LINMA2460 – Project 2
Methods for Nonsmooth Convex Minimization

By Gilles Peiffer

Abstract

In this short paper, we present a study of two methods for nonsmooth convex minimization: the subgradient method and the ellipsoid 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 characteristics of the problem. For reproducibility purposes, the full computation results and source code are also made available.

Contents

1. **Introduction**

Nonsmooth convex minimization is an extensively studied topic in mathematical optimization. In this paper, we look at two of the most well-known methods in this domain: the subgradient method, and the ellipsoid 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 [2].

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

$$\min_{x \in \mathbb{R}^n} f(x), \tag{2.1}$$

where f is a nondifferentiable convex function. For simplicity, we assume that the minimum x^* of this problem exists, and that we know an estimate ϱ:

$$\|x_0 - x^*\| \leqslant \varrho, \tag{2.2}$$

where x_0 is the starting point of the method.

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

$$f(x) = \alpha f_1(x) + \beta f_2(x), \tag{2.3}$$

where $\alpha, \beta \geqslant 0$ and

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

$$f_2(x) = \max_{1 \leqslant i \leqslant n} \left| x^{(i)} \right| - x^{(1)}. \tag{2.5}$$

One can easily observe that the minimum of this function is $f^* = 0$, at point $x^* = 0$.

This then allows one to estimate the Lipschitz continuity parameter L of the objective function as $L = \alpha\sqrt{n} + 2\beta$. We can hence write

$$f \in \mathscr{F}^0_{L=\alpha\sqrt{n}+2\beta}(\mathbb{R}^n). \tag{2.6}$$

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

2.2.1. *Subgradient.* Both methods we use are based on the notion of subgradient, as defined in Definition 2.1.

Definition 2.1 (Definition 3.1.5 of [2]). A vector g is called a *subgradient* of the function f at the point $x_0 \in \operatorname{dom} f$ if for any $y \in \operatorname{dom} f$ we have

$$f(y) \geqslant f(x_0) + \langle g, y - x_0 \rangle. \tag{2.7}$$

The set of all subgradients of f at x_0, $\partial f(x_0)$, is called the *subdifferential* of the function f at the point x_0.

Computing subgradients can be done in our case according to the following set of rules:

(1) If f is differentiable at x_0, then

$$\partial f(x_0) \equiv \{f'(x_0)\}. \tag{2.8}$$

(2) If $f(x) = \alpha f_1(x) + \beta f_2(x)$, with $\alpha, \beta \geqslant 0$, then

$$\partial f(x_0) \equiv \alpha \partial f_1(x_0) + \beta \partial f_2(x_0). \tag{2.9}$$

(3) If $f(x) = \max\{f_1(x), f_2(x)\}$, then

$$\partial f(x_0) \equiv \begin{cases} \partial f_1(x_0), & \text{if } f_1(x_0) > f_2(x_0), \\ \partial f_2(x_0), & \text{if } f_1(x_0) < f_2(x_0), \\ \operatorname{Conv}\{\partial f_1(x_0), \partial f_2(x_0)\}, & \text{if } f_1(x_0) = f_2(x_0), \end{cases} \tag{2.10}$$

where the last case is taken to mean that for any $g_1 \in \partial f_1(x_0), g_2 \in \partial f_2(x_0)$,

$$\alpha g_1 + (1-\alpha) g_2 \in \partial f(x_0), \tag{2.11}$$

where $\alpha \in [0,1]$.

2.2.2. *Subgradient method.* The subgradient method can be defined by the following iteration scheme:

(1) Choose an initial point $x_0 \in \mathbb{R}^n$ and iterate from there.
(2) For $k \geqslant 0$, set

$$x_{k+1} \coloneqq x_k - \frac{\varrho}{\sqrt{k+1}} \frac{g_k}{\|g_k\|}, \tag{2.12}$$

where $g_k \in \partial f(x_k)$.

Let us define the following notation:

$$f_k^* \triangleq \min_{0 \leqslant i \leqslant k} f(x_i). \tag{2.13}$$

We then have Theorem 2.1, based on Theorem 3.2.2 of [2] with step size $h_k \coloneqq \frac{\varrho}{\sqrt{k+1}}$.

THEOREM 2.1 (Theorem 3.2.2 of [2]). *Let a function f be Lipschitz continuous with constant L, with $\|x_0 - x^*\| \leqslant \varrho$. Then*

$$f_k^* - f^* \leqslant \frac{L\varrho}{2} \frac{1 + \sum_{i=0}^{k} \frac{1}{i+1}}{\sum_{i=0}^{k} \frac{1}{\sqrt{i+1}}}. \tag{2.14}$$

2.2.3. *Ellipsoid method.* The ellipsoid method can be defined by the following iteration scheme:

(1) Choose an initial point $x_0 \in \mathbb{R}^n$, and set $H_0 \coloneqq \varrho^2 I_n$, with I_n the $n \times n$ identity matrix.
(2) For $k \geqslant 0$, set

$$x_{k+1} \coloneqq x_k - \frac{1}{n+1} \frac{H_k g_k}{\langle H_k g_k, g_k \rangle^{1/2}}, \tag{2.15}$$

$$H_{k+1} \coloneqq \frac{n^2}{n^2-1} \left(H_k - \frac{2}{n+1} \frac{H_k g_k g_k^T H_k}{\langle H_k g_k, g_k \rangle} \right), \tag{2.16}$$

where $g_k \in \partial f(x_k)$.

We then have Theorem 2.2, based on Theorem 3.2.11 of [2] but adapted for the unconstrained case.

THEOREM 2.2 (Adaptation of Theorem 3.2.11 of [2]). *Let a function f be Lipschitz continuous with constant L, with $\|x_0 - x^*\| \leqslant \varrho$. Then*

$$f_k^* - f^* \leqslant L\varrho \left(1 - \frac{1}{(n+1)^2}\right)^{k/2}, \tag{2.17}$$

where the meaning of f_k^* is the same as in §2.2.2.

2.2.4. *Complexity lower bound.* We also give the adapted result of Theorem 3.2.1 of [2].

THEOREM 2.3 (Theorem 3.2.1 of [2]). *For any k, $0 \leqslant k \leqslant n-1$, there exists a function f such that*

$$f(x_k) - f^* \geqslant \frac{L\varrho}{2(2+\sqrt{k+1})}, \tag{2.18}$$

for both the subgradient and ellipsoid method.

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. One should also take care to notice the stringent condition based on the problem dimension.

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

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

- The type of method (subgradient or ellipsoid).
- The objective function. We choose a function such as the one in §2.1.1, hence two parameters $\alpha, \beta \geqslant 0$ need to be chosen. This choice influences the Lipschitz continuity parameter $L = \alpha\sqrt{n} + 2\beta$ of the function. We make the arbitrary choice to set $\alpha = \beta = \frac{L}{\sqrt{n}+2}$, thus leaving the Lipschitz parameter as a changeable value of the problem.
- The desired accuracy of the final solution, ε. As the objective function we choose has $f^* = 0$, the termination criterion of the methods is $f(x_k) \leqslant \varepsilon$.
- The initial distance to the minimum, $\varrho = \|x_0 - x^*\|$. This distance is taken into account when randomly generating the initial solution.
- The dimension n of the problem.

3.2. *Testing strategy.* Several tests are performed in order to visualize the performance of each method, depending on the parameters of the problem:

(1) A first observation to make is the use of the term $f_k^* - f^*$ in the theoretical predictions. In the first suite of tests, discussed in §4.1, we will show that there is no theoretical guarantee that the objective function will decrease at each iteration.
(2) In the second part, discussed in §4.2, we test different problems with the following parameters (and a maximum number of iterations of 10^5).

Problem size	ε	L	ϱ	n
Small	10^{-12}	10	10	10
Medium	10^{-1}	10^2	10^2	10^2
Large	10	10^3	10^3	10^3

For each of these problems, we show the evolution of $f_k^* - f^*$, and compare this with the theoretical predictions about the rate of convergence of Theorems 2.1 and 2.2, as well as the lower bound given by Theorem 2.3.
(3) In the third part, which is discussed in §4.3, we look at the performance of the methods with respect to their execution time per iteration.[1]

[1] We however do *not* adapt the methods to be optimal for this given number of steps, in order to be consistent with other test suites.

(4) Finally, in the fourth suite of tests, we look at the influence of every parameter individually, while maintaining the others at a fixed value, on the number of iterations needed to reach a given accuracy. For this, we use the default values of the small problem, with $\varepsilon = 10^{-1}$, while varying one of the parameters at a time, for the following values:
- $L = 1, \ldots, 100$;
- $\varrho = 1, \ldots, 100$;
- $n = 2, \ldots, 250$.

We do this for both methods. These results are analyzed in §4.4.

4. **Analysis of the results**

4.1. *Incremental decrease is not guaranteed.* Figure 1 is a simple visualization of both methods operating on a 2-dimensional problem.

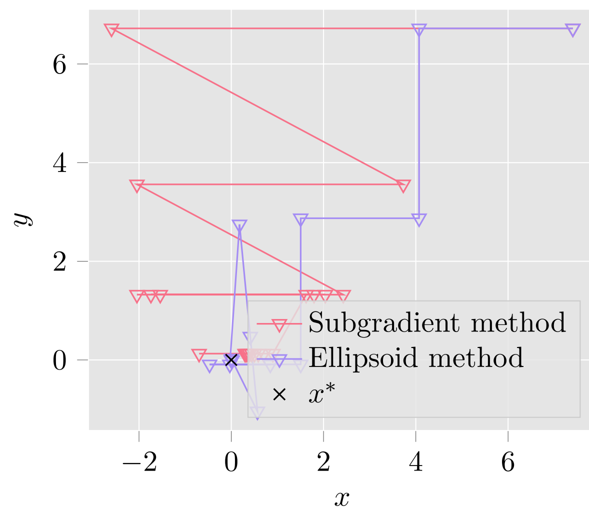

Figure 1. Iterates of the methods.

Figure 2 shows the evolution of the objective value at each iteration for the same problem.

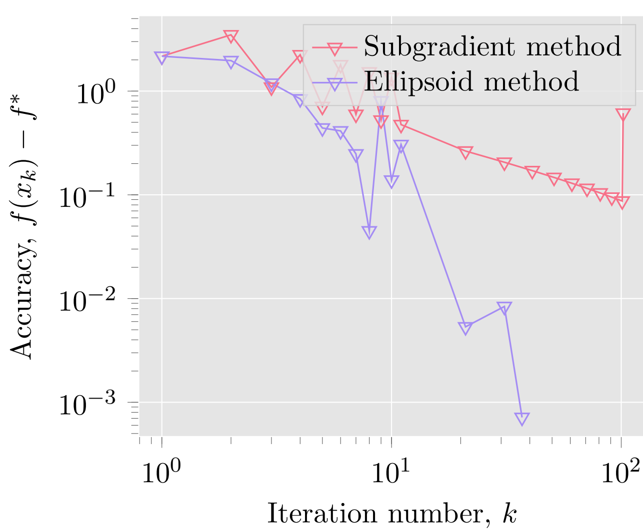

Figure 2. Accuracies of the methods.

One observes that this value is not guaranteed to decrease at each iteration, which explains why theoretical upper bound guarantees only mention $f_k^* \triangleq \min_{0\leqslant i\leqslant k} f(x_i)$.

4.2. *Convergence tests.* The following figures show the results of the second test suite. We first give the various figures, then interpret these results in §4.2.4. All figures represent the evolution of the best accuracy, $f_k^* - f^*$, as a function of the number of iterations k, for both the subgradient and ellipsoid methods, as well as some theoretical bounds taken from [2] (Theorems 2.1, 2.2 and 2.3).

4.2.1. *Small problem.* Figure 3 gives the results for the small problem.

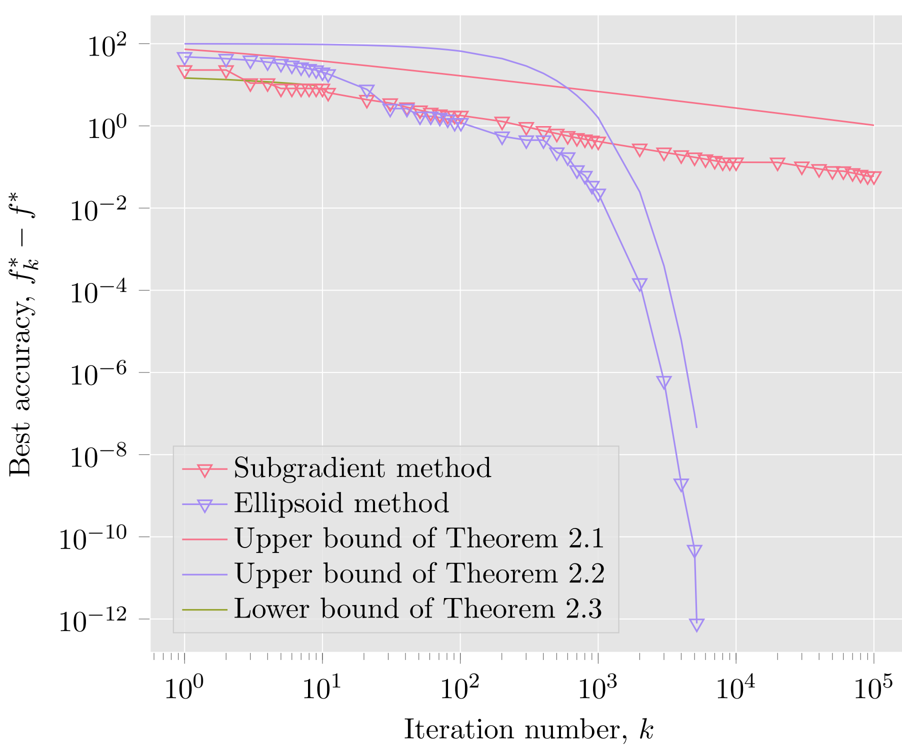

Figure 3. Best accuracy for both methods, on the small problem.

4.2.2. *Medium problem.* Figure 4 gives the results for the medium problem.

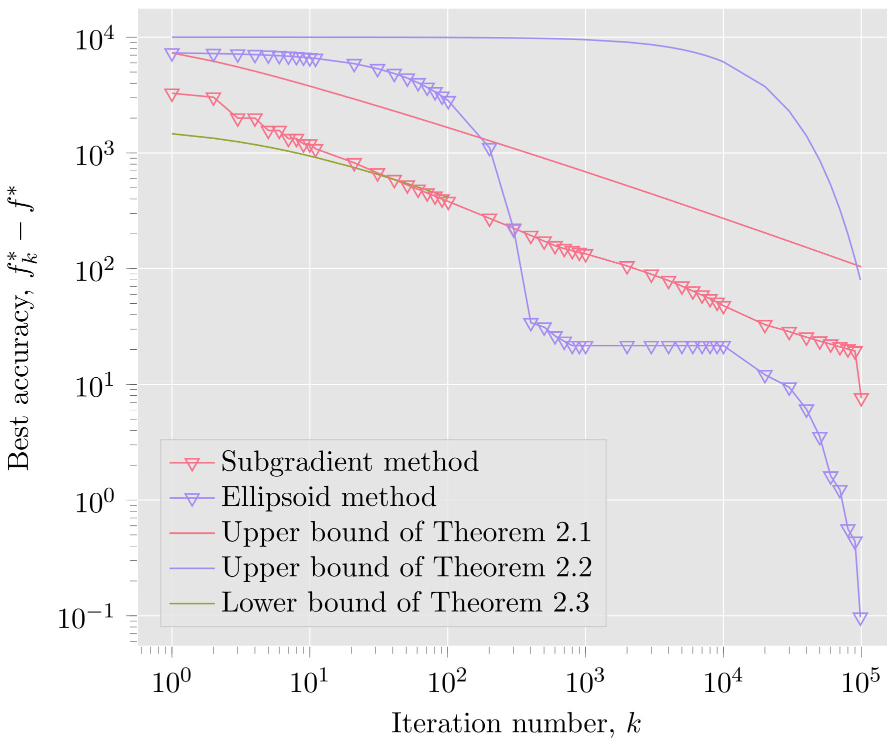

Figure 4. Best accuracy for both methods, on the medium problem.

4.2.3. *Large problem.* Figure 5 gives the results for the large problem.

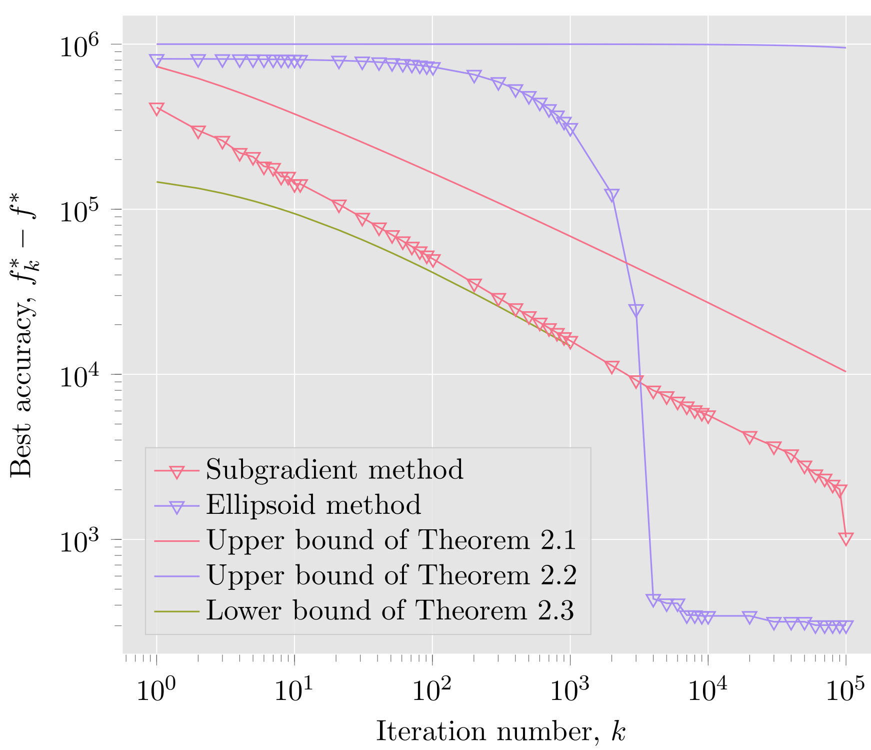

Figure 5. Best accuracy for both methods, on the large problem.

4.2.4. *Conclusion.* Before interpreting the results, one might notice that while Theorem 2.3 mentions $f(x_k) - f^*$, the figures only show $f_k^* - f^*$. However, as the latter is a lower bound on the former, this is not an issue.

A first result one can observe is that the theoretical upper bounds predicted by Theorems 2.1 and 2.2 are always respected, for every problem size.

Similarly, the lower bound of Theorem 2.3 is mostly respected, though its validity is only limited to the earliest iterations. One should note however that this lower bound is not actually a lower bound for our objective function specifically, but rather for a class of functions. With this in mind, one could conjecture that our objective function is among the hardest for that particular class, as the lower bound seems to be rather tight.

Another observation is that the ellipsoid method is the best from a certain number of iterations onward, before which the subgradient method outperforms it. This switching behaviour is also observable in the theoretical bounds, though one the large problem, this is not shown on the figure due to space constraints. The fact that the larger the problem, the larger the iteration threshold for the switch can be explained by the presence of n in the statement of Theorem 2.2.

Finally, one can also observe that problem size seems to influence complexity, as larger problems require more iterations to reach a given accuracy. One should also note that when compared with the smooth convex minimization task of the previous exercise, nonsmooth convex minimization is much harder and converges much slower, with the large problem being particularly hard to solve even with an accuracy of $\varepsilon = 10^3$.

4.3. *Execution time.* Figure 6 gives the execution time per iteration of the solver for a problem with variable dimension n, with a fixed number of iterations (10^3).

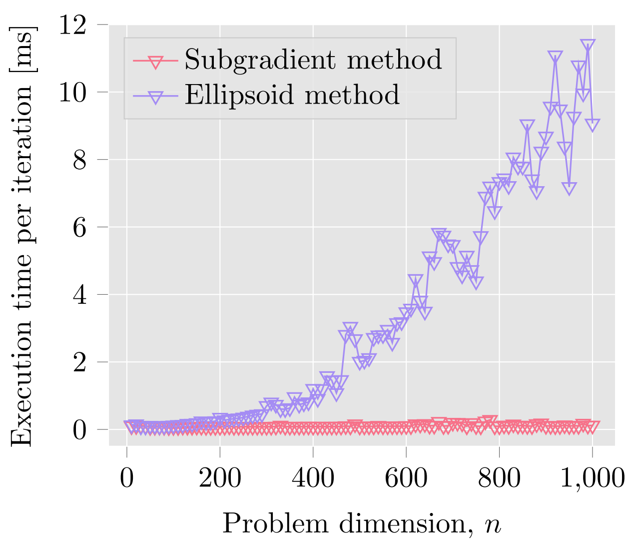

Figure 6. Execution time per iteration as a function of problem dimension.

One can observe on this figure that the ellipsoid method, while having less iterations than the subgradient method, takes a lot longer to perform one iteration of its optimization process.

This should not come as a surprise, as building the successive H_k matrices takes quadratic time in the size n of the problem.

Despite this practical inefficiency, the ellipsoid method was for a long time useful from a theoretical point of view, as only recently have interior-point algorithms been discovered with similar complexity properties. [1]

4.4. *Influence of parameters.* The following figures show the results of the last test suite, which is concerned with the influence of each parameter on the number of iterations required for convergence.

4.4.1. *Lipschitz parameter.* Figure 7 shows the influence of L, the Lipschitz continuity parameter, on the convergence of both methods.

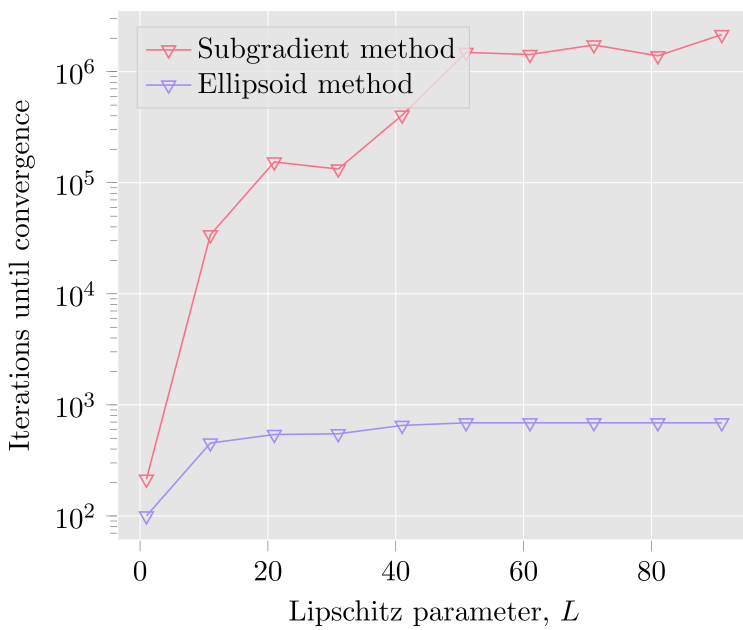

Figure 7. Influence of L on the number of iterations required for convergence.

One can make the observation, based on this figure, that the number of iterations increases with L, a result corroborated by Theorems Theorem 2.1 and 2.2. These theorems also allow one to predict similar behaviour when ϱ is changing, as both are used similarly in the formulas.

Intuitively, this can be explained by the fact that the larger L, the larger the difference in objective value for a given distance from the minimum (which is kept constant for every iteration). This observation is independent of the method used for the optimization process, and hence applies to both the subgradient and ellipsoid methods.

4.4.2. *Initial distance from the minimum.* Figure 8 shows the influence of $\varrho = \|x_0 - x^*\|$, the initial distance from the minimum, on the convergence of both methods.

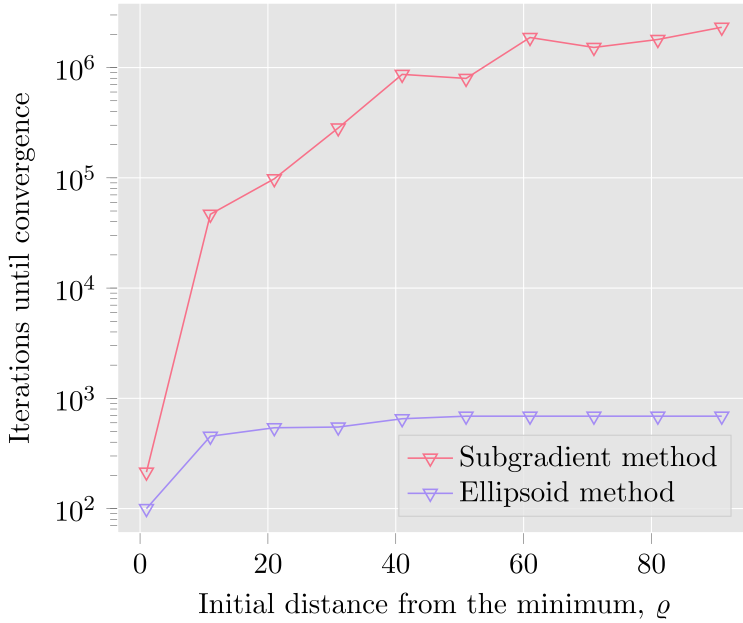

Figure 8. Influence of ϱ on the number of iterations required for convergence.

As with the Lipschitz parameter in §4.4.1, the number of iterations needed for convergence increases with ϱ, which is again to be expected when looking at the theoretical bounds.

Again, the intuition behind why this is the case is fairly trivial. If L is kept constant, then increasing the distance from the minimum increases the gap in objective values between the initial point and the optimum, which would mean more iterations are required to reach a given accuracy. This observation is again independent of the optimization method.

4.4.3. *Problem dimension.* Figure 9 shows the influence of n, the problem dimension, on the convergence of both methods.

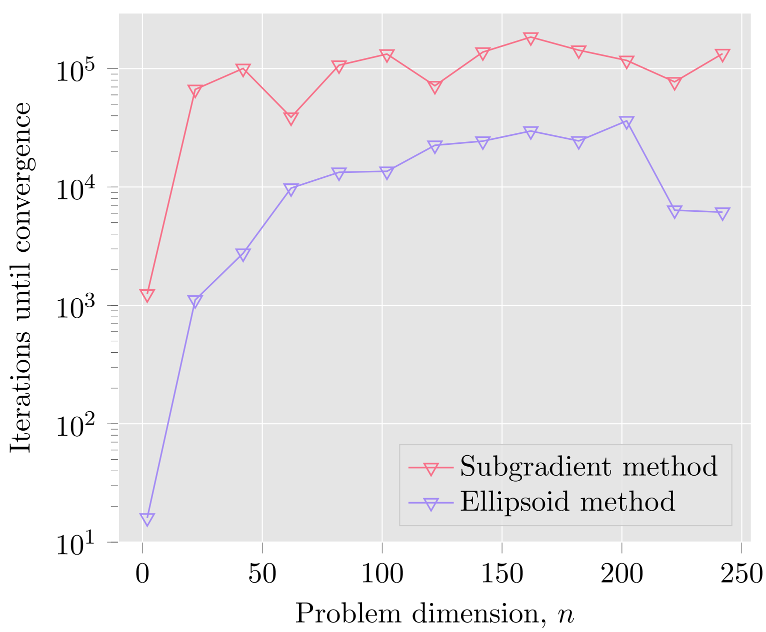

Figure 9. Influence of n on the number of iterations required for convergence.

A first observation one can make is that the subgradient method is minimally affected by an increase of n, the problem dimension, except for very low-dimensional problems ($n < 10$). On the other hand, the ellipsoid method seems to suffer strongly from such an increase, which can be explained by the bound of Theorem 2.2, which decreases more slowly as n increases.

5. **Conclusion**

In this paper, we have looked at two first-order numerical methods for nonsmooth convex minimization, the subgradient method and the ellipsoid method. We have extensively tested both methods, and compared practical results with theoretical findings, from [2].

We have experimentally observed the lack of guarantee on the incremental decrease of the objective value at each iteration, as well as the influence of several problem parameters on the convergence of both methods. Several theoretical bounds were also shown to be consistent with the observations in the paper. Additionally, we have exposed a practical problem with the ellipsoid method, which makes it unfit to handle high-dimensional optimization problems. In a practical problem (which typically entails large n), the subgradient method would thus be a safer choice than the ellipsoid method with respect to the execution time, though more experiences would be necessary to further quantify these findings.

One also notices that nonsmooth convex optimization is noticeably harder than smooth convex minimization as it was explored in the previous exercise.

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.

References

[1] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, *Algorithms and Combinatorics*, Springer-Verlag Berlin Heidelberg, 1993. http://dx.doi.org/10.1007/978-3-642-78240-4.

[2] 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

All computation results are already present in the main text, in figures 1 through 9. Additionally, the code to generate these figures (and the data they represent) is given in §B.

Appendix B. Source code

The source code is divided into three parts:

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

All three are available in full below.

B.1. `solvers.py`.

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

Author: Gilles Peiffer
Date: 2020-06-07

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

import numpy as np
import numba


#@numba.jit(nopython=True)
def solve(n, function, rho, method, eps, maximum_iterations):
    """
    Solve an optimization problem using either the subgradient or the ellipsoid 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.
    function : dict
        Parameters alpha and beta of the objective function.
    rho : float
        Initial distance from the minimum.
    method : {'subgradient', 'ellipsoid'}
        Method to use to solve the problem.
```

```
    eps : float
        Required accuracy.
    maximum_iterations : int
        Maximal number of iterations.

    Returns
    -------
    x : ndarray
        Iterates of the optimization process.
    vals : ndarray
        Function values at each iteration.
    """

    rng = np.random.default_rng(seed=69)
    it = 0

    # Start iterating at distance rho from x_opt = 0.
    x = np.zeros((maximum_iterations + 1, n))
    x[0] = rng.random((n, ))
    x[0] *= rho / np.linalg.norm(x[0])

    alpha = function['alpha']
    beta = function['beta']

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

        Parameters
        ----------
        x : ndarray
            The point at which to evaluate f.

        Returns
        -------
        fx : float
            The value of the objective function at x, f(x).
        """

        xabs = np.abs(x)
        return alpha * np.sum(xabs[:-1]) + beta * (np.max(xabs) - x[0])

    vals = np.zeros((maximum_iterations + 1, ))
    vals[0] = f(x[0])

    H = None
    constants = None

```

```
    # Compute constants for the ellipsoid method.
    if method == 'ellipsoid':
        H = np.identity(n) * rho**2
        constants = [1 / (n + 1), n**2 / (n**2 - 1), 2 / (n + 1)]

    while vals[it] > eps and it < maximum_iterations:
        it += 1

        # Compute subgradient.
        g = np.sign(x[it - 1])
        g[-1] = 0
        g *= alpha
        xabs = np.abs(x[it - 1])
        m = np.max(xabs)
        ind = np.argwhere(xabs == m)
        g[ind] += beta * np.sign(x[it - 1, ind])
        g[0] -= beta

        if method == 'subgradient':
            x[it] = x[it - 1] - rho / np.sqrt(it) * g / np.linalg.norm(g)
        elif method == 'ellipsoid':
            tmp1 = H @ g
            tmp2 = g @ tmp1
            x[it] = x[it - 1] - tmp1 * constants[0] / np.sqrt(tmp2)
            H = constants[1] * (H - constants[2] / tmp2 * np.outer(tmp1, tmp1))

        vals[it] = f(x[it])

    return x[:it + 1], vals[:it + 1]
```

B.2. `benchmarks.py`.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import pickle
import time
import numpy as np

from solvers import solve


def bm_no_incr_decr():
    """
    Show there is no guarantee of incremental decrease at every iteration.
    """
    n = 2
    eps = 0.001
```

```
    L = 1
    rho = 10
    x_subgradient, f_subgradient = solve(n, {
        'alpha': L / (np.sqrt(n) + 2),
        'beta': L / (np.sqrt(n) + 2)
    }, rho, 'subgradient', eps, 101)

    subgradient = zip(x_subgradient, f_subgradient)
    pickle.dump(subgradient,
                open("../report/data/no_incr_decr_subgradient.p", "wb"))

    x_ellipsoid, f_ellipsoid = solve(n, {
        'alpha': L / (np.sqrt(n) + 2),
        'beta': L / (np.sqrt(n) + 2)
    }, rho, 'ellipsoid', eps, 101)

    ellipsoid = zip(x_ellipsoid, f_ellipsoid)
    pickle.dump(ellipsoid, open("../report/data/no_incr_decr_ellipsoid.p",
                                "wb"))


def bm_roc():
    """
    Show theoretical guarantees are satisfied.
    """
    max_it = 100_000
    epss = [1e-12, 1e-1, 10]
    ns = [10, 100, 1000]
    Ls = [10, 100, 1000]
    rhos = [10, 100, 1000]
    names = ['small', 'medium', 'large']
    for eps, n, L, rho, name in zip(epss, ns, Ls, rhos, names):
        print("Size: %s" % (name))
        print(" - subgradient")
        x_subgradient, f_subgradient = solve(n, {
            'alpha': L / (np.sqrt(n) + 2),
            'beta': L / (np.sqrt(n) + 2)
        }, rho, 'subgradient', eps, max_it)

        subgradient = zip(x_subgradient, f_subgradient)
        pickle.dump(subgradient,
                    open("../report/data/roc_subgradient_%s.p" % (name),
                    ↪  "wb"))

        print(" - ellipsoid")

        x_ellipsoid, f_ellipsoid = solve(n, {
            'alpha': L / (np.sqrt(n) + 2),
```

```
            'beta': L / (np.sqrt(n) + 2)
        }, rho, 'ellipsoid', eps, max_it)

        ellipsoid = zip(x_ellipsoid, f_ellipsoid)
        pickle.dump(ellipsoid,
                    open("../report/data/roc_ellipsoid_%s.p" % (name), "wb"))


def bm_time():
    """
    Show that ellipsoid is slow.
    """
    L = 10
    eps = 1e-6
    rho = 10
    max_it = 1000
    sg = []
    el = []
    for n in range(10, 1001, 10):
        print("n: %d" % n)
        s = time.perf_counter()
        _ = solve(n, {
            'alpha': L / (np.sqrt(n) + 2),
            'beta': L / (np.sqrt(n) + 2)
        }, rho, 'subgradient', eps, max_it)
        e = time.perf_counter()

        sg.append(e - s)

        s = time.perf_counter()
        _ = solve(n, {
            'alpha': L / (np.sqrt(n) + 2),
            'beta': L / (np.sqrt(n) + 2)
        }, rho, 'ellipsoid', eps, max_it)
        e = time.perf_counter()

        el.append(e - s)

    pickle.dump(sg, open("../report/data/exec_time_subgradient.p", "wb"))
    pickle.dump(el, open("../report/data/exec_time_ellipsoid.p", "wb"))


def bm_params():
    """
    Show the influence of each parameter.
    """
    default_n = 10
    default_L = 10
```

```
    default_rho = 10
    eps = 1e-1
    max_it = 1_000_000_000

    Ls = range(1, 100, 10)
    rhos = range(1, 100, 10)
    ns = range(2, 250, 20)

    L_influence = {'subgradient': [], 'ellipsoid': []}
    for L in Ls:
        print("L: %d" % (L))
        print(" - subgradient")
        x_subgradient, _ = solve(
            default_n, {
                'alpha': L / (np.sqrt(default_n) + 2),
                'beta': L / (np.sqrt(default_n) + 2)
            }, default_rho, 'subgradient', eps, max_it)

        L_influence['subgradient'].append(len(x_subgradient))

        print(" - ellipsoid")

        x_ellipsoid, _ = solve(
            default_n, {
                'alpha': L / (np.sqrt(default_n) + 2),
                'beta': L / (np.sqrt(default_n) + 2)
            }, default_rho, 'ellipsoid', eps, max_it)

        L_influence['ellipsoid'].append(len(x_ellipsoid))

    pickle.dump(L_influence, open("../report/data/param_L.p", "wb"))

    rho_influence = {'subgradient': [], 'ellipsoid': []}
    for rho in rhos:
        print("rho: %d" % (rho))
        print(" - subgradient")
        x_subgradient, _ = solve(
            default_n, {
                'alpha': default_L / (np.sqrt(default_n) + 2),
                'beta': default_L / (np.sqrt(default_n) + 2)
            }, rho, 'subgradient', eps, max_it)

        rho_influence['subgradient'].append(len(x_subgradient))

        print(" - ellipsoid")

        x_ellipsoid, _ = solve(
            default_n, {
```

```
                'alpha': default_L / (np.sqrt(default_n) + 2),
                'beta': default_L / (np.sqrt(default_n) + 2)
            }, rho, 'ellipsoid', eps, max_it)

        rho_influence['ellipsoid'].append(len(x_ellipsoid))

    pickle.dump(rho_influence, open("../report/data/param_rho.p", "wb"))

    n_influence = {'subgradient': [], 'ellipsoid': []}
    for n in ns:
        print("n: %d" % (n))
        print(" - subgradient")
        x_subgradient, _ = solve(
            n, {
                'alpha': default_L / (np.sqrt(n) + 2),
                'beta': default_L / (np.sqrt(n) + 2)
            }, default_rho, 'subgradient', eps, max_it)

        n_influence['subgradient'].append(len(x_subgradient))

        print(" - ellipsoid")

        x_ellipsoid, _ = solve(
            n, {
                'alpha': default_L / (np.sqrt(n) + 2),
                'beta': default_L / (np.sqrt(n) + 2)
            }, default_rho, 'ellipsoid', eps, max_it)

        n_influence['ellipsoid'].append(len(x_ellipsoid))

    pickle.dump(n_influence, open("../report/data/param_n.p", "wb"))


if __name__ == '__main__':
    #bm_no_incr_decr()
    #bm_roc()
    #bm_time()
    bm_params()
```

B.3. plots.py.

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

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

```python
import pickle

plt.style.use("ggplot")

from solvers import solve

clrs = sns.husl_palette(4)


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

    if len(l) <= 100:
        return r + l[10::10] + [l[-1]]
    r = r + l[10:100:10]

    if len(l) <= 1000:
        return r + l[100::100] + [l[-1]]
    r = r + l[100:1000:100]

    if len(l) <= 10000:
        return r + l[1000::1000] + [l[-1]]
    r = r + l[1000:10000:1000]

    if len(l) <= 100000:
        return r + l[10000::10000] + [l[-1]]
    r = r + l[10000:100000:10000]

    if len(l) <= 1000000:
        return r + l[100000::100000] + [l[-1]]


def plt_no_incr_decr():
    # Plot iterates and function val non-decrease side by side for both methods.
    subgradient = pickle.load(
        open("../report/data/no_incr_decr_subgradient.p", "rb"))
    x_subgradient, f_subgradient = list(zip(*subgradient))

    ellipsoid = pickle.load(
        open("../report/data/no_incr_decr_ellipsoid.p", "rb"))
    x_ellipsoid, f_ellipsoid = list(zip(*ellipsoid))

    l_subgradient = np.array(reduce(range(1, len(f_subgradient) + 1)))
    l_ellipsoid = np.array(reduce(range(1, len(f_ellipsoid) + 1)))
```

```python

    x_subgradient = np.array(x_subgradient)[l_subgradient - 1]
    f_subgradient = np.array(f_subgradient)[l_subgradient - 1]
    x_ellipsoid = np.array(x_ellipsoid)[l_ellipsoid - 1]
    f_ellipsoid = np.array(f_ellipsoid)[l_ellipsoid - 1]

    plt.figure()
    plt.plot([i[0] for i in x_subgradient], [i[1] for i in x_subgradient],
             "-v",
             markerfacecolor='none',
             c=clrs[0])
    plt.plot([i[0] for i in x_ellipsoid], [i[1] for i in x_ellipsoid],
             "-v",
             markerfacecolor='none',
             c=clrs[3])
    plt.plot(0, 0, 'x', c='black')
    plt.xlabel("\\(x\\)")
    plt.ylabel("\\(y\\)")
    plt.legend(["Subgradient method", "Ellipsoid method", "\\(\\xopt\\)"])

    tikzplotlib.save("../report/plots/no_incr_decr_iterates.tikz",
                     axis_width="0.5\\linewidth")

    plt.figure()
    plt.loglog(l_subgradient,
               f_subgradient,
               "-v",
               markerfacecolor='none',
               c=clrs[0])
    plt.loglog(l_ellipsoid,
               f_ellipsoid,
               "-v",
               markerfacecolor='none',
               c=clrs[3])
    plt.xlabel("Iteration number, \\(k\\)")
    plt.ylabel("Accuracy, \\(f(\\xk) - \\fopt\\)")
    plt.legend(["Subgradient method", "Ellipsoid method"])

    tikzplotlib.save("../report/plots/no_incr_decr_vals.tikz",
                     axis_width="0.5\\linewidth")


def plt_roc():
    """
    Show theoretical guarantees are satisfied.
    """
    subgradient_s = pickle.load(
        open("../report/data/roc_subgradient_small.p", "rb"))
```

```
    subgradient_m = pickle.load(
        open("../report/data/roc_subgradient_medium.p", "rb"))
    subgradient_l = pickle.load(
        open("../report/data/roc_subgradient_large.p", "rb"))

    _, f_subgradient_s = list(zip(*subgradient_s))
    _, f_subgradient_m = list(zip(*subgradient_m))
    _, f_subgradient_l = list(zip(*subgradient_l))

    ellipsoid_s = pickle.load(
        open("../report/data/roc_ellipsoid_small.p", "rb"))
    ellipsoid_m = pickle.load(
        open("../report/data/roc_ellipsoid_medium.p", "rb"))
    ellipsoid_l = pickle.load(
        open("../report/data/roc_ellipsoid_large.p", "rb"))

    _, f_ellipsoid_s = list(zip(*ellipsoid_s))
    _, f_ellipsoid_m = list(zip(*ellipsoid_m))
    _, f_ellipsoid_l = list(zip(*ellipsoid_l))

    l_subgradient_s = np.array(reduce(range(1, len(f_subgradient_s))))
    l_subgradient_m = np.array(reduce(range(1, len(f_subgradient_m))))
    l_subgradient_l = np.array(reduce(range(1, len(f_subgradient_l))))

    l_ellipsoid_s = np.array(reduce(range(1, len(f_ellipsoid_s))))
    l_ellipsoid_m = np.array(reduce(range(1, len(f_ellipsoid_m))))
    l_ellipsoid_l = np.array(reduce(range(1, len(f_ellipsoid_l))))

    f_subgradient_s = np.minimum.accumulate(
        np.array(f_subgradient_s)[l_subgradient_s])
    f_subgradient_m = np.minimum.accumulate(
        np.array(f_subgradient_m)[l_subgradient_m])
    f_subgradient_l = np.minimum.accumulate(
        np.array(f_subgradient_l)[l_subgradient_l])

    f_ellipsoid_s = np.minimum.accumulate(
        np.array(f_ellipsoid_s)[l_ellipsoid_s])
    f_ellipsoid_m = np.minimum.accumulate(
        np.array(f_ellipsoid_m)[l_ellipsoid_m])
    f_ellipsoid_l = np.minimum.accumulate(
        np.array(f_ellipsoid_l)[l_ellipsoid_l])

    l_sg = [l_subgradient_s, l_subgradient_m, l_subgradient_l]
    l_el = [l_ellipsoid_s, l_ellipsoid_m, l_ellipsoid_l]
    sg = [f_subgradient_s, f_subgradient_m, f_subgradient_l]
    el = [f_ellipsoid_s, f_ellipsoid_m, f_ellipsoid_l]
    name = ['small', 'medium', 'large']

```

```
    L = {'small': 10, 'medium': 100, 'large': 1000}
    rho = {'small': 10, 'medium': 100, 'large': 1000}
    n = {'small': 10, 'medium': 100, 'large': 1000}

    for l_subgradient, l_ellipsoid, f_subgradient, f_ellipsoid, name in zip(
            l_sg, l_el, sg, el, name):
        plt.figure()
        plt.loglog(l_subgradient,
                   f_subgradient,
                   "-v",
                   markerfacecolor='none',
                   c=clrs[0])
        plt.loglog(l_ellipsoid,
                   f_ellipsoid,
                   "-v",
                   markerfacecolor='none',
                   c=clrs[3])

        thm322 = L[name] * rho[name] / 2 * np.array(
            [(1 + np.sum([1 / (i + 1) for i in range(k + 1)])) /
             np.sum([1 / np.sqrt(i + 1) for i in range(k + 1)])
             for k in l_subgradient])

        plt.loglog(l_subgradient, thm322, '-', c=clrs[0])

        thm3211 = L[name] * rho[name] * (1 - 1 /
                                         (n[name] + 1)**2)**(l_ellipsoid / 2)

        plt.loglog(l_ellipsoid, thm3211, '-', c=clrs[3])

        n_range = np.array(reduce(range(1, n[name])))
        thm321 = L[name] * rho[name] / (2 * (2 + np.sqrt(n_range + 1)))

        plt.loglog(n_range, thm321, '-', c=clrs[1])

        plt.xlabel("Iteration number, \\(k\\)")
        plt.ylabel("Best accuracy, \\(\\foptk - \\fopt\\)")
        plt.legend([
            "Subgradient method", "Ellipsoid method",
            "Upper bound of \\thmref{thm:3.2.2}",
            "Upper bound of \\thmref{thm:3.2.11}",
            "Lower bound of \\thmref{thm:3.2.1}"
        ])

        tikzplotlib.save("../report/plots/roc_%s.tikz" % (name),
                         axis_width="0.7\\linewidth")
```

```
def plt_time():
    """
    Show execution times.
    """
    sg = pickle.load(open("../report/data/exec_time_subgradient.p", "rb"))
    el = pickle.load(open("../report/data/exec_time_ellipsoid.p", "rb"))

    n = np.array(range(10, 1001, 10))

    plt.figure()
    plt.plot(n, sg, "-v", markerfacecolor='none', c=clrs[0])
    plt.plot(n, el, "-v", markerfacecolor='none', c=clrs[3])

    plt.xlabel("Problem dimension, \\(n\\)")
    plt.ylabel("Execution time per iteration [\\si{\\milli\\second}]")
    plt.legend(["Subgradient method", "Ellipsoid method"])

    tikzplotlib.save("../report/plots/exec_time.tikz",
                     axis_width="0.5\\linewidth")


def plt_params():
    """
    Show the influence of each parameter.
    """
    L = pickle.load(open("../report/data/param_L.p", "rb"))
    rho = pickle.load(open("../report/data/param_rho.p", "rb"))
    n = pickle.load(open("../report/data/param_n.p", "rb"))

    L_sg = L['subgradient']
    L_el = L['ellipsoid']
    rho_sg = rho['subgradient']
    rho_el = rho['ellipsoid']
    n_sg = n['subgradient']
    n_el = n['ellipsoid']

    Ls = range(1, 100, 10)
    rhos = range(1, 100, 10)
    ns = range(2, 250, 20)

    plt.figure()
    plt.semilogy(Ls, L_sg, "-v", markerfacecolor='none', c=clrs[0])
    plt.semilogy(Ls, L_el, "-v", markerfacecolor='none', c=clrs[3])
    plt.xlabel("Lipschitz parameter, \\(L\\)")
    plt.ylabel("Iterations until convergence")
    plt.legend(["Subgradient method", "Ellipsoid method"])

    tikzplotlib.save("../report/plots/param_L.tikz",
```

```
                        axis_width="0.6\\linewidth")

    plt.figure()
    plt.semilogy(rhos, rho_sg, "-v", markerfacecolor='none', c=clrs[0])
    plt.semilogy(rhos, rho_el, "-v", markerfacecolor='none', c=clrs[3])
    plt.xlabel("Initial distance from the minimum, \\(\\rho\\)")
    plt.ylabel("Iterations until convergence")
    plt.legend(["Subgradient method", "Ellipsoid method"])

    tikzplotlib.save("../report/plots/param_rho.tikz",
                        axis_width="0.6\\linewidth")

    plt.figure()
    plt.semilogy(ns, n_sg, "-v", markerfacecolor='none', c=clrs[0])
    plt.semilogy(ns, n_el, "-v", markerfacecolor='none', c=clrs[3])
    plt.xlabel("Problem dimension, \\(n\\)")
    plt.ylabel("Iterations until convergence")
    plt.legend(["Subgradient method", "Ellipsoid method"])

    tikzplotlib.save("../report/plots/param_n.tikz",
                        axis_width="0.6\\linewidth")


if __name__ == '__main__':
    #plt_no_incr_decr()
    #plt_roc()
    #plt_time()
    plt_params()
```

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