(num)
    if fin_diff == True:
        e = np.identity(num)
        if (type == "fw" or type == "bw"):
            fx = extnd_powell(\bm{x})
        for i in range(0, num):
            if(type == "fw"):
                grad[i] = (extnd_powell(\bm{x}+h*e[i, :]) - fx) / h
            elif(type == "bw"):
                grad[i] = -(extnd_powell(\bm{x}-h*e[i, :]) - fx) / h
            else:
                grad[i] = (extnd_powell(\bm{x}+h*e[i, :]) - extnd_powell(\bm{x}-h*e[i, :])) / (2*h)
    else:
        for k in range(0, num):
            grad[k] = df(\bm{x}, k+1)
    return grad


def banded_trig(\bm{x}: np.ndarray) -> float:
    #Evaluation of the Banded trigonometric problem in the point x
```

```
    num = x.shape[0]
    if num < 2:
        raise Exception("Array length must be equal or higher than 2.")

    z = np.empty(num)

    #first iteration, different from the others
    z[0] = 1 - cos(\bm{x}[0]) - sin(\bm{x}[1])

    for k in range(1, num-1):
        z[k] = (k + 1) * (1-cos(\bm{x}[k]) + sin(\bm{x}[k-1]) - sin(\bm{x}[k+1]))

    #last iteration, different from the others
    z[num-1] = num * (1 - cos(\bm{x}[num-1]) + sin(\bm{x}[num-2]))
    return (np.sum(z))


def grad_banded_trig(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
    '''
    Compute the appoximation of the gradient via finite differences or with the true gradient

    INPUTS:
    x: point in which I want to evaluate the gradient of the function;
    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;

    OUTPUTS:
    gradfx=the appossimation of the gradient in x;
    '''
    num = x.shape[0]
    if num < 2:
        raise Exception("Array length must be equal or higher than 2.")

    grad = np.empty(num)
    if fin_diff == True:
        e = np.identity(num)
        if (type == "fw" or type == "bw"):
            fx = banded_trig(\bm{x})
        for i in range(0, num):
            if(type == "fw"):
                grad[i] = (banded_trig(\bm{x}+h*e[i, :]) - fx) / h
            elif(type == "bw"):
                grad[i] = -(banded_trig(\bm{x}-h*e[i, :]) - fx) / h
            else:
                grad[i] = (banded_trig(\bm{x}+h*e[i, :]) - banded_trig(\bm{x}-h*e[i, :])) / (2*h)
    else:
        #first iteration, different from the others
        grad[0] = (sin(\bm{x}[0]) + 2*cos(\bm{x}[0]))

        for k in range(2, num):
            grad[k-1] = -(k-1)*cos(\bm{x}[k-1]) + k*sin(\bm{x}[k-1]) + (k+1)*cos(\bm{x}[k-1])

        #last iteration, different from the others
        grad[num-1] = -(num-1)*cos(\bm{x}[num-1]) + num*sin(\bm{x}[num-1])
    return grad


def extnd_rosenb(\bm{x}: np.ndarray) -> float:
    #Evaluation of the Extended Rosenbrock function in the point x
    num = x.shape[0]
    if num % 2 != 0:
        raise Exception("Array length must be multiple of 2.")

    def f(\bm{x}: np.ndarray, k: int) -> float:
        match k % 2:
            case 1:
                k -= 1
                return 10*(\bm{x}[k]**2 - x[k+1])
            case 0:
                k -= 1
                return x[k-1] - 1

    z = np.empty(num)
```

```
    for k in range(0, num):
        z[k] = f(\bm{x}, k+1)
    return (0.5 * np.sum(z**2))


def grad_extnd_rosenb(\bm{x}: np.ndarray, fin_diff: bool, type: str) -> np.ndarray:
    '''
    Compute the appoximation of the gradient via finite differences or with the true gradient

    INPUTS:
    x: point in which I want to evaluate the gradient of the function;
    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;

    OUTPUTS:
    gradfx=the appossimation of the gradient in x;
    '''
    num = x.shape[0]
    if num % 2 != 0:
        raise Exception("Array length must be multiple of 2.")

    def df(\bm{x}: np.ndarray, k: int) -> float:
        xk = x[k-1]
        match k % 2:
            case 1:
                k -= 1
                return 200*xk**3 - 200*xk*x[k+1] + xk - 1
            case 0:
                k -= 1
                return 100*(xk - x[k-1]**2)

    grad = np.empty(num)
    if fin_diff == True:
        e = np.identity(num)
        if (type == "fw" or type == "bw"):
            fx = extnd_rosenb(\bm{x})
        for i in range(0, num):
            if(type == "fw"):
                grad[i] = (extnd_rosenb(\bm{x}+h*e[i, :]) - fx) / h
            elif(type == "bw"):
                grad[i] = -(extnd_rosenb(\bm{x}-h*e[i, :]) - fx) / h
            else:
                grad[i] = (extnd_rosenb(\bm{x}+h*e[i, :]) - extnd_rosenb(\bm{x}-h*e[i, :])) / (2*h)
    else:
        for k in range(0, num):
            grad[k] = df(\bm{x}, k+1)
    return grad
```