LINMA2460 – Project 1
Methods for Smooth Constrained Minimization over a Simple Convex Set

By Gilles Peiffer

Abstract

In this short paper, we present a study of two methods for smooth constrained minimization over a simple convex set: the gradient method and the optimal method. We show that properties predicted by theory are observable in practice on a diverse set of test parameters, while also investigating the performance and sensitivity of the methods with respect to variations of the test set and characteristics of the problem. For reproducibility purposes, the full computation results and source code are also made available.

Contents

© 2020 Gilles Peiffer.

1. Introduction

Smooth constrained minimization is one of the most extensively studied topics in mathematical optimization. In this paper, we look at two of the most well-known methods in this domain: the gradient method, and the optimal method.

In §2, the problem formulation and the mathematical background of the methods is explained in more depth. §3 expands on the evaluation method used to compare the methods on the various test problems, whereas §4 comments on the results of the tests. Finally, in §5, we give a conclusion of the exercise. The appendix contains the full computation results, as well as the source code in Python used to generate these results.

All throughout the paper, we use the notation of [3].

2. Description of problems and numerical methods

In this section, we briefly explain the problem and methods used in the rest of the paper.

2.1. *Problem description.* We are concerned with the problem

$$\min_{x\in Q} f(x), \tag{2.1}$$

where Q is a *simple* convex set (where simple is taken to mean "on which points can be projected analytically"), and f is a continuous, strongly convex function with Lipschitz-continuous gradient. We also assume that f is twice differentiable, and that for all $x \in \mathbb{R}^n$, we have

$$\mu I_n \preceq f''(x) \preceq L I_n, \quad \mu > 0. \tag{2.2}$$

2.1.1. *Objective function.* For simplicity, we define f to be of the form

$$f(x) = \frac{\alpha}{2}\|x - x^*\|^2 + \beta f_1(x - x^*) + \gamma\big(f_2(x) - f_2(x^*) - \langle f_2'(x^*), x - x^*\rangle\big), \tag{2.3}$$

where $\alpha, \beta, \gamma > 0$ and

$$f_1(x) = \frac{1}{2}\left(\left(x^{(1)}\right)^2 + \sum_{i=1}^{n-1}\left(x^{(i)} - x^{(i+1)}\right)^2 + \left(x^{(n)}\right)^2\right), \tag{2.4}$$

$$f_2(x) = \ln\sum_{i=1}^{n}\exp\left(x^{(i)}\right), \tag{2.5}$$

where for numerical stability, we keep f_2 in the form

$$f_2(x) = \delta + \ln\sum_{i=1}^{n}\exp\left(x^{(i)} - \delta\right), \tag{2.6}$$

with $\delta \geqslant \max_i x^{(i)}$.[1]

This then allows one to estimate the parameters of the objective function as $\mu = \alpha$ and $L = \alpha + 4\beta + \gamma$. These values are considered known, and can thus be used in the optimization methods. We can hence write

$$f \in \mathscr{S}^{2,1}_{\mu=\alpha, L=\alpha+4\beta+\gamma}(Q, \|\cdot\|), \tag{2.7}$$

where the norm is always assumed to be the Euclidean norm.

[1] In practice, $\delta = \max_i x^{(i)}$ was chosen.

2.1.2. *Feasible set.*

Definition 2.1. A set $Q \subseteq \mathbb{R}^n$ is called *convex* if for any $x, y \in Q$ and α from $[0, 1]$ we have

$$\alpha x + (1 - \alpha y) \in Q. \tag{2.8}$$

The feasible set Q must be *convex* set, and to simplify the implementation, we also require it to be *simple*, that is, on which points can be projected analytically.

More precisely, we work with three types of sets:[2]

(1) A *ball*:

$$Q = \{x \in \mathbb{R}^n : \|x\| \leqslant R\}, \tag{2.9}$$

where R is the radius of the ball.

(2) A *box*:

$$Q = \left\{x \in \mathbb{R}^n : a^{(i)} \leqslant x^{(i)} \leqslant b^{(i)}, \quad i = 1, \ldots, n\right\}, \tag{2.10}$$

where a and b are vectors delimiting the box.

(3) A *simplex*:

$$Q = \{x \in \mathbb{R}^n : \sum_{i=1}^{n} x^{(i)} = p, x^{(i)} \geqslant 0, i = 1, \ldots, n\}, \tag{2.11}$$

where p is a parameter.

2.2. *Optimization methods.* We use two numerical methods to solve the problem of §2.1: the gradient method and the optimal method.

2.2.1. *Gradient mapping.* Both methods we use are based on the notion of gradient mapping. As defined in [3], the gradient mapping is a mathematical object which preserves the two most important properties of the gradient:

(1) The step along the direction of the anti-gradient decreases the function value by an amount comparable with the squared norm of the gradient:

$$f\left(x - \frac{1}{L} f'(x)\right) \leqslant f(x) - \frac{1}{2L} \left\|f'(x)\right\|^2. \tag{2.12}$$

(2) The inequality

$$\langle f'(x), x - x^* \rangle \geqslant \frac{1}{L} \left\|f'(x)\right\|^2. \tag{2.13}$$

Let us choose some $\bar{x} \in Q$. The gradient mapping $x_Q(\bar{x})$ and the reduced gradient $g_Q(\bar{x})$ of f on Q are then defined as

$$x_Q(\bar{x}) = \arg\min_{x \in Q} \left(f(\bar{x}) + \langle f'(\bar{x}), x - \bar{x} \rangle + \frac{L}{2} \|x - \bar{x}\|^2\right), \tag{2.14}$$

$$g_Q(\bar{x}) = L(\bar{x} - x_Q(\bar{x})). \tag{2.15}$$

Using simple manipulations, one can transform the objective function of (2.14) into the form

$$f(\bar{x}) - \frac{1}{2L} \left\|f'(\bar{x})\right\|^2 + \frac{L}{2} \left\|x - \left(\bar{x} - \frac{1}{L} f'(\bar{x})\right)\right\|^2. \tag{2.16}$$

[2] A fourth type of set, the positive orthant, was also examined, but due to the limitations of the numerical simulations was in practice equivalent to a box.

From this form, it is immediately apparent that computing the gradient mapping $x_Q(\bar{x})$ is equivalent to solving the following optimization problem:

$$\min_{x\in Q}\left\|x-\left(\bar{x}-\frac{1}{L}f'(\bar{x})\right)\right\|^2. \tag{2.17}$$

Solving this problem is equivalent to finding the Euclidean projection of $y=\bar{x}-\frac{1}{L}f'(\bar{x})$ onto the set Q, which we denote by $\pi_Q(y)$.

For the simple sets of §2.1.2, this projection can be found analytically.

(1) For the ball, the projection can simply be computed as

$$\pi_Q(y)=\begin{cases} y, & \text{if } \|y\|\leqslant R,\\ Ry/\|y\|, & \text{otherwise.}\end{cases} \tag{2.18}$$

(2) For the box, the projection becomes

$$\pi_Q(y)^{(i)}=\begin{cases} b^{(i)}, & \text{if } \pi_Q(y)^{(i)}\geqslant b^{(i)},\\ \pi_Q(y)^{(i)}, & \text{if } a^{(i)}\leqslant \pi_Q(y)^{(i)}\leqslant b^{(i)},\\ a^{(i)}, & \text{if } \pi_Q(y)^{(i)}\leqslant a^{(i)}.\end{cases} \tag{2.19}$$

(3) For the simplex, the algorithm presented in [1] is used:
   (a) The algorithm takes an input $y=(y^{(1)},\dots,y^{(n)})^T\in\mathbb{R}^n$.
   (b) Sort y in ascending order as $y^{\{1\}}\leqslant\cdots\leqslant y^{\{n\}}$ and set $i=n-1$.
   (c) Compute $t_i=\frac{\sum_{j=i+1}^n y^{\{j\}}-p}{n-i}$. If $t\geqslant y^{\{i\}}$, set $\hat{t}=t_i$ and go to Step 3e. Otherwise decrement i and go to Step 3d if $i=0$, otherwise repeat Step 3c.
   (d) Set $\hat{t}=\frac{\sum_{j=1}^n y^{\{j\}}-p}{n}$.
   (e) Return $(y-\hat{t})_+$ as the projection of y onto the simplex.

2.2.2. *Gradient method.* The gradient method can be defined by the following iteration scheme:

(1) Choose an initial point $x_0\in Q$ and iterate from there.
(2) For $k\geqslant 0$, set $x_{k+1}=x_Q(x_k)$.

We thus have the following theorem.

THEOREM 2.1 (Theorem 2.2.14 of [3]). *Let $f\in\mathscr{S}^{1,1}_{\mu,L}(\mathbb{R}^n)$. Then*

$$\|x_k-x^*\|\leqslant\left(1-\frac{\mu}{L}\right)^k\|x_0-x^*\|. \tag{2.20}$$

Theorem 2.1 allows us to give an upper bound on the distance from the optimum at every iteration. The result of Theorem 2.1 is also stronger than the one in Theorem 2.2.8 of [2].

We also remember the following result, which holds in the unconstrained, "weakly" convex case, but can still be useful.

COROLLARY 2.2 (Corollary 2.1.2 of [3]). *Let $f\in\mathscr{F}^{1,1}_L(\mathbb{R}^n)$, then*

$$f(x_k)-f(x^*)\leqslant\frac{2L\|x_0-x^*\|^2}{k+4}. \tag{2.21}$$

2.2.3. *Optimal method.* The optimal method can be defined by the following iteration scheme:

(1) Choose an initial point $y_0=x_0\in Q$ and define $\beta=\frac{\sqrt{L}-\sqrt{\mu}}{\sqrt{L}+\sqrt{\mu}}$.
(2) For $k\geqslant 0$, set $x_{k+1}=x_Q(y_k)$, where $y_{k+1}=x_{k+1}+\beta(x_{k+1}-x_k)$.

This is method (2.2.63) of [3], with $q_f=\mu/L$ and $\alpha_k=\sqrt{\mu/L}$. We can thus refer to the following result on its rate of convergence (which is mentioned under the definition of method (2.2.63) in [3]).

THEOREM 2.3 ((2.2.23) of [3]). *The optimal method provides a sequence* $\{x_k\}$ *such that*

$$f(x_k) - f(x^*) \leqslant \frac{L+\mu}{2} \|x_0 - x^*\|^2 \exp(-k\sqrt{\mu/L}). \tag{2.22}$$

Another result is given by Theorem 2.2.3 of [2]:[3]

THEOREM 2.4 (Theorem 2.2.3 of [2]). *The optimal method provides a sequence* $\{x_k\}$ *such that*

$$f(x_k) - f(x^*) \leqslant \min\left\{\left(1-\frac{\mu}{L}\right)^k, \frac{4L}{(2\sqrt{L}+k\sqrt{\mu})^2}\right\}\left(f(x_0) - f(x^*) + \frac{\mu}{2}\|x_0 - x^*\|^2\right). \tag{2.23}$$

2.2.4. *Complexity lower bound.* We also give the result of Theorem 2.1.13 of [3].

THEOREM 2.5 (Theorem 2.1.13 of [3]). *For any* $x_0 \in \mathbb{R}^\infty$ *and any constants* $\mu > 0, L/\mu > 1$, *there exists a function* $f \in \mathscr{S}^{\infty,1}_{\mu,L}(\mathbb{R}^\infty)$ *such that for most first-order methods* (*including the gradient and optimal methods*),

$$\|x_k - x^*\| \geqslant \left(\frac{\sqrt{L/\mu}-1}{\sqrt{L/\mu}+1}\right)^k \|x_0 - x^*\|, \tag{2.24}$$

$$f(x_k) - f(x^*) \geqslant \frac{\mu}{2}\left(\frac{\sqrt{L/\mu}-1}{\sqrt{L/\mu}+1}\right)^{2k} \|x_0 - x^*\|^2. \tag{2.25}$$

While the conditions for applying Theorem 2.5 are not exactly those of our situation, they can still provide useful references for evaluating the various methods. The way to interpret is to consider it as saying that there exists some function which would yield an optimization process which cannot converge faster than a given value by the theorem. As we do not know this function, there are no guarantees that this lower bound would apply to our case. However, we can make the assumption that our function is not too far removed from this pathological one, and would thus give rise to similar optimization processes.

3. **Description of the set of test problems and testing strategy**

3.1. *Parameters.* Several parameters influence the results:

- The type of feasible set (§2.1.2) and its parameters. This also includes the choice of the optimal solution, which is done at random inside Q.
- The type of method (gradient or optimal).
- The objective function. We choose a function such as the one in §2.1.1, hence three parameters α, β, γ need to be chosen. This choice also influences the condition number $\kappa = \frac{L}{\mu} = \frac{\alpha+4\beta+\gamma}{\alpha}$ of the problem. We choose to fix these parameters as $\alpha = 2/(\kappa-1), \beta = 0.25, \gamma = 1$, thus leaving the condition number as a changeable parameter of the problem.
- The desired accuracy of the final solution, ε. As the objective function we choose always has $f(x^*) = 0$, the termination criterion of the methods is $f(x_k) \leqslant \varepsilon$.
- The initial distance to the minimum, $R = \|x_0 - x^*\|$. This distance is taken into account when randomly generating the initial solution.
- The dimension n of the problem.
- (For the box and the simplex.) The number of active constraints m at x^*. If Q is a box, then $\lfloor m/2 \rfloor$ components of x^* are set to a and $\lceil m/2 \rceil$ are set to b. If Q is a simplex, then m components of x^* are set to 0.

[3] In practice, this result is stronger at earlier iterations.

3.2. *Testing strategy.* Our testing procedure can be divided into two parts, each with its own goal:

(1) In the first part, we test different problems with the following parameters (and a maximum number of iterations of 1.5×10^5).

Problem size	ε	κ	R	n	m
Small	10^{-12}	10	10	10	0
Moderate	10^{-12}	10^3	10^2	10^2	0
High	10^{-12}	10^6	10^3	10^3	0

We show the evolution of two values: the accuracy $f(x_k) - f(x^*)$, and the distance from the minimum $\|x_k - x^*\|$, at each iteration.

(2) In the second part, we look at the influence of changing each parameter individually, while maintaining the others fixed, on the number of iterations required to reach a given accuracy. For this, we use the default values of the moderate size problem of the previous tests, while varying the other parameters in the following ranges, and with $\varepsilon = 10^{-12}$:
   - $\kappa \in \{10, 10^2, 10^3, \dots, 10^{10}\}$;
   - $R \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$;
   - $n \in \{10, 20, 50, 100, 200, 500, 1000, 2000\}$;
   - $m \in \{0, 5, 10, \dots, 100\}$.

   We do this for both methods.

4. **Analysis of the results**

4.1. *Convergence tests.* The following figures show the results of the first test suite. For brevity, we only include in the main text the results on the ball; §A contains the full computation results. We first give the various figures, then interpret these results in §4.1.4.

4.1.1. *Small example.* Figure 1 shows the evolution as the number of iterations increases of the distance to the minimum ($\|x_k - x^*\|$), whereas Figure 2 shows the evolution of the accuracy, $f(x_k) - f(x^*)$.

4.1.2. *Moderate example.* Figure 3 shows the evolution as the number of iterations increases of the distance to the minimum ($\|x_k - x^*\|$), whereas Figure 4 shows the evolution of the accuracy, $f(x_k) - f(x^*)$.

4.1.3. *Large example.* Figure 5 shows the evolution as the number of iterations increases of the distance to the minimum ($\|x_k - x^*\|$), whereas Figure 6 shows the evolution of the accuracy, $f(x_k) - f(x^*)$.

4.1.4. *Conclusion.* From the results of this part, we can see several things:

- The various upper bounds predicted by the theory are always respected, when relevant even though in the case of Corollary 2.2, this should not necessarily be the case (as it concerns a relaxed version of the problem).
- The lower bounds are mostly verified, though at the earlier iterations, the distance to the minimum is sometimes less than what would be predicted by Theorem 2.5. This can be explained by the fact that, as mentioned before, the conditions to apply the theorem are not quite satisfied. However, as the figures show, the lower bound seems to be respected as k approaches infinity.
- The optimal method performs significantly better than the gradient method in every single case. This is due to the inertia step of the former, which allows it to converge much faster.

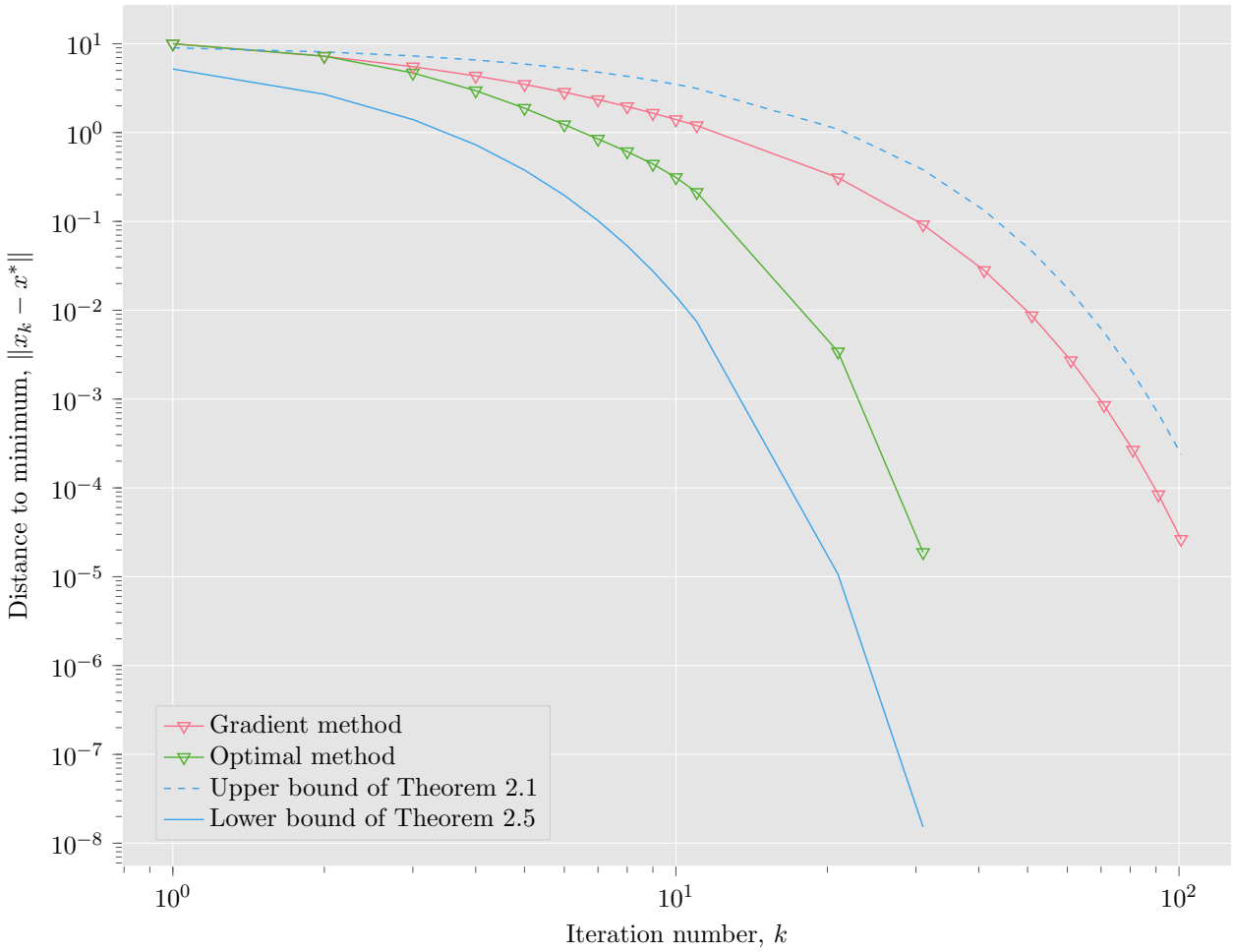


Figure 1. Evolution of the distance to the minimum, for the ball on the small example.

- The larger the test problem, the harder it is to reach a given accuracy. This observation is explored in more detail when considering the influence of n in §4.2. Similarly, the ball seems to be the hardest of the three types of feasible sets, followed by the box and then the simplex. This is also explored further in §4.2.
- There is a fair amount of jitter on the iterations, most notably for the optimal method. This is most likely a byproduct of the random numbers being used and the numerical inaccuracy involved in the calculations.

4.2. *Influence of parameters.* The following figures show the results of the second test suite. For brevity, we only include in the main text the results on the box; §A contains the full computation results.

4.2.1. *Condition number.* Figure 7 shows the influence of the condition number on the required number of iterations.

As we can see, the higher κ, the higher the required number of iterations. This result is expected; in Theorem 2.1, the higher the condition number $\kappa = L/\mu$, the smaller the difference between the initial distance from the minimum and the distance from the minimum at iteration k. Theorem 2.3 and Theorem 2.4 show the same behaviour. Similarly, Corollary 2.2 takes into account the condition number implicitly due to the parameter L, which increases as the condition number increases, hence increasing the lower bound on the number of iterations to reach a given accuracy.

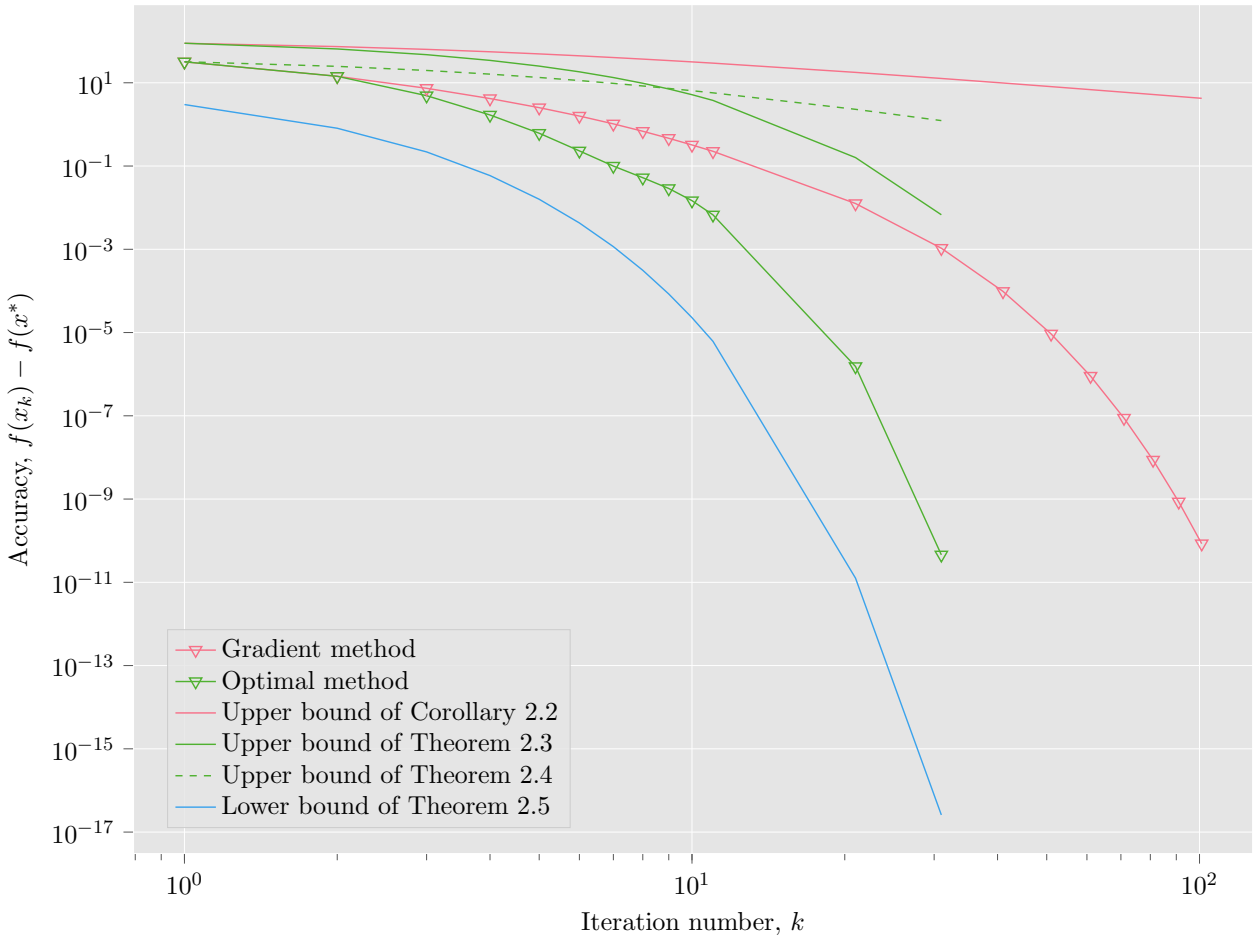


Figure 2. Evolution of the accuracy, for the ball on the small example.

One can also adequately explain the nearly flat curve for higher values of κ, as in the theorems referenced in the previous paragraph, the behaviour is asymptotic (i.e. $1 + \mu/L$ does not diverge, but it gets nearer to one as κ increases).

4.2.2. *Number of active constraints.* Figure 8 shows the influence of the number of active constraints on the required number of iterations.

There is no theoretical argument for why the behaviour we observe shows a decrease of the number of iterations as m increases. However, intuitively, one can see that since we used the projected version of the numerical methods, the iterates are more likely to be close to the optimal solution when it is heavily constrained (and thus lies on the boundaries on which we project iterates).

This also adequately explains why the optimal method seems to have an even stronger rate of decrease, as the inertia step is more likely to fall outside of the feasible set.

4.2.3. *Dimension of the problem.* Figure 9 shows the influence of the dimension of the problem on the required number of iterations.

Increasing the dimension of the problem seems to make it harder up to some point, after which the curve flattens. This result is surprising, as contrary to the case of the condition number, the dimension of the problem does not appear in the formulas given by the theorems we cite.

Again, we also note that the optimal method is better able to deal with the increase in dimensionality.

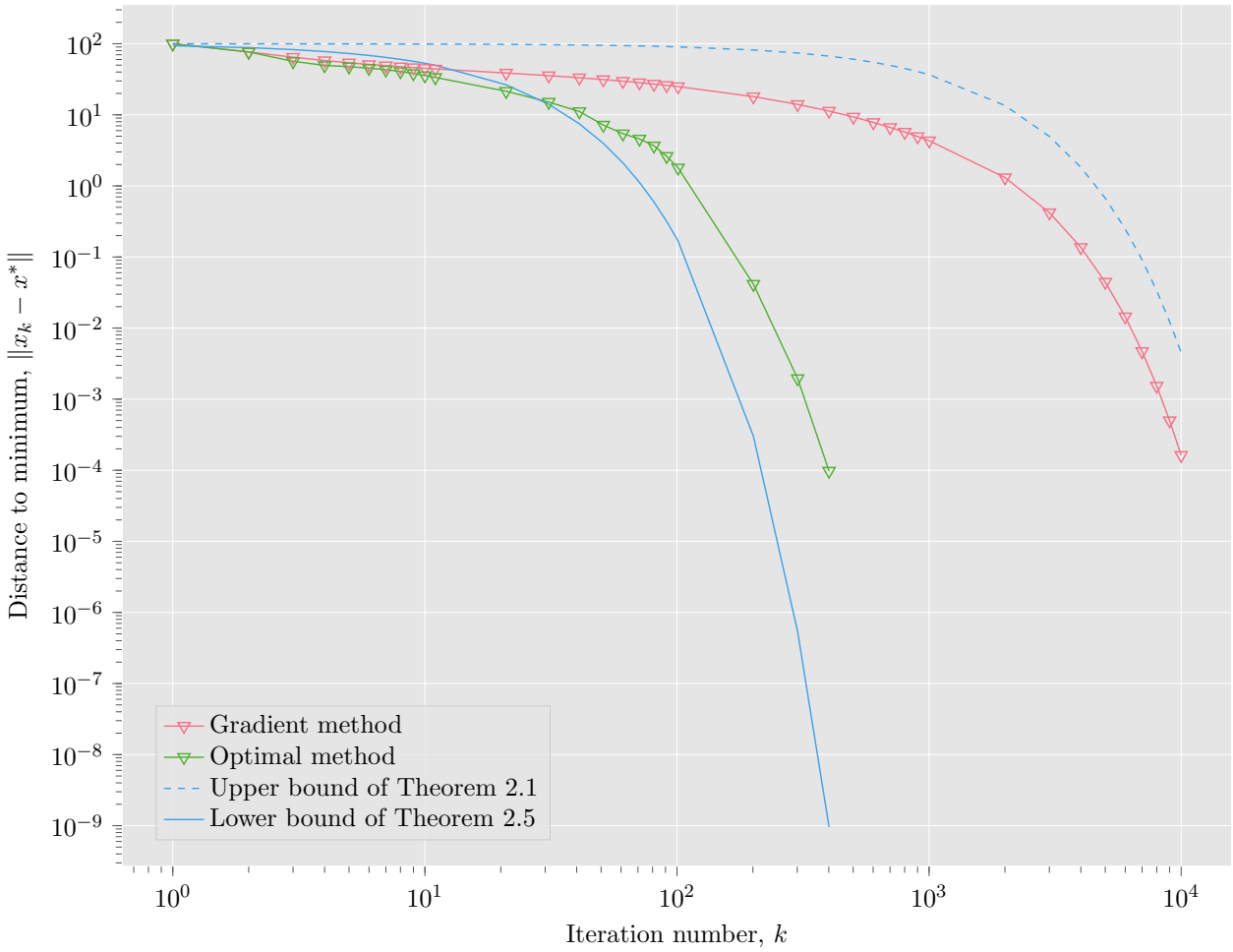


Figure 3. Evolution of the distance to the minimum, for the ball on the moderate example.

4.2.4. *Initial distance from the minimum.* Figure 10 shows the influence of the initial distance to the minimum on the required number of iterations.

The increase of the number of iterations when $\|x_k - x^*\|$ increases is perfectly logical when comparing it with theoretical results. We also note the perennial superiority of the optimal method over the gradient method. In the case of the initial distance from the minimum, this superiority can be explained quite clearly by the inertia step of the optimal method, which allows it to quickly close larger gaps.

5. **Conclusion**

In this paper, we have looked at two first-order numerical methods for smooth constrained minimization over a simplex convex set, the gradient method and the optimal method. We have extensively tested both methods, and compared practical results with theoretical findings, mainly from [3].

We have studied the influence of different set types as well as several problem parameters on the convergence of both methods, leading to the conclusion that in many cases, the optimal method is the best choice due to its efficiency, attained through the use of a so-called "inertia step".

Special care was taken to assure the reproducibility of the experiments, hence full computation results as well as complete source code listings are provided with the paper.

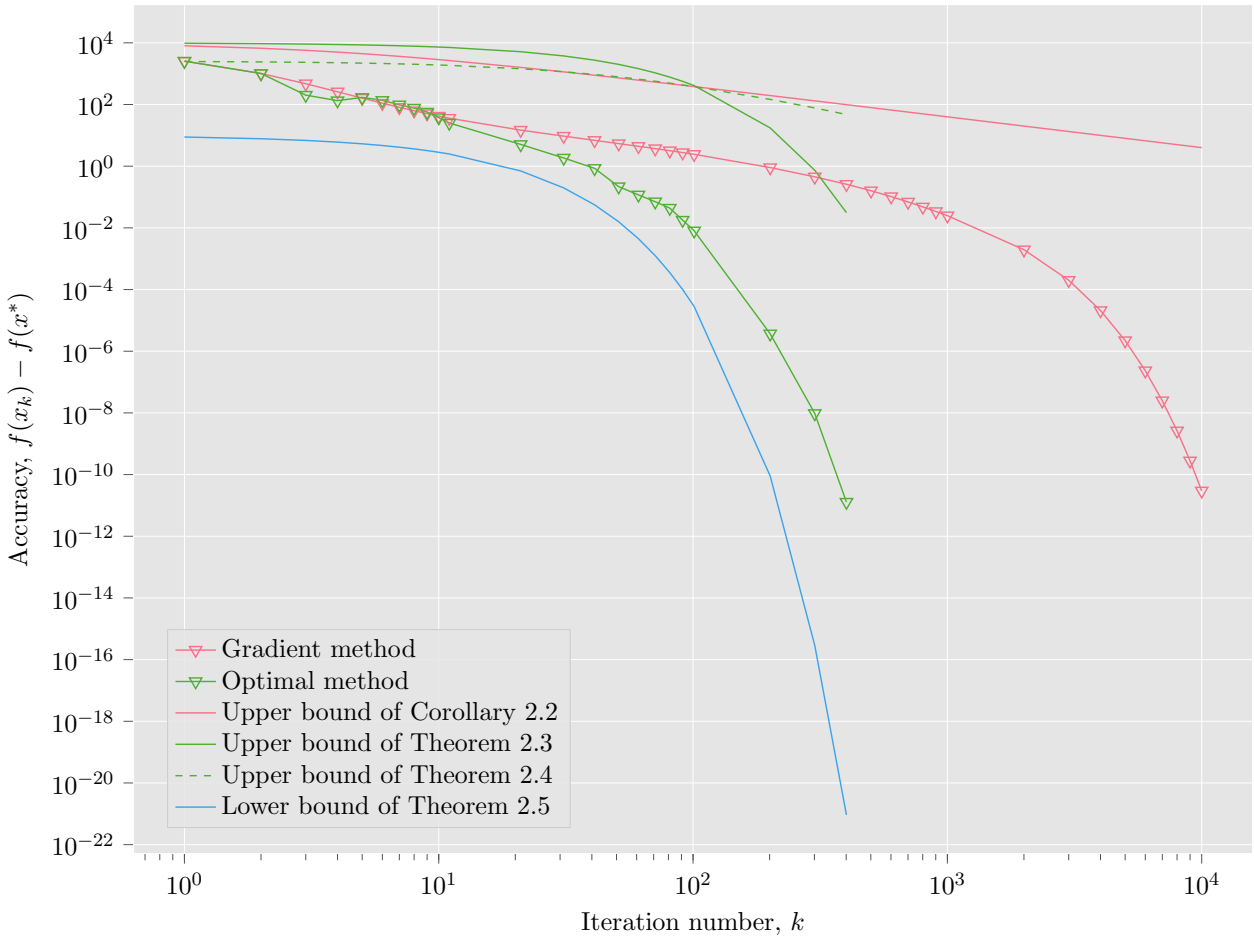


Figure 4. Evolution of the accuracy, for the ball on the moderate example.

References

[1] Y. Chen and X. Ye, Projection onto a simplex, 2011.
[2] Y. Nesterov, *Introductory Lectures on Convex Optimization*, *Applied Optimization*, Springer US, 2004. http://dx.doi.org/10.1007/978-1-4419-8853-9.
[3] Y. Nesterov, *Lectures on Convex Optimization*, *Springer Optimization and Its Applications*, Springer International Press, 2018. http://dx.doi.org/10.1007/978-3-319-91578-4.

Appendix A. Computation results

We give here the full computation results of the various tests (excluding those already presented in the main text).

A.1. *Convergence tests.*

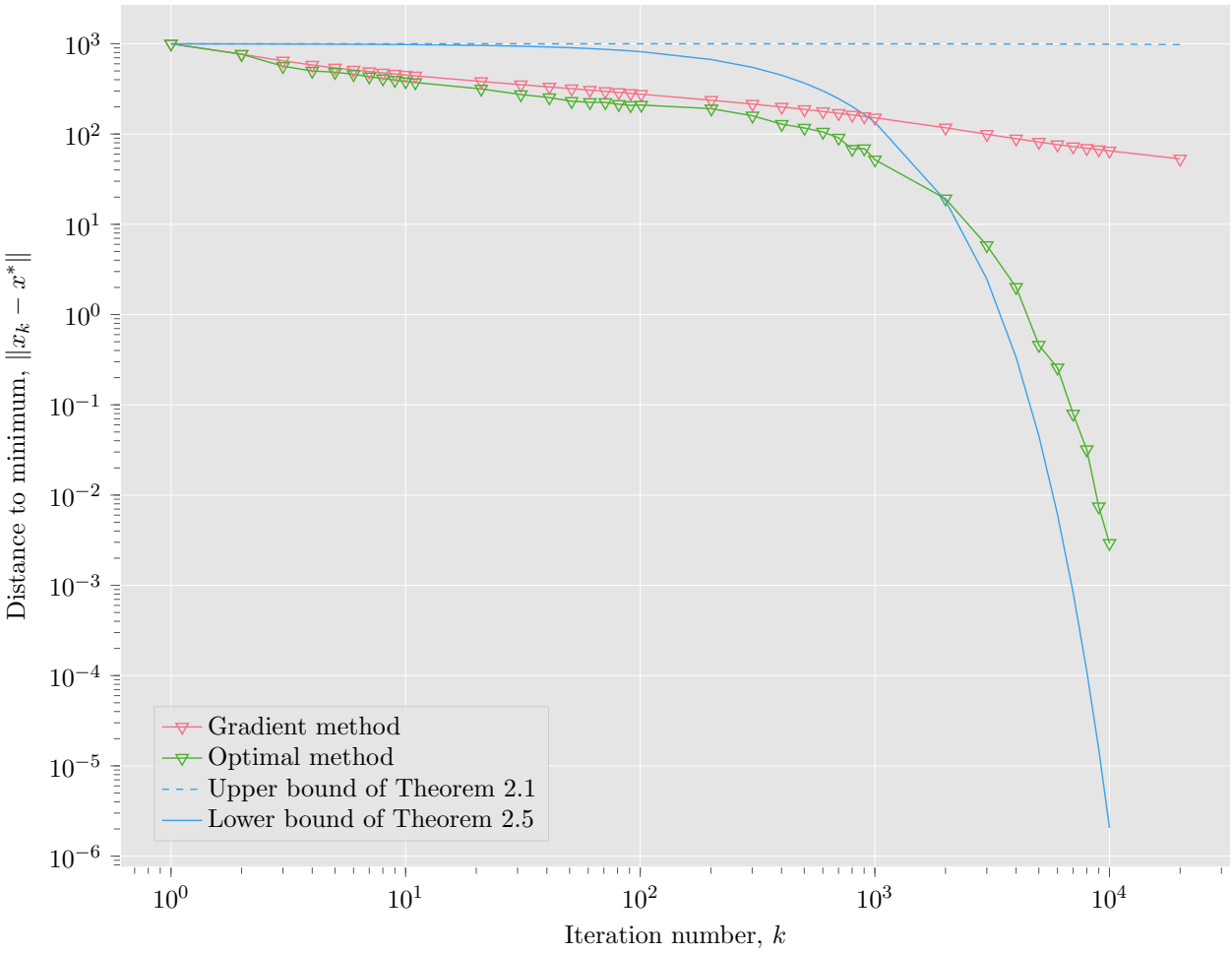


Figure 5. Evolution of the distance to the minimum, for the ball on the large example.

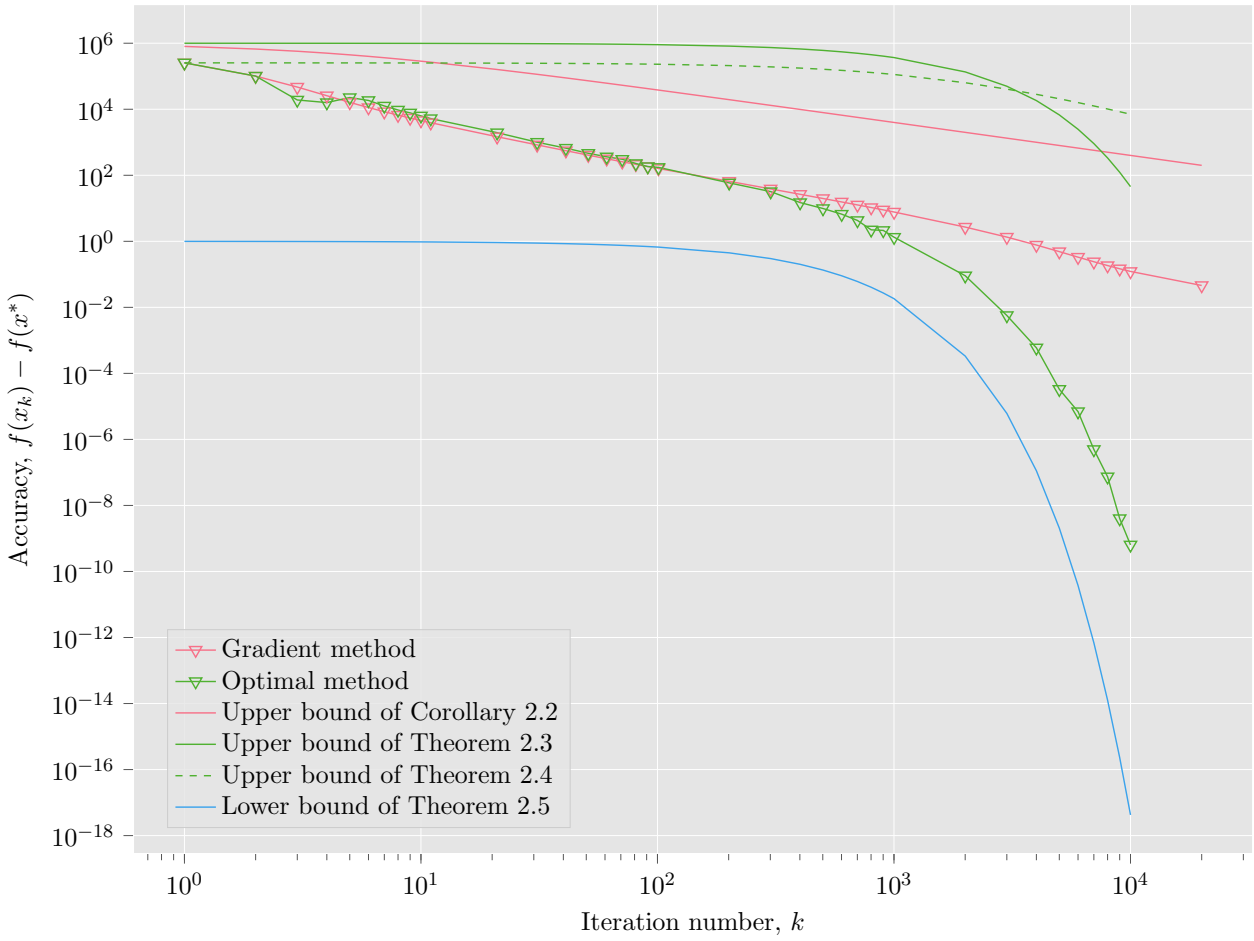
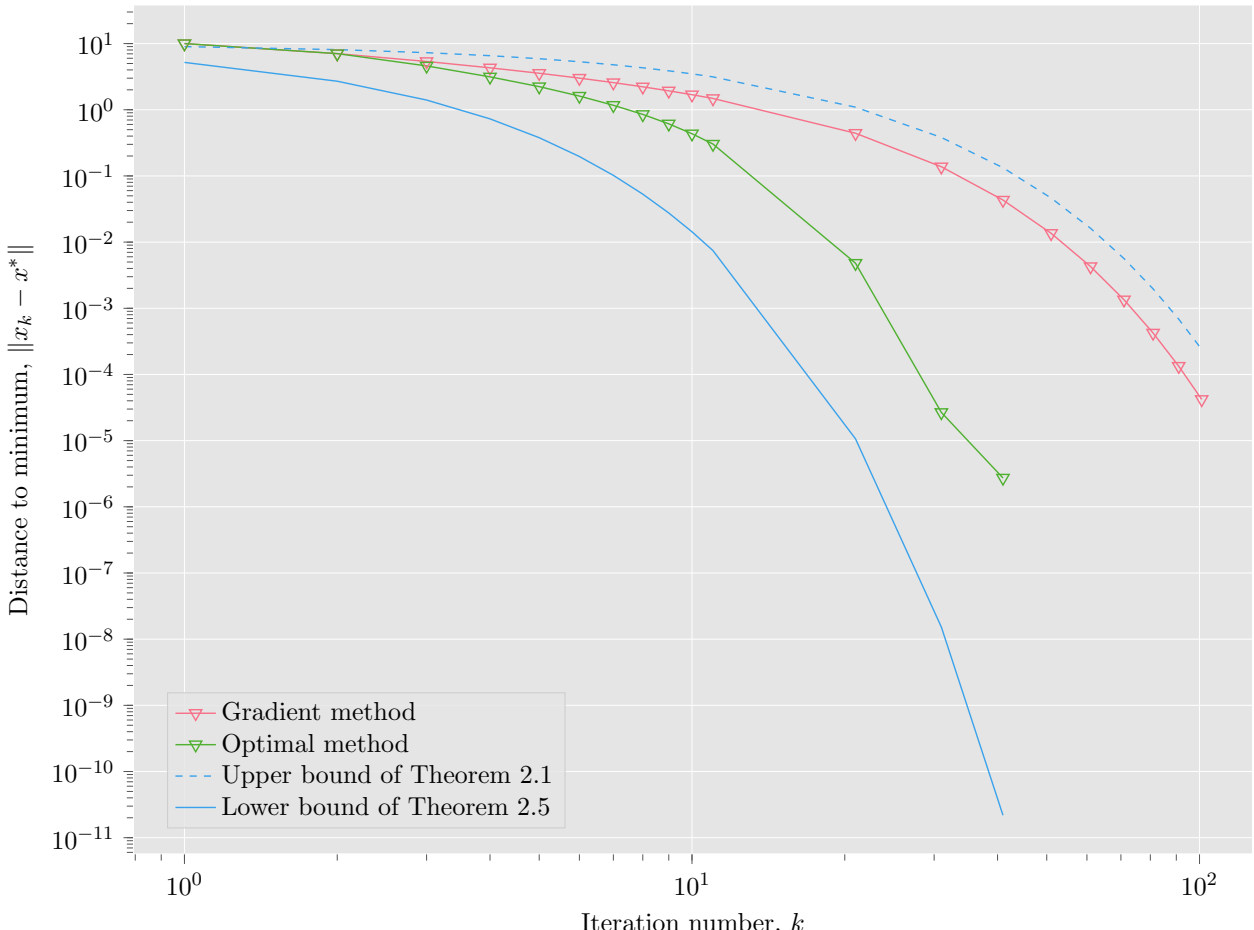


Figure 6. Evolution of the accuracy, for the ball on the large example.



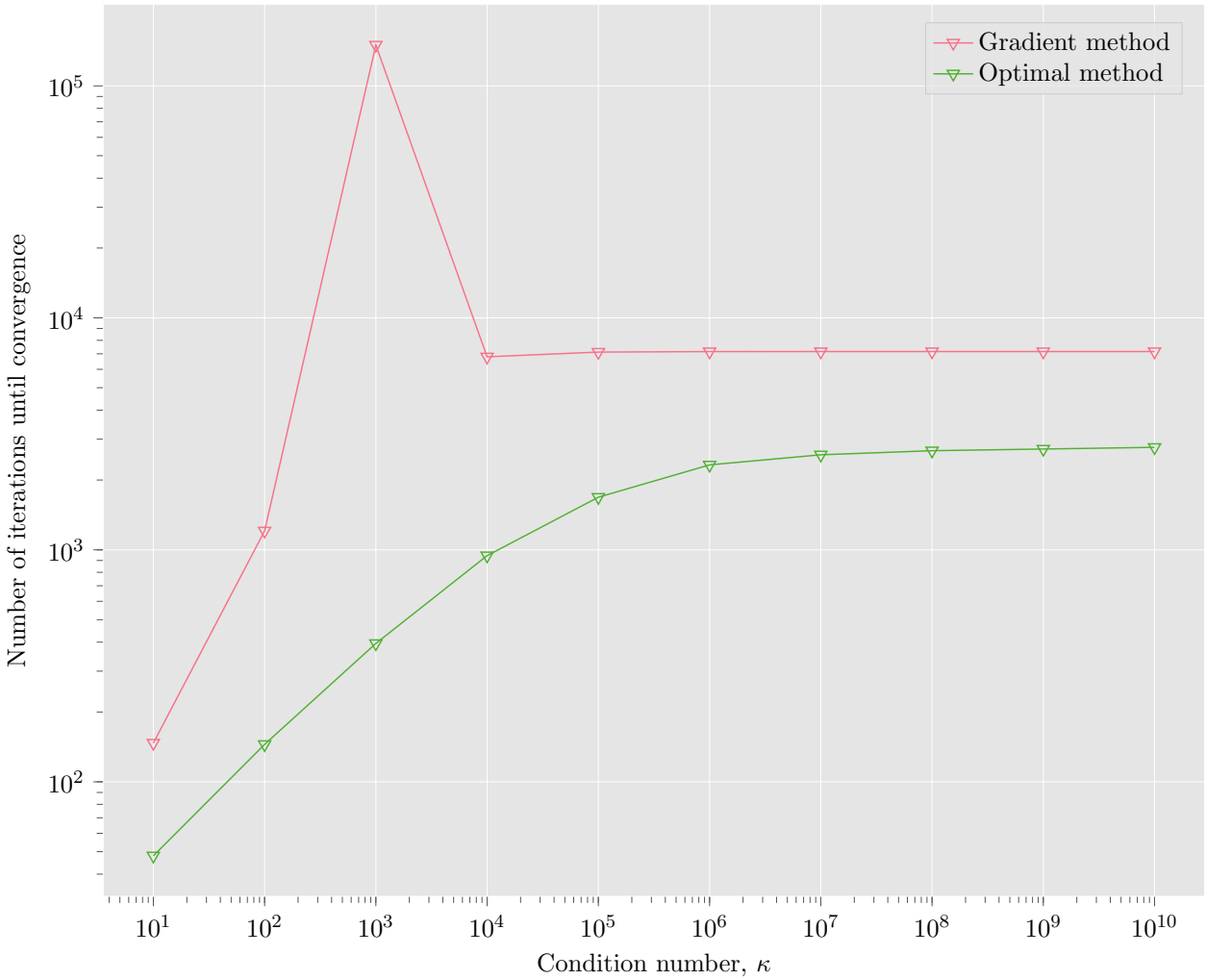
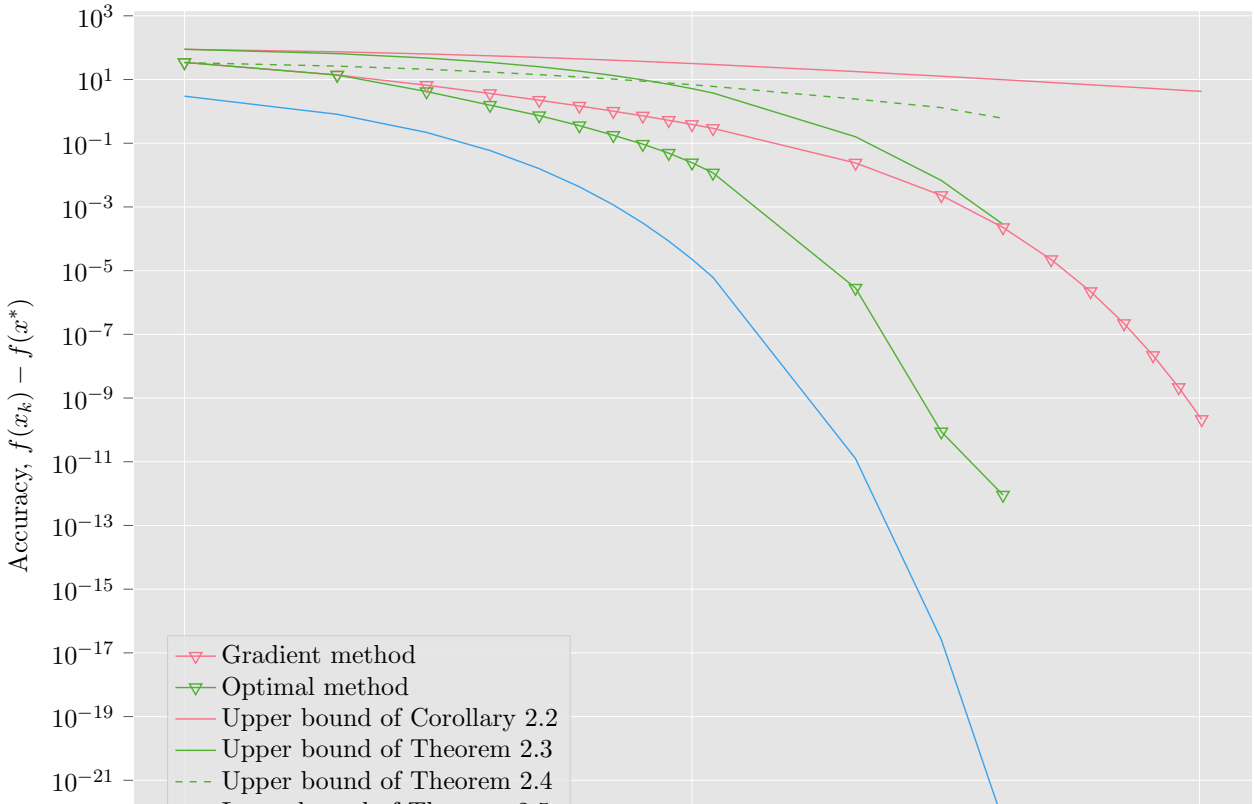


Figure 7. Evolution of the number of iterations required for convergence, for the box, depending on the condition number of the problem.



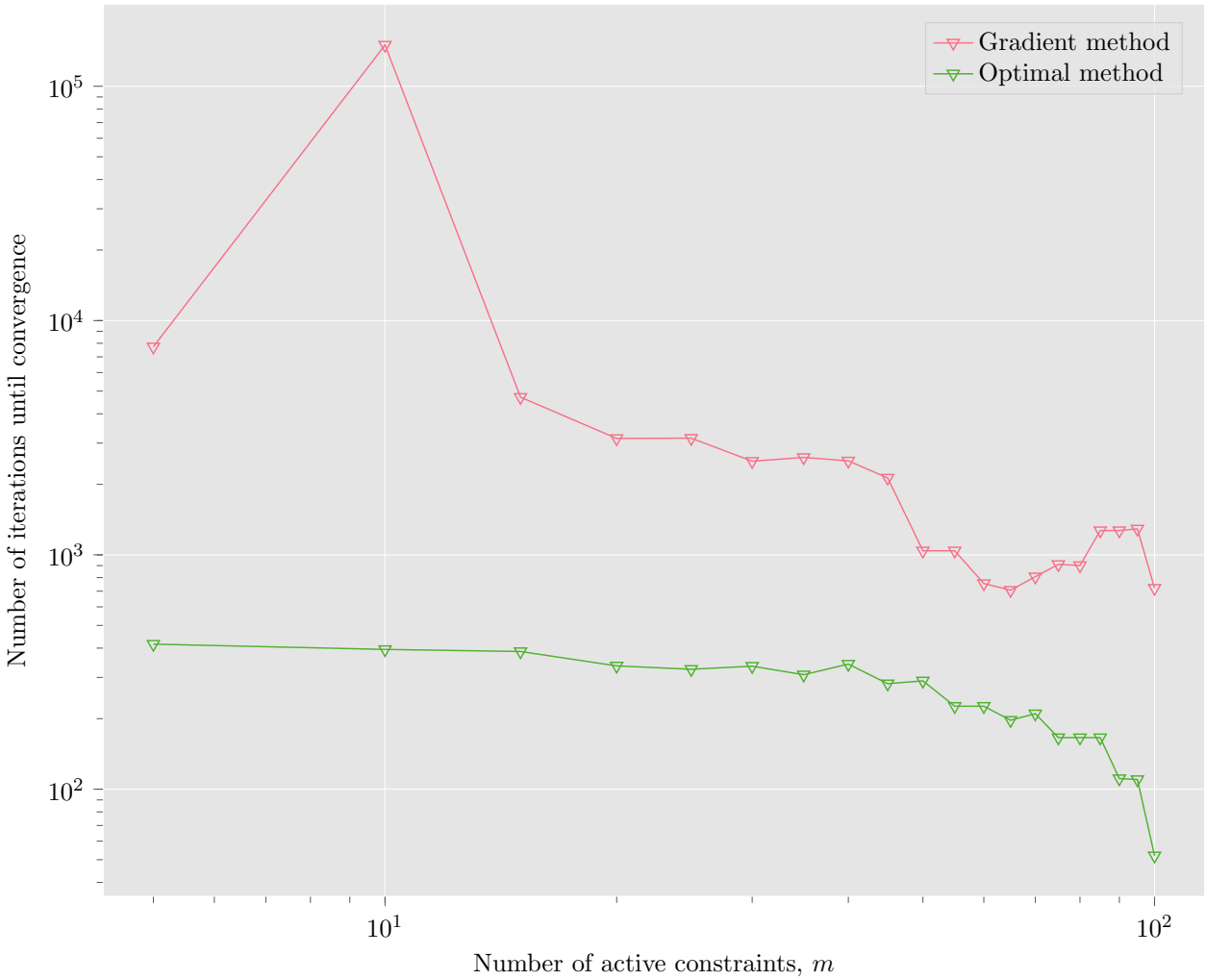
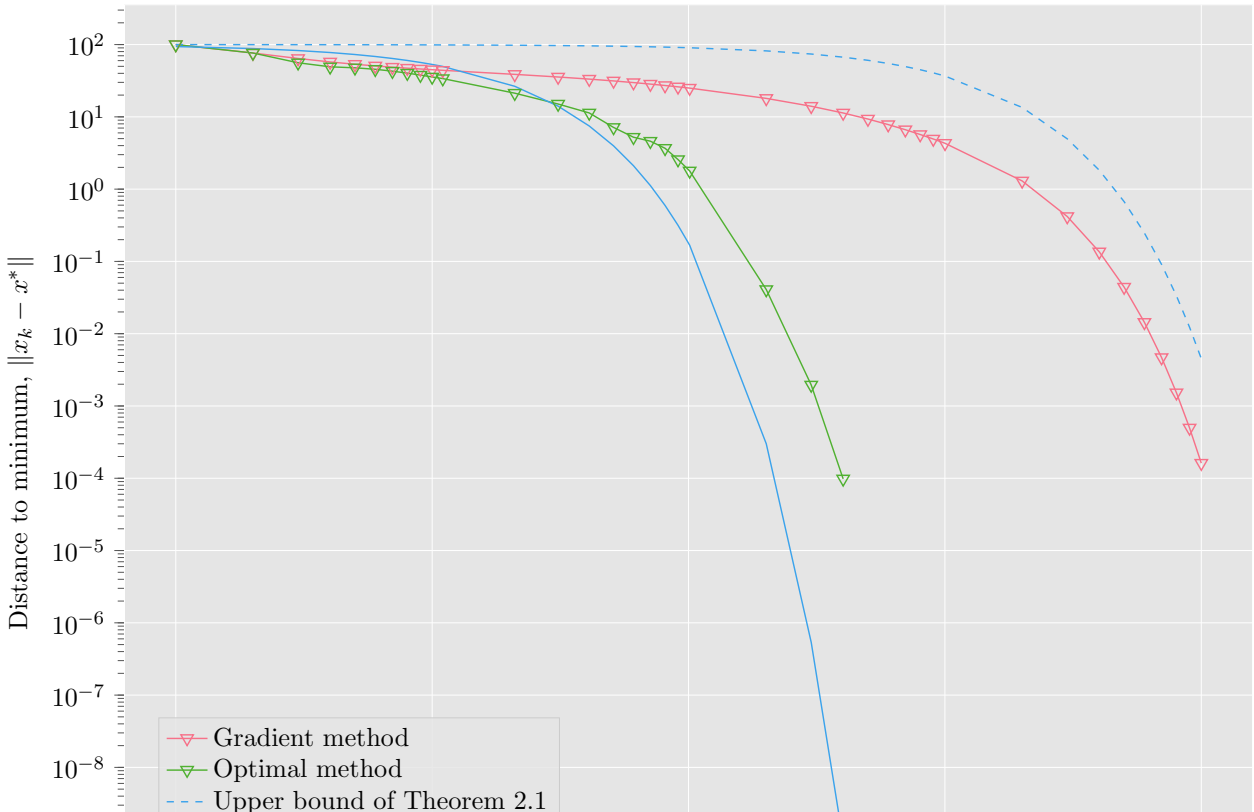


Figure 8. Evolution of the number of iterations required for convergence, for the box, depending on the number of active constraints of the problem.



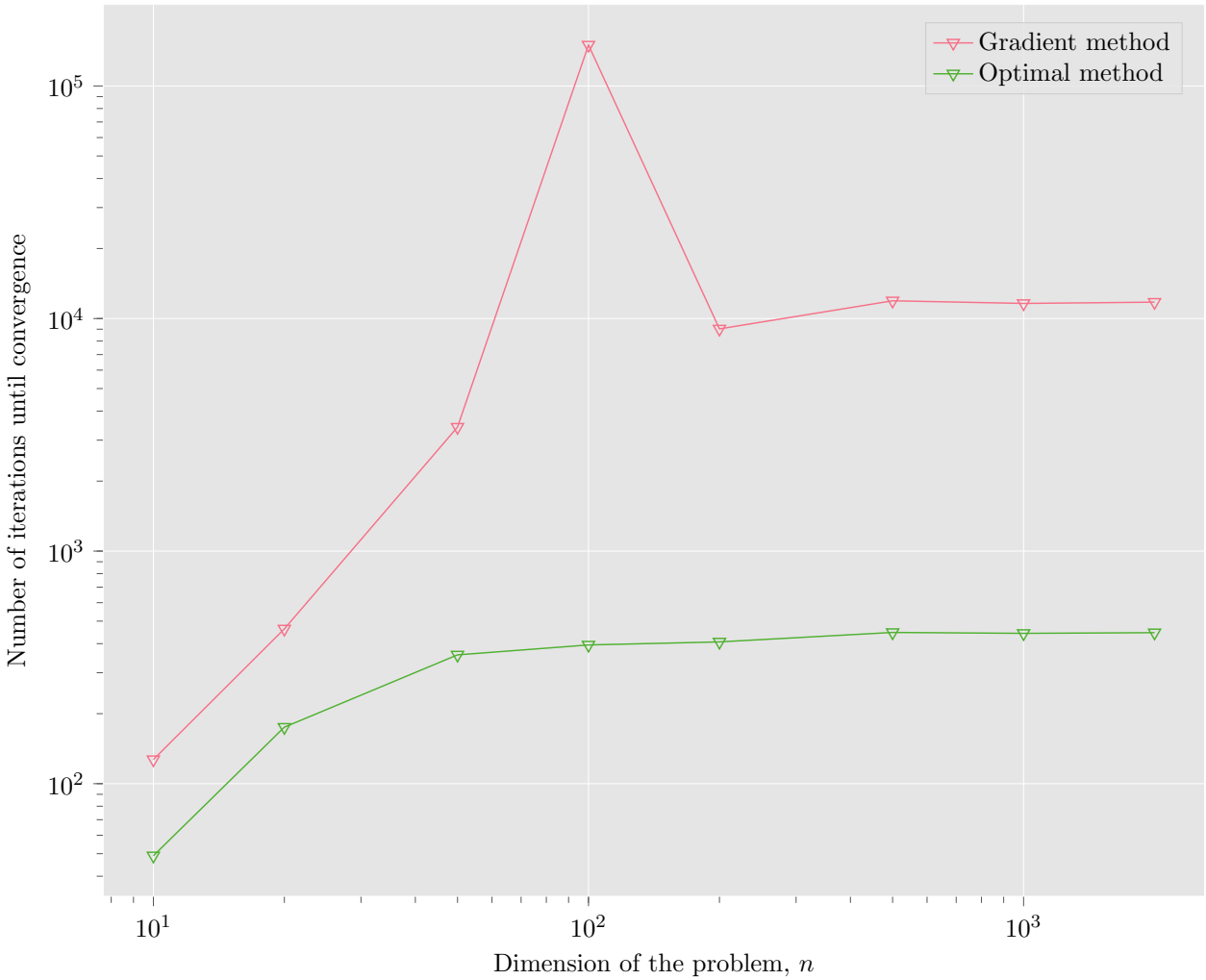
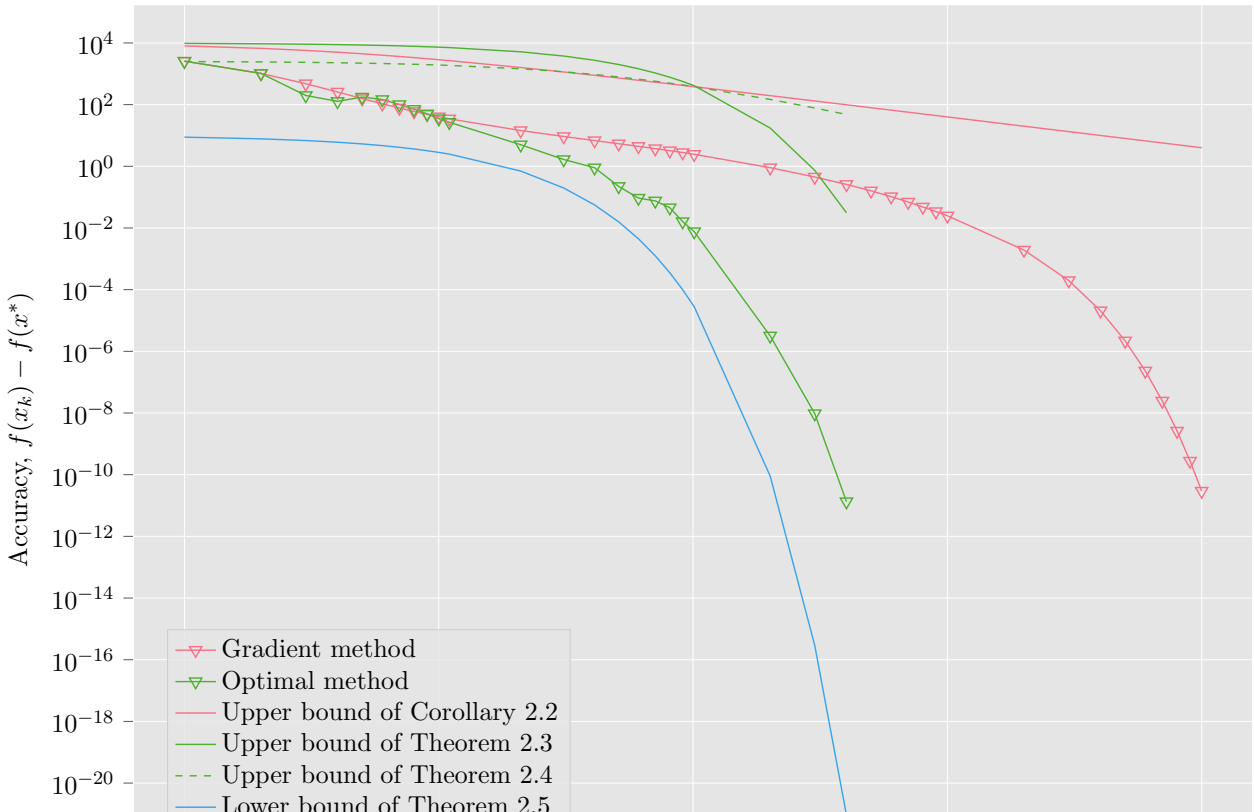


Figure 9. Evolution of the number of iterations required for convergence, for the box, depending on the dimension of the problem.



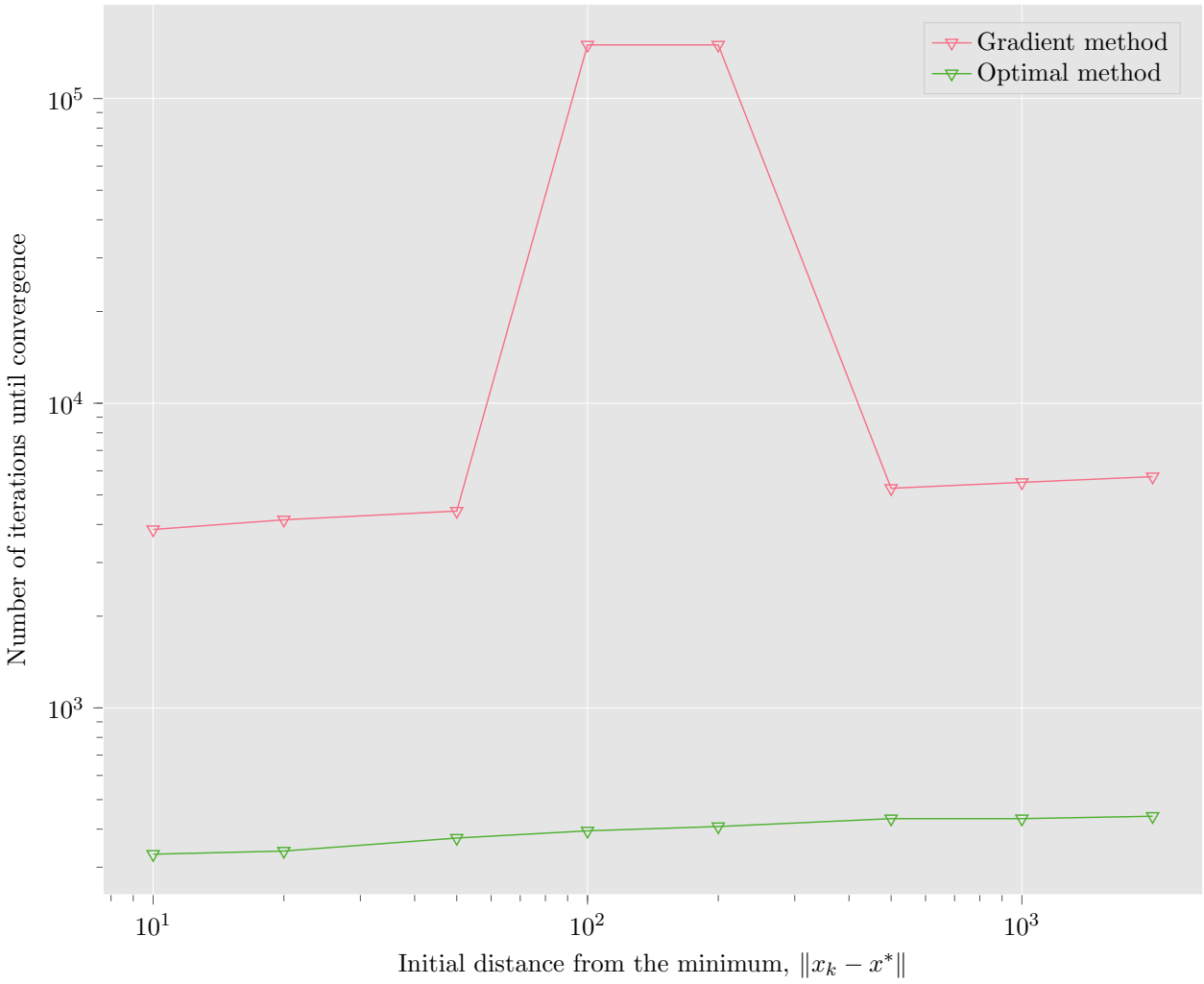
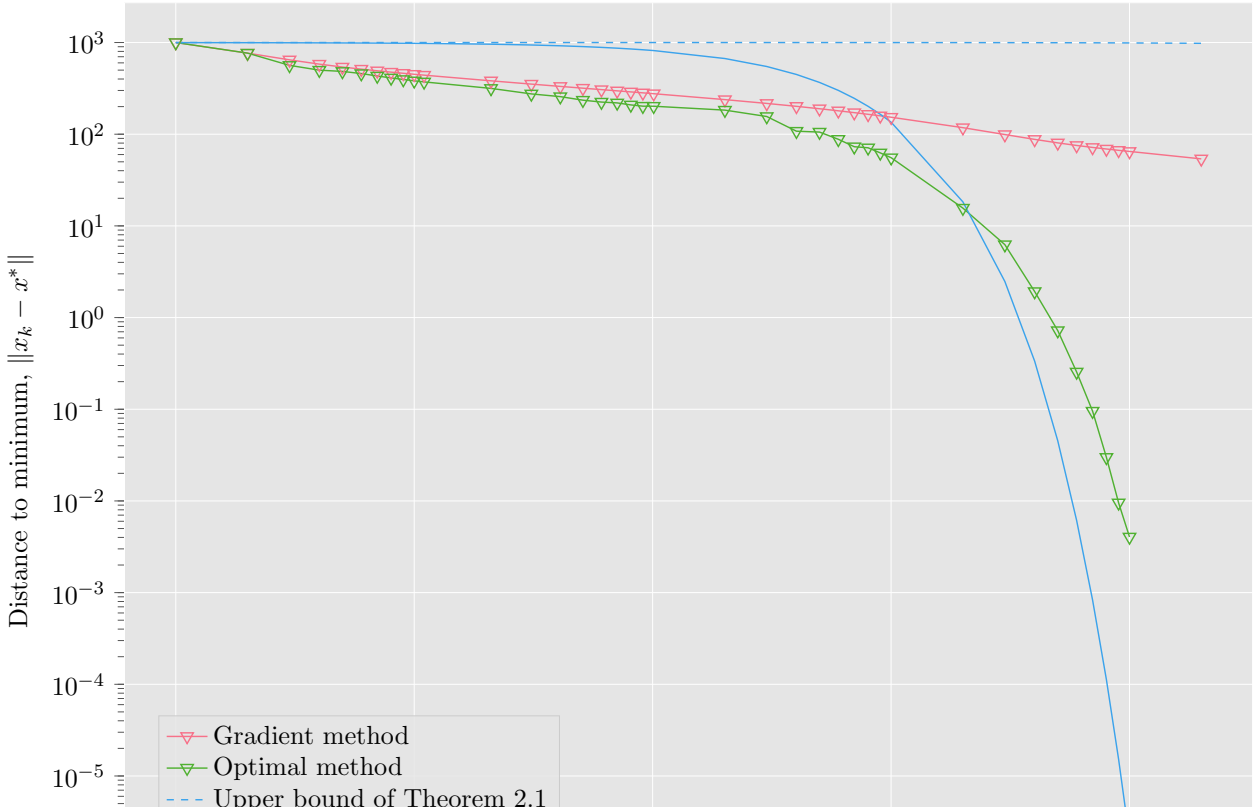


Figure 10. Evolution of the number of iterations required for convergence, for the box, depending on the initial distance to the minimum.



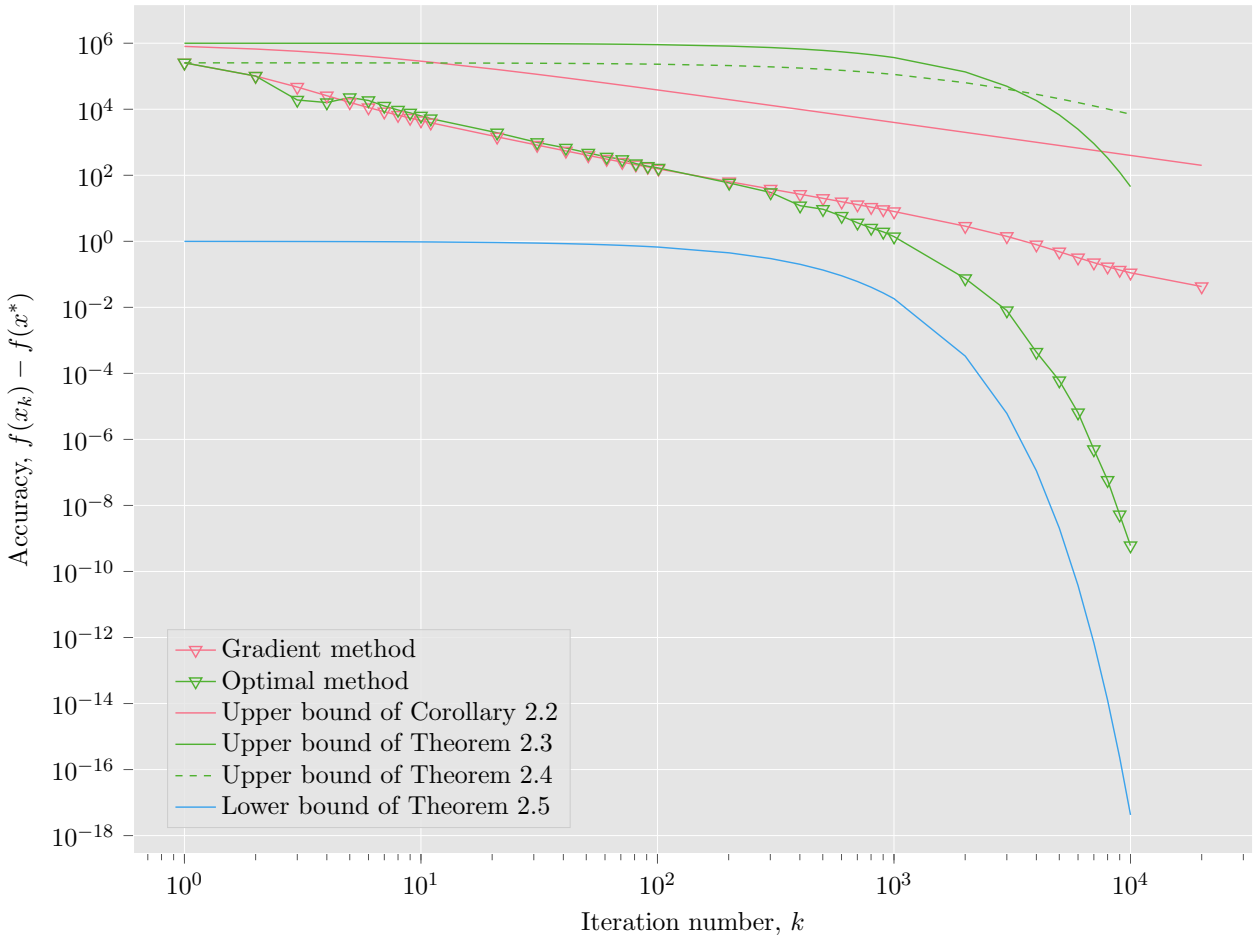


Figure 16. Evolution of the accuracy, for the box on the large example.

A.1.1. *Box.*

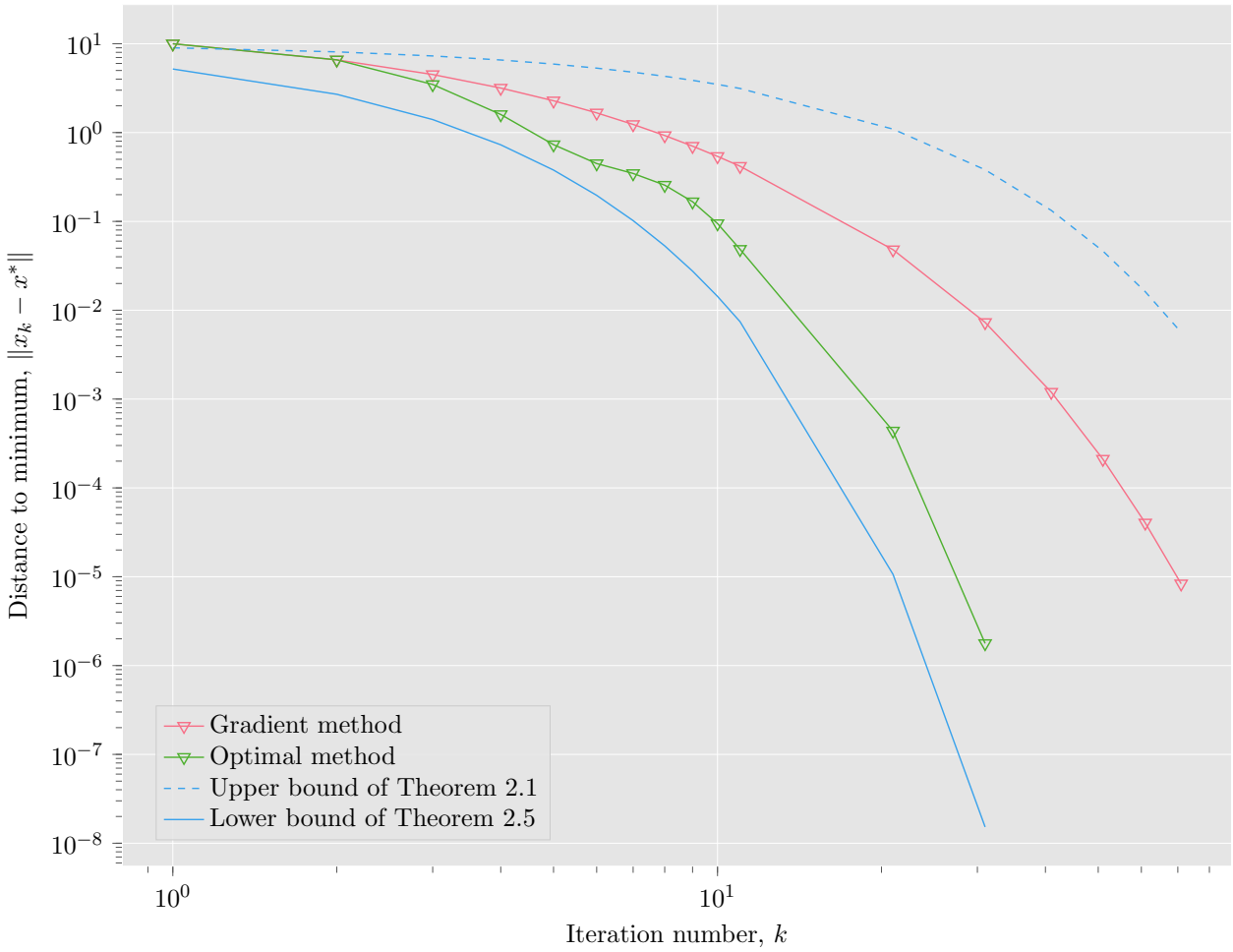


Figure 17. Evolution of the distance to the minimum, for the simplex on the small example.

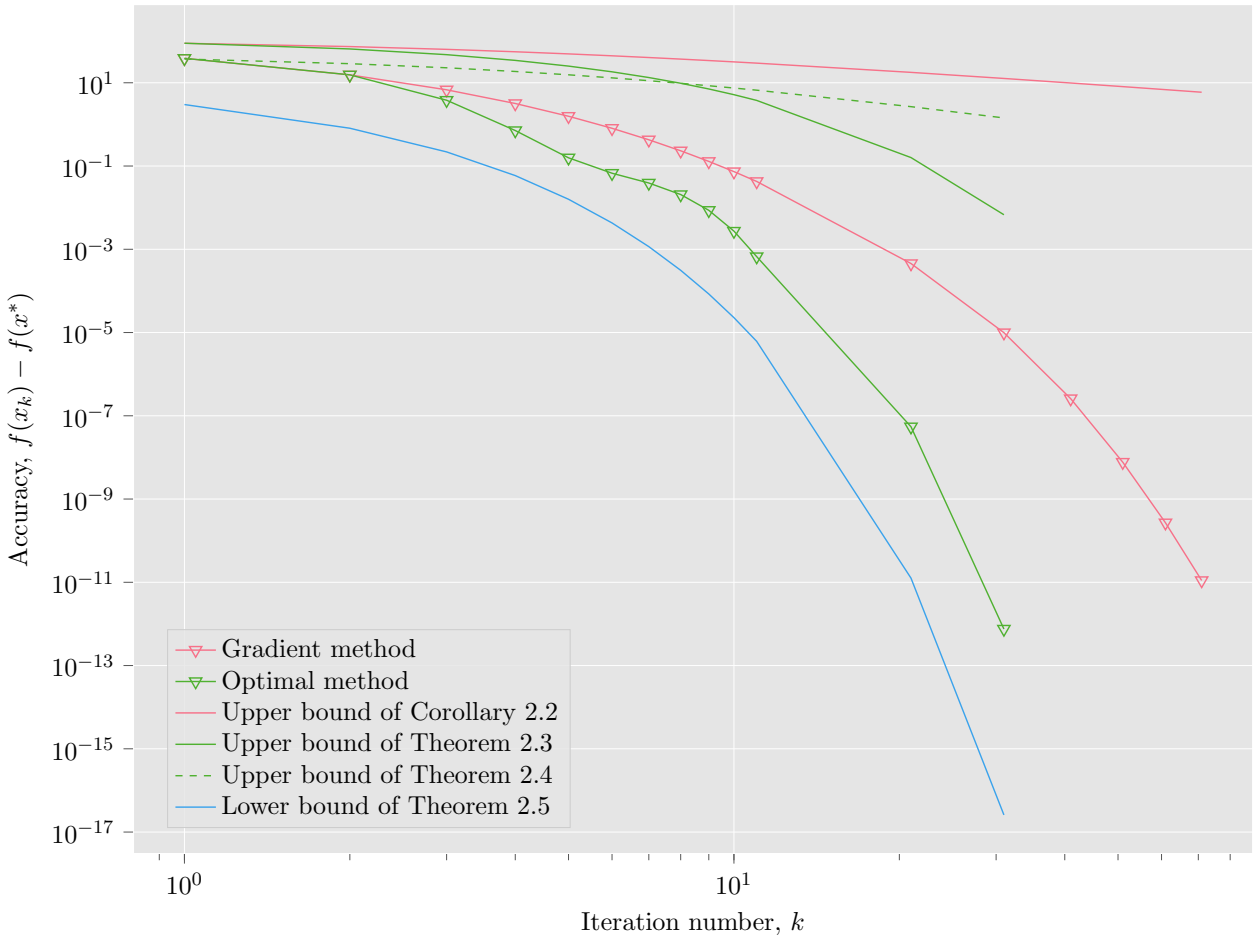


Figure 18. Evolution of the accuracy, for the simplex on the small example.

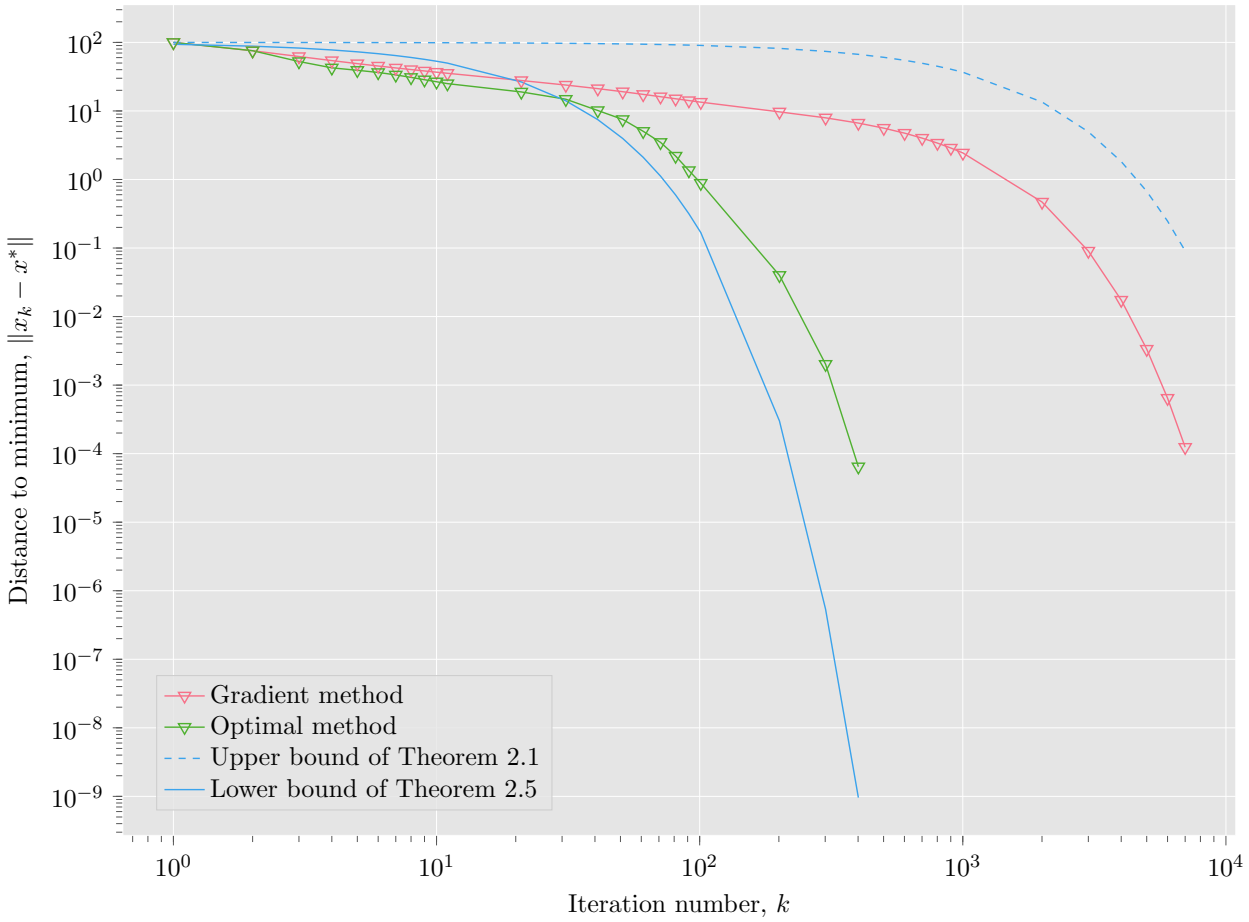


Figure 19. Evolution of the distance to the minimum, for the simplex on the moderate example.

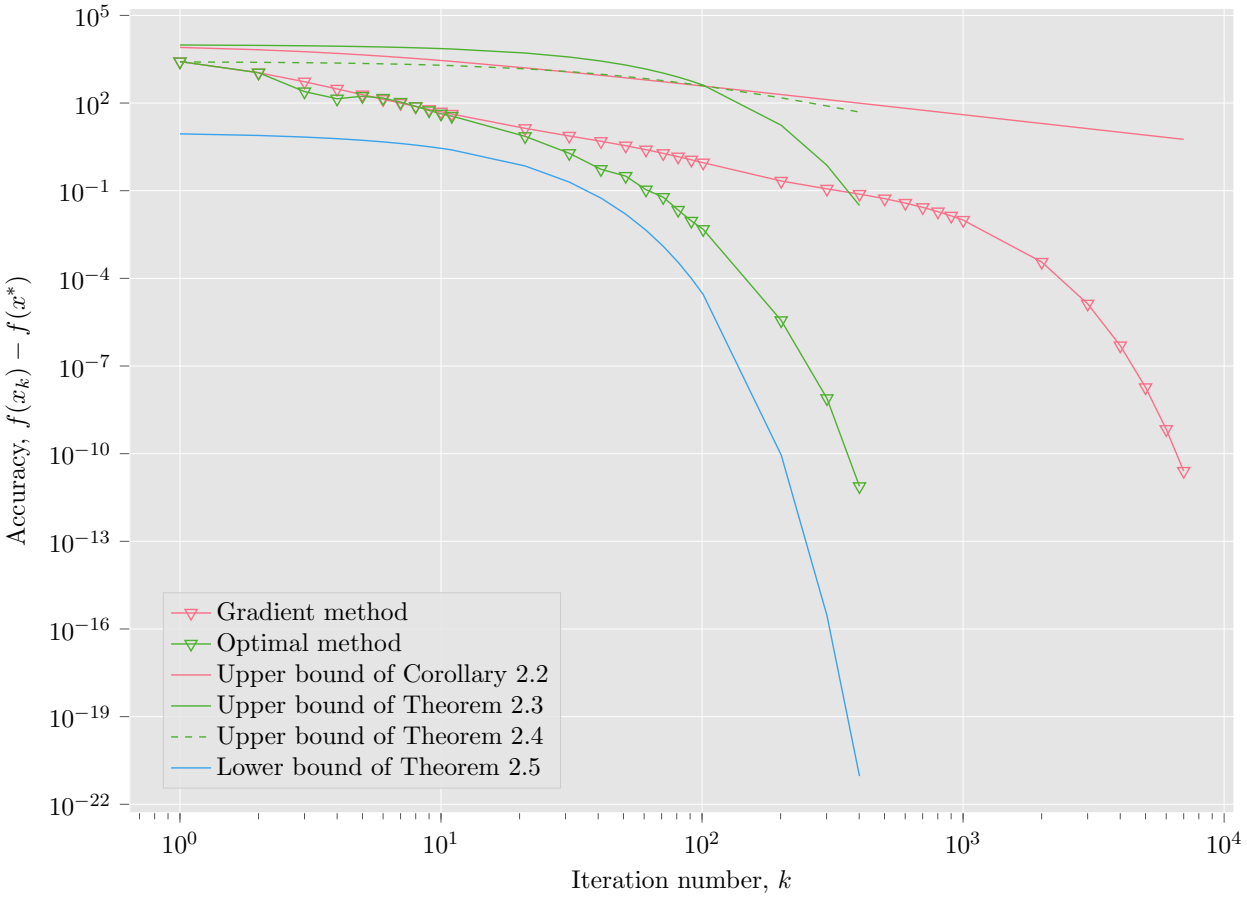


Figure 20. Evolution of the accuracy, for the simplex on the moderate example.

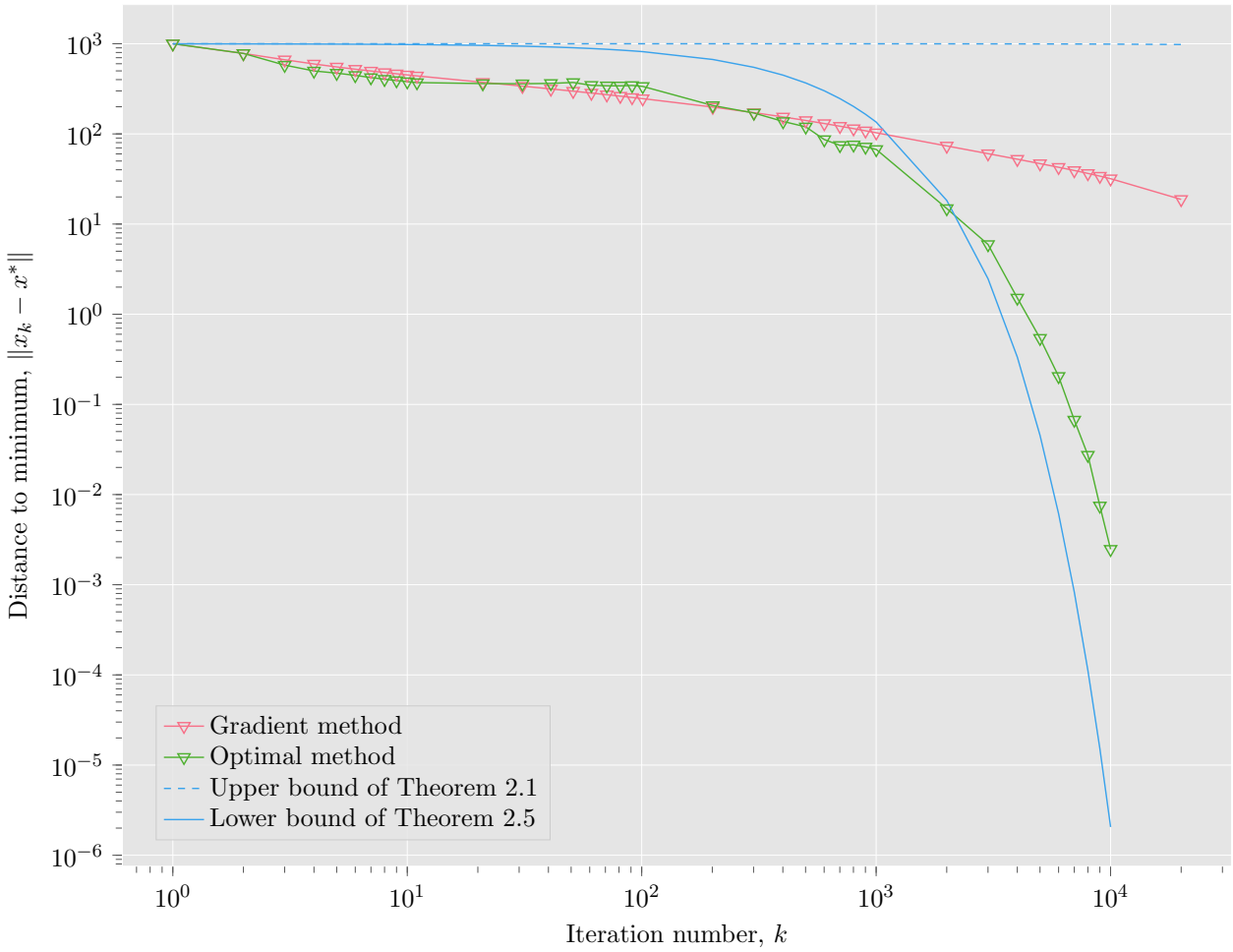


Figure 21. Evolution of the distance to the minimum, for the simplex on the large example.

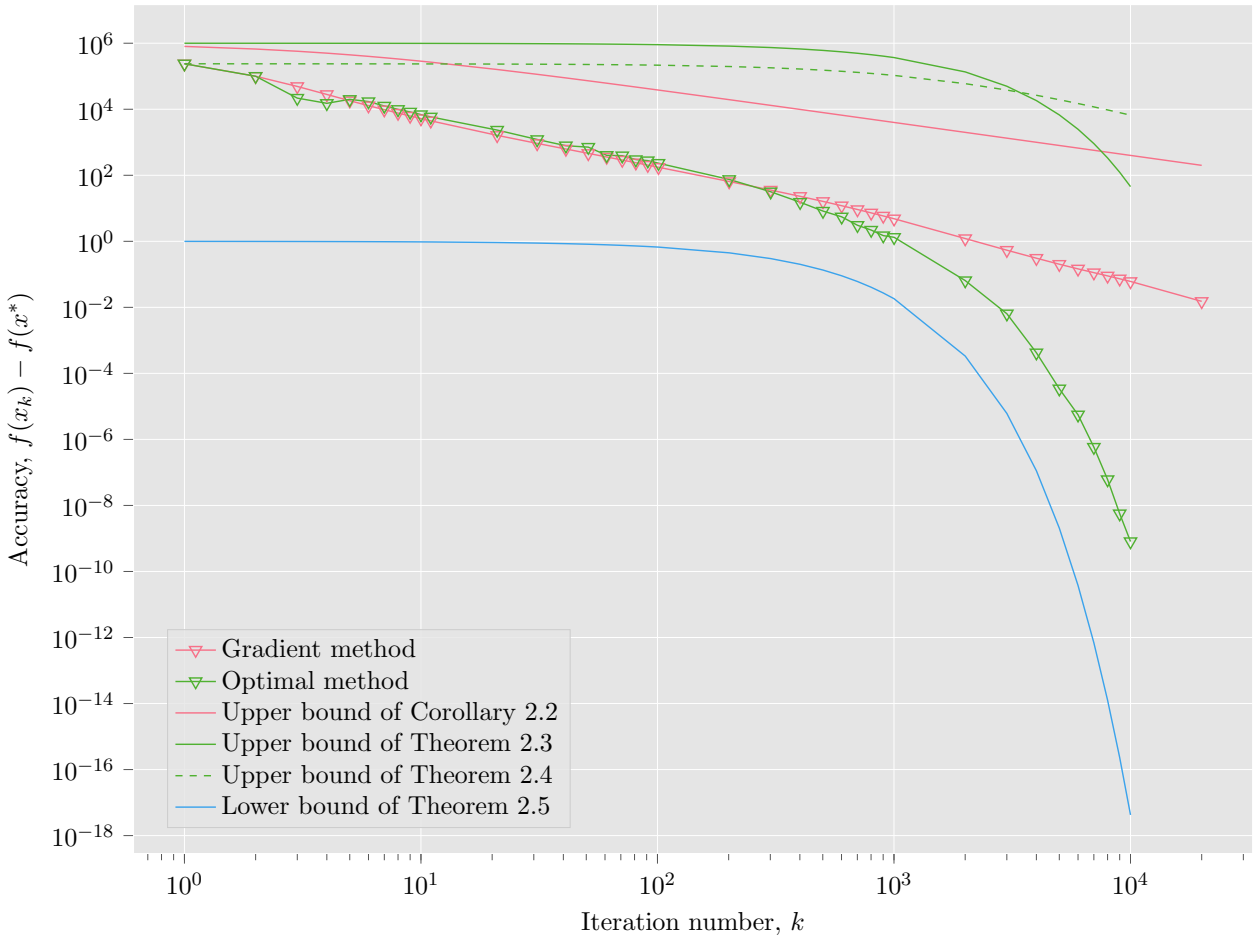


Figure 22. Evolution of the accuracy, for the simplex on the large example.

A.1.2. *Simplex.*

A.2. *Influence of parameters.*

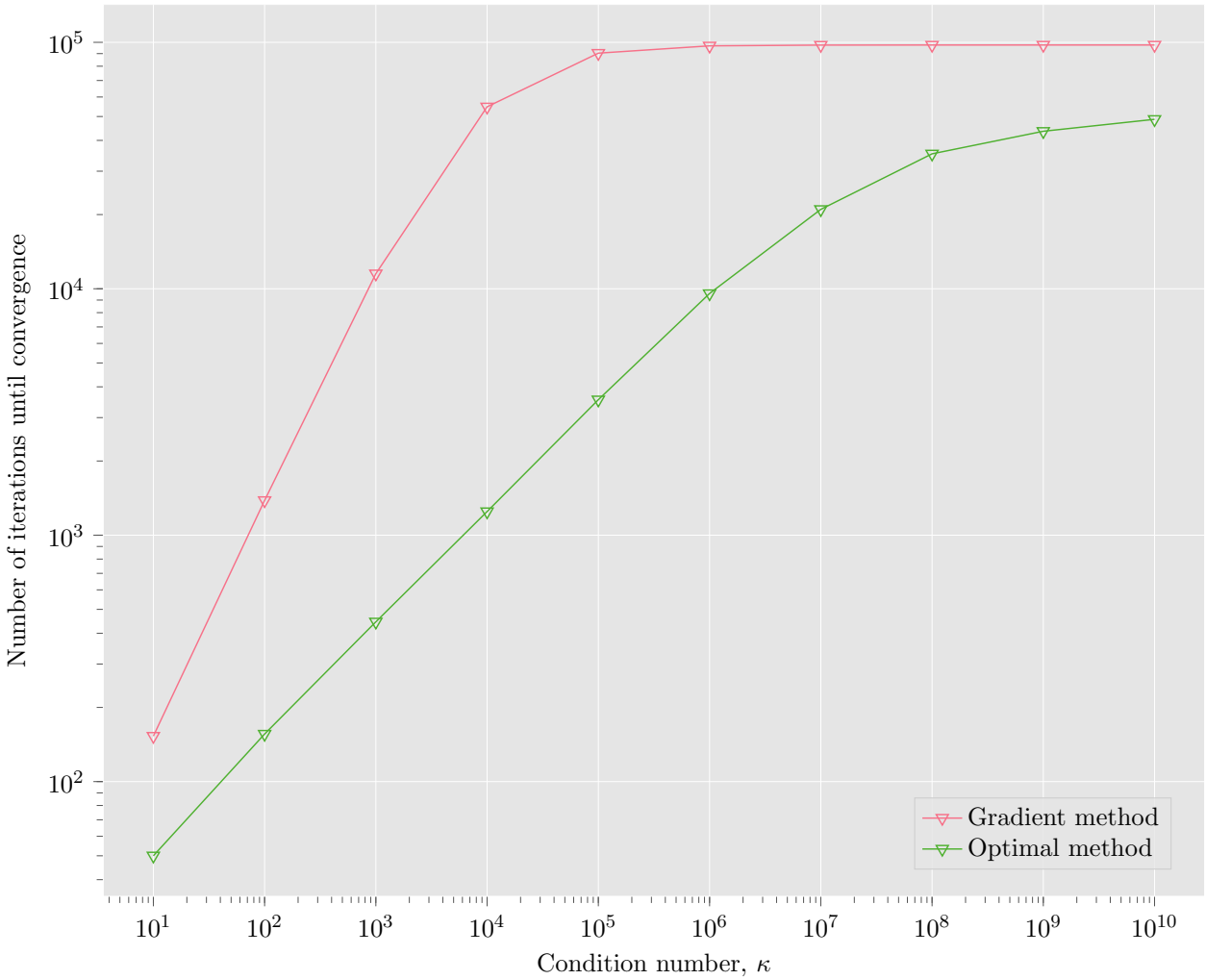


Figure 23. Evolution of the number of iterations required for convergence, for the ball, depending on the condition number of the problem.

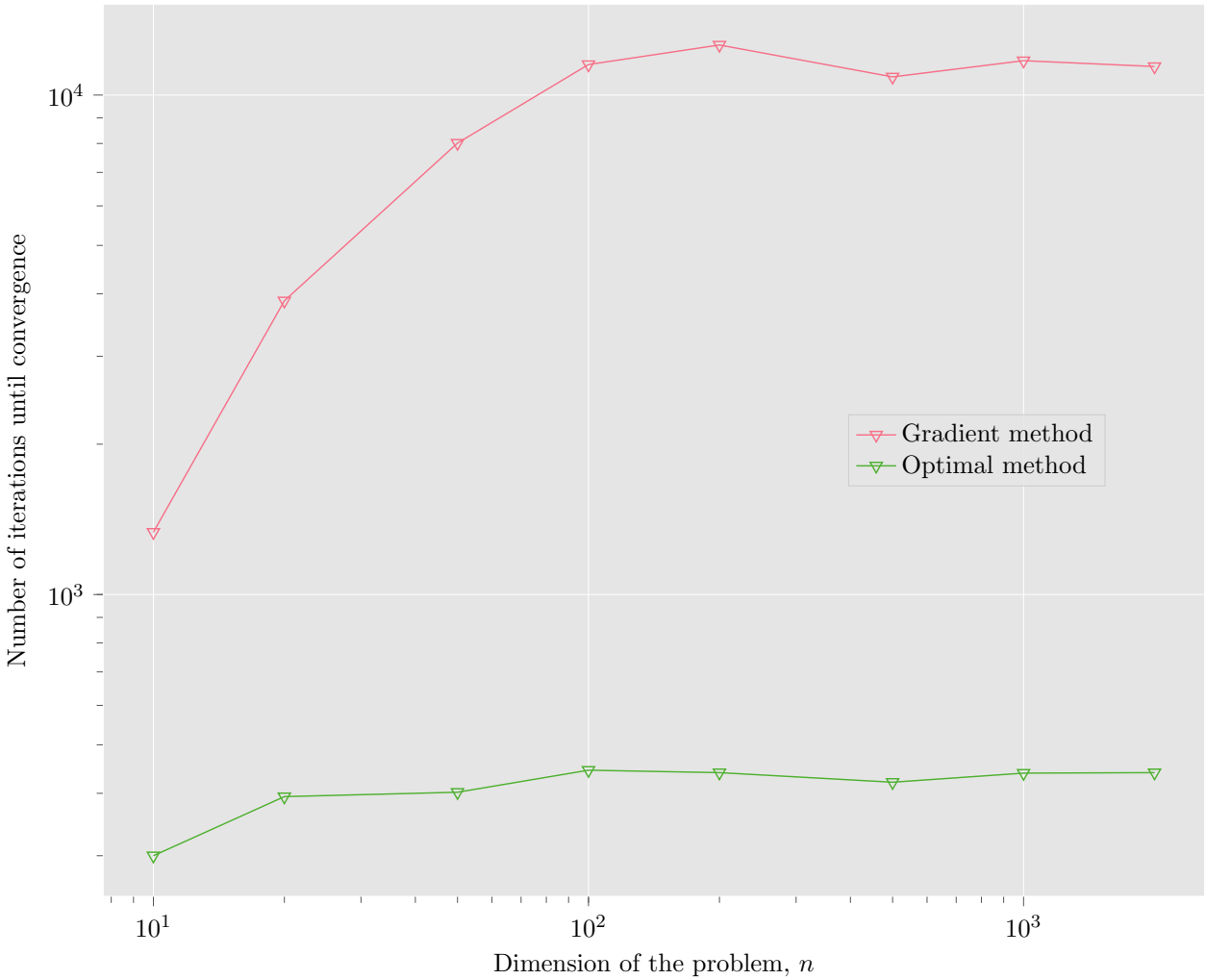


Figure 24. Evolution of the number of iterations required for convergence, for the ball, depending on the dimension of the problem.

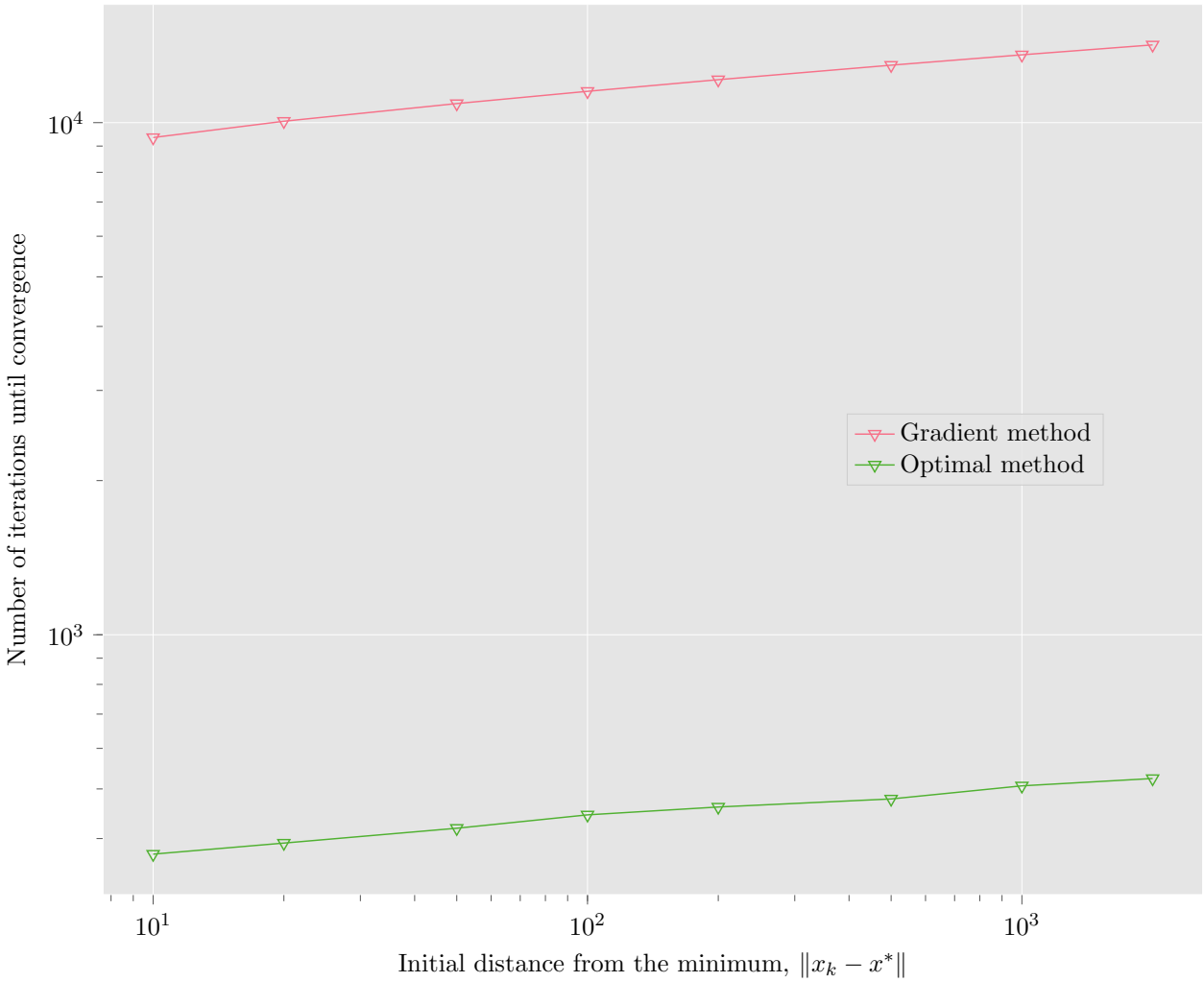


Figure 25. Evolution of the number of iterations required for convergence, for the ball, depending on the initial distance to the minimum of the problem.

A.2.1. *Ball.*

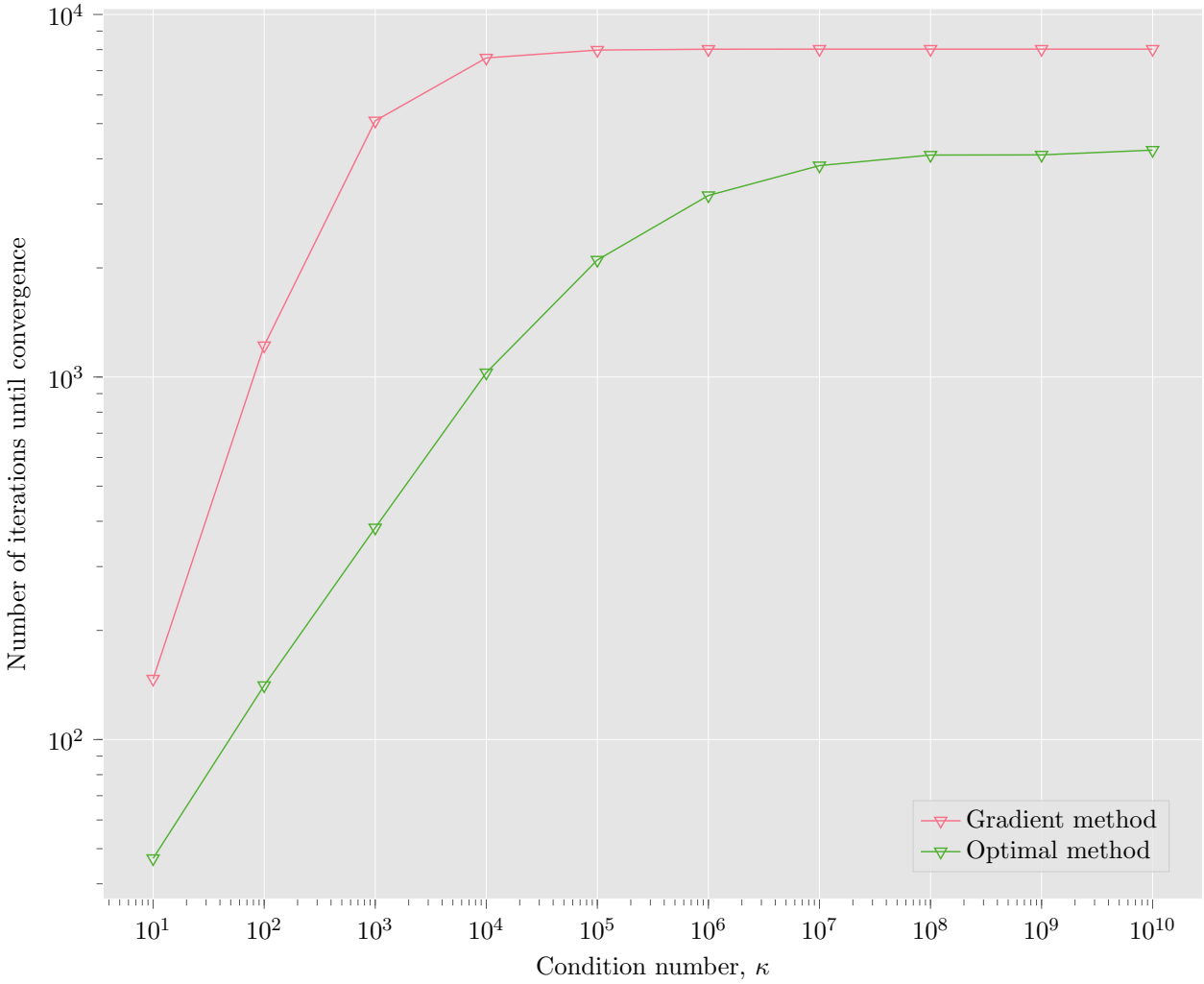


Figure 26. Evolution of the number of iterations required for convergence, for the simplex, depending on the condition number of the problem.

A.2.2. *Simplex.* Other plots for the simplex took several hours to generate, and were hence omitted (number of active constraints, dimension of the problem, initial distance from the minimum).

Appendix B. Source code

The source code is divided into two parts:

- `solvers.py`, which contains the `solve` method.
- `plots.py`, which was used to generate the plots for this paper.

Both are available in full below.

B.1. `solvers.py`.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
solvers.py

Author: Gilles Peiffer
```

```
Date: 2020-05-29

This file contains the numerical methods
needed for the first exercise of LINMA2460.
"""

import numpy as np


def solve(n, m, set_type, parameters, function, eps, residue, method, maximum_iterations):
    """
    Solve an optimization problem using either the gradient or the optimal method.

    Either numerical method can be used to solve the problem,
    depending on which one is specified.

    Parameters
    ----------
    n : int
        Dimension of the problem domain.
    m : int
        Number of active constraints.
        This parameter is only used if 'set' is 'box' or 'simplex'.
    set_type : {'ball', 'box', 'simplex'}
        Type of set being used.
    parameters : dict
        Parameters of the set being used:
         * if set_type is 'ball', this contains R, the radius of the ball;
         * if set_type is 'box', this contains a and b (a[i] <= b[i]), the bounds of the box;
         * if set_type is 'simplex', this contains the parameter p in the simplex definition.
    function : dict
        Contains the parameters of the objective function.
    eps : float
        Desired accuracy of the solution.
    residue : float
        Distance between initial solution and minimum.
    method : {'gradient', 'optimal'}
        Type of numerical method being used.
    maximum_iterations : int
        Maximal number of iterations.

    Returns
    -------
    x : ndarray
        Iterates of the method.
    distances : ndarrray
        Distances between intermediate solutions and optimal solution.
    values : ndarray
```

```
    Values of the objective function at every iteration.
"""

rng = np.random.default_rng(seed=69)

x_opt = np.zeros((n,))
x_init = np.zeros((n,))

# Get parameters based on set type.
if set_type == 'ball':
    # Radius of the ball.
    radius = parameters['R']

    # Randomly generate optimal solution inside ball.
    r_opt = radius * rng.random()
    x_opt = rng.random((n,))
    x_opt *= r_opt / np.linalg.norm(x_opt)

    # Generate initial point at a distance residue from x_opt inside the ball.
    # A random point at the correct distance is generated,
    # until one falls inside the ball.
    shift = -1 + 2*rng.random((n,))
    x_init = x_opt + residue * shift / np.linalg.norm(shift)
    while np.linalg.norm(x_init) > radius:
        shift = -1 + 2 * rng.random((n,))
        x_init = x_opt + residue * shift / np.linalg.norm(shift)
elif set_type == 'box':
    # Parameters of the box.
    a = parameters['a']
    b = parameters['b']

    # Randomly generate optimal solution inside box.
    x_opt = (b - a) * rng.random((n,)) + a
    if m > 0:
        # Generate a random set of indices
        # at which to constrain the problem.
        # Half of them are set to a[i], the other half to b[i].
        permutation = rng.permutation(n)[:m]
        x_opt[permutation[:m//2]] = a[permutation[:m//2]]
        x_opt[permutation[m//2:]] = b[permutation[m//2:]]

    # Generate initial point at a distance residue from x_opt inside the box.
    # A random point at the correct distance is generated,
    # until one falls inside the box.
    shift = -1 + 2*rng.random((n,))
    # Help by always moving away from borders on constrained parts.
    shift[x_opt == a] = np.abs(shift[x_opt == a])
    shift[x_opt == b] = np.abs(shift[x_opt == b])
```

```
        x_init = x_opt + residue * shift /  np.linalg.norm(shift)
        while not np.all(np.less_equal(a, x_init)) or not np.all(np.less_equal(x_init, b)):
            shift = -1 + 2 * rng.random((n,))
            # Help by always moving away from borders on constrained parts.
            shift[x_opt == a] = np.abs(shift[x_opt == a])
            shift[x_opt == b] = -np.abs(shift[x_opt == b])
            x_init = x_opt + residue * shift /  np.linalg.norm(shift)
    elif set_type == 'simplex':
        # Parameter of the simplex.
        p = parameters['p']

        # Randomly generate optimal solution inside simplex.
        permutation = rng.permutation(n)[:n-m]
        x_opt = np.zeros((n,))
        x_opt[permutation] = rng.random((n-m),)
        x_opt[permutation] = p * x_opt[permutation] / np.sum(x_opt[permutation])

        # Generate initial point at a distance residue from x_opt inside the simplex.
        # A random point with zero sum is generated and added to x_opt.
        shift = rng.random((n,))
        shift -= np.mean(shift)
        shift *= residue / np.linalg.norm(shift)
        indices_opt = np.argsort(x_opt)
        shift[::-1].sort()
        x_init[indices_opt] = x_opt[indices_opt] + shift
        while not np.all(x_init >= 0):
            shift = rng.random((n,))
            shift -= np.mean(shift)
            shift *= residue / np.linalg.norm(shift)
            shift[::-1].sort()
            x_init[indices_opt] = x_opt[indices_opt] + shift

    # Number of iterations.
    it = 0

    # Iterates, starting from initial position
    x = np.zeros((maximum_iterations+1, n))
    x[0] = x_init

    # Parameters of the objective function.
    alpha = function['alpha']
    beta = function['beta']
    gamma = function['gamma']

    # Lipschitz-continuity coefficients.
    L = alpha + 4*beta + gamma
    mu = alpha
```

```
# Set initial parameters for the optimal method.
y = None
beta_opt = None
if method == 'optimal':
    y = np.zeros((maximum_iterations + 1, n))
    y[0] = x[0]
    beta_opt = (np.sqrt(L) - np.sqrt(mu)) / (np.sqrt(L) + np.sqrt(mu))

def f(x):
    """
    Compute the value of the function at a point x.

    The function is built as a closure using the values of the parameters
    and of the optimal solution.
    It is kept in a numerically stable version, as mentioned in the pdf.

    Parameters
    ----------
    x : ndarray
        Point at which to evaluate f.

    Returns
    -------
    fx: float
        Value of f(x).
    """
    xdif = x - x_opt
    fx = alpha/2 * np.linalg.norm(xdif)**2  # First term.
    fx += beta/2 * (xdif[0]**2 +
                    np.sum((xdif[:-1] - xdif[1:])**2)
                    + xdif[-1]**2)  # Second term.

    def f2(x):
        """
        Compute f2 in definition of objective function.

        The computation is done so as to preserve numerical stability.

        Parameters
        ----------
        x : ndarray
            Point at which to compute the function value.

        Returns
        -------
        fx : float
            Value of f2(x).
        """
```

```
            delta = np.max(x)
            return delta + np.log(np.sum(np.exp(x - delta)))

        # Compute f2'(x_opt) more efficiently.
        tmp = np.exp(x_opt - np.max(x_opt))

        fx += gamma * (f2(x) -
                       f2(x_opt) -
                       np.dot(tmp / np.sum(tmp), xdif))  # Third term.

        return fx

    def fp(x):
        """
        Compute value of f' at point x.

        Parameters
        ----------
        x : ndarray
            Point at which we evaluate f'.

        Returns
        -------
        fpx : float
            Value of f'(x).
        """
        # For more efficient computation.
        xdif = x - x_opt
        tmp_x_opt = np.exp(x_opt - np.max(x_opt))
        tmp_x = np.exp(x - np.max(x))

        aux = np.zeros((n,))
        aux[:-1] += xdif[1:]
        aux[1:] += xdif[:-1]

        # Derivative of f evaluated at x.
        fpx = ((alpha + 2*beta) * xdif -
               beta * aux +
               gamma * (tmp_x / np.sum(tmp_x) - tmp_x_opt / np.sum(tmp_x_opt)))

        return fpx

    # Distance between iterate and optimal solution.
    distances = np.zeros((maximum_iterations+1,))
    distances[0] = residue

    # Values of the objective function.
    values = np.zeros((maximum_iterations+1,))
```

```
values[0] = f(x[0])

it = 1
# Iterate while convergence hasn't been reached.
# Optimal value of the function is 0, hence we can simply compare to eps.
while np.abs(values[it - 1]) > eps and 0 < it <= maximum_iterations:
    # Decide which method to use.

    # Compute next iterate.
    if method == 'gradient':
        x[it] = x[it - 1] - 1/L * fp(x[it - 1])
    elif method == 'optimal':
        x[it] = y[it - 1] - 1/L * fp(y[it - 1])

    # Project iterate if needed.
    if set_type == 'ball':
        radius = parameters['R']

        norm = np.linalg.norm(x[it])
        if norm > radius:
            x[it] *= radius / norm
    elif set_type == 'box':
        a = parameters['a']
        b = parameters['b']

        # Project components outside of the box on the box.
        tmpa = x[it] < a
        tmpb = x[it] > b
        x[it][tmpb] = b[tmpb]
        x[it][tmpa] = a[tmpa]
    elif set_type == 'simplex':
        p = parameters['p']

        # Project onto simplex;
        # https://arxiv.org/abs/1101.6081v2

        # If there is only one point in the domain, it must be p.
        if n == 1:
            x[it] = p
        else:
            x_sorted = np.sort(x[it])
            i = n-1

            t_opt = None
            while True:
                ti = (np.sum(x_sorted[i:]) - p) / (n - i)

                if ti >= x_sorted[i]:
```

```
                        t_opt = ti
                        break

                i = i - 1
                if i == 0:
                    t_opt = np.mean(x_sorted) - p / n
                    break

            np.maximum(x[it] - t_opt, 0, out=x[it])

        # Update y if necessary.
        if method == 'optimal':
            y[it] = x[it] + beta_opt * (x[it] - x[it - 1])

        # Store values.
        distances[it] = np.linalg.norm(x[it] - x_opt)
        values[it] = f(x[it])

        it += 1

    # Prune empty preallocated space.
    return x[:it], distances[:it], values[:it]
```

B.2. plots.py.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tikzplotlib

plt.style.use("ggplot")

from solvers import solve


def reduce(l):
    l = list(l)
    if len(l) <= 10:
        return l
    r = l[:10]

    if len(l) <= 100:
        return r + l[10::10]
    r = r + l[10:100:10]

    if len(l) <= 1000:
        return r + l[100::100]
```

```
    r = r + l[100:1000:100]

    if len(l) <= 10000:
        return r + l[1000::1000]
    r = r + l[1000:10000:1000]

    if len(l) <= 100000:
        return r + l[10000::10000]



# Ball small
#    accuracy
#    distance


# Ball moderate
#    accuracy
#    distance

# Ball large
#     accuracy
#     distance

# ----------------

# Box small
#     accuracy
#     distance

# Box moderate
#     accuracy
#     distance

# Box large
#     accuracy
#     distance

# ----------------

# Simplex small
#     accuracy
#     distance

# Simplex moderate
#     accuracy
#     distance
```

```
# Simplex large
#     accuracy
#     distance
def part1():
    maximum_iterations = 20_000
    set_types = ['ball', 'box', 'simplex']
    sizes = ['small', 'moderate', 'large']
    problems = {'small': {'n': 10,
                          'm': 0,
                          'function': {'alpha': 2/(10 - 1),
                                       'beta': 0.25,
                                       'gamma': 1},
                          'eps': 1e-12,
                          'residue': 10},
                'moderate': {'n': 100,
                             'm': 0,
                             'function': {'alpha': 2/(1_000 - 1),
                                          'beta': 0.25,
                                          'gamma': 1},
                             'eps': 1e-12,
                             'residue': 1e2},
                'large': {'n': 1_000,
                          'm': 0,
                          'function': {'alpha': 2/(1_000_000 - 1),
                                       'beta': 0.25,
                                       'gamma': 1},
                          'eps': 1e-12,
                          'residue': 1e3}}


    clrs = sns.color_palette('husl', 3)

    for set_type in set_types:
        for size in sizes:
            print("Size: %s, set: %s" % (size, set_type))

            params = None
            if set_type == 'ball':
                params = {'R': 100 * problems[size]['residue']}
            elif set_type == 'box':
                params = {'a': np.zeros((problems[size]['n'],)),
                          'b': 100 * problems[size]['residue'] * np.ones((problems[size]['n'],)
            elif set_type == 'simplex':
                params = {'p': 100 * problems[size]['residue']}

            x_grad, dist_grad, vals_grad = solve(problems[size]['n'],
                                                 problems[size]['m'],
                                                 set_type,
```

```
                                        params,
                                        problems[size]['function'],
                                        problems[size]['eps'],
                                        problems[size]['residue'],
                                        'gradient',
                                        maximum_iterations)

x_opt, dist_opt, vals_opt = solve(problems[size]['n'],
                                  problems[size]['m'],
                                  set_type,
                                  params,
                                  problems[size]['function'],
                                  problems[size]['eps'],
                                  problems[size]['residue'],
                                  'optimal',
                                  maximum_iterations)

L = problems[size]['function']['alpha'] + 4 * problems[size]['function']['beta'] +
mu = problems[size]['function']['alpha']
kappa = L/mu

residue = problems[size]['residue']

l_grad = np.array(reduce(range(1, len(x_grad) + 1)))
l_opt = np.array(reduce(range(1, len(x_opt) + 1)))

vals_grad = vals_grad[l_grad-1]
vals_opt = vals_opt[l_opt-1]

dist_grad = dist_grad[l_grad-1]
dist_opt = dist_opt[l_opt-1]

cor212 = 2*L * residue**2 / (l_grad+4)
eq2223 = (L + mu)/2 * residue**2 * np.exp(-l_opt * np.sqrt(mu/L))
thm223 = [min(np.power(1 - mu/L, k), 4*L / (2 * np.sqrt(L) + k * np.sqrt(mu))**2) *
thm2113acc = mu/2 * ((np.sqrt(kappa) - 1)/(np.sqrt(kappa) + 1))**(2*l_opt) * residu

fig = plt.figure()
plt.loglog(l_grad, vals_grad, '-v', markerfacecolor='none', c=clrs[0])
plt.loglog(l_opt, vals_opt, '-v', markerfacecolor='none', c=clrs[1])
plt.loglog(l_grad, cor212, '-', c=clrs[0])
plt.loglog(l_opt, eq2223, '-', c=clrs[1])
plt.loglog(l_opt, thm223, '--', c=clrs[1])
plt.loglog(l_opt, thm2113acc, '-', c=clrs[2])
plt.xlabel("Iteration number, \\(k\\)")
plt.ylabel("Accuracy, \\(f(\\xk) - f(\\xopt)\\)")
plt.legend(["Gradient method",
            "Optimal method",
```

```
                        "Upper bound of Corollary~\\ref{cor:2.1.2}",
                        "Upper bound of \\thmref{thm:2.2.23}",
                        "Upper bound of \\thmref{thm:2.2.3}",
                        "Lower bound of \\thmref{thm:2.1.13}"])

            tikzplotlib.save("../report/plots/p1_%s_%s_acc.tikz" % (set_type, size), figure=fig


            thm2214 = (1 - mu/L)**l_grad * residue
            thm2113dist = ((np.sqrt(kappa) - 1)/(np.sqrt(kappa) + 1))**l_opt * residue

            fig = plt.figure()
            plt.loglog(l_grad, dist_grad, '-v', markerfacecolor='none', c=clrs[0])
            plt.loglog(l_opt, dist_opt, '-v', markerfacecolor='none', c=clrs[1])
            plt.loglog(l_grad, thm2214, '--', c=clrs[2])
            plt.loglog(l_opt, thm2113dist, '-', c=clrs[2])
            plt.xlabel("Iteration number, \\(k\\)")
            plt.ylabel("Distance to minimum, \\(\\norm{\\xk - \\xopt}\\)")
            plt.legend(["Gradient method",
                        "Optimal method",
                        "Upper bound of Theorem~\\ref{thm:2.2.14}",
                        "Lower bound of \\thmref{thm:2.1.13}"])

            tikzplotlib.save("../report/plots/p1_%s_%s_dist.tikz" % (set_type, size), figure=fi


# Ball
#     kappa
#     R
#     n

# Box
#     kappa
#     R
#     n
#     m

# Simplex
#     kappa
#     R
#     n
#     m
def part2():
    maximum_iterations = 20_000
    set_types = ['ball', 'box', 'simplex']

    default_kappa = 1000
    default_residue = 100
```

```
default_n = 100
default_m = 10
epsilon = 1e-12

clrs = sns.color_palette('husl', 3)


variab = {'kappa': [10**i for i in range(1, 11)],
          'residue': [10, 20, 50, 100, 200, 500, 1000, 2000],
          'n': [10, 20, 50, 100, 200, 500, 1000, 2000],
          'm': [5*i for i in range(21)]}


for set_type in set_types:
    print(set_type)

    params = None

    if set_type == 'ball':
        params = {'R': 100 * default_residue}
    elif set_type == 'box':
        params = {'a': np.zeros((default_n,)),
                  'b': 100 * default_residue * np.ones((default_n,))}
    elif set_type == 'simplex':
        params = {'p': 100 * default_residue}

    n_iter_grad = []
    n_iter_opt = []
    for kappa in variab['kappa']:

        x_grad, dist_grad, vals_grad = solve(default_n,
                                             default_m,
                                             set_type,
                                             params,
                                             {'alpha': 2/(kappa - 1),
                                              'beta': 0.25,
                                              'gamma': 1},
                                             epsilon,
                                             default_residue,
                                             'gradient',
                                             maximum_iterations)


        x_opt, dist_opt, vals_opt = solve(default_n,
                                          default_m,
                                          set_type,
                                          params,
                                          {'alpha': 2/(kappa - 1),
```

```
                                                'beta': 0.25,
                                                'gamma': 1},
                                               epsilon,
                                               default_residue,
                                               'optimal',
                                               maximum_iterations)

        n_iter_grad.append(len(x_grad))
        n_iter_opt.append(len(x_opt))


    plt.figure()
    plt.loglog(variab['kappa'], n_iter_grad, '-v', markerfacecolor='none', c=clrs[0])
    plt.loglog(variab['kappa'], n_iter_opt, '-v', markerfacecolor='none', c=clrs[1])
    plt.xlabel("Condition number, \\(\\kappa\\)")
    plt.ylabel("Number of iterations until convergence")
    plt.legend(["Gradient method",
                "Optimal method"])

    tikzplotlib.save("../report/plots/p2_%s_kappa.tikz" % (set_type), axis_width="\\linewid

    n_iter_grad = []
    n_iter_opt = []

    if set_type == 'box' or set_type == 'simplex':
        for m in variab['m']:
            x_grad, dist_grad, vals_grad = solve(default_n,
                                                 m,
                                                 set_type,
                                                 params,
                                                 {'alpha': 2/(default_kappa - 1),
                                                  'beta': 0.25,
                                                  'gamma': 1},
                                                 epsilon,
                                                 default_residue,
                                                 'gradient',
                                                 maximum_iterations)


            x_opt, dist_opt, vals_opt = solve(default_n,
                                              m,
                                              set_type,
                                              params,
                                              {'alpha': 2/(default_kappa - 1),
                                               'beta': 0.25,
                                               'gamma': 1},
                                              epsilon,
                                              default_residue,
```

```
                                        'optimal',
                                        maximum_iterations)

            n_iter_grad.append(len(x_grad))
            n_iter_opt.append(len(x_opt))


        plt.figure()
        plt.loglog(variab['m'], n_iter_grad, '-v', markerfacecolor='none', c=clrs[0])
        plt.loglog(variab['m'], n_iter_opt, '-v', markerfacecolor='none', c=clrs[1])
        plt.xlabel("Number of active constraints, \\(m\\)")
        plt.ylabel("Number of iterations until convergence")
        plt.legend(["Gradient method",
                    "Optimal method"])

        tikzplotlib.save("../report/plots/p2_%s_m.tikz" % (set_type), axis_width="\\linewi

        n_iter_grad = []
        n_iter_opt = []

    for n in variab['n']:

        if set_type == 'ball':
            params = {'R': 100 * default_residue}
        elif set_type == 'box':
            params = {'a': np.zeros((n,)),
                      'b': 100 * default_residue * np.ones((n,))}
        elif set_type == 'simplex':
            params = {'p': 100 * default_residue}

        x_grad, dist_grad, vals_grad = solve(n,
                                             default_m,
                                             set_type,
                                             params,
                                             {'alpha': 2/(default_kappa - 1),
                                              'beta': 0.25,
                                              'gamma': 1},
                                             epsilon,
                                             default_residue,
                                             'gradient',
                                             maximum_iterations)


        x_opt, dist_opt, vals_opt = solve(n,
                                          default_m,
                                          set_type,
                                          params,
                                          {'alpha': 2/(default_kappa - 1),
```

```
                                          'beta': 0.25,
                                          'gamma': 1},
                                         epsilon,
                                         default_residue,
                                         'optimal',
                                         maximum_iterations)

        n_iter_grad.append(len(x_grad))
        n_iter_opt.append(len(x_opt))

    plt.figure()
    plt.loglog(variab['n'], n_iter_grad, '-v', markerfacecolor='none', c=clrs[0])
    plt.loglog(variab['n'], n_iter_opt, '-v', markerfacecolor='none', c=clrs[1])
    plt.xlabel("Dimension of the problem, \\(n\\)")
    plt.ylabel("Number of iterations until convergence")
    plt.legend(["Gradient method",
                "Optimal method"])

    tikzplotlib.save("../report/plots/p2_%s_n.tikz" % (set_type), axis_width="\\linewidth")

    n_iter_grad = []
    n_iter_opt = []

    for residue in variab['residue']:

        if set_type == 'ball':
            params = {'R': 100 * residue}
        elif set_type == 'box':
            params = {'a': np.zeros((default_n,)),
                      'b': 100 * residue * np.ones((default_n,))}
        elif set_type == 'simplex':
            params = {'p': 100 * residue}

        x_grad, dist_grad, vals_grad = solve(default_n,
                                             default_m,
                                             set_type,
                                             params,
                                             {'alpha': 2/(default_kappa - 1),
                                              'beta': 0.25,
                                              'gamma': 1},
                                             epsilon,
                                             residue,
                                             'gradient',
                                             maximum_iterations)

        x_opt, dist_opt, vals_opt = solve(default_n,
                                          default_m,
                                          set_type,
```

```
                                                params,
                                                {'alpha': 2/(default_kappa - 1),
                                                 'beta': 0.25,
                                                 'gamma': 1},
                                                epsilon,
                                                residue,
                                                'optimal',
                                                maximum_iterations)

            n_iter_grad.append(len(x_grad))
            n_iter_opt.append(len(x_opt))

        plt.figure()
        plt.loglog(variab['residue'], n_iter_grad, '-v', markerfacecolor='none', c=clrs[0])
        plt.loglog(variab['residue'], n_iter_opt, '-v', markerfacecolor='none', c=clrs[1])
        plt.xlabel("Initial distance from the minimum, \\(\\norm{\\xk - \\xopt}\\)")
        plt.ylabel("Number of iterations until convergence")
        plt.legend(["Gradient method",
                    "Optimal method"])

        tikzplotlib.save("../report/plots/p2_%s_residue.tikz" % (set_type), axis_width="\\linew


if __name__ == '__main__':
    #part1()
    part2()
```

Université catholique de Louvain, Ottignies-Louvain-la-Neuve, Belgium
E-mail: gilles.peiffer@student.uclouvain.be