Computational mathematics for learning and data analysis project report

Dalla Noce Niko, Ristori Alessandro, Rizzo Simone
Master Degree in Computer science.
n.dallanoce@studenti.unipi.it, a.ristori5@studenti.unipi.it, s.rizzo14@studenti.unipi.it.
Computational mathematics for learning and data analysis, Academic Year: 2021/2022
Wildcard project
Date: 26/07/2022
`https://github.com/nikodallanoce/ComputationalMathematics`

Abstract

(P) is the linear least squares problem

$$\min_w \|\hat{X}w - \hat{y}\|$$

where

$$\hat{X} = \begin{bmatrix} X^T \\ \lambda I \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} y \\ 0 \end{bmatrix},$$

with X the (tall thin) matrix from the ML-cup dataset by prof. Micheli, and y is a random vector.

(A1) is an algorithm of the class of limited-memory quasi-Newton methods [4].

(A2) is thin QR factorization with Householder reflectors, in the variant where one does not form the matrix Q, but stores the Householder vectors u_k and uses them to perform (implicitly) products with Q and Q^T.

(A3) is an algorithm of the class of Conjugate Gradient methods [15].

(A4) is a standard momentum descent (heavy ball) approach [11].

Contents

List of Figures

List of Tables

1 Introduction

In this section we will provide an inside view about the task by showing a brief description of it, its convexity study and Lipschitz continuity, needed to introduce the algorithm properties over the next section.

1.1 Linear Least Squares

Linear least squares problem (we will call it **LLS** for the rest of the report) is an optimization problem in which we need to find a vector $x \in \mathbb{R}^n$) such that it minimizes $\|Ax - b\|_2$, formally

$$\min_x f(x) \quad \text{where} \quad f(x) = \|Ax - b\|_2 \tag{1.1}$$

with:

- $A \in \mathbb{R}^{m \times n}$ is a tall thin matrix where, typically, $m >> n$, like in a model fitting setting.
- $b \in \mathbb{R}^m$ is a vector, or, using a model fitting vocabulary, the expected values.
- $\|x\|_2$, is the euclidean norm (or 2-norm) such that $\|x\|_2 = \sqrt{\sum_i {x_i}^2} = \sqrt{x^T x}$.

by keeping in mind that the square root function is monotone, minimizing this function is equal to minimize its argument, formally $\min_x \sqrt{f(x)} = \min_x f(x)$. In our case, $\min_w \|\hat{X}w - \hat{y}\|_2 = \min_w \|\hat{X}w - \hat{y}\|_2^2$.

The LLS solution is obtained by finding the non-stationary points of $f(w)$, by, first, rewriting it as following

$$\begin{aligned} f(w) &= \|\hat{X}w - \hat{y}\|_2^2 = (\hat{X}w - \hat{y})^T(\hat{X}w - \hat{y}) = ((\hat{X}w)^T - \hat{y}^T)(\hat{X}w - \hat{y}) \\ &= (w^T\hat{X}^T - \hat{y}^T)(\hat{X}w - \hat{y}) = w^T\hat{X}^T\hat{X}w - w^T\hat{X}^T\hat{y} - \hat{y}^T\hat{X}w + \hat{y}^T\hat{y} \\ &\overset{1}{=} w^T\hat{X}^T\hat{X}w - 2\hat{y}^T\hat{X}w + \hat{y}^T\hat{y} \end{aligned} \tag{1.2}$$

Now, we define the gradient of $f(w)$ as

$$\nabla f(w) = 2w^T\hat{X}^T\hat{X} - 2\hat{y}^T\hat{X} \tag{1.3}$$

The solution can therefore be defined by putting $\nabla f(w) = 0$, thus leading us to

$$w^T\hat{X}^T\hat{X} - \hat{y}^T\hat{X} = 0 \tag{1.4}$$

which is known as the normal equation of the LLS, but the latter tends to be ill-conditioned in practice, so although using (1.4) could be a good way to solve the task, there exist way better methods for such purpose (which is also the aim of this project).

Let's now define the Hessian of $f(w)$ as

$$\nabla^2 f(w) = 2\hat{X}^T\hat{X} \tag{1.5}$$

[1] this holds because $(w^T\hat{X}^T\hat{y})^T = \hat{y}^T\hat{X}w$ and they are both scalars.

1.2 Convexity

By looking at the Hessian matrix (1.5) we can say that is always positive definite, this statement, without taking into account the constant 2, holds for our case since $\hat{X}$ has full-column rank and also

$$z^T(\hat{X}^T\hat{X})z = (\hat{X}z)^T(\hat{X}z) = \|\hat{X}z\|_2^2 > 0, \quad \forall z \neq 0 \tag{1.6}$$

Knowing that the Hessian is always positive definite, we can assert that LLS problems are convex. In fact, from the literature, it is known that a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is convex iff the Hessian $\nabla^2 f(x)$ is positive semi-definite (which, of course, is true since positive definiteness implies positive semi-definiteness), $\forall x \in \mathbb{R}^n$.

Global minima Thanks to (1.6) we know that the Hessian (1.5) is positive definite in any point $z \in \mathbb{R}^n \backslash 0$ and, therefore, our function is strongly convex, this implies that the stationary point is a global unique minimum.

1.3 Lipschitz continuous

The gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$ is Lipschitz continuous with Lipschitz constant $L \geq 0$ if

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|(y - x)\|, \forall x, y \in \mathbb{R}^n \tag{1.7}$$

The Lipschitz constant of the gradient of the LLS is equal to the square of the largest singular value s_1 of the feature matrix $\hat{X} \in \mathbb{R}^{m\times n}$ since for any $\beta_1, \beta_2 \in \mathbb{R}^n$:

$$\|\nabla f(\beta_2) - \nabla f(\beta_1)\| \overset{2}{=} \|\hat{X}^T\hat{X}(\beta_2 - \beta_1)\| \leq s_1^2\|\beta_2 - \beta_1\| \tag{1.8}$$

We can show that the second part of (1.8) holds by exploiting the fact that the norm product is less or equal to the product of the norms: $\|Av\| \leq \|A\| * \|v\|$, therefore we can state the following

$$\|\hat{X}^T\hat{X}(\beta_2 - \beta_1)\| \leq \|\hat{X}^T\hat{X}\| * \|\beta_2 - \beta_1\| \tag{1.9}$$

Now, for any symmetric matrix (like the one we have, $\hat{X}^T\hat{X}$), $\|\hat{X}^T\hat{X}\|$ is equal to the squared largest singular value of the matrix $\hat{X}$, corresponding to the largest eigenvalue of $\hat{X}^T\hat{X}$. In fact, lets consider the SVD factorization of $\hat{X}$ defined as $\hat{X} = U\Sigma V^T$, we can write $\hat{X}^T\hat{X} = V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T$, since U is orthogonal ($U^T U = UU^T = I$).

Moreover, $\hat{X}^T\hat{X}$ is symmetric and this implies that

$$\|\hat{X}^T\hat{X}\| = \rho(\hat{X}^T\hat{X}) = \max_i |\lambda_i| \tag{1.10}$$

where $\rho(\hat{X}^T\hat{X})$ is the spectral radius of $\hat{X}^T\hat{X}$. Recalling that $eig(\hat{X}^T\hat{X}) = \Sigma^2(\hat{X})$ we conclude that $\|\hat{X}^T\hat{X}\| = \max\{s_i^2 \in \Sigma^2\} = s_1^2$ since the largest singular value is the first one on the matrix diagonal, this leads us to

$$\|\hat{X}^T\hat{X}\| * \|\beta_2 - \beta_1\| = s_1^2\|\beta_2 - \beta_1\| \tag{1.11}$$

Which is what we stated in (1.8), therefore s_1^2 is the Lipschitz constant of the LLS.

[2]Holds due to the gradient definition for the LLS problem as shown in (1.3), we do not consider the constant since it has no effect.

2 Limited-memory quasi-Newton method

Over this section we will explore one of the most popular and efficient algorithm of of the quasi-Newton methods, we will show its convergence, complexity and, finally, we will analyze its performances on the LLS task.

2.1 Overview on quasi-Newton methods

The main feature of quasi-Newton methods, as opposed to pure Newton methods, is the fact that they do not need to compute the Hessian at each iteration and, therefore, are a cheaper solution, albeit coming with worse convergence speed.

First of all, lets introduce the following quadratic model of the objective function at the *k-th* iteration, defined as

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p \tag{2.1}$$

As we can see from (2.1), we use B_k as an approximation of the true Hessian matrix, which is typical in the context of the quasi-Newton methods. B_k is an $n \times n$ symmetric positive definite matrix that is updated at each iteration.

We can explicitly write the minimizer p_k (which indicates the search direction) of this convex quadratic model as

$$p_k = -B_k^{-1} \nabla f_k \tag{2.2}$$

The new step $k+1$ is defined as the following

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.3}$$

where α_k is the step length that should be chosen in order to satisfy the Wolfe conditions

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha_k p_k^T \nabla f(x_k), \tag{2.4a}$$

$$-p_k^T \nabla f(x_k + \alpha_k p_k) \leq -c_2 p_k^T \nabla f(x_k) \tag{2.4b}$$

with $0 < c_1 < c_2 < 1$.

Since our function is quadratic and convex, α_k can be actually computed by an exact line search in the following way

$$\alpha_k = -\frac{\nabla f_k^T p_k}{p_k^T A p_k} \tag{2.5}$$

where $A = \hat{X}^T \hat{X}$, (2.5) was taken from chapter 3.5 by Nocedal et al. [8].

At the new iterate x_{k+1}, the model defined in (2.1) has gradient

$$\nabla m_{k+1}(p) = \nabla f_{k+1} + p^T B_{k+1} \tag{2.6}$$

The gradient of m_{k+1} should match the one of the objective function f at the latest two iterates x_k and x_{k+1}. The second of these conditions is verified because $\nabla m_{k+1}(0) = \nabla f_{k+1}$, while the

first one can be written as $\nabla m_{k+1}(-\alpha_k p_k) = \nabla f_k$, as defined in (2.6). By rearranging the latter we obtain

$$B_{k+1}\alpha_k p_k = \nabla f_{k+1} - \nabla f_k \tag{2.7}$$

Now, lets define the displacement and the difference between gradients

$$s_k = x_{k+1} - x_k = \alpha_k p_k, \quad y_k = \nabla f_{k+1} - \nabla f_k \tag{2.8}$$

then the (2.7) becomes

$$B_{k+1}s_k = y_k \tag{2.9}$$

The (2.9) is known as the *secant equation* and its purpose is to map s_k into y_k, this is achievable iff the *curvature condition* holds

$$s_k^T y_k > 0 \tag{2.10}$$

luckily for us, (2.10) always holds for convex functions like our case, so we can be sure that B_{k+1} will inherit the positive definiteness from B_k.

To determine B_{k+1} uniquely, an additional condition must be imposed such that among all symmetric matrices satisfying the secant equation we take the closest to the current matrix B_k. Knowing that $B = B^T$ and $By_k = s_k$, the condition is the following [3]

$$\min_B \|B - B_k\|_W \tag{2.11}$$

We will discuss more deeply about one of the the quasi-Newton methods in section 2.2, where we will also show how to implement it.

2.2 Limited memory BFGS

The chosen quasi-Newton algorithm is the BFGS method, named after its developers Broyden, Fletcher, Goldfarb, and Shanno, specifically we are going to implement a lighter in memory version of this called L-BFGS [7] which does not need save the entire Hessian approximation, but it updates the matrix by exploit the previous m iterations.

By introducing $H_k = B_k^{-1}$, we can impose the same conditions that B_k had on H_k, therefore the latter must be symmetric and positive definite and should satisfy both (2.9), $H_k s_k = y_k$, and (2.11), hence

$$\min_H \|H - H_k\|_W \tag{2.12}$$

The $k+1$ step in (2.3), now becomes

$$x_{k+1} = x_k - \alpha_k H_k \nabla f_k \tag{2.13}$$

With a BFGS approach, H is then updated at every iteration by means of the following formula

$$H_{k+1} = V_k^T H_k V_k + \rho_k s_k s_k^T \tag{2.14}$$

[3] supposing that $\|A\|_W = \|W^{\frac{1}{2}} A W^{\frac{1}{2}}\|_F$ is the weighted Frobenius norm, where W is any matrix such that $W s_k = y_k$.

where

$$\rho = \frac{1}{y_k^T s_k}, \quad V_k = I - \rho_k s_k y_k^T \tag{2.15}$$

Most of the times H_k will be dense, hence the cost of storing and manipulating will be prohibitive when the number of variables becomes larger. The solution consists in dealing with a modified version of H_k that stores a l vector pairs s_i, y_i.

The product $H_k \nabla f_k$ can be approximated by performing a sequence of products and vector summations involving the pairs $\{s_i, y_i\}$. After the new iterate is computed, the oldest vector pair in the set of pairs $\{s_i, y_i\}$ is replaced by the new pair $\{s_k, y_k\}$ obtained from (2.8).

After choosing some initial Hessian approximation H_k^0 (that is allowed to vary from iteration to iteration) and by repeating the formula (2.14), we find that the H_k satisfies the following

$$\begin{aligned} H_k =& (V_{k-1}^T \dots V_{k-l}^T) H_k^0 (V_{k-l} \dots V_{k-1}) \\ &+ \rho_{k-l} (V_{k-1}^T \dots V_{k-l+1}^T) s_{k-l} s_{k-l}^T (V_{k-l+1} \dots V_{k+1}) \\ &+ \rho_{k-l+1} (V_{k-1}^T \dots V_{k-l+2}^T) s_{k-l} s_{k-l+1}^T (V_{k-l+2} \dots V_{k+1}) \\ &+ \dots \\ &+ \rho k - 1 s_{k-1} s_{k-1}^T. \end{aligned} \tag{2.16}$$

Thanks to (2.16) we can compute $H_k \nabla f_k$ in (2.13) with algorithm 1

Algorithm 1: L-BFGS two-loop recursion

```
q = ∇f_k;
for i = k − 1, k − 2, ..., k − l do
    α_i = ρ_i s_i^T q;
    q = q − α_i y_i;
r = H_k^0 q;
for i = k − l, k − l + 1, ... k − 1 do
    β = ρ_i y_i^T r;
    r = r + s_i(α_i − β);
return r;
```

A renowned method for choosing H_k^0 that has been proved effective in practice is to set $H_k^0 = \gamma_k I$. γ_k is the scaling factor that attempts to estimate the size of the true Hessian matrix along the most recent search direction and it is defined as

$$\gamma_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} \tag{2.17}$$

Algorithm 1 has the advantage that the multiplication by the initial matrix H_0^k is isolated from the rest of the computations, allowing this matrix to be chosen freely and to vary between iterations.

Algorithm 2: L-BFGS

Choose starting point x_0, integer $l > 0$;
$k = 0$;
while *convergence not reached* **do**
    Choose H_0;
    Compute $p_k = -H_k \nabla f_k$ from algorithm 1;
    Compute $x_{k+1} = x_k + \alpha_k p_k$, where α_k is chosen by an exact line search;
    if $k > l$ **then**
        Discard the vector pair $\{s_{k-l}, y_{k-l}\}$ from storage;
    Compute and save $s_k \leftarrow x_{k+1} - x_k, y_k = \nabla f_{k+1} - \nabla f_k$;
    $k = k + 1$;

2.3 Convergence and complexity

It is already well-known that BFGS converges for convex minimization problems like LLS (we showed this property in subsection 1.2), moreover, it has been shown that even L-BFGS converges on such tasks [7], so we will not dive to much in details about all the literature about the convergence properties of such methods, but it is somewhat interesting showing the convergence rate of L-BFGS.

Keeping in mind that BFGS, and therefore its limited-memory verison, converges on strongly convex tasks, like LLS, since it satisfies the assumptions that [8] states in theorem 6.5:

1. *The objective function f is twice continuously differentiable*, which is trivial as we know that $\nabla^2 f = 2\hat{X}^T\hat{X}$ and this is also positive definite.

2. *B_k needs to be positive definite at each iteration*, this holds only if the *curvature condition* defined by (2.10) is satisfied at each iteration, but, as we already stated, the latter always holds for convex functions so even this assumption is trivially fine for our case.

From [8] (section 6.4, until theorem 6.5) we can retrieve the fact that the convergence of the iterates is linear. In particular, the sequence $\|x_k - x^*\|$ converges to zero rapidly enough that

$$\sum_{k=1}^{\infty} \|x_k - x^*\| < \infty \tag{2.18}$$

where x^* is the minimizer of the function.

First, we know that quasi-newton methods converge with superlinear rate under the existence of step length α_k that satisfies the Wolfe conditions (2.4). The following theorem 2.1 states the existence of such α_k ([8], theorem 3.6):

Theorem 2.1 *Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable. Consider the iteration $x_{k+1} = x_k + \alpha_k p_k$, where p_k is a descent direction and α_k satisfies the Wolfe conditions*

with $c_1 \leq \frac{1}{2}$. *If the sequence* x_k *converges to a point* x^* *such that* $\nabla f(x^*) = 0$ *and* $\nabla^2 f(x^*)$ *is positive definite, and if the search direction satisfies*

$$\lim_{k \to \infty} \frac{\|\nabla f_k + \nabla^2 f_k p_k\|}{\|p_k\|} = 0, \tag{2.19}$$

then

1. *the step length* $\alpha_k = 1$ *is admissible for all* k *greater than a certain index* k_0*; and*
2. *if* $\alpha_k = 1$ *for all* $k > k_0$, x_k *converges to* x^* *superlinearly.*

In quasi-Newton methods, applied in this case for LLS but this also works generally, since p_k is the search direction of the form (2.2) then Equation 2.19 can be rewritten as

$$\lim_{k \to \infty} \frac{\|(\mathbf{B}_k - \nabla^2 f(x^*))p_k\|}{\|p_k\|} = 0, \tag{2.20}$$

In order to obtain a superlinear convergence speed is enough that $\mathbf{B}_k$ become increasingly accurate approximations to $\nabla^2 f(x^*)$ along the search directions p_k. Importantly, condition (2.20) is *both necessary and sufficient* for the superlinear convergence of quasi-Newton methods in convex cases thanks to Theorem 2.2 ([8], theorem 3.7).

Theorem 2.2 *Suppose that* $f : \mathbb{R}^n \to \mathbb{R}$ *is twice continuously differentiable. Consider the iteration* $x_{k+1} = x_k + p_k$ *(that is, the step length* α_k *is uniformly 1) and that* p_k *is given by* (2.2)*. Let us assume also that* x_k *converges to a point* x^* *such that* $\nabla f(x^*) = 0$ *and* $\nabla^2 f(x^*)$ *is positive definite. Then* x_k *converges superlinearly if and only if* (2.20) *holds.*

The main issue is that we can not safely assert (2.20) holds for our case, but making the assumption that the Hessian matrix $G(x) = \nabla^2 f(x)$ is Lipschitz continuous at x^*, that is

$$\|G(x) - G(x^*)\| \leq L\|x - x^*\| \tag{2.21}$$

for every x near x^* where L is a positive constant, then Theorem 2.3 (from [8], theorem 6.6) comes in help for us.

Theorem 2.3 *Suppose that* f *is twice continuously differentiable and that the iterates generated by the BFGS algorithm converge to a minimizer* x^* *at which* (2.21) *holds. Suppose also that* (2.18) *holds. Then* x_k *converges to* x^* *at a superlinear rate.*

In our case, the first assumption is verified

$$\|2\hat{X}^T\hat{X} - 2\hat{X}^T\hat{X}\| = 0 \leq L\|x - x^*\| \tag{2.22}$$

The second assumption holds as described in [8], section 6.4.

Both assumptions of Theorem 2.3 are satisfied, therefore we can state that L-BFGS converges superlinearly and, subsequently, thanks to Theorem 2.2, (2.20) holds.

Complexity The interesting part is the role that l assumes on the convergence rate of the method. Keeping in mind that the memory/time cost per iteration is $\mathcal{O}(ln)$ [8], l represents a trade-off in the convergence rate:

- with small l, the convergence tends to be like the gradient approach.
- with large l, the convergence tends to be like the standard Newton methods.

During our analysis in subsection 2.4 we will show the aforementioned trade-off by using different values for l.

2.4 Performance analysis

In this subsection we will show the results achieved by our implementation of the L-BFGS method by also showing empirically its superlinear convergence. The hyper-parameters considered are the following:

- $l \in \{5, 10, 15, 20\}$.
- $\lambda \in \{1e+4, 1e+2, 1, 1e-2, 1e-4\}$.
- the tolerance for $\|\nabla f\|$ was chosen to be $1e-14$.
- the maximum number of iterations was set to 1000.
- the starting point was chosen to be $w_0 = 0$, since we have seen empirically to be the best choice when $\hat{X}$ becomes ill-conditioned.

For a total of 20 configurations and each of one of them was run ten times in order to obtain the averages and the standard deviation, for a more reliable analysis.

The considered metrics are:

- Relative error $\frac{\|w-w^*\|}{\|w^*\|}$, where w^* is the solution obtained using the Matlab solver $\hat{X}\backslash\hat{y}$. w can be used in this case instead of $f(w)$ thanks to fact that the global optimum w^* is unique, see 1.2 for a deeper understanding.
- Relative residual, $\frac{\|\hat{X}w-\hat{y}\|}{\|\hat{y}\|}$.
- Execution time.
- Number of iterations.

For correctness, we will first show in Table 1 the results and then we will go deeper into to the other aspects by displaying the plots.

l	λ	Residual	Error	Iterations
5	10^4	$9.9 \times 10^{-1} \pm 5.74 \times 10^{-5}$	$3.48 \times 10^{-14} \pm 2.15 \times 10^{-14}$	4 ± 0
5	10^2	$8.68 \times 10^{-1} \pm 1.39 \times 10^{-2}$	$2.82 \times 10^{-14} \pm 1.64 \times 10^{-14}$	10 ± 0
5	1	$5.77 \times 10^{-2} \pm 5.80 \times 10^{-3}$	$3.24 \times 10^{-12} \pm 8.77 \times 10^{-13}$	28 ± 14
5	10^{-2}	$5.88 \times 10^{-4} \pm 6.02 \times 10^{-5}$	$5.36 \times 10^{-12} \pm 2.32 \times 10^{-12}$	29 ± 10
5	10^{-4}	$5.80 \times 10^{-6} \pm 1.65 \times 10^{-7}$	$6.30 \times 10^{-12} \pm 7.09 \times 10^{-13}$	26 ± 8
10	10^4	$9.99 \times 10^{-1} \pm 3.02 \times 10^{-4}$	$3.28 \times 10^{-14} \pm 2.69 \times 10^{-14}$	4 ± 0
10	10^2	$9.2 \times 10^{-1} \pm 9.37 \times 10^{-2}$	$4.76 \times 10^{-15} \pm 7.90 \times 10^{-16}$	10 ± 0
10	1	$4.64 \times 10^{-2} \pm 2.34 \times 10^{-3}$	$5.67 \times 10^{-14} \pm 7.72 \times 10^{-15}$	14 ± 0
10	10^{-2}	$5.11 \times 10^{-4} \pm 1.06 \times 10^{-4}$	$4.80 \times 10^{-14} \pm 1.57 \times 10^{-14}$	14 ± 0
10	10^{-4}	$5.85 \times 10^{-6} \pm 5.75 \times 10^{-7}$	$3.18 \times 10^{-14} \pm 2.92 \times 10^{-15}$	14 ± 0
15	10^4	$9.99 \times 10^{-1} \pm 3.60 \times 10^{-4}$	$3.27 \times 10^{-14} \pm 2.97 \times 10^{-14}$	4 ± 0
15	10^2	$9.07 \times 10^{-1} \pm 6.69 \times 10^{-2}$	$6.42 \times 10^{-14} \pm 4.89 \times 10^{-16}$	10 ± 0
15	1	$5.12 \times 10^{-2} \pm 6.35 \times 10^{-3}$	$2.41 \times 10^{-14} \pm 9.32 \times 10^{-15}$	13 ± 0
15	10^{-2}	$4.96 \times 10^{-4} \pm 8.81 \times 10^{-5}$	$4.28 \times 10^{-14} \pm 2.69 \times 10^{-14}$	13 ± 0
15	10^{-4}	$3.95 \times 10^{-6} \pm 2.17 \times 10^{-7}$	$3.26 \times 10^{-14} \pm 7.74 \times 10^{-15}$	13 ± 0
20	10^4	$9.99 \times 10^{-1} \pm 9.09 \times 10^{-4}$	$1.40 \times 10^{-14} \pm 2.63 \times 10^{-15}$	4 ± 0
20	10^2	$9.37 \times 10^{-1} \pm 2.69 \times 10^{-2}$	$5.62 \times 10^{-15} \pm 1.42 \times 10^{-15}$	10 ± 0
20	1	$5.89 \times 10^{-2} \pm 3.58 \times 10^{-3}$	$1.73 \times 10^{-14} \pm 8.87 \times 10^{-16}$	13 ± 0
20	10^{-2}	$5.02 \times 10^{-4} \pm 1.82 \times 10^{-5}$	$2.72 \times 10^{-14} \pm 3.60 \times 10^{-15}$	13 ± 0
20	10^{-4}	$6.08 \times 10^{-6} \pm 4.01 \times 10^{-7}$	$4.07 \times 10^{-14} \pm 3.04 \times 10^{-14}$	13 ± 0

Table 1: L-BFGS results

Analyzing the results in Table 1 it is evident that increasing the memory l will lower the number of iteration and will result in a smaller error, since having a bigger l means that we are able to approximate the Hessian better and, therefore, have a better descent direction.

We will talk about the effects of l and λ in 2.4.1 and 2.4.2 where we will show the convergence plots with our results. Furthermore, we will not plot the relative residual, but by looking at Table 1, we can see that it decreases when the error increases, this is typical when working with ill-conditioned matrices.

Before we dive deeper into the analysis, we have to recall that we showed the super-linear convergence of L-BFGS theoretically, we now aim to show it empirically through our plots. First, we need to mention that the error achieved by our method step by step is plotted alongside the linear and quadratic convergence rates, the former and the latter are defined as the following

$$linear_k = \{initErr, \frac{initErr}{2}, \frac{initErr}{4}, \frac{initErr}{8}, \frac{initErr}{16}, \dots, \frac{initErr}{2^k}\} \tag{2.23}$$

for the linear convergence rate and

$$quadratic_k = \{initErr, \frac{initErr}{4}, \frac{initErr}{16}, \frac{initErr}{256}, \frac{initErr}{65536}, \dots, \frac{initErr}{2^{2^k}}\} \tag{2.24}$$

for the quadratic one, $initErr$ is the initial error for both cases.

2.4.1 Analysis on l

As we know, the memory size l assumes an important role on the convergence rate, since for low values of l the method achieves an almost linear rate, while an high value emulates the behavior of the Newton methods. For showing this we opted to plot the results of our method considering a case in which $\lambda = 1$ as we can see in Figure 2.1.

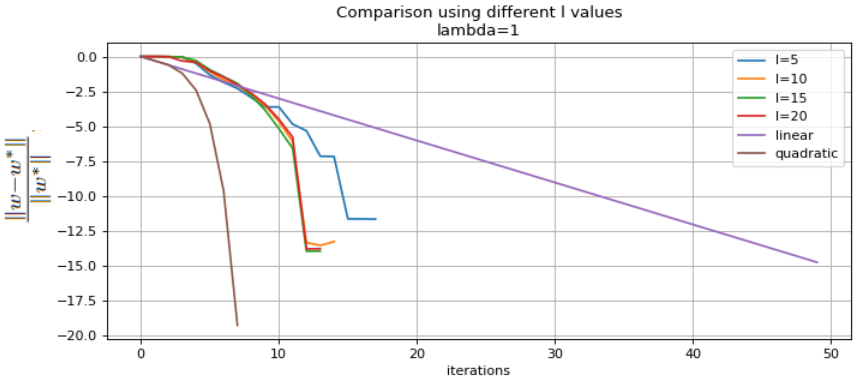


Figure 2.1: Convergence plot with $\lambda = 1$ at different values of l

The chosen l values were taken from the suggested ones by Nocedal [8], that is those inside the range $[3, 20]$ for keeping a good compromise between used memory and execution time, since we have already seen that l influences the number of iterations and, therefore, the execution time of the method. We can also notice from Table 1 that larger values of l implies a smaller error (more evident when working with ill-conditioned matrices), this comes at no surprise since, as we already know, the more memory we use the more the Hessian is accurately approximated.

It's easy to notice from Figure 2.1 that for a smaller l the convergence gets a little bit closer to the linear rate, meanwhile for bigger values it stays, more or less, in the middle, suggesting a super-linear convergence of the method. In our case the method was so fast that there were no relevant changes for $l > 5$, since the algorithm does not reach enough iterations to show the real effect of such hyper-parameter.

2.4.2 Analysis on λ

Analyzing the behavior at various values of λ is extremely interesting and fundamental, as it explains what we will see over the plots in this subsection and in the rest of this report. The results can be briefly explained by the following observations

- For larger values of λ ($1e4$ and $1e2$) the method converges in few steps and obtains excellent results;
- For small λ, the algorithm requires more steps and the results get worse.

Such behavior arises when our matrix is ill-conditioned, that is when λ is extremely small. This is caused by the fact that λ is the smallest singular value of $\hat{X}$ (see A.1 and A.2 for a better understanding) and, therefore, has a strong influence on the condition number of $\hat{X}$ making the method very unstable.

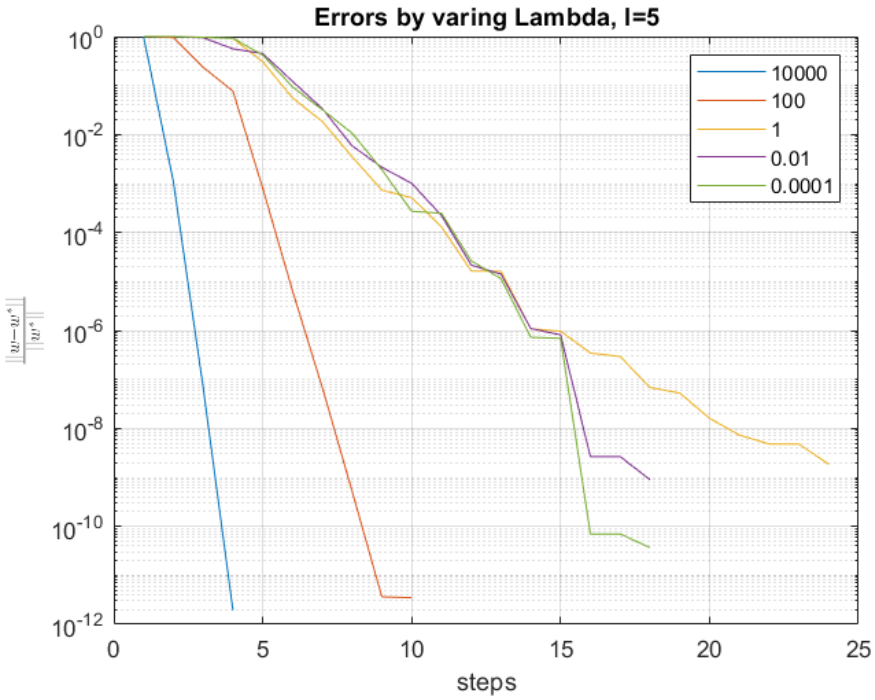


Figure 2.2: Errors by varying λ with $l = 5$.

As we can see from Figure 2.2, the greater λ is the better the algorithm converges even achieving a quadratic convergence rate (see Figure B.1 for a better view) and maintaining a super-linear one for small values, confirming our previous observations and what we have stated for the convergence of L-BFGS. More plots of the L-BFGS method can be found in B.1.

3 Thin QR factorization with Householder reflectors

Inside this section we will first introduce QR factorization and Householder reflectors, then we will talk about solving our task with the thin variant of the former, finally, just like we did for the L-BFGS method, we will analyze its performances.

3.1 Overview on thin QR factorization

For any matrix $A \in \mathbb{R}^{m \times n}$ there exist $Q \in \mathbb{R}^{m \times m}$ orthogonal and $R \in \mathbb{R}^{m \times n}$ upper triangular such that

$$A = QR \tag{3.1}$$

By diving into our case, we must introduce the thin variant of (3.1). Given that $\hat{X} \in \mathbb{R}^{m \times n}$, then R has the following structure

$$\begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & \dots & r_{1,n-1} & r_{1,n} \\ 0 & r_{2,2} & r_{2,3} & \dots & r_{2,n-1} & r_{2,n} \\ 0 & 0 & r_{3,3} & \dots & r_{3,n-1} & r_{3,n} \\ \vdots & & & & & \vdots \\ 0 & 0 & 0 & \dots & r_{n-1,n-1} & r_{n-1,n} \\ 0 & 0 & 0 & \dots & 0 & r_{n,n} \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & & & & & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

Hence, R can be split into two parts where the first one is $R_1 \in \mathbb{R}^{n \times n}$ and, following the same reasoning, Q is split into $Q_1 \in \mathbb{R}^{m \times n}$ and $Q_2 \in \mathbb{R}^{(m-n) \times n}$. Therefore, by looking at (3.1) and taking into account what we said just before, $\hat{X}$ can be factorized as following

$$\hat{X} = QR = \begin{bmatrix} Q1 & Q2 \end{bmatrix} \begin{bmatrix} R1 \\ 0 \end{bmatrix} = Q_1 R_1 \tag{3.2}$$

Before we discuss how to solve the LLS with the assigned method we need to introduce the Householder reflectors. The QR factorization, and its thin variant, can be obtained by storing the so called Householder reflectors instead of computing Q at each step, so let $x \in \mathbb{R}^n$ be any vector and let $u \in \mathbb{R}^n$ be the normalized Householder vector of x such that

$$u = \frac{v}{\|v\|} \tag{3.3a}$$

where

$$v = \begin{bmatrix} x_1 - \|x\| \\ x_2 \\ x3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \tag{3.3b}$$

The Householder vector defined in (3.3a) is computed by algorithm(3)

Algorithm 3: Householder vector

```
s = ‖x‖;
if x(1) ≥ 0 then
    s=-s;
v = x;
v(1) = v(1) - s;
u = v / ‖v‖;
return u, s;
```

The Householder reflector of u is then defined as

$$H_u = I - 2uu^T \tag{3.4}$$

Householder reflectors are essential for computing the QR factorization without needing to formulate Q. More formally, given a matrix $\hat{X} \in \mathbb{R}^{m \times n}$, the algorithm we have to use consists in building a list of Householder reflectors (as defined in (3.4)) that transform $\hat{X}$ into an un upper triangular matrix $R \in \mathbb{R}^{m \times n}$ and in storing their products as the orthogonal matrix $Q \in \mathbb{R}^{m \times m}$.

3.2 Solving LLS with thin QR

Starting from (3.2), we can write

$$\begin{aligned} \|\hat{X}w - \hat{y}\| &\overset{4}{=} \|Q^T(\hat{X}w - \hat{y})\| = \|Q^T\hat{X}w - Q^T\hat{y}\| = \|Q^TQRw - Q^T\hat{y}\| = \|Rw - Q^T\hat{y}\| \\ &= \|\begin{bmatrix} R_1 \\ 0 \end{bmatrix} w - \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}^T \hat{y}\| = \|\begin{bmatrix} R_1 w \\ 0 \end{bmatrix} - \begin{bmatrix} Q_1^T\hat{y} \\ Q_2^T\hat{y} \end{bmatrix}^T\| = \|\begin{bmatrix} R_1 w - Q_1^T\hat{y} \\ Q_2^T\hat{y} \end{bmatrix}\| \end{aligned} \tag{3.5}$$

where R_1 and Q_1 are the matrices as seen in (3.2) and $Q_2^T\hat{y}$ is the residual of the norm, that is the optimum of our optimization problem.

Moreover, from the theory is known that if a matrix is factorized as in (3.2) and has full column rank, then the solution of LLS is given by

$$w = R_1^{-1}Q_1^T\hat{y} \tag{3.6a}$$

by stating that

$$c = Q_1^T\hat{y} \tag{3.6b}$$

then (3.6a) can be rewritten as

$$R_1 w = c \tag{3.6c}$$

[4]Holds since $\|Qx\| = \|x\|$ when Q is orthogonal.

Therefore, solving the LLS problem with QR factorization is the same as solving the linear system (3.6c), which is an extremely easy task once R_1 and Q_1 are computed. We now show algorithm 4 that computes the thin QR factorization, without storing the Householder matrices and keeping in memory only the $m \times n$ strictly needed elements of the Q_1 matrix at each step. We will not show the part that solves the upper triangular system as defined by (3.6c) since it is not relevant to the aim of this section.

Algorithm 4: Compute thin QR factorization

1 Given $A \in R^{m \times n}$
2 $Q_1 y = y$
3 **for** $j = 1$ *to* $min(m-1, n)$ **do**
4 $\quad [u, s] = householder\_vector(A[j : end, j]);$ ▷ **algorithm 3**
5 $\quad A[j, j] = s;$
6 $\quad A[j+1 : end, j] = 0;$
7 $\quad A[j : end, j+1 : end] = A[j : end, j+1 : end] - 2u * (u^T * A[j : end, j+1 : end]);$
8 $\quad Q_1 y[j : end] = Q_1 y[j : end] - 2u * (u^T * Q_1 y[j : end]);$
9 $R_1 = A[1 : n, :];$
10 $Q_1 y = Q_1 y[1 : n];$
11 **return** $Q_1 y, R_1;$

As you may notice our implementation is not a strict thin qr factorization since we do not even store $Q1$, instead at each step we compute the product $Q * \hat{y}$ considering only the affected rows starting from the j-th one. Both $Q * \hat{y}$ and R are computed in the main loop from algorithm 4, keep in mind that $Q_1\hat{y}$ and R_1 are taken from the first n rows of $Q * \hat{y}$ and the starting matrix respectively at the end of the computation.

$Q * \hat{y}$ is computed as

$$Q\hat{y} = H_n \dots H_1 \hat{y} \tag{3.7}$$

By following the definition of Householder reflector from (3.4), we can see (3.7) as

$$Q\hat{y} = (I - 2u_n u_n^T) \dots (I - 2u_1 u_1^T)\hat{y} \tag{3.8}$$

For a single iteration, (3.8) can be rewritten in a more easier way

$$Q_i\hat{y} = (I - 2u_i u_i^T)Q_i\hat{y} = Q_i\hat{y} - 2u_i(u_i^T Q_i\hat{y}) \tag{3.9}$$

The same reasoning can be applied to R

$$R = H_n \dots H_1 \hat{X} \tag{3.10}$$

Without explaining again the same process seen in (3.8), we can see a single iteration for computing R as

$$R_i = (I - 2u_i u_i^T)R_i = R_i - 2u_i(u_i^T R_i) \tag{3.11}$$

(3.9) and (3.11) are dimensionally feasible since we operate on submatrices of $Q * y$ and R as we can see from algorithm 4 (row 7 and 8). To give a reason on why this is more efficient, for R this means doing a product between matrices of size

$$\mathbb{R}^{(m-i)\times 1} \times (\mathbb{R}^{1\times(m-i)} \times \mathbb{R}^{(m-i)\times(n-i)}) \tag{3.12a}$$

instead of doing a product between two matrices of size

$$\mathbb{R}^{(m-i)\times(m-i)} \times \mathbb{R}^{(m-i)\times(n-i)} \tag{3.12b}$$

By following (3.12a) we avoid doing any product between matrices improving the execution time of our implementation, since such product require one more order of magnitude with respect to products between vectors, hence our choice.

The same reasoning, of course, can be applied to $Q * \hat{y}$ where we compute products of matrices of size

$$\mathbb{R}^{(m-i)\times 1} \times (\mathbb{R}^{1\times(m-i)} \times \mathbb{R}^{(m-i)\times 1}) \tag{3.13}$$

In this way we do not need to compute Q_1 and then multiplying it by $\hat{y}$, instead their product is computed step by step inside the main loop of algorithm 4.

Complexity Analyzing the computational cost of Equation 3.6, we notice that the vector-matrix multiplication $Q_1^T \hat{y}$ requires $\mathcal{O}(mn)$ operations, plus the time to solve the triangular system $R_1 w = Q_1^T \hat{y}$, that is $\mathcal{O}(n^2)$. Anyway, the time spent to compute this part is lower than the one required to compute the matrices Q_1 and R_1, which is $\mathcal{O}(mn^2)$.

3.3 Backward stability

We know aim to show the backward stability of the QR factorization (the same reasoning applies to the thin version), by first recalling the upper bound on the relative error for the LLS as we can see from (3.14)

$$\frac{\|\tilde{w} - w\|}{\|w\|} \leq u \tag{3.14}$$

where w is any number and the actual solution to $\hat{X}w - \hat{y} = 0$, $\tilde{w}$ is the perturbed solution of such problem and u is the so called machine precision $2^{-52} = 10^{-16}$.

Assume that an algorithm is used to find $\hat{y} = f(w)$, we know that due to the machine precision (which is an hardware limitation) we will not get such perfect results, instead we will obtain a perturbed value $\tilde{y} = f(\tilde{w})$ that, in turn, will define

$$\Delta_{\hat{y}} = \tilde{y} - \hat{y} \tag{3.15a}$$

as the forward error and

$$\Delta_w = \tilde{w} - w \tag{3.15b}$$

as the backward error.

Recall also that an algorithm is called backward stable if the relative backward error from (3.15b) is stable, formally

$$\frac{\|\tilde{w} - w\|}{\|w\|} = \mathcal{O}(u) \tag{3.16}$$

We first notice that doing a product with an orthogonal matrix Q is backward stable, since the product $Q \circ A = Q * A + E$ (given any matrix $A^{m \times n}$) has forward error E whose norm is

$$\|E\| \leq \mathcal{O}(u)\|Q\|\|A\| = \mathcal{O}(u)\|A\| \tag{3.17}$$

moreover $Q \circ A = Q(A + Q^{-1}E)$ defines the backward error $\Delta_A = Q^{-1}E$ whose norm is

$$\|\Delta_A\| \leq \|Q^{-1}\|\|E\| = \mathcal{O}(u)\|A\| \tag{3.18}$$

With (3.17) and (3.18) it's easy to show that the QR factorization is backward stable by perturbing A with Δ_A such that $\tilde{Q}\tilde{R} = qr(A + \Delta_A)$. Recall that QR factorization is made up of steps as follows

$$Q_k A_{k-1} = A_k \tag{3.19}$$

considering machine precision u, each step starts with $\tilde{A}_{k-1}$ (which contains even the previous errors) and produces $\tilde{A}_k$, each single step is backward stable since the computed $\tilde{A}_k$ and $\tilde{u}_k$ satisfy the perturbed version of (3.19)

$$\tilde{Q}_k(\tilde{A}_{k-1} + \Delta_{k-1}) = \tilde{A}_k \tag{3.20}$$

with the norm of the accumulated error at step $k-1$ as

$$\|\Delta_{k-1}\| \leq \mathcal{O}(u)\|\tilde{A}_{k-1}\| = \mathcal{O}(u)\|A\| \tag{3.21}$$

Then, if we consider n steps we can retrieve the exact result

$$\tilde{R} = A + \Delta_0 + \tilde{Q}_1^T\Delta_1 + \tilde{Q}_1^T\tilde{Q}_2^T\Delta_2 + \cdots + \tilde{Q}_1^T\tilde{Q}_2^T \ldots \tilde{Q}_{n-1}^T\Delta_{n-1} \tag{3.22}$$

where each of the error terms is what we have seen in (3.21), by always keeping in mind that every computation is orthogonal.

Therefore, the QR factorization is backward stable and the computed Q and R are the exact results of $qr(A + \Delta)$ with $\|\Delta\| \leq \mathcal{O}(u)\|A\|$.

Moreover, while going back to our case, backward stable algorithms usually produces small residuals $r = \hat{X}w - \hat{y}$, but this does not imply that $\tilde{w}$ is close to the actual solution w as we have seen in subsection 2.4. Even then is still possible to find an upper bound to such error as theorem 3.1 states.

Theorem 3.1 *Let $\hat{X} = Q_1R_1$ be a thin QR factorization and let $r_1 = Q_1^T(\hat{X}\tilde{w} - \hat{y})$. Then, it can be proven that $\tilde{w}$ is the exact solution of the least square problem*

$$min\|\hat{X}w - (\hat{y} + Q_1r_1)\| \tag{3.23}$$

thus

$$\frac{\|\tilde{w} - w\|}{\|w\|} \leq \kappa_{rel}(LS, \hat{y})\frac{\|Q_1r_1\|}{\|\hat{y}\|} = \kappa_{rel}(LS, \hat{y})\frac{\|r_1\|}{\|\hat{y}\|} \tag{3.24}$$

(3.24) states that the relative error of the LLS problem solved with thin QR factorization is bounded by the condition number relative to such problem and by the ratio between the residual and the expected values.

3.4 Performance analysis

We have two different purposes for the thin QR factorization algorithm:

1. we have to show that our implementation scales linearly with respect to the number of rows of the input matrix.
2. we need to show the results for varying values of λ.

3.4.1 Linearity of thin QR with respect to the rows

The project requires us to implement a thin QR factorization algorithm that scales linearly with respect to the biggest dimension of the number of rows of a matrix, in order to prove this behavior, we created 20 $m \times n$ random matrices, with n fixed and equal to 200, whereas m spaces from 1000 to 5750 with a step size equal to 250. For each configuration, we ran our thin QR factorization algorithm 10 times, measuring the average completion time and then we plotted the results as shown in Figure 3.1.

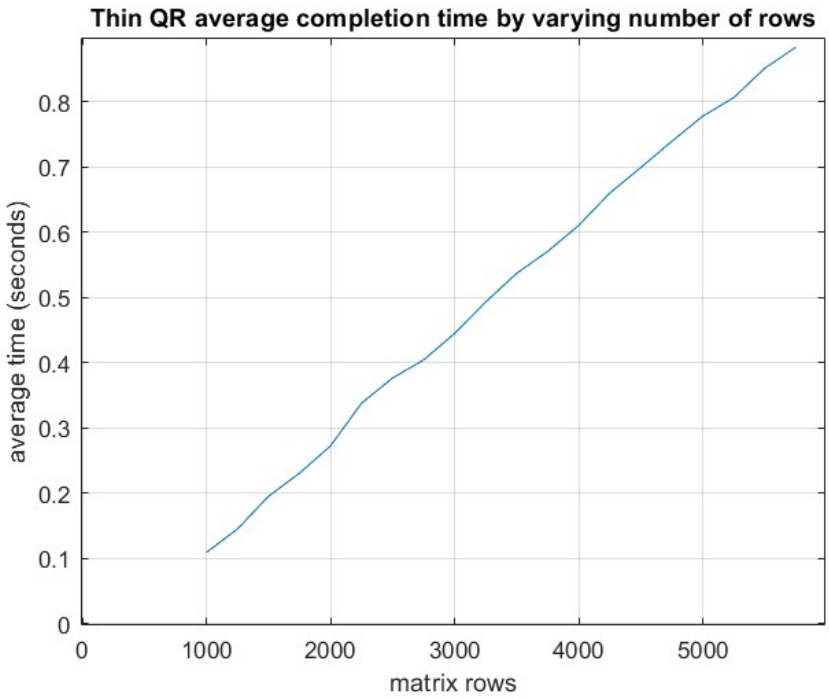


Figure 3.1: Thin QR average completion time by varying number of rows.

As Figure 3.1 shows, our implementation of thin QR scales linearly with the number of rows which is expected as we know that its complexity is $\mathcal{O}(mn^2)$ from the last paragraph of subsection 3.2.

3.4.2 Accuracy of thin QR

We are also interested in the accuracy of our thin QR implementation to know how it factorizes our matrix $\hat{X}$ when $\kappa(\hat{X})$ becomes high (we already know this is caused by λ as talked in subsection A.2), such metric is defined as

$$\frac{\|\hat{X} - Q_1 R_1\|}{\|\hat{X}\|} \tag{3.25}$$

where Q_1 and R_1 are the matrices obtained by our thin qr factorization.

In short, we want to know how accurately our thin QR rebuilds the starting matrix. For such purpose, Table 2 shows the accuracies achieved by our method compared to the ones from the matlab implementation (called "economical thin QR") and the distance between them.

λ	Our accuracy	Matlab accuracy	Δ accuracies
10^4	1.865168×10^{-15}	3.088531×10^{-15}	1.223363×10^{-15}
10^2	9.94962×10^{-16}	1.602198×10^{-15}	6.072354×10^{-16}
1	7.463726×10^{-16}	7.450506×10^{-16}	1.321951×10^{-18}
10^{-2}	7.908642×10^{-16}	7.630684×10^{-16}	2.779588×10^{-17}
10^{-4}	7.542911×10^{-16}	7.424282×10^{-16}	1.186286×10^{-17}

Table 2: Thin QR accuracy.

From Table 2 we can see that our accuracy is close to $\mathcal{O}(u) = 1e-16$ meaning that our implementations is, indeed, correct, as expected from a backward stable algorithm. Moreover our accuracy is extremely close to the one achieved by matlab, giving us a nice confident boost for the next steps in this performance analysis.

3.4.3 Results

Since the only hyper-parameter for the thin QR is λ we will show how the results change when the latter assumes different values, moreover we will not show the standard deviation for each configuration since it's very close to 0 given the fact that thin QR is not iterative.

The considered values for λ are the same seen in subsection 2.4, but this time we consider one more metric which is the gradient of the function $\nabla f(w)$ in w, since we did not use it as stop condition. As for the L-BFGS, using the relative error on w is fine given the fact that w^* is unique, thanks to what we have seen in 1.2.

We also consider the upper bound from (3.24) since it gives us a nice information about the maximum relative error we can achieve by computing the solution using non-exact arithmetic. Moreover, (3.24) can be rewritten as

$$\frac{\|w - w^*\|}{\|w^*\|} \leq \kappa_{rel}(LS, \hat{y}) \frac{\|r_1\|}{\|\hat{y}\|} \overset{5}{=} \frac{\kappa(\hat{X})}{\|\hat{X}w\|} \|r_1\| \tag{3.26}$$

We can now show the results achieved by our implementation in Table 3 before doing a brief explanation about them.

λ	$\frac{\|w-w^*\|}{\|w^*\|}$	$\frac{\|\hat{X}w-y\|}{\|y\|}$	$\|\nabla f(w)\|$	$\frac{\kappa(\hat{X})}{\|\hat{X}w\|}\|r_1\|$ (3.26)
10^4	7.3825×10^{-14}	9.9959×10^{-1}	1.8069×10^{-10}	1.0512×10^{-13}
10^2	1.5650×10^{-14}	9.1160×10^{-1}	6.0329×10^{-12}	2.7837×10^{-14}
1	2.0354×10^{-14}	5.3405×10^{-2}	2.7046×10^{-12}	1.1681×10^{-12}
10^{-2}	9.0120×10^{-14}	5.4414×10^{-4}	4.6584×10^{-12}	2.0006×10^{-10}
10^{-4}	8.1724×10^{-14}	5.4071×10^{-6}	6.1100×10^{-12}	1.0513×10^{-8}

Table 3: Thin QR results. w^* is the Matlab solution from $\hat{X}\backslash\hat{y}$.

The execution time is more or less $9.8s$ for each configuration so we decided not to report it. By considering the upper bound from (3.26), we can notice how it decreases as λ increases since $\kappa(\hat{X})$ starts to get bigger and bigger. Anyway, the bound always holds for our implementation, even for small values of λ, as we expected from the theory.

Furthermore, we noticed that the residuals follow the same behavior as we have seen for the L-BFGS in subsection 2.4, that is the decreasing of the relative residual as the value of λ decreases.

[5]recall that $\kappa_{rel}(LS, \hat{y}) = \frac{\kappa(\hat{X})}{\cos\theta}$, where $\cos\theta = \frac{\|\hat{X}w\|}{\|\hat{y}\|}$.

4 Conjugate gradient

In this section we will talk about the conjugate gradient method in the same way we did for L-BFGS and thin QR, by first introducing the algorithm itself and then by analyzing its complexity and the performances obtained by applying to our case.

4.1 Overview on conjugate gradient

The conjugate gradient is an iterative algorithm for solving linear systems of equations

$$Ax = b \tag{4.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a symmetric positive definite matrix [6] and $x, b \in \mathbb{R}^n$, where x is called as the unique solution. Since A is symmetric positive definite, then (4.1) can be stated as a minimization problem (a strictly convex one to be more precise)

$$min\ \phi(x) \stackrel{def}{=} \frac{1}{2} x^T A x - b^T x \tag{4.2a}$$

$$\nabla \phi(x) = Ax - b \stackrel{def}{=} r(x) \tag{4.2b}$$

In short, minimizing (4.2a) means that we need to solve either (4.2b) (with $r(x) = 0$) or (4.1).

4.1.1 Conjugate direction methods

As the name suggests the conjugate gradient method exploits a property called *conjugacy* to generate a set of vectors. A set of non-zero vectors $\{p_0, p_1 \dots p_l\}$ is known as *conjugate* of a symmetric definite positive matrix A if

$$p_i^T A p_j = 0, \quad \forall i \neq j \tag{4.3}$$

(4.3) is vital to minimize (4.2a) in, at maximum, n steps, but first we need to consider the *conjugate direction* method. Given a starting point $x_0 \in \mathbb{R}^n$ a a set of conjugate directions $\{p_0, p_1 \dots p_{n-1}\}$, the sequence $\{x_k\}$ is generated step by step by

$$x_{k+1} = x_k + \alpha_k p_k \tag{4.4}$$

(4.4) seems identical to what we have seen for the L-BFGS in (2.3) and also the step size α will be similar, which is now defined as

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T A p_k} \tag{4.5}$$

We now have enough tools for theorem 4.1 (theorem 5.1 from [8], the proof can also be found there).

[6] $X = X^T$ and $zAz^T > 0\ \forall z \neq 0 \in \mathbb{R}^n$

Theorem 4.1 *For any $x_0 \in \mathbb{R}^n$ the sequence $\{x_k\}$ generated by the conjugate direction algorithm* (4.4)*, then* (4.5) *converges to the solution x^* of the linear system* (4.1) *in at most n steps.*

Theorem 4.1 tell us some properties of conjugate directions:

1. If A is diagonal, then the contours of (4.2a) are ellipses whose axes are aligned with the coordinate directions, therefore, we can find the minimum by performing one-dimensional minimizations along the coordinate directions $e_1, e_2, \dots e_n$. If A is not diagonal we need to precondition it in order to obtain the same behaviour.

2. When the Hessian is diagonal, each coordinate minimization determines one of the components of the solution x^*.

The second property holds even when the Hessian is not diagonal and this is proven by theorem 4.2 (theorem 5.2 and proof from [8]).

Theorem 4.2 *Let $x_0 \in \mathbb{R}^n$ be any starting point and suppose that the sequence $\{xk\}$ is generated by the conjugate direction algorithm* (4.4)*,* (4.5)*. Then*

$$r_k^T p_i = 0 \quad for\ i = 0, ...k-1 \tag{4.6}$$

and x_k is the minimizer of (4.2a) *over the set*

$$\{x | x = x_0 + span\{p_0, p_1, \dots, p_{k-1}\}\} \tag{4.7}$$

4.1.2 Conjugate gradient method

As the name suggest, the conjugate gradient method is a conjugate direction method with the very nice fact that to compute a new vector p_k based only on p_{k-1}, hence it does not need the entire conjugate set. This implies that the method requires little storage and computations.

In the conjugate gradient method, each direction p_k is chosen to be a linear combination of the negative residual $-rk$ (which, by (4.2b) is the steepest direction for (4.2a)) and the previous direction p_{k-1}, formally

$$p_k = -r_k + \beta_k p_{k-1} \tag{4.8}$$

where β_k is a value determined by the requirement that p_k and p_{k_1} must be conjugate with respect to A, moreover, by doing the product of (4.8) with $p_{k-1}^T A$ and imposing the condition that $p_{k-1}^T A p_k = 0$, we find that

$$\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}} \tag{4.9}$$

We need to show that the directions $p_0, p_1, \dots, p_{n-1}$ are indeed conjugate, which by theorem 4.1 implies that the method terminates in n steps. Theorem 4.3 (theorem 5.3 from [8]) establishes this property and two other important properties:

1. The residuals r_i are mutually orthogonal.

2. Each search direction p_k and residual r_k is contained in the *Krylov subspace of degree k for* r_0, defined as
$$K(r_0, k) \overset{def}{\approx} span\{r_0, Ar_0, \ldots, A^k r_0\} \tag{4.10}$$

Theorem 4.3 *Suppose that the k-th iterate generated by the conjugate gradient method is not the solution point* x^**. The following four properties hold*

$$r_k^T r_i = 0, \quad for\ i = 0, \ldots, k-1 \tag{4.11a}$$

$$span\{r_0, r_1, \ldots, r_k\} = span\{r_0, Ar_0, \ldots, A^k r_0\} \tag{4.11b}$$

$$span\{p_0, p_1, \ldots, p_k\} = span\{r_0, Ar_0, \ldots, A^k r_0\} \tag{4.11c}$$

$$p_k^T A p_i = 0, \quad for\ i = 0, \ldots, k-1 \tag{4.11d}$$

The proof of theorem 4.3 can be found after theorem 5.3 from [8]. As last thing for this part, we can rewrite (4.5) by using (4.8) and (4.6) to obtain

$$\alpha_k = -\frac{r_k^T p_k}{p_k^T A p_k} = -\frac{r_k^T(-r_k + \beta_k p_{k-1})}{p_k^T A p_k} = \frac{r_k^T r_k}{p_k^T A p_k} \tag{4.12}$$

Moreover, noting the relation between (4.4), and the k-th residual $r_k = Ax_k - b$, we can write

$$r_{k+1} = rk + \alpha_k A p_k \tag{4.13}$$

We can also rewrite (4.9) based, again, on (4.8), (4.6) and (4.13)

$$\beta_k = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}} = \frac{r_k^T \frac{r_k - r_{k-1}}{\alpha_{k-1}}}{p_{k-1}^T \frac{r_k - r_{k-1}}{\alpha_{k-1}}} = \frac{r_k^T(r_k - r_{k-1})}{p_{k-1}^T(r_k - r_{k-1})} = -\frac{r_k^T r_k}{r_{k-1}^T p_{k-1}} = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}} \tag{4.14}$$

We now have everything needed to build the conjugate gradient algorithm and to apply to our case.

4.2 Solving LLS with conjugate gradient

Diving into our lls problem we first notice that $\hat{X}$ is not symmetric positive definite, hence we can not apply the conjugate gradient method in a straightforward way. But, as we have seen in subsection 1.1, the solution to the lls problem is obtained by solving (1.4) and, luckily for us, $\hat{X}^T\hat{X}$ is symmetric positive definite (this is also the Hessian of the lls which we have shown to be positive definite in subsection 1.2), so we can apply that to our case.

The optimal solution w_*, therefore, can be found in the following way

$$w_*^T \hat{X}^T \hat{X} = \hat{y}^T \hat{X} \iff \hat{X}^T \hat{X} w_* = \hat{X}^T \hat{y} \tag{4.15}$$

$\hat{X}^T \hat{X} \in \mathbb{R}^{n \times n}$ has full column rank, therefore is symmetric positive definite and we can now solve the lls with the conjugate gradient, but, before we proceed further, we should rewrite (4.15) in a more readable way

$$\hat{X}^T \hat{X} w_* = \hat{X}^T \hat{y} \iff A w_* = b \tag{4.16}$$

From now on we will refer to the right side of (4.16) over the next sections. We now have everything needed to show the conjugate gradient method in algorithm 5

Algorithm 5: Conjugate gradient

Given $x_0 \in R^m$;
$r_0 = Ax_0 - b$;
$p_0 = -r_0$;
$k = 0$;
while $r_k \neq 0$ **do**
    $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$;
    $x_{k+1} = x_k + \alpha_k p_k$;
    $r_{k+1} = r_k + \alpha_k A p_k$;
    $\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$;
    $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$;
    $k = k + 1$;
return x_{k+1};

Of course, we can improve algorithm 5 by computing Ap_k, $r_k^T r_k$ before α_k and β_{k+1} in order to obtain a better execution time.

Moreover, in order to apply the conjugate gradient algorithm to solve the lls problem, we need to compute the matrix product $A = \hat{X}^T \hat{X}$, as we can see from (4.16). Therefore, the resulting matrix has a conditioning number $\kappa(A) = \kappa(\hat{X})^2$ and this implies that A is extremely ill-conditioned when λ is small (recall the conditioning number of our matrices $\hat{X}$ from Table 9). This issue can be mitigated by avoiding to form the matrix A explicitly, because it is less sparse than $\hat{X}$.

By looking at algorithm 5, we can compute r_0 by first finding the product $\hat{X}x_0$, and then $\hat{X}^T \hat{X} x_0$. Also, the update of α_k should be modified as following

$$\alpha_k = \frac{r_k^T r_k}{(\hat{X} p_k)^T \hat{X} p_k} \tag{4.17}$$

knowing that the inner product $\langle p, \hat{X}^T \hat{X} p \rangle = \langle \hat{X} p, \hat{X} p \rangle$.

Finally, we modify the update of r_k with

$$r_{k+1} = r_k + \alpha_k \hat{X}^T (\hat{X} p_k) \tag{4.18}$$

This normal equations approach was suggested by Björck and Elfving [2], and it allowed us to reach a solution that is an order of magnitude better than our original approach.

4.3 Convergence and complexity

From Theorem 4.1 we know that conjugate gradient converges at the solution in at most n steps, this can be improved when the distribution of the eigenvalues of A has certain features. First, we can notice that at each step of the conjugate gradient method, the error term $e_{k+1} = w_{k+1} - w^*$ can be rewritten as

$$e_{k+1} = e_0 + span\{r_0, Ar_0, A^2 r_0, \ldots, A^k r_0\} = e_0 + span\{Ae_0, A^2 e_0, \ldots, A^k e_0\} \tag{4.19}$$

then by considering a polynomial $P_k(A)$ of degree k, we can state that the error term assumes the following form

$$e_{k+1} = w_{k+1} - w^* = P_k(A)(w_0 - w^*) = P_k(A)e_0 = \left(I + \sum_{i=1}^{k} \gamma_i A^i \right) e_0 \tag{4.20}$$

for some constants γ_i related to β_i and α_i (this fact is not important for the pursuance of this section). What is interesting to us is the fact that the algorithm chooses each γ_i in order to minimize

$$\|e_{k+1}\|_A^2 = e_{k+1}^T A e_{k+1} = (w_{k+1} - w^*)^T A (w_{k+1} - w^*) \tag{4.21}$$

Moving now forward on our analysis, let $0 < \theta_1 \leq \cdots \leq \theta_m$ be the eigenvalues of A and let $v_1, \ldots, v_n$ be the corresponding orthonormal eigenvectors, such that

$$A = \sum_{i=1}^{m} \theta_i v_i v_i^T \tag{4.22}$$

then, we can rewrite e_0 as

$$e_0 = w_0 - w^* = \sum_{i=1}^{n} \xi_i v_i \tag{4.23}$$

for some coefficient ξ_i. Moreover, P_k can take either a scalar or a matrix as argument and the results will be the same, so it can be shown that

$$P_k(A)v_i = P_k(\theta_i)v_i \quad \forall i = 1, \ldots, m \tag{4.24}$$

Therefore, by plugging (4.23) into (4.20) we obtain that

$$e_{k+1} = w_{k+1} - w^* = \sum_{i=1}^{n} \xi_i P_k(\theta_i) v_i \tag{4.25}$$

By looking at (4.21), we can also rewrite (4.25) as

$$\|e_{k+1}\|_A^2 = \|w_{k+1} - w^*\|_A^2 = \sum_{i=1}^{n} \xi_i^2 [P_k(\theta_i)]^2 \theta_i \tag{4.26}$$

Then, by extracting the largest of the terms $[P_k(\theta_i)]^2$ we retrieve that

$$\|w_{k+1} - w^*\|_A^2 \leq \min_{P_k} \max_{1 \leq i \leq n} [P_k(\theta_i)]^2 (\sum_{j}^{n} \theta_j \xi_j^2) = \min_{P_k} \max_{1 \leq i \leq n} [P_k(\theta_i)]^2 \|w_0 - w^*\|_A^2 \tag{4.27}$$

(4.27) gives us an upper bound, allowing us to estimate the convergence rate of the method by estimating

$$\min_{P_k} \max_{1 \leq i \leq n} [P_k(\theta_i)]^2 \tag{4.28}$$

In short, we need to look for a P_k that minimizes (4.28).

Following what we have seen in this section until now, we can introduce Theorem 4.4 and Theorem 4.5 (theorems 5.4 and 5.5 from [8], their proofs are also there).

Theorem 4.4 *If A has only r distinct eigenvalues, then the CG iteration will terminate at the solution in at most r iterations.*

Following the same reasoning, Leunberger derived an estimate

Theorem 4.5 *If A has eigenvalues $\theta_1 \leq \cdots \leq \theta_n$, we have that*

$$\|w_{k+1} - w^*\|_A^2 \leq \left(\frac{\theta_{n-k} - \theta_1}{\theta_{n-k} + \theta_1} \right)^2 \|w_0 - w^*\|_A^2 \tag{4.29}$$

By defining $\epsilon = \theta_{n-j} - \theta_1$, Theorem 4.5 tells us that after $j+1$ steps of applying the method, we obtain

$$\|e_j\|_A \approx \epsilon \|e_0\|_A \tag{4.30}$$

Another last thing is the fact that ϵ from (4.30) can be rewritten in terms of $\kappa(A) = \frac{\theta_m}{\theta_1}$, giving us the following

$$\|e_j\|_A = \|w_j - w^*\|_A \approx 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^j \|e_0\|_A \tag{4.31}$$

The bound from (4.31) is a large overestimation of the error, but it could be useful when the only information we know about A are its maximum and minimum eigenvalues.

Complexity The most expensive operations inside the conjugate gradient algorithm are the matrix-vector products that, in general, require $\mathcal{O}(z)$ operations, where z is the number of non-zero entries of the matrix. If the matrix $A \in \mathbb{R}^{n \times n}$ is sparse, then $z \in \mathcal{O}(n)$. This holds also for our case, in fact the $\hat{X}$ matrix is sparse and almost square: the non-zeros entries are only in the diagonal of λI and in the first 12 rows as we can see from Figure 4.1.

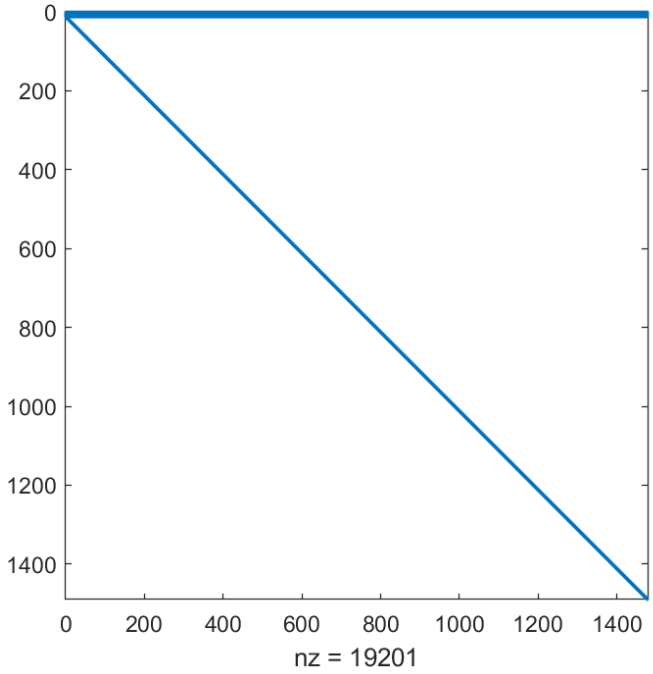


Figure 4.1: Sparsity of $\hat{X}$, nz is the number of non-zero entries.

Now, suppose we wish to perform enough iterations to reduce the norm of the error by a factor of ϵ, that is $\|w_i - w^*\|_A \leq \epsilon \|w_i - w^*\|_A$, then (4.31) comes at our help suggesting that the maximum number of iterations required from this algorithm is

$$i \leq \left\lceil \frac{1}{2}\sqrt{\kappa} \log\left(\frac{2}{\epsilon}\right) \right\rceil \tag{4.32}$$

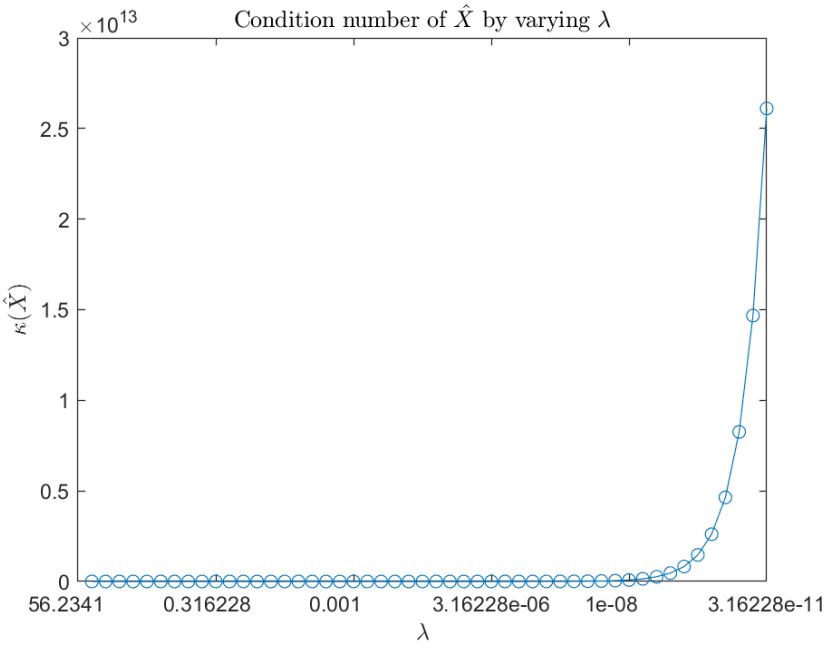
and we can conclude that the conjugate gradient has a time complexity of $\mathcal{O}(z\sqrt{\kappa})$. This analysis was inspired by the one from Shewchuk et al. [10].

4.4 Performance analysis

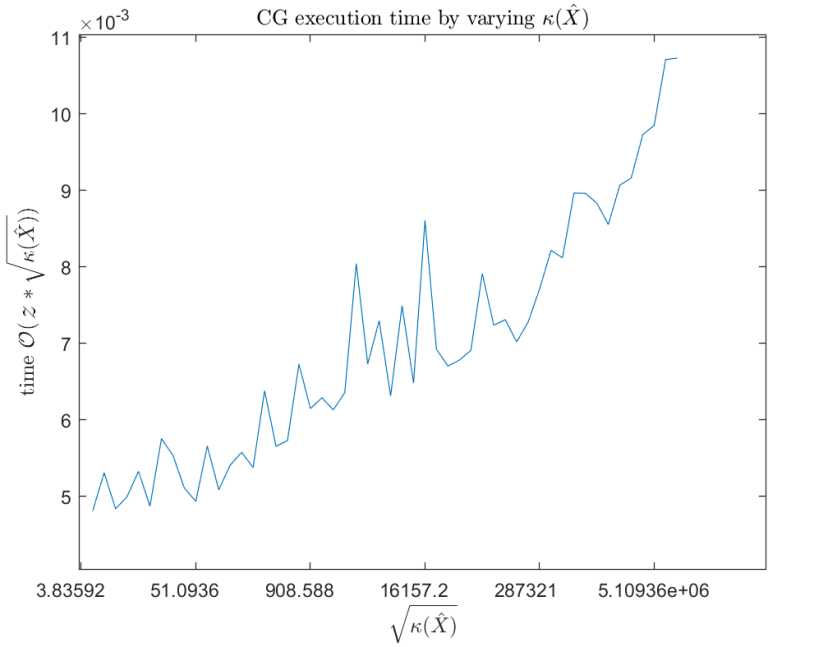
In this section we will take into account the condition number of the matrix $\hat{X}$ rather than the one of $A = \hat{X}^T\hat{X}$ because the CG algorithm (5) does not explicitly compute it.

4.4.1 Empirical observation on the execution time

We know that the algorithm runs in $\mathcal{O}(z\sqrt{\kappa(\hat{X})})$, so we might expect a linear behavior till $\sqrt{\kappa(\hat{X})} < z$. From this point on, the complexity only depends on $\sqrt{\kappa(\hat{X})}$. Unfortunately, this quantity grows exponentially and this implies that when it exceeds the amount of non-zero values of $\hat{X}$, the execution time of the algorithm starts to grow exponentially, as shown in Figure 4.2.



(a) Condition number of $\hat{X}$ by varying λ.



(b) Execution time of CG algorithm by varying $\kappa(\hat{X})$.

Figure 4.2: Side by side comparison between $\kappa(\hat{X})$ and the execution time of the CG algorithm.

4.4.2 Superlinear convergence ratio

By plotting the relative errors $\frac{\|w-w^*\|}{\|w^*\|}$ we observed that the conjugate gradient algorithm has a super-linear convergence ratio, this is also confirmed by Beckermann et al. [1]. We decided to show how the conjugate gradient convergence rate changes as λ assumes different values, just as we did for the others methods.

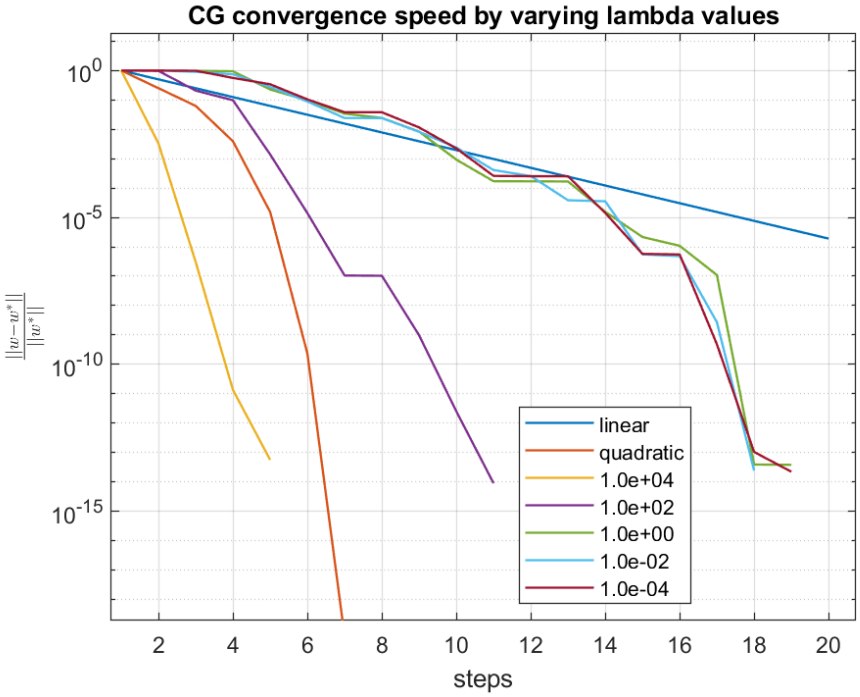


Figure 4.3: CG convergence speed by varying lambdas values.

As we can see from Figure 4.3, the method reaches a worse convergence ratio as λ decreases, like we have seen for L-BFGS.

4.4.3 Results

The number of the distinct eigenvalues of $A = \hat{X}^T\hat{X}$ is 13, as stated in A.3, so we expect to find a solution at most in 13 steps if we were in exact arithmetic as stated by Theorem 4.4. As shown in Table 4, CG algorithm found a solution requiring a less number of steps with respect to the theoretical maximum until the matrix $\hat{X}$ is well-conditioned. When $\kappa(\hat{X})$ increases, the algorithm took some more steps to converge. Furthermore, by storing the matrix $\hat{X}$ as sparse, the algorithm runs two order of magnitude faster, doing the same number of iterations. The solution has a relative error of 10^{-14} with respect to the Matlab solution $w^* = \hat{X}\backslash\hat{y}$, moreover, we need to point out, again, that using the relative error on w is doable since w^* is unique.

λ	$\frac{\|w-w^*\|}{\|w^*\|}$	$\frac{\|\hat{X}w-y\|}{\|y\|}$	steps	time (sec)
10^4	$(2.768 \pm 2.726) \times 10^{-14}$	$9.996 \times 10^{-1} \pm 3.268 \times 10^{-4}$	4	5.5×10^{-4}
10^2	$1.477 \times 10^{-14} \pm 5.881 \times 10^{-15}$	$9.336 \times 10^{-1} \pm 4.214 \times 10^{-2}$	10	1.2×10^{-3}
1	$2.032 \times 10^{-14} \pm 4.859 \times 10^{-15}$	$5.606 \times 10^{-2} \pm 6.847 \times 10^{-3}$	17	1.8×10^{-3}
10^{-2}	$2.754 \times 10^{-14} \pm 4.481 \times 10^{-15}$	$5.541 \times 10^{-4} \pm 6.994 \times 10^{-5}$	17	1.9×10^{-3}
10^{-4}	$2.798 \times 10^{-14} \pm 1.002 \times 10^{-15}$	$5.226 \times 10^{-6} \pm 7.742 \times 10^{-7}$	18	2.0×10^{-3}

Table 4: Conjugate gradient results.

The last comparison we did was about the performance of the algorithm with and without explicitly the building of the A matrix. What we noticed is that, in the first case, with the same number of steps, the relative error was one order of magnitude worse than the second one and this happens when the matrix $\hat{X}$ becomes more and more ill-conditioned, that is, when $\lambda < 1$. Moreover, the run time was also an order of magnitude slower.

λ	$\frac{\|w-w^*\|}{\|w^*\|}$	$\frac{\|w-w^*\|}{\|w^*\|}$ A explicitly computed	$\frac{\|w-w^*\|}{\|w^*\|}$ Matlab lsqr
10^4	2.768×10^{-14}	1.522×10^{-14}	3.729×10^{-14}
10^2	1.477×10^{-14}	2.343×10^{-14}	4.206×10^{-14}
1	2.032×10^{-14}	3.487×10^{-13}	1.410×10^{-14}
10^{-2}	2.754×10^{-14}	3.487×10^{-13}	2.051×10^{-14}
10^{-4}	2.798×10^{-14}	3.605×10^{-13}	1.749×10^{-14}

Table 5: Comparison between the conjugate gradient with and without explicitly computing A, the former results are taken from Table 4

.

Comparing our implementation with an off-the-shelf resolver of Matlab, we can state that our implementation works nicely. We first tested our implementation versus the *pcg* method (preconditioned conjugate gradient) of matlab without using preconditioning and then versus *lsqr*. The pcg method requires a square matrix in input, so in order to use it we had to explicitly compute $A = \hat{X}^T\hat{X}$ and the performance on the errors were almost identical to the ones reported in the third column of Table 5. Instead, the lsqr algorithm carried out results closer to ours, in terms of relative errors and run time. In order to avoid "verbosity", in Table 5 we only report the relative errors of this last algorithm.

5 Standard momentum descent (heavy ball)

In this section we will talk about the fourth and last method considered for solving the lls problem, we will proceed as we did for the L-BFGS, thin QR and conjugate gradient.

5.1 Overview on gradient descent

Gradient descent is the most well-known way to minimize an objective function $J(W)$ parameterized by a model's parameters $W \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_W J(W)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

Let's take a look at the easiest implementation that computes the gradient of the cost function w.r.t. to the parameters W for the entire training data-set (the so called batch version).

$$W = W - \eta \nabla J(W) \tag{5.1}$$

The main issue of such method is the slow convergence, resulting in an extremely high number of iterations. To alleviate this behavior the momentum descend approach can be used.

5.2 Standard momentum descent

The standard momentum descent is a technique first presented by Polyak [9] that introduces a variable V which accumulates the velocity towards directions of persisted reduction in the objective across iterates [12]. The momentum algorithm allows to accelerate the gradient descent as in the detected direction it adds a fraction β of the vector calculated in the previous step to the current update vector. The hyper-parameters are the learning rate η and the momentum $\beta \in [0, 1)$, typically chosen as 0.9. If $\beta = 0$, then classical momentum reduces to the gradient method. The parameter update works as follow:

$$V_{dw} = \beta V_{dw} - \eta * dW \tag{5.2a}$$

$$W = W + V_{dw} \tag{5.2b}$$

5.3 Solving LLS with standard momentum descent

The idea behind gradient descent is pretty similar to the one seen for conjugate gradient. Gradient descent can be used to solve a system of linear equation $Ax - b = 0$, where A is symmetric and positive definite, but, as we know, the $\hat{X}$ matrix has not the same proprieties of A, so in order to apply gradient descent to our problem we need to define $A = \hat{X}^T \hat{X}$ and $b = \hat{X}^T \hat{y}$. We have already explained this fact when we described how to solve LLS

problem while using conjugate gradient in 4.2. For this reason, we will not talk again about the preliminary part to do before starting to apply the algorithm.

The gradient descent algorithm differs a bit from the conjugate gradient one, so we are going to report it. Also, even for this algorithm, we do not explicitly compute the matrix A, even if it would have saved us time because we preferred to get more qualitative results, limiting the condition number of A.

Algorithm 6: Standard momentum descent

Given $w_0 \in R^m$, β, $v = 0$;
while $k < max\_iterations$ *and* $error \geq tol$ **do**
    $r_k = b - Aw_k$;
    $\eta = \frac{r_k^T r_k}{r_k^T A r_k}$;
    $v = \beta v + \eta r_k$;
    $w_k = w_k + v$;
    $k = k + 1$;
return w;

The computational cost of gradient descent is dominated by matrix-vector products and fortunately, one can be eliminated. If we do not consider the momentum update rule, we get $w = w + \eta r_k$ and by premultiplying both sides of this equation by $-A$ and adding b, we have that

$$r_k = r_k - \eta A r_k \tag{5.3}$$

so we can store the result of Ar_k in order to avoid to compute it twice per iteration. The starting point for r still remains $r_0 = b - Aw_0$, then it gets updated by using the new formula. The disadvantage of using this recurrence is that the sequence defined by (5.3) is generated without any feedback from the value of w_i, so that accumulation of floating point round-off error may cause w_i to converge to some point near w. This effect can be avoided by periodically using $r_k = b - Aw_k$ to recompute the correct residual as suggested by Shewchuk et al. [10].

5.4 Convergence analysis with exact line search

Assume that f is strongly convex on S, so there are positive constants θ_1 and θ_n such that $\theta_1 I \leq \nabla^2 f(x) \leq \theta_n I, \forall x \in S$. By using the lighter notation $x^{(k+1)} = x^{(k)} + \eta^{(k)} \Delta x^{(k)}$, Boyd and Vandenberghe [3] state that

$$f(x^{(k)}) - p^* \leq c^k (f(x^{(0)}) - p^*) \tag{5.4}$$

where p^* is the optimal value of the function f and $c = 1 - \theta_1/\theta_n < 1$ [7], which shows that $f(x^{(k)})$ converges to p^* as $k \to \infty$. In particular, we must have $f(x^{(k)}) - p^* \leq \epsilon$ after at most

$$\frac{log((f(x^{(0)}) - p^*)/\epsilon)}{log(1/c)} \tag{5.5}$$

[7] recalling that $\kappa = \theta_n/\theta_1$ is an upper bound on the condition number of the matrix $\nabla^2 f(x)$, *i.e.* the ratio of its largest eigenvalue to its smallest eigenvalue.

iterations of the gradient method with exact line search. We can now analyze what kind of information can be derived from both the numerator and the denominator. The first one can be interpreted as the log of the ratio of the initial sub-optimality (*i.e.*, gap between $f(x^0)$ and p^*, to the final sub-optimality (*i.e.* less than ϵ). This term suggests that the number of iterations depends on how good the initial point is, and what the final required accuracy is. Instead, the denominator of (5.5) is a function of θ_n/θ_1, which we have seen is a bound on the condition number of $\nabla^2 f(x)$ and for large condition number bound θ_n/θ_1, we have

$$log(1/c) = -log(1 - \theta_1/\theta_n) \approx \theta_1/\theta_n \tag{5.6}$$

meaning that our bound on the number of iterations required increases approximately linearly following the increasing of θ_n/θ_1. This fact lead the gradient method to require a large number of iterations when the Hessian of f, near x^*, has a large condition number. Finally, the bound (5.4) shows that the error $f(x^{(k)}) - p^*$ converges to zero at least as fast as a geometric series that, in the context of iterative numerical methods, is called linear convergence.

5.5 Convergence analysis of heavy ball

Again, we know that f is strongly convex on S, so there are positive constants θ_1 and θ_n such that $\theta_1 I \leq \nabla^2 f(x) \leq \theta_n I, \forall x \in S$. The heavy ball method adds a momentum term in gradient descent:

$$x_{k+1} = x_k - \eta_k \nabla_x f(x_k) + \underbrace{\beta_k(x_k - x_{k-1})}_{\text{HBM momentum}} \tag{5.7}$$

Consider $x_{k+1} - x^*$. By definition of HBM update (5.7),

$$x_{k+1} - x^* = x_k - \eta_k \nabla_x f(x_k) + \beta_k(x_k - x_{k-1}) - x^* \tag{5.8}$$

As $\nabla_x f(x_k) = Ax_k - b$ and $b = Ax^*$, we have $\nabla_x f(x_k) = Ax_k - Ax^*$. By developing (5.8), we have:

$$\begin{aligned} x_{k+1} - x^* &= x_k - x^* - \eta_k A(x_k - x^*) + \beta(x_k - x_{k-1}) \\ &= (I - \eta_k A)(x_k - x^*) + \beta_k(x_k - x_{k-1} - x^* + x^*) \\ &= (I - \eta_k A)(x_k - x^*) - \beta_k(x_{k-1} - x^*) + \beta(x_k - x^*) \\ &= ((1 + \beta_k)I - \eta_k A)(\, x_k - x^*) - \beta_k(x_{k-1} - x^*) \end{aligned} \tag{5.9}$$

In this sense, we have to consider $x_k - x^*$ and $x_{k-1} - x^*$ at the same time

$$\begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} = \underbrace{\begin{bmatrix} (1 + \beta_k)I - \eta_k A & -\beta_k I \\ I & 0 \end{bmatrix}}_{T_k(\eta,\beta)} \begin{bmatrix} x_k - x^* \\ x_{k-1} - x^* \end{bmatrix} \tag{5.10}$$

where $T_k(\eta, \beta)$ is the transition matrix. For simplicity we consider the compact expression

$$\begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} = T_k(\eta, \beta) \begin{bmatrix} x_k - x^* \\ x_{k-1} - x^* \end{bmatrix} \tag{5.11}$$

Take constant η_k and β_k in T_k, so $T_k = T$ and

$$\begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} = T^k \begin{bmatrix} x_1 - x^* \\ x_0 - x^* \end{bmatrix} \tag{5.12}$$

Take the norm

$$\left\| \begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} \right\| = \left\| T^k \begin{bmatrix} x_1 - x^* \\ x_0 - x^* \end{bmatrix} \right\| \leq \|T^k\| \left\| \begin{bmatrix} x_1 - x^* \\ x_0 - x^* \end{bmatrix} \right\| \tag{5.13}$$

So, if $\|T^k\|$ is bounded, the series x_k produced by the heavy ball method converges. In order to explain this we need two lemmas, for which we omit the proof.

Lemma 5.1 *For a matrix $T \in \mathbb{R}^{n \times n}$, there exists a sequences $\epsilon_k \geq 0$ such that $\|T^k\| \leq (\rho(T) + \epsilon_k)^k$ where $\lim_{k \to \infty} \epsilon_k = 0$.*

Lemma 5.2 *For $\beta > (1 - \sqrt{\eta\theta_n})^2$, $\rho(T) < \beta$, where $\rho(T) = \max\{|\psi_1|, |\psi_2|, \ldots, |\psi_n|\}$ is the spectral radius of matrix T and ψ_i are the eigenvalues of T.*

Now, assume $\beta > (1 - \sqrt{\eta\theta_n})$, by lemma 5.2 we have $\rho(T) = \max|\psi_i(T)| \leq \beta$. By lemma 5.1, we have $\|T^k\| \leq (\rho(T) + \epsilon_k)^k$ with $\lim_{k \to \infty} \epsilon_k = 0$. Putting lemma 5.2 into lemma 5.1 we obtain:

$$\|T^k\| \leq (\beta + \epsilon_k)^k \tag{5.14}$$

Lastly, let $\eta = \frac{4}{(\sqrt{\theta_n} + \sqrt{\theta_1})^2}$ and $\beta = \frac{\sqrt{\theta_n} - \sqrt{\theta_1}}{\sqrt{\theta_n} + \sqrt{\theta_1}}$ in T we have:

$$\left\| \begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} \right\| \leq \left(\frac{\sqrt{\theta_n} - \sqrt{\theta_1}}{\sqrt{\theta_n} + \sqrt{\theta_1}} + \epsilon \right)^k \left\| \begin{bmatrix} x_1 - x^* \\ x_0 - x^* \end{bmatrix} \right\| \tag{5.15}$$

or:

$$\|x_k - x^*\| \leq \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} + \epsilon \right)^k \|x_0 - x^*\| \tag{5.16}$$

where $\kappa = \frac{\theta_n}{\theta_1}$. By exploiting momentum, we benefit of an improvement from $\left(\frac{\kappa - 1}{\kappa + 1} + \epsilon \right)^k$ of the standard gradient descent to $\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} + \epsilon \right)^k$ of the heavy ball method. This result is similar to the one obtained for the conjugate gradient method, in (4.31).

5.6 Performance analysis

5.6.1 Linear rate of convergence

In 5.5, we stated that the standard momentum descent algorithm has a linear convergence rate due to the fact that our function is strongly convex.

In Figure 5.1 we can empirically confirm that our implementation has the theoretically expected convergence rate. Moreover we can notice that when $\hat{X}^T \hat{X}$ becomes more and more ill-conditioned, the GD method requires more steps to converge, as discussed in 5.4.

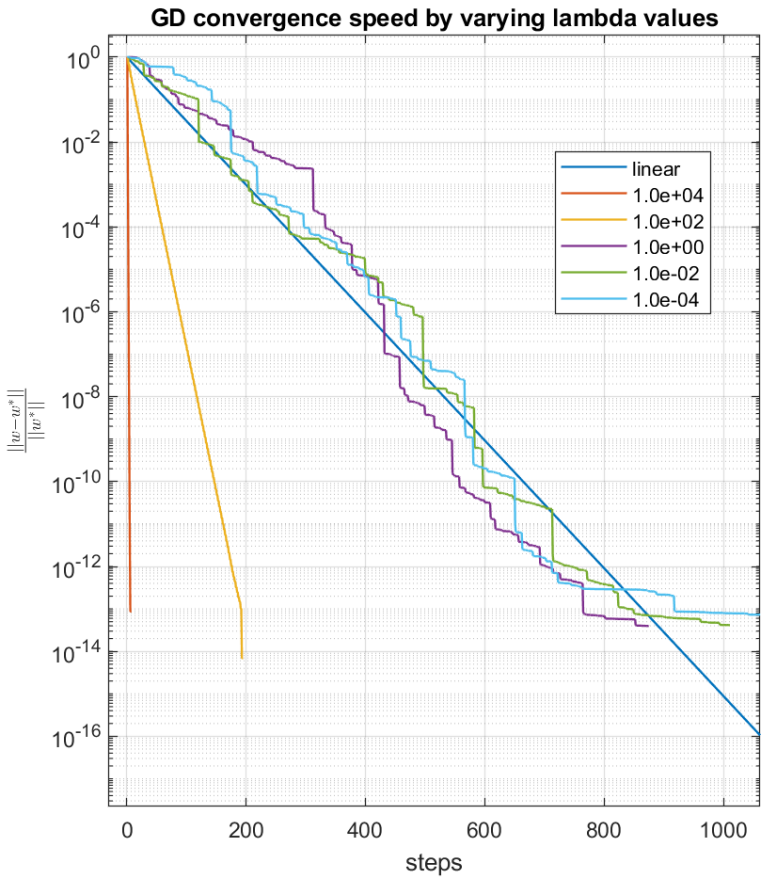


Figure 5.1: GD convergence speed by varying lambdas values, with momentum.

5.6.2 Effect of momentum

A fundamental analysis involves the role of β in our method, therefore, in order to find the best value for such hyper-parameter, we ran a grid search. For instance, in Figure 5.2, we fixed the problem (same input matrix, expected values and starting point) and we found that the best value for β was equal to 0.05, of course, this value can greatly differ from problem to problem. In our case, a too high momentum slows down the convergence speed, increasing the total amount of steps needed to reach the convergence.

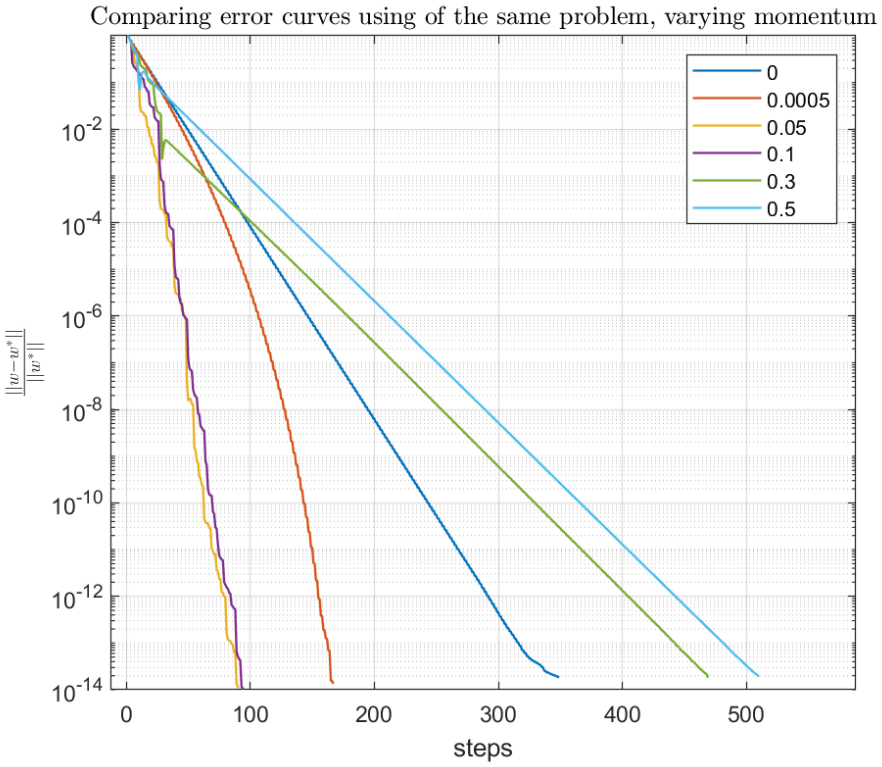
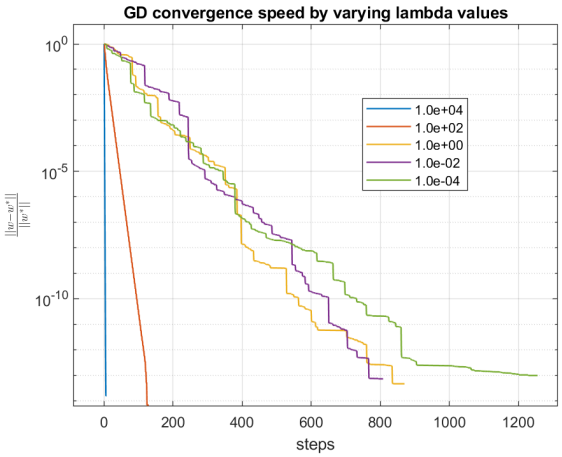


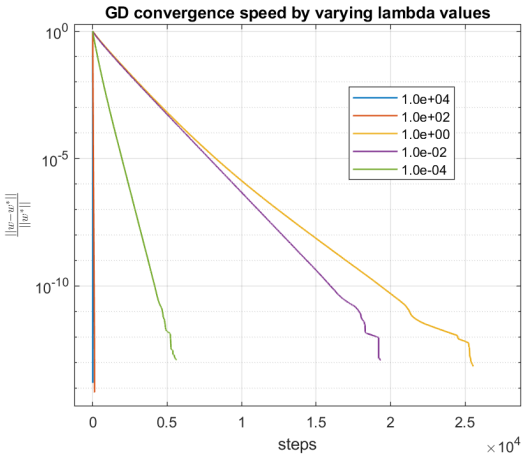
Figure 5.2: GD convergence speed by varying β values.

It is interesting to notice that the introduction of momentum makes our method faster, when chosen appropriately, but, on the other side, we can say that it becomes more unstable by looking at Figure 5.1.

Side by side comparison In order to graphically view the effect of momentum in reducing the total amount of steps required by the gradient descent to converge. In Figure 5.3 it is clearly visible that momentum plays an important role, in fact it allowed us to reduce the total amount of steps to reach the convergence especially when the Hessian becomes ill-conditioned.



(a) Gradient descent with momentum



(b) Gradient descent without applying momentum

Figure 5.3: Side by side comparison between the steps necessary to converge by GD method, with and without momentum.

5.6.3 Results

By looking at Table 6, it is possible to notice that the relative errors and the relative residuals are pretty similar to the conjugate gradient ones from Table 4, since the two methods are related. Furthermore, in terms of relative error and amount of steps to reach the convergence, the algorithm performed better without using momentum only for $\lambda = 10^4$, whereas, in the remaining cases we tested, adding momentum allowed us to reach the solution in a faster way as already seen from Figure 5.3. As usual, w^* is the Matlab solution $\hat{X}\backslash\hat{y}$ and, from 1.2, such minima is unique and global.

λ	β	$\frac{\|w-w^*\|}{\|w^*\|}$	$\frac{\|\hat{X}w-y\|}{\|y\|}$	steps	time (sec)
10^4	0	$3.49 \times 10^{-14} \pm 2.15 \times 10^{-14}$	$9.99 \times 10^{-1} \pm 3.04 \times 10^{-4}$	19 ± 9	9.56×10^{-4}
10^2	0.05	$8.67 \times 10^{-15} \pm 8.08 \times 10^{-16}$	$9.39 \times 10^{-1} \pm 4.21 \times 10^{-2}$	106 ± 10	4.12×10^{-3}
1	0.05	$3.51 \times 10^{-14} \pm 1.85 \times 10^{-14}$	$5.61 \times 10^{-2} \pm 5.66 \times 10^{-3}$	1350 ± 342	4.91×10^{-2}
10^{-2}	0.05	$5.77 \times 10^{-14} \pm 3.08 \times 10^{-14}$	$5.07 \times 10^{-4} \pm 7.58 \times 10^{-5}$	1075 ± 78	3.84×10^{-2}
10^{-4}	0.05	$6.42 \times 10^{-14} \pm 2.57 \times 10^{-14}$	$5.42 \times 10^{-6} \pm 8.26 \times 10^{-7}$	1148 ± 59	4.10×10^{-2}

Table 6: Standard momentum descent results.

6 Comparison between methods

As last step for this report we will blend together all the results achieved by the four different methods illustrated by, first, comparing the three iterative methods on time metrics and, then, by comparing all of them based on errors and residuals. Keep in mind that the results shown here and in all the tables before were achieved with starting point $x_0 = 0$, since we have empirically seen it to be the best choice.

6.1 Comparison on time metrics

We will start by considering only three of the methods seen until now (L-BFGS, CG and SMD) since thin QR is not iterative. Table 7 shows the execution times and number of iterations achieved by the best configuration of these three methods:

- For the L-BFGS method the best results were retrieved from Table 1 where $l = 20$.
- Conjugate gradient had no hyper-parameters, so the results are those seen from Table 4.
- For standard momentum descent the results are those from Table 6, following the grid search for the best values of β.

λ	L-BFGS		CG		SMD	
	time	steps	time	steps	time	steps
10^4	3.0×10^{-3}	4	5.5×10^{-4}	4	9.5×10^{-4}	19
10^2	3.5×10^{-3}	10	1.2×10^{-3}	10	4.1×10^{-3}	106
1	3.8×10^{-3}	13	1.8×10^{-3}	18	4.9×10^{-2}	1350
10^{-2}	3.9×10^{-3}	13	1.9×10^{-3}	17	3.8×10^{-2}	1075
10^{-4}	4.0×10^{-3}	13	2.0×10^{-3}	18	4.1×10^{-2}	1148

Table 7: Comparison on execution time (in seconds) and number of iterations between the three iterative methods.

As we can see from Table 7 L-BFGS and conjugate gradient methods performs in a similar way, instead SMD has an higher number of iterations and execution time as λ starts to get smaller and smaller, this is expected since standard momentum descent is just a poorer version of conjugate gradient.

Between L-BFGS and conjugate gradient, the former has a low number of iteration but double the execution time, so, at the end, the best iterative algorithm, in our case, would be conjugate gradient.

6.2 Comparison on metrics

Let us now talk about the comparison on errors and residuals, this time including the results achieved by the thin QR factorization. The results are shown in Table 8.

λ	L-BFGS		Thin QR	
	error	residual	error	residual
10^4	1.40×10^{-14}	9.99×10^{-1}	7.38×10^{-14}	9.99×10^{-1}
10^2	5.62×10^{-15}	9.37×10^{-1}	1.56×10^{-14}	9.11×10^{-1}
1	1.73×10^{-14}	5.89×10^{-2}	2.03×10^{-14}	5.34×10^{-2}
10^{-2}	2.72×10^{-14}	5.02×10^{-4}	9.01×10^{-14}	5.44×10^{-4}
10^{-4}	4.07×10^{-14}	6.08×10^{-6}	8.17×10^{-14}	5.40×10^{-6}
λ	CG		SMD	
	error	residual	error	residual
10^4	2.76×10^{-14}	9.99×10^{-1}	3.49×10^{-14}	9.99×10^{-1}
10^2	1.47×10^{-14}	9.33×10^{-1}	8.67×10^{-14}	9.39×10^{-1}
1	2.03×10^{-14}	5.60×10^{-2}	3.51×10^{-14}	5.61×10^{-2}
10^{-2}	2.75×10^{-14}	5.54×10^{-4}	5.77×10^{-14}	5.07×10^{-4}
10^{-4}	2.79×10^{-14}	5.22×10^{-6}	6.42×10^{-14}	5.42×10^{-6}

Table 8: Comparison on relative errors and residuals.

By analyzing the errors and residuals obtained by the algorithms, we notice that they are pretty similar, and this is maybe due to the fact that our function is strongly convex, therefore, it was quite easy to find the unique global minimum and, as a consequence, we had the nice possibility to use the exact line search for those algorithm that required a line search. As we can see in Table 8, in terms of relative errors, L-BFGS performs better with respect to the others, but not significantly noticing that the order of magnitude is the same of the other algorithms.

Another aspect to consider is that the performances of SMD and L-BFGS depend on their hyper-parameters. We are referring to the memory size l for L-BFGS and to the step size η and the momentum β for SMD. The first hyper-parameter of the latter is useful only for the fixed step length version, that often requires a grid search before being chosen, as well as for the momentum. To clarify, the choice of a too low memory size of L-BFGS could slow down the convergence rate, but eventually the algorithm finds a solution, instead this does not happen if the learning rate is chosen wrongly, since it could cause the SMD to diverge. Rather, thin QR and conjugate gradient are ready-to-use algorithms that do not require hyper-parameters tuning, so they are easier to set up.

6.3 Comparison on space

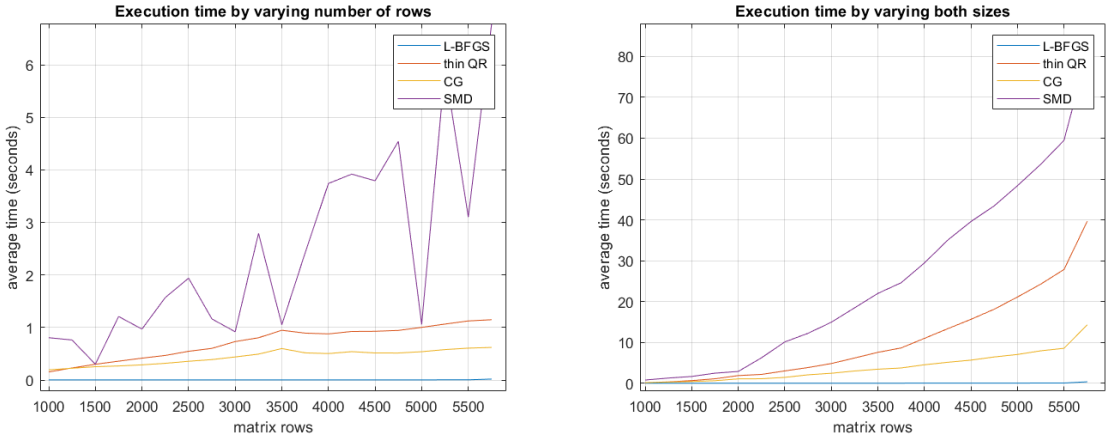
Now, we can analyze these algorithms in terms of space complexity. Starting with thin QR, we notice that is the one using more memory because it stores the upper triangular matrix R_1 and the vector resulting from the product $Q_1\hat{y}$, because we do not explicitly compute Q_1.

Then, we have conjugate gradient and standard momentum descent that are the lightest, in fact they only need few arrays: both require one for the solution, and one for the residual r_k whereas CG needs one more for p_k and SMD needs another one for the v array in order to update the solution. In the end we have L-BFGS, whose space complexity is dominated by the

two matrices $S, Y \in \mathbb{R}^{n \times l}$, where n is the size of the solution and l is the memory size, plus the same space required by the last two aforementioned algorithms. Let us say that L-BFGS uses less memory than thin QR when $l < \frac{n}{2}$, which always holds for our case since $n = 1477$ and $l \in [3, 20]$.

6.4 Comparison on scalability

As final comparison we will compare the scalability of each method in a similar fashion to what we have done in 3.4.1. We generated 20 matrices and we run each every method five times on them, then we averaged the results in order to compare the methods' behavior as the size of the problem increases. As first test we fixed the number of columns to 200 and we let the number of rows grow by 250 at every new generated matrix, starting from 1000. Afterwards, for the second test, we performed the same task of the first one but, this time, we also increased the number of columns by 50 at each step to see how the methods scale when both sizes increase.



(a) Scalability by varying the number of rows

(b) Scalabilty by varying both sizes

Figure 6.1: Scalability comparison between the four methods, first by only varying the number of rows and then by varying both sizes.

As we can see from Figure 6.1 standard momentum descent performs worse than the other three methods since we could not do an exhaustive grid search to look for the best value of β for each configuration. Other than that, from both figures we can notice that L-BFGS (with $l = 20$) is the uncontested winner since it suffers a little from the increase in size of the matrix. Moreover, by looking at 6.1b we can see that thin QR is the one suffering the most by the increase in columns, as expected from the theory, meanwhile the others maintain, more or less, the same behavior, except for SMD which seems more stable than what we can see in 6.1a at the cost of an even worse scalability.

7 How to run the project

First, you can clone the project repo from github in the following way

```
git clone https://github.com/nikodallanoce/ComputationalMathematics/
```

Then you can work with each method separately on Matlab by going inside the chosen algorithm folder and running *run_method.m*, change the hyper-parameters as you prefer.

A Appendix of proofs

A.1 λ is the smallest singular value of $\hat{X}$

For that we need to follow what [13] has stated in theorem 1 and apply it to our case, therefore: let $\hat{X}$ be an $m \times n$ matrix where $\hat{X} = \begin{bmatrix} X^T \\ \lambda I \end{bmatrix}$ with $X^T \in \mathbb{R}^{k \times n}$, whose singular values are $\sigma_1(\hat{X}) \geq \cdots \geq \sigma_n(\hat{X})$. Let λI be a $n \times n$ submatrix of $\hat{X}$, with singular values $\sigma_1(\lambda I) \geq \cdots \geq \sigma_n(\lambda I)$, then

$$\sigma_i(\hat{X}) \geq \sigma_i(\lambda I), \quad for \quad i = 1, \ldots, n \tag{A.1a}$$

$$\sigma_i(\lambda I) \geq \sigma_{i+k}(\hat{X}), \quad for \quad i \leq min(n-k, n) \tag{A.1b}$$

with $k = m - n$.

Let's consider the $(n-k)th$ singular value of λI, following (A.1b) we have that

$$\sigma_n(\hat{X}) \leq \sigma_{n-k}(\lambda I) \tag{A.2}$$

Then by putting (A.1a) into (A.2) we obtain

$$\sigma_n(\lambda I) \leq \sigma_n(\hat{X}) \leq \sigma_{n-k}(\lambda I) \tag{A.3}$$

λI is an $n \times n$ matrix whose singular values are all equal to λ, therefore from (A.3) we can easily state that

$$\sigma_n(\hat{X}) = \lambda \tag{A.4}$$

A.2 Condition number of $\hat{X}$

As we know from the theory, the condition number κ of a matrix is the following

$$\kappa(\hat{X}) = \frac{\sigma_1}{\sigma_n} \tag{A.5}$$

where σ_1 and σ_n are the biggest and smaller singular values of $\hat{X}$ and we already know from subsection A.1 that the latter is λ. Therefore as λ decreases the more $\kappa(\hat{X})$ increases, formally

$$\lim_{\lambda \to 0} \kappa(\hat{X}) = \lim_{\lambda \to 0} \frac{\sigma_1}{\lambda} = \infty \tag{A.6}$$

We will show, as last thing, that (A.6) holds by looking at how $\kappa(\hat{X})$ changes at different values of λ in Table 9

λ	$\kappa(\hat{X})$
10^4	1.0034
10^2	8.32
1	825.532
10^{-2}	8.255×10^4
10^{-4}	8.255×10^6

Table 9: $K(\hat{X})$ at varying values of λ

A.3 Distinct eigenvalues of $\hat{X}^T\hat{X}$

We know from subsection A.1 that the submatrix $\lambda I \in \mathbb{R}^n$ has only one distinct singular value, that is λ. Moreover, the upper component of $\hat{X}$, X^T has $m - n = 12$ distinct singular values.

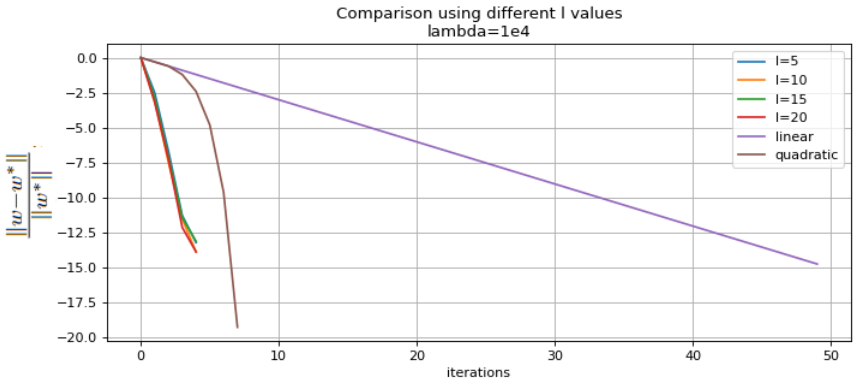
Now, since we know that

$$eig(\hat{X}^T\hat{X}) = \Sigma^2(\hat{X}) \tag{A.7}$$

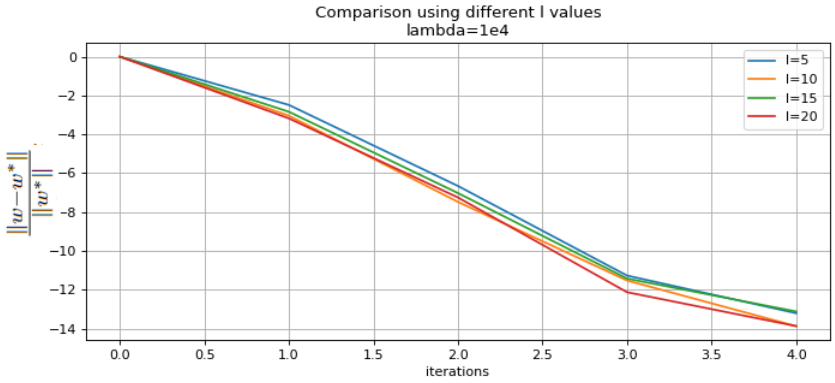
we can safely state that $\hat{X}^T\hat{X}$ has $m - n + 1$ distinct eigenvalues, for the case under our consideration there are 13 of them.

B Appendix of plots

B.1 L-BFGS

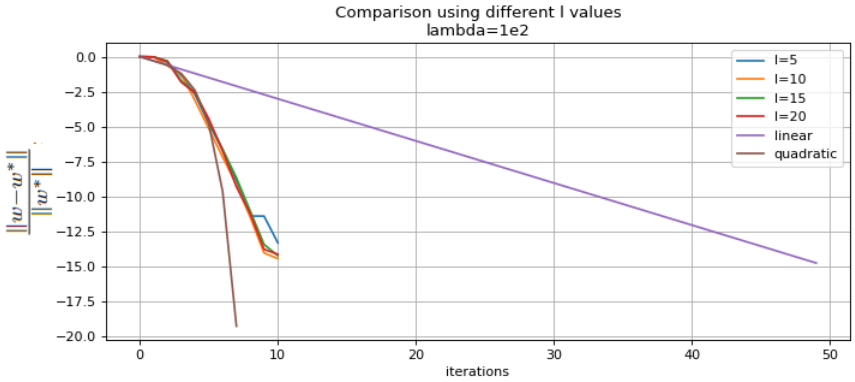


(a) Considering the convergence rates

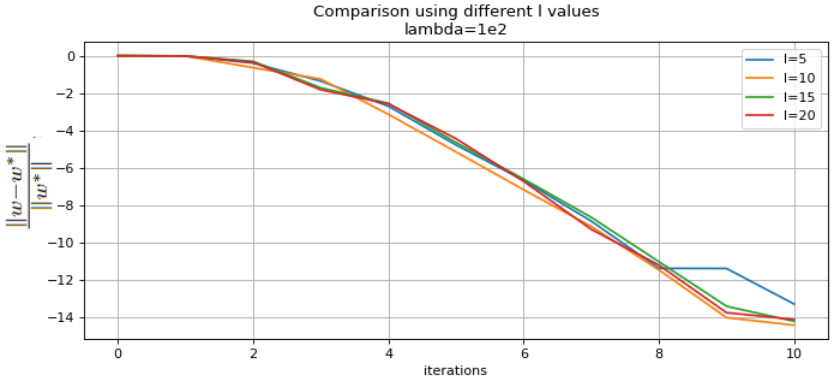


(b) Without the convergence rates

Figure B.1: Convergence plot with $\lambda = 1e4$ at varying values of l

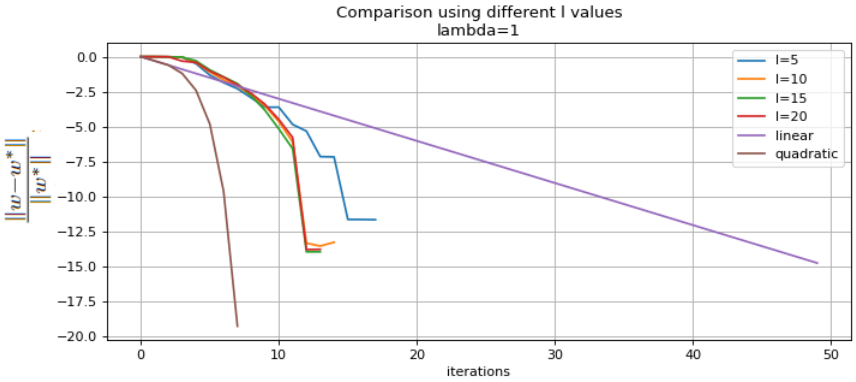


(a) Considering the convergence rates

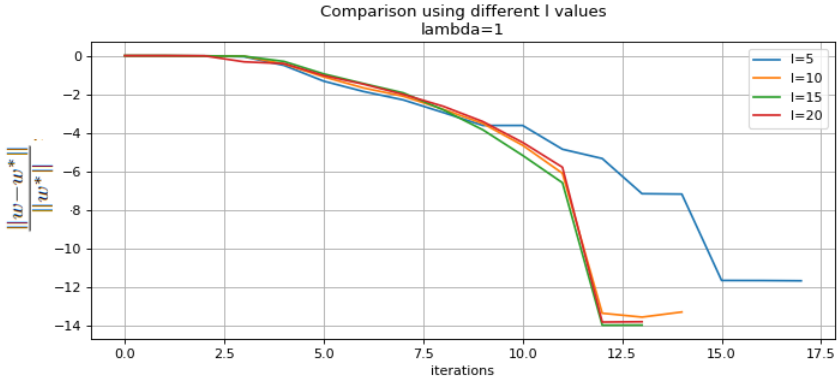


(b) Without the convergence rates

Figure B.2: Convergence plot with $\lambda = 1e2$ at varying values of l

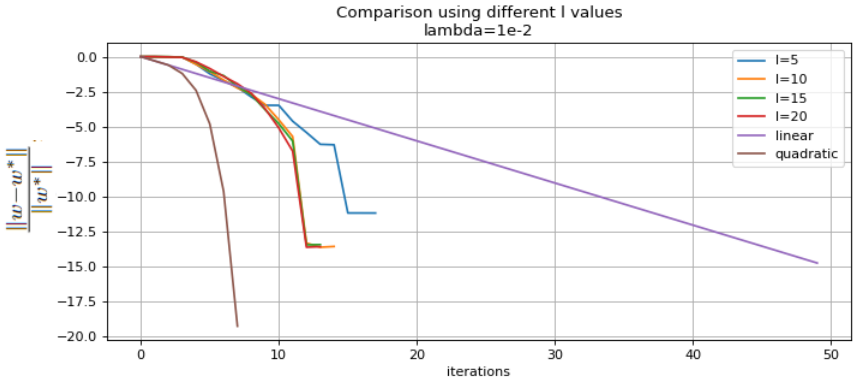


(a) Considering the convergence rates

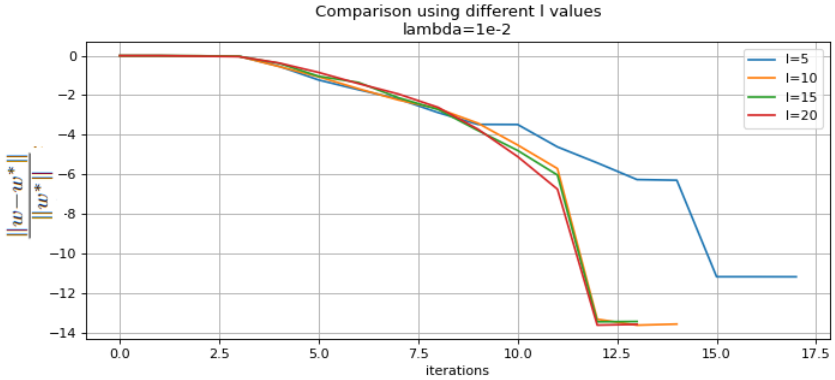


(b) Without the convergence rates

Figure B.3: Convergence plot with $\lambda = 1$ at varying values of l

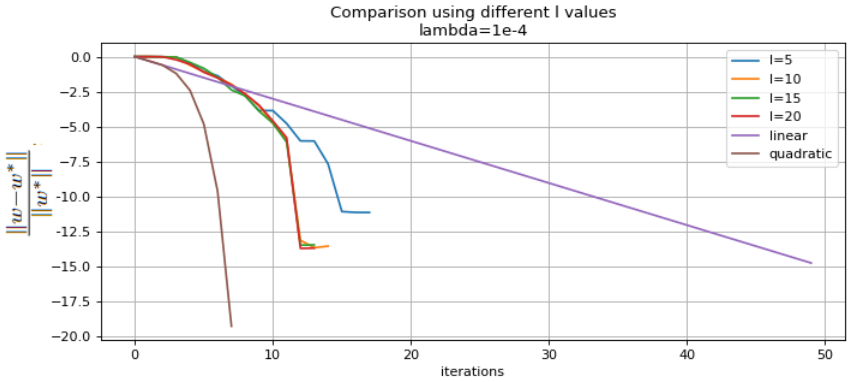


(a) Considering the convergence rates

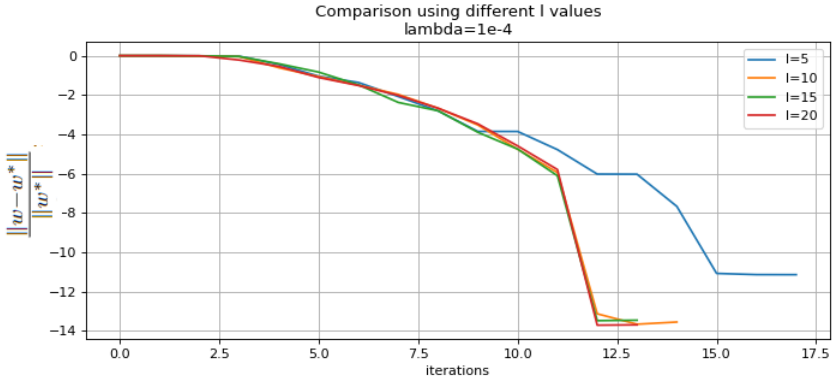


(b) Without the convergence rates

Figure B.4: Convergence plot with $\lambda = 1e-2$ at varying values of l



(a) Considering the convergence rates



(b) Without the convergence rates

Figure B.5: Convergence plot with $\lambda = 1e-4$ at varying values of l

References

[1] Bernhard Beckermann and Arno BJ Kuijlaars. Superlinear convergence of conjugate gradients. *SIAM Journal on Numerical Analysis*, 39(1):300–329, 2001.

[2] Åke Björck and Tommy Elfving. Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. *BIT Numerical Mathematics*, 19(2):145–163, 1979.

[3] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[4] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, 2014.

[5] Igor Gitman, Hunter Lang, Pengchuan Zhang, and Lin Xiao. Understanding the role of momentum in stochastic gradient methods. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9630–9640, 2019.

[6] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[7] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[8] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[9] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[10] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[11] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[12] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[13] R.C. Thompson. Principal submatrices ix: Interlacing inequalities for singular values of submatrices. *Linear Algebra and its Applications*, 5(1):1–12, 1972.

[14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[15] Stephen Wright, Jorge Nocedal, et al. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

[16] Tianbao Yang, Qihang Lin, and Zhe Li. Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization, 2016.