CUED - Engineering Tripos Part IIB 2023-2024 **Module Coursework**

Module	4M17	Title of report	Coursework Assignment 1
Date submitted: 08/12/2023		Assessment for this module is ☑ 100% / ☐ 25% coursework of which this assignment forms _50__ %	
UNDERGRADUATE and POST GRADUATE STUDENTS			
Candidate number:	5553C	☑ Undergraduate ☐ Post graduate	

Feedback to the student ☐ **See also comments in the text**		Very good	**Good**	Needs improvmt
CONTENT	**Completeness, quantity of content:** Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly?			
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?			
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?			
	Comments:			
PRE	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			

4M17 Coursework Assignment 1

December 8, 2023

Abstract

Explore the l_1, l_2 and l_∞ norm; compare minimisation problems utilising each norm and show how the results differ. The interior point method for the l_1 norm is implemented as we show how a constrained problem can be cast as an unconstrained one, whilst highlighting the need for optimised implementations. Finally, we reconstruct a signal using LASSO (l_2 norm on the residuals and an l_1 regulariser) and observe the balance one must find between the regulariser and residuals in the cost function.

1 Comparison of Norms

When approaching optimisation problems, such as regression, we must decide which norm we wish to minimise; each norm has its uses and is particularly desirable under certain circumstances; No, given norm is universally better than another [1]. Note, it is common, such as regularisation in TensorFlow, to use a mixture of different norms, e.g. "tf.keras.regularizers.L1L2()".

1.1 A)

Each of the norms on $\mathcal{R}^m$ that we will be exploring are shown in Equation 1. We will be applying them to the residual given by $\mathbf{Ax} - \mathbf{b}$.

$$\begin{aligned} l_1\text{: } ||\mathbf{x}||_1 &= |x_1| + ... + |x_m| \\ l_2\text{: } |||\mathbf{x}||_2 &= \sqrt{|x_1|^2 + ... + |x_m|^2} \\ l_\infty\text{: } |||\mathbf{x}||_\infty &= max(|x_1|, ..., |x_m|) \end{aligned} \tag{1}$$

The l_1 norm on the residual is given below in Equation 2. This is simply a minimisation of the sum of the modulus of each entry in the residual vector.

$$min_x||\mathbf{Ax} - \mathbf{b}||_1 = \sum_{i=1}^{m} |\mathbf{a_i}^T\mathbf{x} - b_i| \tag{2}$$

Equation 3 gives the l_2 norm on the residual which is widely used (least-squares regression). It places greater emphasis on minimising larger residuals; however, the cost is that outliers will have an unfavourably large impact, which is not seen in the l_1 norm.

$$min_x||\mathbf{Ax} - \mathbf{b}||_2 = \sqrt{\sum_{i=1}^{m} (\mathbf{a_i}^T\mathbf{x} - b_i)^2} \tag{3}$$

Lastly, we have the l_∞ norm, which seeks to minimise the largest residual present.

$$min_x||\mathbf{Ax} - \mathbf{b}||_\infty = max_i|\mathbf{a_i}^T\mathbf{x} - b_i| \tag{4}$$

The l_2 norm is unique from the above, as the square of it can be minimised analytically, as shown in Equation 5 (This gives the least-squares regression solution).

$$
\begin{aligned}
min_x||\mathbf{Ax}-\mathbf{b}||_2^2 &= min_x(\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b}) \\
&= min_x\mathbf{f} \\
\Rightarrow \frac{\partial \mathbf{f}}{\partial \mathbf{x}} &= 0 \\
0 &= 2\mathbf{A}^T\mathbf{Ax}-2\mathbf{A}^T\mathbf{b} \\
\Rightarrow x &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}
\end{aligned} \tag{5}
$$

The solution of x is achieved by pre-multiplying $\mathbf{b}$ by the Moore-Penrose inverse. As this is analytic we expect this to be, by far, the fastest minimisation problem explored in this report.

1.2 B)

As we cannot solve the l_∞ and l_1 norms analytically, we seek to pose the minimisation as a constrained optimisation problem, as shown below in Equation 6.

$$
\begin{aligned}
min_x||\mathbf{Ax}-\mathbf{b}||_\infty &= min_x(max_i|\mathbf{a_i}^T\mathbf{x}-b_i|) \\
&= min_x(min_t(-t \leq \mathbf{a_i}^T\mathbf{x}-b_i \leq t))
\end{aligned} \tag{6}
$$

Equation 6 can now be cast as $min_{x,t}t$ subject to the constraint $-t \leq \mathbf{a_i}^T\mathbf{x}-b_i \leq t$ for $i = 1, ...m$. This can be solved as a linear program as shown in Equation 7.

$$
\begin{aligned}
&min_x\tilde{\mathbf{c}}^T\tilde{\mathbf{x}} \\
&\text{subject to } \tilde{\mathbf{A}}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}} \\
&\text{where } \tilde{\mathbf{c}} = \begin{pmatrix}\mathbf{0_n}\\ 1\end{pmatrix}, \tilde{\mathbf{x}} = \begin{pmatrix}\mathbf{x}\\ t\end{pmatrix} \\
&\tilde{\mathbf{A}} = \begin{pmatrix}\mathbf{A} & -\mathbf{1_m}\\ -\mathbf{A} & -\mathbf{1_m}\end{pmatrix}, \tilde{b} = \begin{pmatrix}\mathbf{b}\\ -\mathbf{b}\end{pmatrix}
\end{aligned} \tag{7}
$$

Where $\mathbf{A} \in \mathcal{R}^{m\times n}$, $\mathbf{b} \in \mathcal{R}^m$ and $\mathbf{x} \in \mathcal{R}^n$ results in $\tilde{\mathbf{A}} \in \mathcal{R}^{2m\times n+1}$, $\tilde{\mathbf{b}} \in \mathcal{R}^{2m}$, $\tilde{\mathbf{x}} \in \mathcal{R}^{n+1}$ and $\tilde{\mathbf{c}} \in \mathcal{R}^{n+1}$

We now repeat a similar process for the l_1 norm.

$$
\begin{aligned}
min_x||\mathbf{Ax}-\mathbf{b}||_1 &= min_x\sum_{i=1}^{m}|\mathbf{a_i}^T\mathbf{x}-b_i| \\
&= min_x\sum_{i=1}^{m}(min_{t_i}(-t_i \leq \mathbf{a_i}^T\mathbf{x}-b_i \leq t_i))
\end{aligned} \tag{8}
$$

For the l_1 norm, we cast Equation 8 as $min_{x,t}\sum_{i=1}^{m}t_i$ subject to the constraint $-t_i \leq \mathbf{a_i}^T\mathbf{x}-b_i \leq t_i$ for $i = 1, ...m$, which is shown as a linear program in Equation 7.

$$
\begin{aligned}
&min_x\tilde{\mathbf{c}}^T\tilde{\mathbf{x}} \\
&\text{subject to } \tilde{\mathbf{A}}\tilde{\mathbf{x}} \leq \tilde{\mathbf{b}} \\
&\text{where } \tilde{\mathbf{c}} = \begin{pmatrix}\mathbf{0_n}\\ \mathbf{1_m}\end{pmatrix}, \tilde{\mathbf{x}} = \begin{pmatrix}\mathbf{x}\\ \mathbf{t}\end{pmatrix} \\
&\tilde{\mathbf{A}} = \begin{pmatrix}\mathbf{A} & -\mathbf{I_m}\\ -\mathbf{A} & -\mathbf{I_m}\end{pmatrix}, \tilde{\mathbf{b}} = \begin{pmatrix}\mathbf{b}\\ -\mathbf{b}\end{pmatrix}
\end{aligned} \tag{9}
$$

Where now $\tilde{\mathbf{A}} \in \mathcal{R}^{2m\times n+m}$, $\tilde{\mathbf{b}} \in \mathcal{R}^{2m}$, $\tilde{\mathbf{x}} \in \mathcal{R}^{n+m}$ and $\tilde{\mathbf{c}} \in \mathcal{R}^{n+m}$

1.3 C)

By obtaining the dual problem and solving it, we can obtain a lower bound on the primal minimisation problems described above

We start by taking the constrained problems in Equations 7 and 9 and transforming them into unconstrained problems. We use the Lagrangian function as shown in Equation 10, which introduces a penalty cost for violating the constraint.

$$
\begin{aligned}
\mathcal{L}(\tilde{\mathbf{x}},\lambda) &= \tilde{\mathbf{c}}^T\tilde{\mathbf{x}} + \lambda^T(\tilde{\mathbf{A}}\tilde{\mathbf{x}}-\tilde{\mathbf{b}}) \\
&= -\tilde{\mathbf{b}}^T\lambda + (\tilde{\mathbf{A}}^T\lambda + \tilde{\mathbf{c}})^T\tilde{\mathbf{x}}
\end{aligned} \tag{10}
$$

Note that as $\lambda \geq 0$, when we minimise the Lagrange function (Equation 11), it must yield a solution lower than the optimal cost for the primal problem.

$$
min_x\mathcal{L}(\tilde{\mathbf{x}},\lambda) = \begin{cases} -\tilde{\mathbf{b}}^T\lambda, & \text{if } \tilde{\mathbf{A}}^T\lambda + \tilde{\mathbf{c}} = 0 \\ -\infty, & \text{otherwise} \end{cases} \tag{11}
$$

We seek to have a lower bound as close to the optimal cost, therefore, using the constrained solution above, we now set up the dual cost as a constrained maximisation problem as shown in 12.

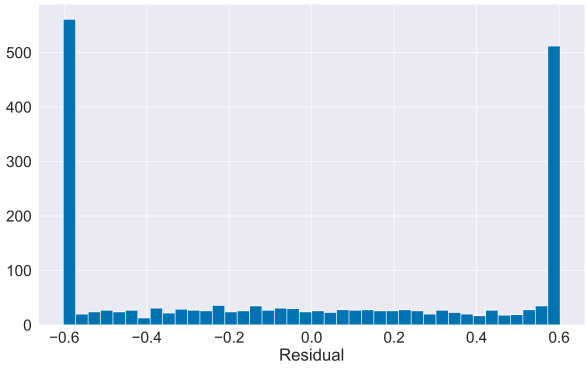
$$\begin{aligned} &max_{\lambda} - \tilde{\mathbf{b}}^T \lambda \\ &\text{subject to } \tilde{\mathbf{A}}^T \lambda + \tilde{\mathbf{c}} = 0 \\ &\text{and } \lambda \geq 0 \end{aligned} \tag{12}$$

Duality is useful as it provides a lower bound that can be used to calculate the distance to the primal cost, which is then used as a stopping criterion. Furthermore, it is also always convex, even if the primal problem is not.

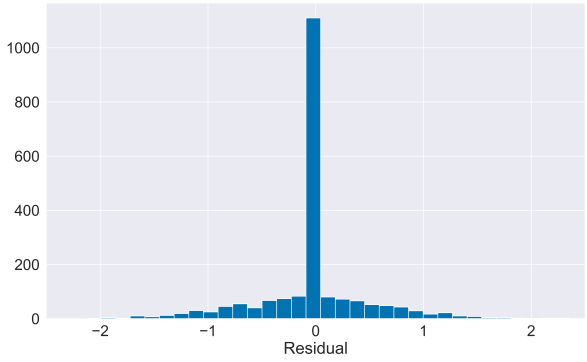
1.4 D)

Now we minimised the residuals using the data CSV files 'A0, b0', 'A1, b1', 'A2, b2', 'A3, b3', 'A4, b4', with $n = 16, 64, 256, 512, 1024$, respectively ($m = 2n$). Note that when using the l_1 and l_∞ norms, we performed the minimisation twice: once with a simplex method (SciPy 'highs-ds') and once with an interior point method (SciPy 'highs-ipm').

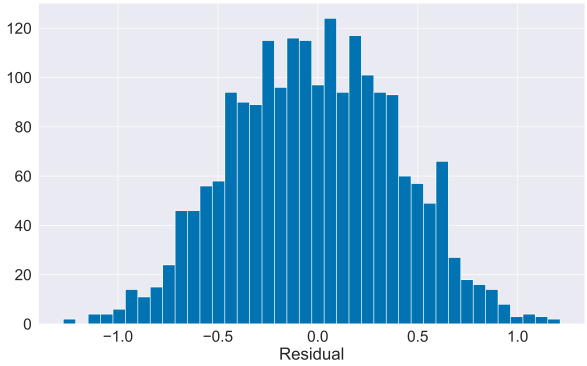
The residuals after minimisation for each of the norms and methods are shown in Figure 1. We observe that the l_∞ norm produces the tightest range of residual values, which is expected as this norm seeks to minimise the largest residual; although as it only focuses on the largest value and not the distribution of residuals, the vast majority of residuals end up grouping around the maximum residual value. This changes dramatically when using the l_1 norm as shown in Figure 1b: the vast majority of residuals are roughly 0 as this norm minimises the sum of residual magnitudes. As the focus isn't on the largest residual, large residuals up to 1.5 are present. This isn't the case in Figure 1c; the l_2 norm seeks to minimise the square of the residuals summed together; the larger outliers are minimised to a greater degree to prevent the larger impact they have. This is shown by the tighter range the residuals are distributed within compared to the l_1 norm. Although the cost of this is that far fewer residuals are around 0, instead of the spike seen at 0 for the l_1 norm, the l_2 norm produces more of a Gaussian-shaped histogram of residuals.



(a) l_∞ norm residuals



(b) l_1 norm residuals



(c) l_2 norm residuals

Figure 1: Histogram of the minimised residuals for the data 'A4' and 'b4'. For the l_1 and l_∞ residuals the Interior Points method was used.

From Figure 2 we observe that as expected the l_2 norm analytic solution was by far the quickest, as no iterations were required as it is solved directly. Furthermore, the l_2 norm was able to keep the cost far lower than the methods achieved for the l_1 norm; the ease of use and accuracy do much of the explaining for the reasons behind the l_2 norm's popularity (note that the l_∞ norm values are not comparable as it is simply the value of the largest residual, not the sum).

Both the interior point and simplex methods require similar amounts of time as shown in Figure 2 (IPM is consistently faster than SP due to the SciPy 'highs-ds' method solving both the primal and dual problem. Once considered speeds are very close). Also, both methods reach the same optimal costs. The reason for this is the strong duality present in these convex problems (see Subsections 1.3 and 1.5).

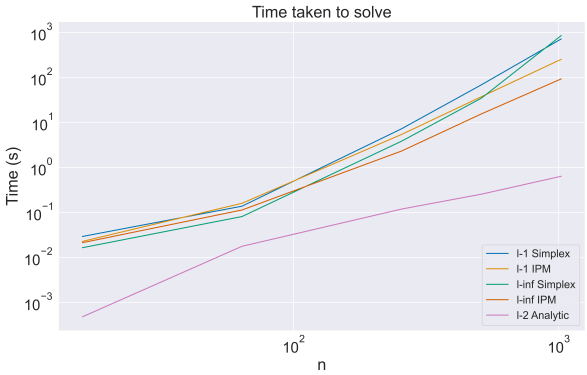


Figure 2: Time taken for each of the optimisation problems to complete for varying n.

		Minimised Value						Time Taken (s)				
Norm	Method	n =	16	64	256	512	1024	16	64	256	512	1024
l-1	Simplex		8.64	40.5	132	266	577	1.77e-2	8.66e-2	5.13	4.56e1	4.51e2
	IPM		8.64	40.5	132	266	577	4.77e-2	7.46e-2	2.75	2.47e1	1.61e2
l-inf	Simplex		0.532	0.655	0.579	0.581	0.602	1.12e-2	7.82e-2	2.05	2.14e1	5.77e2
	IPM		0.532	0.655	0.579	0.581	0.602	1.30e-2	6.13e-2	1.58	9.44	6.31e1
l-2	Analytic		2.12	5.19	8.79	12.5	18.6	3.63e-4	1.73e-1	1.12	1.79	2.66

Table 1: Table of minimised costs and time taken for different norm and optimisation methods performed on data sets with $n = 16, 64, 256, 512, 1024$. 3 s.f. entries

1.5 E)

For each of the methods used to minimise the l_∞ and l_1 norms, the dual problems described in Subsection 1.3, using the same methods, produced a duality gap of zero for each of the five data-sets. This indicates that the solutions found have strong duality and are at their respective lower bounds. This is expected as both the primal and dual solutions are feasible. In general, convex problems tend to reach a strong duality.

2 Interior Point Method

In this section, we will explore the implementation of the interior point method on the l_1 norm approximation problem.

2.1 A)

Starting from a constrained problem in Equation 9, we will design a barrier function that will indicate when the constraints are violated.

$$min_x \tilde{\mathbf{c}}^T\tilde{\mathbf{x}} + \phi(\tilde{\mathbf{x}})$$
$$\text{where } \phi(\tilde{\mathbf{x}}) = \sum_{i=1}^{2m} I_-(\tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}} - \tilde{b}_i) \tag{13}$$

Equation 13 now poses the problem as unconstrained optimisation, albeit rather ineffectively due to the lack of smoothness because the indicator function goes immediately from 0 to ∞ when the constraint is violated. Instead, we use a log barrier function, as given in Equation 14, which is ∞ when violated, but smooth when not and is easy to differentiate.

$$min_x t\tilde{\mathbf{c}}^T\tilde{\mathbf{x}} + \phi(\tilde{\mathbf{x}})$$
$$\text{where } \phi(\tilde{\mathbf{x}}) = -\sum_{i=1}^{2m} log(\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}}) \tag{14}$$

Now we can evaluate the gradient (see Equation 15) when within the feasible area and minimise the cost using a first-order gradient method. Note the introduction of t. As $t \to 0$ the log function tends to the indicator function. As we get closer to the infeasible region, we will increase the value of t to allow us to approach it more closely; otherwise, the gradient of the barrier function will keep the solution away from the boundary.

$$\nabla(f) = t\tilde{\mathbf{c}} + \sum_{i=1}^{2m} \frac{1}{\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}}}\tilde{\mathbf{a}}_i \tag{15}$$

2.2 B)

Algorithm 1 First-order gradient method with backtracking line-search

Require: $\tilde{\mathbf{x_0}}$ such that $\tilde{\mathbf{A}}\tilde{\mathbf{x}} - \tilde{\mathbf{b}} \leq 0$
$\alpha = 0.2$
$\beta = 0.3$
$f(x) = t\tilde{\mathbf{c}}^T\tilde{\mathbf{x}} - \sum_{i=1}^{2m} log(\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}})$
$\nabla(f) = t\tilde{\mathbf{c}} + \sum_{i=1}^{2m} \frac{1}{\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}}}\tilde{\mathbf{a}}_i$
while $||\nabla(f)||_2^2 \geq \epsilon$ **do**
    $t_{step} = 1$
    while $\tilde{\mathbf{A}}(\tilde{\mathbf{x}} - t_{step}\nabla(f)) - \tilde{\mathbf{b}} > 0$ **do**
        $t_{step} = beta \times t_{step}$
    while $f(\tilde{\mathbf{x}} - t_{step}\nabla(f)) > f(\tilde{\mathbf{x}}) - \alpha \times t||\nabla(f)||_2^2$ **do**
        $t_{step} = beta \times t_{step}$
        $\tilde{\mathbf{x}} = \tilde{\mathbf{x}} - t_{step}\nabla(f)$
        $f(x) = t\tilde{\mathbf{c}}^T\tilde{\mathbf{x}} - \sum_{i=1}^{2m} log(\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}})$
        $\nabla(f) = t\tilde{\mathbf{c}} + \sum_{i=1}^{2m} \frac{1}{\tilde{b}_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}}}\tilde{\mathbf{a}}_i$

After implementing Pseudo-code 1 we minimise the residual for A3 and b3. When using $t_{step} = 2$ and $\epsilon = 0.01$, we obtain a l_1 norm minimised residual of 311 (3 s.f.) and a run-time of 57.7 seconds. This is 17% above 266 which was the minimised value using SciPy interior points solver in Subsection 1.4 and equal to the lower bound. This is of course expected as the barrier function will have a gradient pushing the optimal solution away from the infeasible region.

2.3 C)

To achieve greater accuracy we should repeat the above code several times with a larger t_{step} and initialised at the solution for the previous t_{step}. This means as we get closer to the barrier, we increase t_{step} which will decrease the influence of the barrier function on the cost and hence allow the gradient of $\tilde{\mathbf{c}}^T\tilde{\mathbf{x}}$ to overcome the gradient of the barrier function and take the solution closer to the infeasible region.

We performed 3 loops with t =

1, 10, 100, which achieved a minimised value of 269 (3 s.f.); this is significantly closer than 311 to the optimal cost of 266 (Subsection 1.4. A major drawback of this implementation was that it took a substantial amount of time to complete (45 minutes), although this is expected as it is a crude implementation, whereas the SciPy 'highs-ipm' solver is a C++ wrapper and far more refined [2].

3 Regularisation in Signal Reconstruction

When reconstructing signals, a regulariser may be implemented to keep the size of $\mathbf{x}$ small while the residuals, $\mathbf{Ax} - \mathbf{b}$, are minimised. Here we explore LASSO, which uses an l_2 norm to minimise residuals and an l_1 norm to regularise (see Equation 16); this performs well on sparse data sets.

$$min_x||\mathbf{Ax} - \mathbf{b}||_2^2 + \lambda||\mathbf{x}||_1 \tag{16}$$

3.1 A)

Due to the introduction of the l_1 norm term in Equation 16, the solution to LASSO does not exist in closed form; therefore we seek to pose the minimising problem as an unconstrained problem.

Using Equations 8 and 9, it is easy to repeat the same process on $||\mathbf{x}||_1$.

$$\begin{aligned} min_x||\mathbf{x}||_1 &= min_x \sum_{j=1}^{n} |x_j| \\ &= min_x \sum_{j=1}^{n} (min_{t_j}(-t_j \leq x_j \leq t_j)) \end{aligned} \tag{17}$$

Now we can reconstruct the problem as Equation 18.

$$\begin{aligned} &min_{x,u}||\mathbf{Ax} - \mathbf{b}||_2^2 + \lambda \sum_{j=1}^{n} u_j \\ &\text{subject to } \; -t_j \leq x_j \leq t_j \text{ for } i = 1, 2, ..., m \end{aligned} \tag{18}$$

Finally, we propose a barrier function, that will go to ∞ when the constraints are violated, to pose the problem as an unconstrained one, as seen in Equation 19.

$$\begin{aligned} &min_{x,u}\Phi_t(\mathbf{x}, \mathbf{u}) \\ &= min_{x,u} t||\mathbf{Ax} - \mathbf{b}||_2^2 + t\lambda \sum_{j=1}^{n} u_j + \phi(\mathbf{x}, \mathbf{u}) \end{aligned}$$

where

$$\phi(\mathbf{x}, \mathbf{u}) = -\sum_{j=1}^{n} log(u_i - x_i) + log(u_i + x_i) \tag{19}$$

3.2 B)

Unlike Section 2, we will be using the Newton method, which is a second-order descent method, utilising both the gradient and the hessian of the unconstrained problem. The first-order partial derivatives are defined in Equations 20 and 21.

$$\frac{\partial \Phi_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}} = 2t(\mathbf{A}.T\mathbf{Ax} - \mathbf{A}.T\mathbf{b}) + \frac{1}{\mathbf{u} - \mathbf{x}} - \frac{1}{\mathbf{u} + \mathbf{x}} \tag{20}$$

$$\frac{\partial \Phi_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}} = t\lambda + \frac{1}{\mathbf{u} - \mathbf{x}} - \frac{1}{\mathbf{u} + \mathbf{x}} \tag{21}$$

Now using the above equation we will calculate the hessian as shown in Equation 22.

$$\mathbf{H} = \begin{pmatrix} \frac{\partial^2 \Phi_t(\mathbf{x},\mathbf{u})}{\partial \mathbf{x}^2} & \frac{\partial^2 \Phi_t(\mathbf{x},\mathbf{u})}{\partial \mathbf{x} \partial \mathbf{u}} \\ \frac{\partial^2 \Phi_t(\mathbf{x},\mathbf{u})}{\partial \mathbf{u} \partial \mathbf{x}} & \frac{\partial^2 \Phi_t(\mathbf{x},\mathbf{u})}{\partial \mathbf{u}^2} \end{pmatrix} \tag{22}$$

$$\mathbf{H_{11}} = \frac{\partial^2 \Phi_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}^2} = 2t\mathbf{A}^T\mathbf{A}Diag\left(\frac{1}{(\mathbf{u}+\mathbf{x})^2} + \frac{1}{(\mathbf{u}-\mathbf{x})^2}\right) \tag{23}$$

$$\mathbf{H_{12}} = \mathbf{H_{21}} = \frac{\partial^2 \Phi_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}\partial \mathbf{u}} = Diag\left(\frac{1}{(\mathbf{u}+\mathbf{x})^2} - \frac{1}{(\mathbf{u}-\mathbf{x})^2}\right) \tag{24}$$

$$\mathbf{H_{22}} = \frac{\partial^2 \Phi_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}^2} = Diag\left(\frac{1}{(\mathbf{u}+\mathbf{x})^2} + \frac{1}{(\mathbf{u}-\mathbf{x})^2}\right) \tag{25}$$

3.3 C)

We aim to reconstruct $\mathbf{x_0}$ ($\in \mathcal{R}^{256}$), as shown in Figure 3, from the measurement matrix $\mathbf{A}$ ($\in \mathcal{R}^{60\times256}$), which is given, and the noisy measurement of $\mathbf{x_0}$ $\mathbf{b} = \mathbf{A}\mathbf{x_0} + \epsilon$ ($\in \mathcal{R}^{60}$), where $\epsilon = UniformDist(-0.005, +0.005)$.

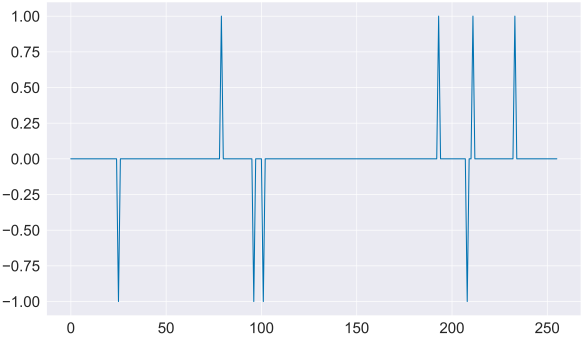


Figure 3: Plot of the signal x_0.

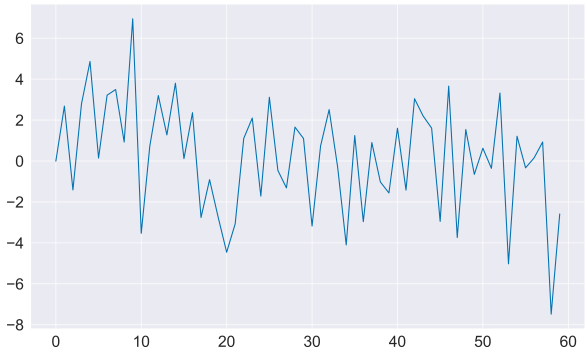


Figure 4: Plot of the noisy measurement of $\mathbf{x_0}$ ($\mathbf{b}$).

Now following the Newton interior point method as shown in Pseudo-code 2, we will reconstruct the original signal from the noisy measurements.

Algorithm 2 The Newton Method

Require: $\tilde{\mathbf{x}_0}$ such that $\tilde{\mathbf{A}}\tilde{\mathbf{x}} - \tilde{\mathbf{b}} \leq 0$
$\beta = 0.8$
Evaluate $\Phi_t(\tilde{\mathbf{x}})$
Evaluate $\nabla(\Phi_t(\tilde{\mathbf{x}}))$
Evaluate $Hessian(\Phi_t(\tilde{\mathbf{x}}))$
$Step = -Hessian(\Phi_t(\tilde{\mathbf{x}}))^{-1}\nabla(\Phi_t(\tilde{\mathbf{x}}))$
while $\nabla(-\Phi_t(\tilde{\mathbf{x}}))^T Step/2 \geq \epsilon$ **do**
    $t_{step} = 1$
    while $\Phi_t(\tilde{\mathbf{x}} + t_{step}Step) == '$ $infeasible'$ **do**
        $t_{step} = beta \times t_{step}$
    $\tilde{\mathbf{x}} + t_{step}Step$
    Evaluate $\Phi_t(\tilde{\mathbf{x}})$
    Evaluate $\nabla(\Phi_t(\tilde{\mathbf{x}}))$
    Evaluate $Hessian(\Phi_t(\tilde{\mathbf{x}}))$
    $Step = -Hessian(\Phi_t(\tilde{\mathbf{x}}))^{-1}\nabla(\Phi_t(\tilde{\mathbf{x}}))$

This algorithm uses the Newton decrement to define the stopping criterion, with $\epsilon = 0.1$.

When reconstructing the signal we used $\lambda = 0.01\lambda_{max}$, where $\lambda_{max} = ||2\mathbf{A}^T\mathbf{b}||_\infty$, and looped through the algorithm 5 times with $t = 1, 10, 100, 1000, 10000$, to solve as close to the constraints as possible (see 2.3). This took 5 seconds to run and yielded Figure 5.

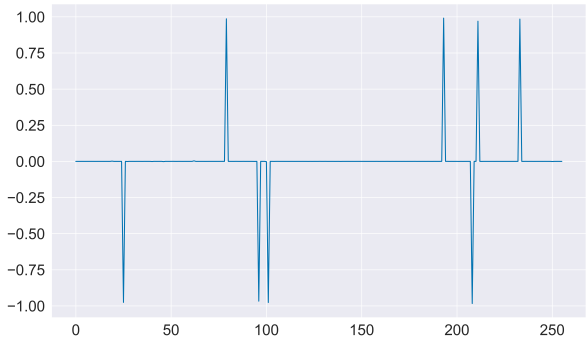


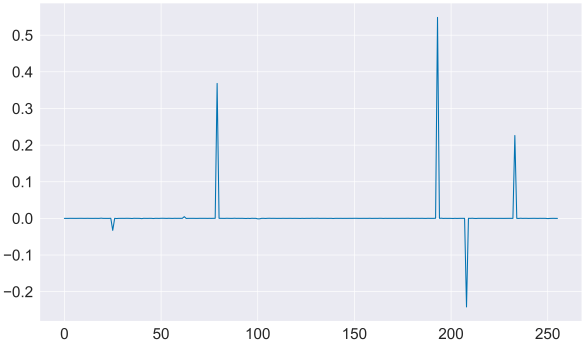
Figure 5: Plot of the reconstructed signal with: $\lambda = 0.01\lambda_{max}$, $t = 1, 10, 100, 1000, 10000$ and $\epsilon = 0.1$.

We observe the reconstruction is extremely accurate, with the peaks just falling short of their original magnitude of 1.

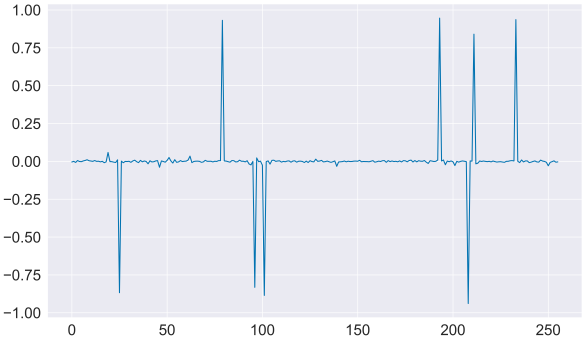
3.4 D)

We now repeat the above, but for $t = 1, 10, 100$ and with λ varied. We reduce the number of different t that we loop through so the variations due to λ are more noticeable.

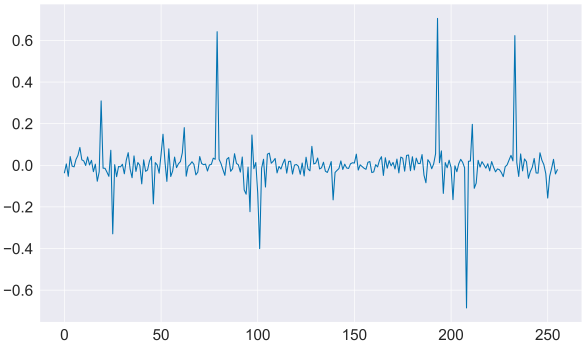
From Figure 6a we see that if the regulariser has a larger amount of influence, it keeps all the entries of $\mathbf{x}$ small, hence the signal values are greatly reduced and some peaks aren't even visible; decreasing λ allows the peaks to take larger values, as seen in Figure 6b. But from Figure 6c we see that decreasing it too much has a negative drawback too; when lambda is small, it is more rewarding for the algorithm to minimise the residuals than it is to regularise $\mathbf{x}$. As a result, the signal as a whole starts to get noisier because focusing on minimising the residuals means that the algorithm will begin trying to reproduce the noise present in $\mathbf{b}$ due to the addition of the uniform distribution.



(a) $\lambda = 0.5\lambda_{max}$



(b) $\lambda = 0.01\lambda_{max}$



(c) $\lambda = 0.001\lambda_{max}$

Figure 6: Plots of reconstructed signals with varying λ and $t = 1, 10, 100$.

4 Appendix

4.1 References

[1] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," in IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pp. 67-82, April 1997, doi: 10.1109/4235.585893.

[2] Huangfu, Q., Galabova, I., Feldmeier, M., and Hall, J. A. J. "HiGHS - high performance software for linear optimization." https://highs.dev/

4.2 Code

4.2.1 Exercise 1

```
import numpy as np
from numpy.linalg import inv
import timeit
import cvxpy as cp

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(rc={'figure.figsize':(14,8.27)}, font_scale=2)
sns.set_palette('colorblind')

Ab = []
for i in range(5):
    A = np.genfromtxt('2023-data/A' + str(i) + '.csv', delimiter=",")
    b = np.genfromtxt('2023-data/b' + str(i) + '.csv', delimiter=",")
    Ab.append((A, b))

def lp(data, norm, method, iterations):
    times = np.zeros(5)
    residuals = [None] * 5
    opt = np.zeros(5)
    low_bound = np.zeros(5)
    for iter in range(iterations):
        for i, (A,b) in enumerate(data):
            m, n = A.shape
            if norm == 'l1':
                x = cp.Variable(n+m)
                c_til = np.zeros((n + m,))
                c_til[-m:] = 1
                b_til = np.hstack((b, -b))
                I = np.identity(m)
                A_til = np.vstack((np.hstack((A, -I)), np.hstack((-A, -I))
                    ))
            elif norm == 'linf':
                x = cp.Variable(n+1)
                c_til = np.zeros((n + 1,))
                c_til[-1:] = 1
                b_til = np.hstack((b, -b))
                A_til = np.vstack((np.hstack((A, -np.ones((m, 1)))), np.
                    hstack((-A, -np.ones((m, 1))))))
            else:
                print('Error: incorrect norm')
                return

            start_time = timeit.default_timer()
            prob = cp.Problem(cp.Minimize(c_til.T@x),
                              [A_til @ x <= b_til])
            prob.solve(solver=cp.SCIPY, scipy_options={"method": method})
            time = timeit.default_timer() - start_time

            lam = cp.Variable(2*m)
```

```
49              dual_prob = cp.Problem(cp.Maximize(-b_til.T@lam),
50                                     [A_til.T @ lam == -c_til,   lam >= np.zeros((2*
                                           m,))])
51              dual_prob.solve(solver=cp.SCIPY, scipy_options={"method":
                    method})
52
53              opt[i] = prob.value
54              low_bound[i] = dual_prob.value
55              residuals[i] = A @ x.value[:n] - b
56              times[i] += time / iterations
57
58      return opt, times, residuals, low_bound
59
60  ## $l_1$ norm
61
62  ### i) Simplex Method
63
64  l1_smp_opt, l1_smp_times, l1_smp_residuals, l1_smp_low = lp(Ab, 'l1', '
        highs-ds', 1)
65
66  print(l1_smp_opt)
67  print(l1_smp_low)
68  print(l1_smp_times)
69
70  ### ii) Interior Point Method
71
72  l1_ipm_opt, l1_ipm_times, l1_ipm_residuals, l1_ipm_low = lp(Ab, 'l1', '
        highs-ipm', 1)
73
74  print(l1_ipm_opt)
75  print(l1_ipm_low)
76  print(l1_ipm_times)
77
78  plt.hist(l1_ipm_residuals[-1], bins=40)
79  plt.xlabel('Residual')
80  plt.xlim(-2.5,2.5)
81  plt.savefig('Figures/l1residual.png', format="png", dpi=800, bbox_inches="
        tight")
82  plt.show()
83  plt.close()
84
85  ## $l_\infty$ norm
86
87  ### i) Simplex Method
88
89  linf_smp_opt, linf_smp_times, linf_smp_residuals, linf_smp_low = lp(Ab, '
        linf', 'highs-ds', 1)
90
91  print(linf_smp_opt)
92  print(linf_smp_low)
93  print(linf_smp_times)
94
95  ### ii) Interior Point LP
96
97  linf_ipm_opt, linf_ipm_times, linf_ipm_residuals, linf_ipm_low = lp(Ab, '
        linf', 'highs-ipm', 1)
98
99  print(linf_ipm_opt)
```

```
print(linf_ipm_low)
print(linf_ipm_times)

plt.hist(linf_ipm_residuals[-1], bins=40)
plt.xlabel('Residual')
plt.savefig('Figures/linfresidual.png', format="png", dpi=800, bbox_inches="tight")
plt.show()
plt.close()

## $l_2$ norm

l2_lp_times = [0] * 5
l2_lp_opt = [None] * 5
l2_lp_residuals = [None] * 5
iterations = 10
for n in range(iterations):
    for i, (A,b) in enumerate(Ab):

        start_time = timeit.default_timer()
        x = inv(A.T @ A) @ A.T @ b
        time = timeit.default_timer() - start_time

        l2_lp_residuals[i] = A @ x - b

        l2_lp_opt[i] = np.sqrt(l2_lp_residuals[i].T @ l2_lp_residuals[i])
        l2_lp_times[i] += time / iterations
```

4.2.2 Exercise 2

```
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(rc={'figure.figsize': (14, 8.27)}, font_scale=2)
sns.set_palette('colorblind')

A = np.genfromtxt('2023-data/A3.csv', delimiter=",")
b = np.genfromtxt('2023-data/b3.csv', delimiter=",")

m, n = A.shape
b = b.reshape((m, 1))
c_til = np.zeros((n + m, 1))
c_til[-m:] = 1
b_til = np.vstack((b, -b))
I = np.identity(m)
A_til = np.vstack((np.hstack((A, -I)), np.hstack((-A, -I))))

def f(A, b, c, x, t):
    if (A @ x - b <= 0).all():
        function = t * c.T @ x - np.sum(np.log(b - A @ x))
    else:
        function = 'infeasible'
    return function

def grad_f(x, A, b, c, t):
    grad = t * c + A.T @ (1 / (b - A @ x))
    return grad
```

```
29
30 def backtracking(x, A, b, c, grad, t, alpha = 0.2, beta= 0.3):
31     t_b = 1
32     while f(A, b, c, x - t_b * grad, t) == 'infeasible':
33         t_b = beta*t_b
34     while f(A, b, c, x - t_b * grad, t) > f(A, b, c, x, t) - alpha * t_b *
           grad.T @ grad:
35         t_b = beta * t_b
36     x = x - t_b * grad
37     return x
38
39 opt_x_t = []
40 x = c_til
41 costs = []
42 for t, epsilon in [(1, 0.001), (10, 0.1), (100, 1)]:
43     iteration = 0
44     grad = grad_f(x, A_til, b_til, c_til, t)
45     while grad.T @ grad >= epsilon:
46         x = backtracking(x, A_til, b_til, c_til, grad, t)
47         grad = grad_f(x, A_til, b_til, c_til, t)
48         func = f(A_til, b_til, c_til, x, t)
49         costs.append(func)
50         iteration += 1
51         if iteration % 500 == 0:
52             print(f'Iteration:{iteration}, func: {func}')
53         if iteration % 2000 == 0:
54             residual = A @ x[:n] - b
55             print(np.sum(np.sqrt(residual*residual)))
56             opt_x_t.append(x)
57
58 residual = A @ opt_x_t[-1][:n] - b
59
60 costs = np.asarray(costs)
61 costs = costs.reshape((costs.shape[0]))
62
63 plt.plot(costs)
64 plt.ylabel('Cost')
65 plt.xlabel('Iterations')
66 plt.xlim(-5, 17000)
67 plt.savefig('Figures/ex2_costViterations.png', format="png", dpi=800,
       bbox_inches="tight")
68 plt.show()
69 plt.close()
```

4.2.3 Exercise 3

```
1  import numpy as np
2  from numpy.linalg import inv
3
4  import matplotlib.pyplot as plt
5  import seaborn as sns
6  sns.set(rc={'figure.figsize': (14, 8.27)}, font_scale=2)
7  sns.set_palette('colorblind')
8
9  def f(A, b, x_til, lam, t):
10     n =256
11     x, u = x_til[:n], x_til[n:]
12     if (u-x < 0).any() or (u+x < 0).any():
```

```
13          return 'infeasible'
14      residual = A @ x - b
15      return t*residual.T @ residual + t*lam*np.sum(u) - np.sum(np.log(u-x) +
            np.log(u+x))
16
17  ## Gradient
18
19  def grad_f_x(A, b, x_til, t):
20      n =256
21      x, u = x_til[:n], x_til[n:]
22      gradient = 2*t*(A.T @ A @ x - A.T @ b) + 1/(u - x) - 1/(u + x)
23      return gradient
24
25  def grad_f_u(x_til, t, lam):
26      n =256
27      x, u = x_til[:n], x_til[n:]
28      gradient = t*lam - 1/(u - x) - 1/(u + x)
29      return gradient
30
31  def grad_f(A, b, x_til, t, lam):
32      return np.concatenate((grad_f_x(A, b, x_til, t), grad_f_u(x_til, t, lam
            )))
33
34  ## Hessian
35
36  def grad_f_xx(A, x_til, t):
37      n =256
38      x, u = x_til[:n], x_til[n:]
39      gradient = 2*t*A.T @ A + np.diag(1/np.square(u + x) + 1/np.square(u - x
            ))
40      return gradient
41
42  def grad_f_xu(x_til):
43      n =256
44      x, u = x_til[:n], x_til[n:]
45      gradient = np.diag(1/np.square(u + x) - 1/np.square(u - x))
46      return gradient
47
48  def grad_f_uu(x_til):
49      n =256
50      x, u = x_til[:n], x_til[n:]
51      gradient = np.diag(1/np.square(u + x) + 1/np.square(u - x))
52      return gradient
53
54  def hessian_f(A, x_til, t):
55      h_11 = grad_f_xx(A, x_til, t)
56      h_12 = grad_f_xu(x_til)
57      h_22 = grad_f_uu(x_til)
58      return np.block([[h_11, h_12],[h_12, h_22]])
59
60  A = np.genfromtxt('2023-data/A.csv', delimiter=",")
61  x_0 = np.genfromtxt('2023-data/x.csv', delimiter=",")
62
63  b = A @ x_0 + np.random.uniform(-0.005, 0.005, A.shape[0])
64
65  plt.plot(x_0)
66  plt.show()
67
```

```
68  def N_decrement(A, b, x_til, t, lam):
69      jacobian = grad_f(A, b, x_til, t, lam)
70      return (jacobian.T @ np.linalg.inv(hessian_f(A, x_til, t)) @ jacobian)
            /2
71  #
72  def Newtons_method(x_til_start, epsilon, t, lam_factor):
73      lam_max = np.linalg.norm(2 * A.T @ b, ord=np.inf)
74      lam = lam_factor * lam_max
75      beta = 0.8
76      iterations = 0
77      x_til = x_til_start
78      func = f(A, b, x_til, lam, t)
79      costs = [func]
80      grad = grad_f(A, b, x_til, t, lam)
81      hessian = hessian_f(A, x_til, t)
82      step = - np.linalg.inv(hessian) @ grad
83
84      print(f't: {t}, Iteration: {iterations}, func: {func}')
85
86      while (grad.T @ - step)/2 >= epsilon:
87          t_b = 1
88          while f(A, b, x_til + t_b * step, lam, t) == 'infeasible':
89              t_b = t_b * beta
90          x_til = x_til + t_b * step
91          iterations += 1
92          grad = grad_f(A, b, x_til, t, lam)
93          hessian = hessian_f(A, x_til, t)
94          step = - np.linalg.inv(hessian) @ grad
95          func = f(A, b, x_til, lam, t)
96          costs.append(func)
97          print(f't: {t}, Iteration: {iterations}, func: {func}')
98      return x_til, costs
99
100 x_til = np.concatenate((np.zeros(256), np.ones(256)))
101 ep = 0.1
102 for lam_factor in [0.01]:
103     costs = []
104     for t in [1, 10, 100, 1000]:
105         x_til, cost = Newtons_method(x_til, ep, t, lam_factor)
106         costs = costs + cost
107
108 plt.plot(x_til_opt[:256])
109 plt.show()
110
111 x_til_opt = np.concatenate((np.zeros(256), np.ones(256)))
112 ep = 0.1
113 x_opt = []
114 for lam_factor in [10, 1, 0.5, 0.1, 0.05,  0.01, 0.005]:
115     for t in [1, 10]:
116         x_til_opt, cost = Newtons_method(x_til_opt, ep, t, lam_factor)
117     x_opt.append(x_til_opt[:256])
118
119 plt.plot(x_opt[4])
120 plt.show()
```