4M17 Coursework Assignment 1

Norm Approximation

Candidate Number: 5746G

Section 1

(a) Let $\|\cdot\|$ be a norm on $\mathbb{R}^m$. For a real number $p \geq 1$, the l_p-norm of a variable $y \in \mathbb{R}^m$ is a scalar value defined as:

$$\|y\|_p = \left(\sum_{i=1}^{m} |y_i|^p \right)^{1/p}. \tag{1}$$

The forms of the l_1, l_2 and l_∞-norms follow from this definition:

$$\|y\|_1 = \sum_{i=1}^{m} |y_i|, \quad \|y\|_2 = \left(\sum_{i=1}^{m} |y_i|^2 \right)^{1/2}, \quad \|y\|_\infty = \max_{1 \leq i \leq m} |y_i|.$$

For the simplest, unconstrained *norm approximation problem* of the form:

$$\text{minimise} \quad \|Ax - b\|_2, \tag{2}$$

where $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$ are given as problem data, $x \in \mathbb{R}^n$ is the variable, and $\|\cdot\|$ is a norm on $\mathbb{R}^m$, the *residual vector* r is defined as:

$$r = Ax - b.$$

The *least-squares approximation problem* is obtained by squaring the l_2-norm of the objective:

$$\text{minimise} \quad \|Ax - b\|_2^2 = r_1^2 + r_2^2 + \cdots + r_m^2, \tag{3}$$

where the objective minimises the *sum of squares* of residuals. Such problem can be expressed as an optimisation problem with a convex quadratic function:

$$f(x) = x^T A^T A x - 2b^T A x + b^T b$$

which, by the first-order optimality condition, is minimised at the point where its derivative is zero:

$$\nabla f(x) = 2A^T A x - 2A^T b = 0 \tag{4}$$

$$(A^T A)x = A^T b. \tag{5}$$

Assuming, without loss of generality, that $m \geq n$ and the columns of A are independent, such that A is full-rank and $\det(A) \neq 0$, then:

$$\det(A^T A) = \det(A^T)\det(A) = \det(A)\det(A) \neq 0$$

guaranteeing $A^T A$ is full-rank and thus invertible. The unique solution of eq. (5), and thus the least-squares approximation problem, is:

$$x = \left(A^T A\right)^{-1} A^T b. \tag{6}$$

(b) The norm approximation problem under the l_1-norm minimises the sum of absolute residuals:

$$\text{minimise} \quad \|Ax - b\|_1 = \sum_{i=1}^{m} |r_i|. \tag{7}$$

Using the identity:

$$\begin{aligned} |z_i| &= \max\{-z_i, z_i\} \\ &= \min\{t_i \mid z_i \leq t_i, -z_i \leq t_i\}, \end{aligned} \tag{8}$$

the l_1-norm approximation problem can be cast as a *linear program* (LP):

$$\begin{array}{ll} \underset{x,t_i}{\text{minimize}} & \sum_{i=1}^{m} t_i \\ \text{subject to} & -t_i \leq a_i^T x - b_i \leq t_i \quad i = 1, \ldots, m \end{array} \tag{9}$$

which expressed in standard form is:

$$\begin{array}{ll} \underset{\tilde{x}}{\text{minimize}} & \tilde{c}^T \tilde{x} \\ \text{subject to} & \tilde{A}\tilde{x} \leq \tilde{b} \end{array} \tag{10}$$

where:

$$\tilde{x} = \begin{bmatrix} x \\ t \end{bmatrix}, \quad \tilde{c} = \begin{bmatrix} \mathbf{0}_n \\ \mathbf{1}_m \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & -I \\ -A & -I \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} b \\ -b \end{bmatrix}$$

for $\tilde{x} \in \mathbb{R}^{(n+m)}$, $\tilde{c} \in \mathbb{R}^{(n+m)}$, $\tilde{A} \in \mathbb{R}^{2m \times (n+m)}$ and $\tilde{b} \in \mathbb{R}^{2m}$.

Using the l_∞-norm, the approximation problem:

$$\text{minimise} \quad \|Ax - b\|_\infty = \max\left\{|r_1|, \ldots, |r_m|\right\} \tag{11}$$

minimises the maximum absolute value residual. Employing the identity:

$$\begin{aligned} \max_i |z_i| &= \max\{z_i, -z_i\} \\ &= \min\left\{t \mid z_i \leq t, -z_i \leq t\right\} \\ &= \min\left\{t \mid -t \leq z_i \leq t\right\}, \end{aligned} \tag{12}$$

the l_∞-norm approximation problem can be cast as an LP:

$$\begin{array}{ll} \underset{x,t}{\text{minimize}} & t \\ \text{subject to} & -t \leq a_i^T x - b_i \leq t \quad i = 1, \ldots, m \end{array} \tag{13}$$

which again can be expressed in standard form, as in problem (10), where now:

$$\tilde{x} = \begin{bmatrix} x \\ t \end{bmatrix}, \quad \tilde{c} = \begin{bmatrix} \mathbf{0}_n \\ 1 \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} A & -\mathbf{1}_m \\ -A & -\mathbf{1}_m \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} b \\ -b \end{bmatrix}$$

for $\tilde{x} \in \mathbb{R}^{(n+1)}$, $\tilde{c} \in \mathbb{R}^{(n+1)}$, $\tilde{A} \in \mathbb{R}^{2m \times (n+1)}$ and $\tilde{b} \in \mathbb{R}^{2m}$ respectively.

(c) For the five pairs of problem data (`A1`, `b1`),...,(`A5`, `b5`) with $n = 16, 64, 256, 512, 1024$ and $m = 2n$, the MATLAB LP solver `linprog`[1] was used to solve the l_1 and l_∞-norm LP problems outlined in (9) and (13) respectively. The l_2-norm for the problem in (3) was minimised through use of the `lsqminnorm`[2] function. Values of the minimised norms and runtime of algorithms are given in Table 1.

Table 1: Values of the minimised l_1, l_2, and l_∞-norms $\|Ax - b\|$, and runtime of algorithms.

Data set	$\|Ax-b\|_1$	$\|Ax-b\|_2$	$\|Ax-b\|_\infty$	l_1 runtime /s	l_2 runtime /s	l_∞ runtime /s
(`A1`, `b1`)	9.06	2.32	0.585	0.0349	5.70×10^{-4}	0.0262
(`A2`, `b2`)	42.4	5.52	0.707	0.0588	0.0029	0.0584
(`A3`, `b3`)	141	9.18	0.593	2.92	0.0177	1.42
(`A4`, `b4`)	309	13.9	0.622	30.2	0.0747	14.0
(`A5`, `b5`)	576	18.7	0.602	424	0.546	174

These results demonstrate that for a given norm approximation problem, the value of the minimised residual is dependent on the choice of norm and size of data set: the minimised residual of the l_1-norm approximation problem scales linearly with the dimensionality of the input data, whilst for the l_2-norm, the residuals scale with the square root of the size of data. This is attributed to the fact that the l_1 and l_2-norms penalise the absolute and squared values of residuals respectively. Penalising only the absolute value of the largest magnitude residual, the minimised l_∞-norm residual stays roughly constant, irrespective of the size of the data set.

[1]MATLAB `linprog`: `https://uk.mathworks.com/help/optim/ug/linprog.html`
[2]MATLAB `lsqminnorm`: `https://uk.mathworks.com/help/matlab/ref/lsqminnorm.html`

Likewise, depending on the norm in which the residuals are minimised, the runtime of each algorithm scales differently with the size of the data set. The least-squares approximation problem has significantly smaller runtime than the l_1 and l_∞-norm approximation problems. This is due to the fact it is an unconstrained optimisation problem with an analytic solution, as given in eq. (5). The MATLAB `lsqminnorm` function solves the least-squares problem using the QR factorisation $A = QR$:

$$\begin{aligned} x = \left(A^T A\right)^{-1} A^T b &= \left((QR)^T (QR)\right)^{-1} (QR)^T b \\ &= \left(R^T Q^T Q R\right)^{-1} R^T Q^T b \\ &= \left(R^T R\right)^{-1} R^T Q^T b \\ &= R^{-1} R^{-T} R^T Q^T b \\ &= R^{-1} Q^T b \end{aligned} \tag{14}$$

The algorithmic complexity of performing the QR factorisation of $A \in \mathbb{R}^{m \times n}$ is $2mn^2$ flops [2]. The matrix-vector product $d = Q^T b$ costs $2mn$ flops, and solving $Rx = Q^T b$ by back-substitution n^2 flops. Asymptotically, the $2mn^2$ term dominates, giving an overall complexity of $2mn^2$ flops. Since for the problem data $m = 2n$, this complexity can be simplified to $4n^3$ flops. This scaling of runtime with the size of data set is observed in the l_2 runtime column in Table 1.

The l_1 and l_∞-norm approximation problems, on the other hand, are constrained optimisation problems with no analytic solutions. The MATLAB LP solver `linprog` was used with the default option of the *dual-simplex* algorithm, which performs a simplex algorithm on the *dual-problem* [7]. In the worst-case scenario, every vertex of the simplex must be visited. With $2m$ inequality constraints, this yields an algorithmic complexity of 2^{2m} flops [5]. Recent advancements in [3], focused on applying polynomially small perturbations to the vector $\tilde{b}$, have reduced this to polynomial time complexity. However, the cost of running the LP solver `linprog` is still significantly greater than the least-squares solver.

(d) The histogram of residuals for the norm approximation problem in (2) for the fifth data pair (`A5`, `b5`) are shown for the l_1, l_2 and l_∞-norms in Figure 1. In l_p-norm approximation problems, the choice of p dictates the measure to which the i-th component of the residual r_i is penalised. Comparing l_1 and l_2-norm approximations, for $r_i = 1$, $|r_i| = |r_i|^2$, and so the two norms assign equal penalty to the residual. For small r_i, where $|r_i| \ll 1$, the penalty associated with the l_1-norm $|r_i|$ is very much smaller than that associated with the l_2-norm $|r_i|^2$. On the contrary, for large r_i, $|r_i|^2 \gg |r_i|$, hence the penalty induced by the l_2-norm is very much greater than that from the l_1-norm.

The amplitude distribution of the optimal residuals in Fig. (1) reflects this difference in relative weightings. Putting the most weight on small residuals and the least weight on large residuals, the l_1-norm approximation solution yields many more zero and small residuals, and relatively more large residuals. This means a large number of the equations are satisfied exactly, such that $a_i^T x = b_i$ for many i.

The l_2-norm approximation problem, on the other hand, puts small weight on small residuals, but significant weight on large residuals. Since there is little incentive to drive small residuals smaller, the l_2-norm approximation problem yields optimal residuals which are small, but not very small. The solution therefore solution produces mostly modest residuals, with very few larger ones.

The l_∞-norm approximation problem penalty takes only into account the component of the residual with maximum magnitude, thus giving many residuals at the positive and negative limits.

In the case of estimation or regression problems, an *outlier* is a measurement $y_i = a_i^T x + \epsilon_i$ for which the noise ϵ_i is large compared to the observation x. Such outliers result in estimates of x with a residual vector with significant components. Ideally, these outliers would be identified, and either removed from the data fitting problem, or reduced in weight when forming the estimate.

The penalty associated with l_1-norm approximation problem is least sensitive to the relative value of large residuals; the penalty growing linearly with the absolute value of r_i. Such a penalty function is termed *robust*, being much less sensitive to outliers or large errors than, for example, least-squares.

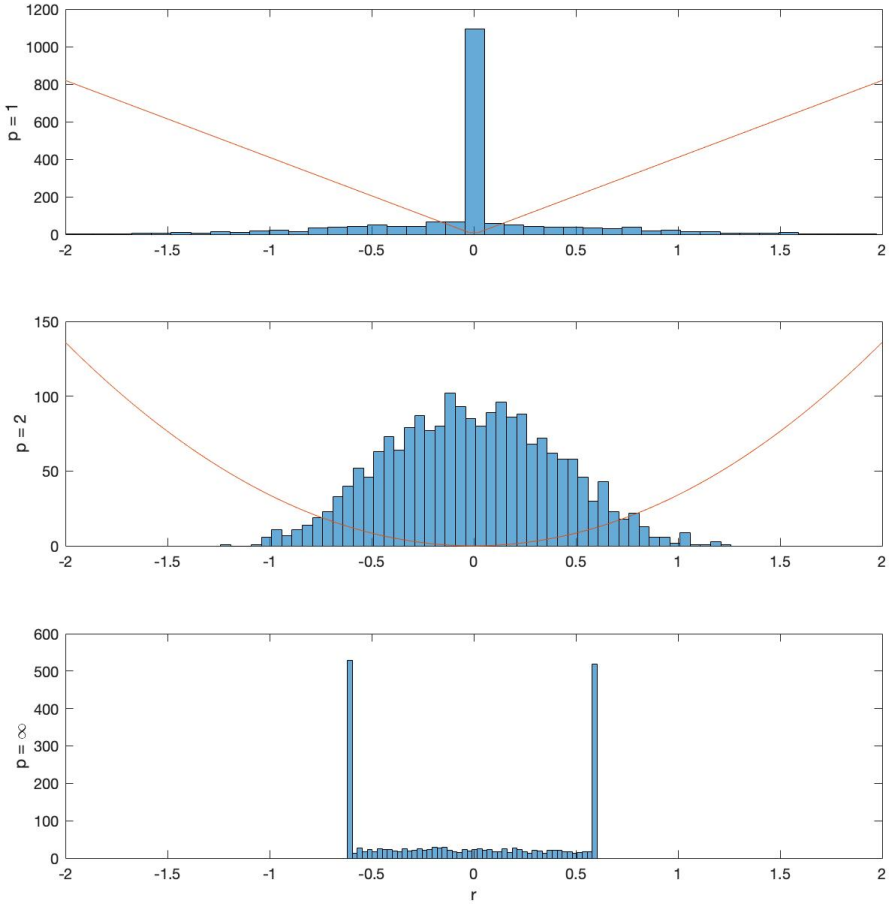


Figure 1: Histogram of residual amplitudes for the data set (`A5`, `b5`) for three norm approximation problems. Top panel: l_1-norm. Middle panel: l_2-norm. Bottom panel: l_∞-norm. The (scaled) penalty functions are also shown for reference for the l_1 and l_2-norm approximation problems.

Section 2

(a) Consider the *constrained* optimisation problem given by:

$$\begin{array}{ll} \text{minimise} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \end{array} \tag{15}$$

where f_0, f_i are convex and twice continuously differentiable functions for $i = 1, \ldots, m$. *Interior-point methods* approximate this problem as an unconstrained problem. The *central path* associated with problem (15) is the set of points $\{x^*(t) \mid t \geq 0\}$ which are the solution of:

$$\text{minimise} \quad tf_0(x) + \phi(x) \tag{16}$$

for any $t \geq 0$, where the function:

$$\phi(x) = -\sum_{i=1}^{m} \log\left(-f_i(x)\right) \tag{17}$$

with domain of the feasabile set $S = \{x \in \mathbb{R}^n \mid f_i(x) \leq 0, i = 1, \ldots, m\}$ is the *logarithmic barrier*. The log barrier is an appropriate choice of approximating function to the indicator function:

$$I_-(f_i(x)) = \begin{cases} 0 & f_i(x) \leq 0 \\ \infty & f_i(x) > 0 \end{cases} \tag{18}$$

being convex, twice differentiable and growing unbounded as $f_i(x) \to 0$, independent of the value of the positive parameter t. The parameter t sets the accuracy of the approximation: as t increases, the approximation becomes more accurate. However, a large t causes the Hessian of the objective function to vary rapidly near the boundaries of S, making minimisation by Newton's method difficult.

The explicit form of problem (16) corresponding to the LP formulation of the l_1-norm approximation problem outlined in (10) is:

$$\begin{array}{ll} \underset{\tilde{x}}{\text{minimize}} & f_0(\tilde{x}) = \tilde{c}^T\tilde{x} \\ \text{subject to} & f_i(\tilde{x}) = \tilde{a}_i^T\tilde{x} - \tilde{b}_i \leq 0 \quad i = 1, \ldots, 2m \end{array} \tag{19}$$

where $\tilde{a}_1^T, \ldots, \tilde{a}_{2m}^T$ are the rows of $\tilde{A}$, giving the logarithmic barrier function:

$$\phi(\tilde{x}) = -\sum_{i=1}^{2m} \log\left(\tilde{b}_i - \tilde{a}_i^T\tilde{x}\right)$$

with $\mathbf{dom}\ \phi = \left\{\tilde{x} \in \mathbb{R}^{2m} \mid \tilde{a}_i^T\tilde{x} - \tilde{b}_i \leq 0, i = 1, \ldots, 2m\right\}$. Denoting the objective function in problem (16) as J, the gradient of the cost is:

$$\begin{aligned} \nabla J(\tilde{x}) &= t\nabla f_0(\tilde{x}) + \sum_{i=1}^{2m} \frac{1}{-f_i(\tilde{x})}\nabla f_i(\tilde{x}) \\ &= t\tilde{c} + \sum_{i=1}^{2m} \frac{1}{\tilde{b}_i - \tilde{a}_i^T\tilde{x}}\tilde{a}_i. \end{aligned} \tag{20}$$

By the first-order optimality condition, any point on the central path $\tilde{x} = \tilde{x}^*(t)$ must satisfy:

$$t\tilde{c} + \sum_{i=1}^{2m} \frac{1}{\tilde{b}_i - \tilde{a}_i^T\tilde{x}}\tilde{a}_i = 0. \tag{21}$$

The geometric interpretation of this condition is that the gradient of $\nabla\phi(\tilde{x}^*(t))$, which is perpendicular to the level sets of ϕ through the central point $\tilde{x}^*(t)$, is parallel to $-\tilde{c}$.

(b) The problem in (19) is solved by applying a first order gradient descent method with *backtracking line search* for $t = 1$ in the case corresponding to the data set (`A3`, `b3`). The initial starting point $\tilde{x}_0$ is found by finding the solution to the normal equations:

$$\tilde{A}^T\tilde{A}\tilde{x}_0 = \tilde{A}^T(\tilde{b} - \lambda\mathbf{1}_{2m}), \tag{22}$$

where λ is small and positive, thus ensuring that the point $\tilde{x}_0$ is in the domain of the feasible set $\tilde{S} = \{\tilde{x} \in \mathbb{R}^{2m} \mid f_i(\tilde{x}) = \tilde{b}_i - \tilde{a}_i^T\tilde{x} \leq 0, i = 1, \ldots, 2m\}$. Denoting the optimal value $\inf_{\tilde{x}} J(\tilde{x}) = J(\tilde{x}^*(t)) = p^*$, a conceptual stopping criterion can be determined, in which the algorithm is terminated when the gradient of J at $\tilde{x}$ is small enough such that the difference between $J(\tilde{x})$ and p^* is small. Given the Hessian of the objective function:

$$\nabla^2 J(\tilde{x}) = \sum_{i=1}^{2m} \frac{1}{(\tilde{b}_i - \tilde{a}_i^T\tilde{x})^2}\tilde{a}_i\tilde{a}_i^T, \tag{23}$$

such a stopping criterion is defined in the form:

$$\|\nabla^2 J(\tilde{x})\|_2 \leq \eta, \tag{24}$$

where η is small and positive. Provided η is chosen small enough, then $J(\tilde{x}) - p^* \leq \epsilon$, where ϵ is some positive tolerance. Assuming the objective function to be *strongly convex* on $\tilde{S}$, it is shown in [1] that for constants k and K, where:

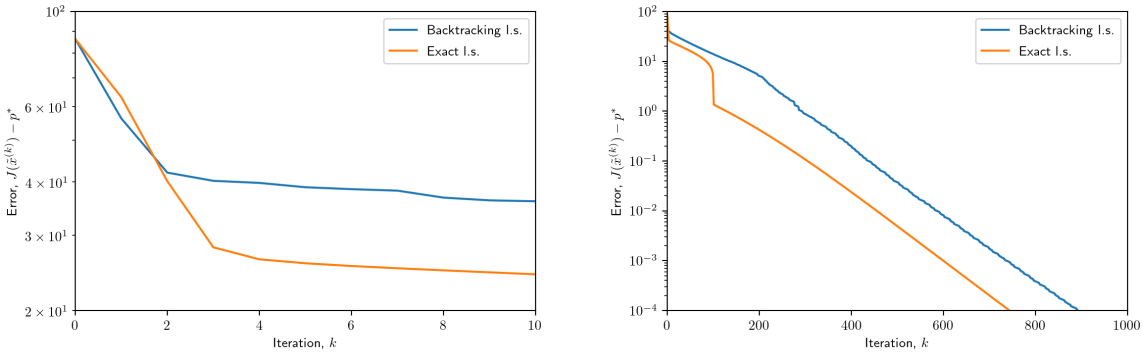
$$kI \preceq \nabla^2 J(\tilde{x}) \preceq KI \tag{25}$$

for all $\tilde{x} \in \tilde{S}$, then η must be chosen to be smaller than $(k\epsilon)^{1/2}$ (very likely) to guarantee $J(\tilde{x}) - p^* \leq \epsilon$.

With backtracking line search parameters of $\alpha = 0.1$ and $\beta = 0.4$, and stopping threshold $\eta = 10^{-3}$, the algorithm converged in 1229 iterations to a minimised l_1-norm of 168.3 (4 s.f.).

(c) The backtracking line search from (a) is compared to an *exact line search*, in which the step size s is chosen to minimise the objective function J along the ray $\{\tilde{x} + s\Delta\tilde{x} | s \geq 0\}$:

$$s = \underset{s \geq 0}{\operatorname{argmin}}\ J(\tilde{x} + s\Delta\tilde{x}). \tag{26}$$



(a) **(b)**

Figure 2: Panel (a) plots the variation of the error $J(\tilde{x}^{(k)}) - p^*$ with the first 10 iterations for both backtracking and exact line search. Panel (b) plots the same error over 1000 iterations.

Practically, this is implemented by approximating the objective function by it's second-order Taylor expansion:

$$J(\tilde{x} + s\Delta\tilde{x}) = J(\tilde{x}) + s\nabla J(\tilde{x})^T \Delta\tilde{x} + \frac{1}{2}s^2 \Delta\tilde{x}^T \nabla^2 J(\tilde{x}) \Delta\tilde{x} + \mathcal{O}(s^3), \tag{27}$$

and finding the s which 'loosely' minimises J in the direction by the first-order opitmality condition:

$$\begin{aligned} \frac{\partial J(\tilde{x} + s\Delta\tilde{x})}{\partial s} &\approx 0 \\ \nabla J(\tilde{x})^T \Delta\tilde{x} + s\Delta\tilde{x}^T \nabla^2 J(\tilde{x}) \Delta\tilde{x} &= 0 \\ s &= -\frac{\nabla J(\tilde{x})^T \Delta\tilde{x}}{\Delta\tilde{x}^T \nabla^2 J(\tilde{x}) \Delta\tilde{x}}. \end{aligned} \tag{28}$$

The variation in error with number of iterations for backtracking line search with parameters $\alpha = 0.1$ and $\beta = 0.4$ is shown in Figure 2. The convergence of this algorithm is said to be *linear*, since the error lies below a line on a log-linear plot of error versus iteration number [1]. Figure 2(a) shows a fairly rapid linear convergence for the first 2 iterations, followed by slower linear convergence. Figure 2(b) reveals that this slower convergence continued for the remaining 900 iterations. The initial convergence rate is around a factor of 0.7; the remaining iterations converge at a slower rate of around 0.99 per iteration. Overall, the error is reduced by a factor of around 10^6 in 900 iterations, giving an average error reduction by a factor of around $10^{-6/900} \approx 0.985$ per iteration.

Figure 2(b) shows the convergence of the gradient descent method with an exact line search to again be approximately linear, with an overall average error reduction by a factor of $10^{-6/750} \approx 0.982$ per iteration. This is only a marginal improvement over the backtracking line search.

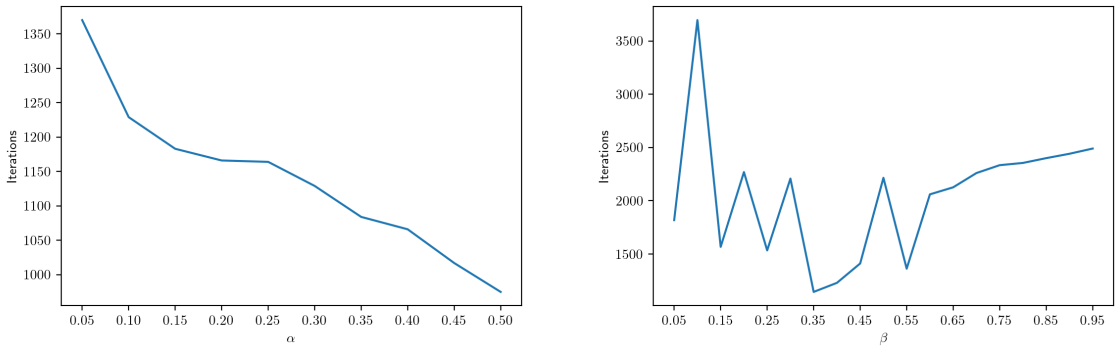
In this case, the Hessian has a closed analytic form, and so the cost of computing the exact step size from eq. (28) is similar to the cost of computing the search direction $\Delta\tilde{x} = -\nabla J(\tilde{x})$ itself. Providing a negligible performance gain, there is little advantage in implementing an exact line search for this problem, other than for the academic purpose of comparison with backtracking line search.

The *condition number* of the Hessian is defined as the ratio of its largest eigenvalue to its smallest eigenvalue, and has an upper bound of K/k [1]:

$$\kappa(\nabla^2 J(\tilde{x})) = \frac{\lambda_{\max}\left(\nabla^2 J(\tilde{x})\right)}{\lambda_{\min}\left(\nabla^2 J(\tilde{x})\right)} \leq \frac{K}{k}. \tag{29}$$

$J(\tilde{x}^{(k)})$ converges to p^* at least as fast as a geometric series with exponent that has a dependency on the condition number bound K/k. Consequently, the rate of convergence of gradient descent methods are highly dependent on the condition number of the Hessian.

The Hessian detailed in eq. (23) corresponding to the l_1-norm approximation problem (19) has a condition number of 768, and so is moderately well conditioned. Despite this fact, convergence is still very slow. This highlights the main disadvantage of gradient descent as an optimisation approach: convergence rates are slow unless the condition number is small (typically, less than 100).



(a) **(b)**

Figure 3: Panel (a) plots the number of iterations for convergence of gradient descent with backtracking line search for varying α with fixed $\beta = 0.4$. Panel (b) plots the number of iterations for convergence with fixed $\alpha = 0.1$ and varying β.

The effect of varying the backtracking parameters $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$ was investigated by establishing the number of iterations required for the convergence criterion in eq. (24) with $\eta = 10^{-3}$ to be met. In the first instance, the value of β was fixed to be 0.4, and α varied between 0.05 and 0.5. The number of iterations decreased monotonically with increasing α, as shown in Figure 3(a), suggesting this problem is best suited to values of α in the upper end.

Likewise, the effect of β on the total number of iterations was explored by fixing $\alpha = 0.1$ and sweeping through values of β from 0.05 to 0.95, the results of which are shown in Figure 3(b). This investigation suggests that intermediate values of β in the range 0.35-0.45 result in fastest convergence for this particular problem.

Section 3

(a) The l_1-regularised least squares problem:

$$\underset{x}{\text{minimise}} \quad \|Ax - b\|_2^2 + \lambda \|x\|_1 \tag{30}$$

can be transformed to a convex quadratic problem with linear inequality constraints using the identity outlined in eq. (8) from Section 1(b):

$$\begin{aligned} \underset{x,u_i}{\text{minimise}} \quad & \|Ax - b\|_2^2 + \lambda \textstyle\sum_{i=1}^n u_i \\ \text{subject to} \quad & -u_i \leq x_i \leq u_i \quad i = 1, \ldots, n \end{aligned} \tag{31}$$

The logarithmic barrier function Φ defined jointly on x, u which characterises the inequality constraints is defined as:

$$\begin{aligned} \Phi(x, u) &= \Phi_1(x, u) + \Phi_2(x, u) \\ &= -\sum_{i=1}^n \log(u_i - x_i) - \sum_{i=1}^n \log(u_i + x_i) \\ &= -\sum_{i=1}^n \log\big((u_i - x_i)(u_i + x_i)\big) \end{aligned} \tag{32}$$

with $\textbf{dom}\ \Phi = \big\{x, u \in \mathbb{R}^n \mid -u_i \leq x_i \leq u_i, i = 1, \ldots, n\big\}$. The central path corresponding to problem (32) is the set of points $\big\{x(t), u(t) \mid t \geq 0\big\}$ which are the solution of:

$$\text{minimise} \quad \|Ax - b\|_2^2 + \lambda \sum_{i=1}^n u_i + \frac{1}{t}\Phi(x, u). \tag{33}$$

Multiplying by t gives the central path formulation:

$$\phi_t(x, u) = t\|Ax - b\|_2^2 + t\lambda \sum_{i=1}^n u_i + \Phi(x, u). \tag{34}$$

(b) The gradient of $\phi_t(x,u)$ takes the form:

$$\nabla\phi_t(x,u) = \begin{bmatrix} \nabla_x\phi_t(x,u) \\ \nabla_u\phi_t(x,u) \end{bmatrix}. \tag{35}$$

The first component of this derivative $\nabla_x\phi_t(x,u) \in \mathbb{R}^n$ is given by:

$$\nabla_x\phi_t(x,u) = 2tA^T(Ax-b)+c \tag{36}$$

where the elements of $c \in \mathbb{R}^n$ are:

$$\begin{aligned} c_i &= \frac{\partial\Phi(x,u)}{\partial x_i} \\ &= \frac{1}{u_i - x_i} - \frac{1}{u_i + x_i}. \end{aligned} \tag{37}$$

Similarly, the second component of the derivative $\nabla_u\phi_t(x,u) \in \mathbb{R}^n$ is derived as:

$$\nabla_u\phi_t(x,u) = t\lambda\mathbf{1} + d \tag{38}$$

where the elements of $d \in \mathbb{R}^n$ are:

$$\begin{aligned} d_i &= \frac{\partial\Phi(x,u)}{\partial u_i} \\ &= -\frac{1}{u_i - x_i} - \frac{1}{u_i + x_i}. \end{aligned} \tag{39}$$

Taking second derivatives of $\phi_t(x,u)$ yields the Hessian $\nabla^2\phi_t(x,u) \in \mathbb{R}^{2n\times 2n}$, a square matrix defined as:

$$\nabla^2\phi_t(x,u) = \begin{bmatrix} \nabla_x^2\phi_t(x,u) & \nabla_x\nabla_u\phi_t(x,u) \\ \nabla_u\nabla_x\phi_t(x,u) & \nabla_u^2\phi_t(x,u) \end{bmatrix}. \tag{40}$$

The terms of this matrix are as follows:

$$\begin{aligned} \nabla_x^2\phi_t(x,u) &= \nabla_x\left(2tA^T(Ax-b)+c\right) \\ &= 2tA^TA + C \end{aligned} \tag{41}$$

where the elements of the square matrix $C \in \mathbb{R}^{n\times n}$ are:

$$\begin{aligned} C_{i,j} &= \frac{\partial^2\Phi(x,u)}{\partial x_i\partial x_j} = \frac{\partial}{\partial x_i}\frac{\partial\Phi(x,u)}{\partial x_j} \\ &= \frac{\partial c_j}{\partial x_i} \\ &= \begin{cases} \frac{1}{(u_i+x_i)^2} + \frac{1}{(u_i-x_i)^2} & \text{for } i = j \\ 0 & \text{for } i \neq j. \end{cases} \end{aligned} \tag{42}$$

Likewise:

$$\nabla_u^2\phi_t(x,u) = \nabla_u\left(t\lambda\mathbf{1}+d\right) = \nabla_u d = D \tag{43}$$

where the elements of the square matrix $D \in \mathbb{R}^{n\times n}$ are given by:

$$\begin{aligned} D_{i,j} &= \frac{\partial^2\Phi(x,u)}{\partial u_i\partial u_j} = \frac{\partial}{\partial u_i}\frac{\partial\Phi(x,u)}{\partial u_j} \\ &= \frac{\partial d_j}{\partial u_i} \\ &= \begin{cases} \frac{1}{(u_i+x_i)^2} + \frac{1}{(u_i-x_i)^2} & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases} \end{aligned} \tag{44}$$

the result of which is the same as the entries found for the matrix C in eq. (42).

By definition, the Hessian is symmetric, such that:

$$\nabla_x\nabla_u\phi_t(x,u) = \nabla_u\nabla_x\phi_t(x,u) = E. \tag{45}$$

The terms in E are given by:

$$\begin{aligned} E_{i,j} &= \left[\nabla_u \left(2tA^T(Ax-b)+c\right)\right]_{i,j} = [\nabla_u c]_{i,j} \\ &= \frac{\partial c_j}{\partial u_i} \\ &= \begin{cases} \frac{1}{(u_i+x_i)^2} - \frac{1}{(u_i-x_i)^2} & \text{for } i=j \\ 0 & \text{for } i \neq j. \end{cases} \end{aligned} \tag{46}$$

The full form of the Hessian is then:

$$\nabla^2 \phi_t(x,u) = \begin{bmatrix} 2tA^TA + C & E \\ E & C \end{bmatrix}. \tag{47}$$

(c) The sparse signal reconstruction of the signal $x_0 \in \mathbb{R}^{256}$ given the measurement matrix $A \in \mathbb{R}^{60\times 256}$ and observations $b = Ax_0$ is performed by applying a primal Newton interior-point method to problem (34). Defining $\lambda_{max} = \|2A^Tb\|_\infty$, the regularisation parameter $\lambda = 0.01\lambda_{max}$ is set as a trade off between the quality of the fit to the data and the sparsity of the coefficient vector x [1].

The starting point $(x_0, u_0) = (\mathbf{0}, \mathbf{1})$ is chosen as trivial point which lies in the feasible region **dom** ϕ_t. For $(x,u) \in$ **dom** ϕ_t, the *Newton step* for ϕ_t at (x,u) is the vector:

$$\begin{bmatrix} \Delta x_{\text{nt}} \\ \Delta u_{\text{nt}} \end{bmatrix} = -\nabla^2\phi_t(x,u)^{-1}\nabla\phi_t(x,u), \tag{48}$$

and the step size is found through backtracking line search, with search parameters $(\alpha, \beta) = (0.1, 0.4)$. The convergence criterion is established as $\sigma(x,u)^2/2 \leq \epsilon$, where the quantity:

$$\sigma(x,u) = \left(\nabla\phi_t(x,u)^T\nabla^2\phi_t(x,u)^{-1}\nabla\phi_t(x,u)\right)^{1/2} \tag{49}$$

is the *Newton decrement* at (x,u), and ϵ is small and positive.

Figure 4(a) plots the variation of the minimised l_1-regularised least squares objective $\|Ax^*-b\|_2^2 + \lambda\|x^*\|_1$ as a function of t, with a positive tolerance $\epsilon = 10^{-3}$; these results confirm the notion that as $t \to \infty$ and the accuracy of the logarithmic barrier function improves, $x^*(t)$ converges to an optimal point. This too is reflected in the properties of the reconstructed signals, which approach the original signal as the value of t is increased. Despite the number of measurements being far less than the number of unknowns, the sparse signal reconstruction approach finds the position of the 10 spikes of amplitude ± 1 in the original signal x_0 for sufficiently large t.

(d) Since A is a 'fat matrix' (number of rows = 60 < number of columns = 256), the solution of the least-squares problem:

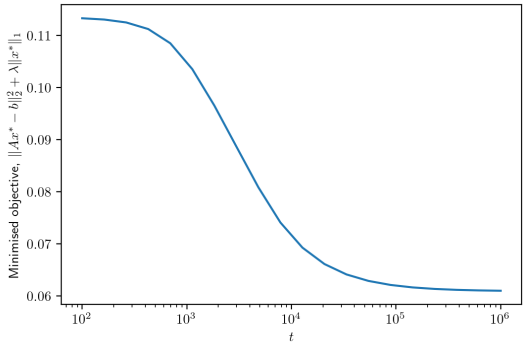
$$\text{minimise} \quad \|Ax-b\|_2 \tag{50}$$

is not unique. The *minimum energy reconstruction* is the point in the set $\{x \in \mathbb{R}^{256} : A^TAx = A^Tb\}$ that is closest to the origin in l_2-norm. It has solution:

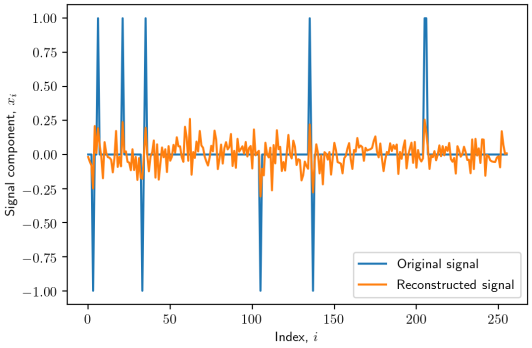
$$\begin{aligned} x_{\text{m.e.}} &= A^+b & (51) \\ &= A^T(AA^T)^{-1}b & (52) \end{aligned}$$

where A^+ is the *pseudoinverse* or *Moore–Penrose inverse* of the matrix A [1]. It is a *right-inverse*, as $AA^+ = I$.

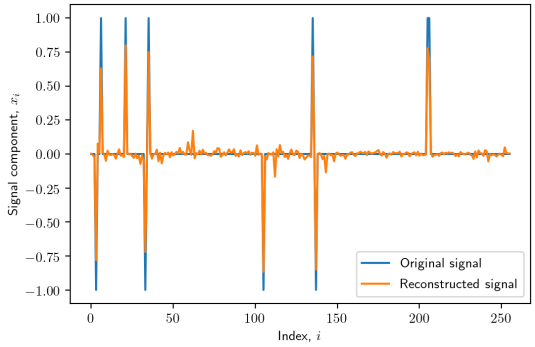
Figure 4(e) plots the minimum energy reconstruction alongside the original signal. Without the l_1-norm term $\lambda\|x\|_1$ in the objective function, there is no penalty on the sparsity of the of the coefficient vector x. Consequently, the optimised solution fits best to the data, at the expense of high cardinality.



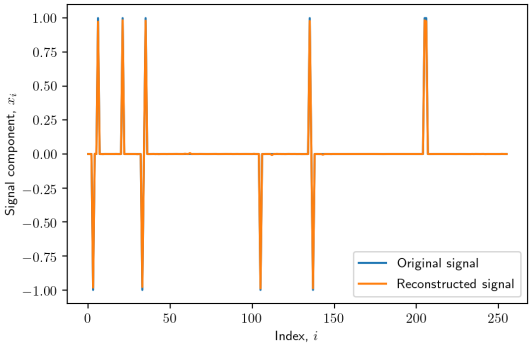
(a)



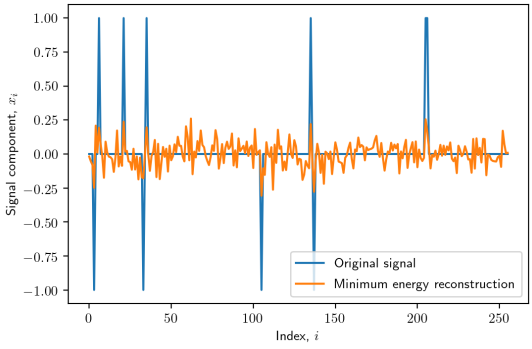
(b) $t = 10^2$



(c) $t = 10^4$



(d) $t = 10^6$



(e)

Figure 4: Panel (a) plots the minimised objective of the l_1-regularised least squares problem as a function of t. Panels (b), (c) and (d) plot the original and reconstructed signals for $t = 10^2$, 10^4 and 10^6 respectively. Panel (d) plots the minimum energy reconstruction.

(e) The main disadvantage of a Newton interior-point method is the cost required to form and store the Hessian. Furthermore, for large l_1-norm regularised least squares problems, solving the system of equations:

$$\nabla^2\phi_t(x,u)\begin{bmatrix}\Delta x_{\text{nt}}\\ \Delta u_{\text{nt}}\end{bmatrix} = -\nabla\phi_t(x,u), \tag{53}$$

to obtain the Newton step at $(x,u) \in \mathbf{dom}\ \phi_t$ is computationally expensive. The paper [4] exploits the problem structure to approximate the solution to the Newton system in eq. (53) using *preconditioned conjugate gradient* (PCG) steps. In doing so, this version of the optimisation algorithm does not require calculation of the full Hessian. Since an iterative method is used to approximately solve for the search direction, the overall method is called a *truncated Newton interior-point method* (TNIPM).

The *Lagrange dual* of the primal problem in eq. (30) is derived in Seciton III B of [4] as:

$$\begin{array}{ll}\text{minimise} & G(\nu) = -\frac{1}{4}\nu^T\nu - \nu^T b\\ \text{subject to} & |A^T\nu|_i \leq \lambda_i \quad i = 1,\ldots,m\end{array} \tag{54}$$

where the vectors λ and ν are called the *dual variables* or *Lagrange multiplier vectors*. The dual problem in (54) is convex, and any dual feasible point ν provides a lower-bound on the optimal value p^* of the primal problem:

$$p^* \geq G(\nu). \tag{55}$$

The *duality gap* η is defined as the gap between the optimal primal and dual solutions:

$$\eta = \|Ax - b\|_2^2 + \lambda\|x\|_1 - G(\nu). \tag{56}$$

Since the primal problem is convex and *Slater's condition* holds, by *strong duality* there exists a dual feasible point ν^* where the optimal values of the primal and dual are equal, and the *optimal duality gap* is zero [1]:

$$p^* = G(\nu^*); \quad \eta^* = \|Ax^* - b\|_2^2 + \lambda\|x^*\|_1 - G(\nu^*) = 0. \tag{57}$$

By the lower-bound property of the dual feasible point ν, the ratio:

$$\frac{f(x) - p^\star}{p^\star} \leq \frac{\eta}{G(\nu)} \tag{58}$$

is an upper bound on the relative suboptimality, where $f(x)$ is the primal objective in problem (30) evaluated at the point x. Consequently, the convergence criterion:

$$\frac{\eta}{G(\nu)} \leq \epsilon \tag{59}$$

where ϵ is small and positive (very likely) guarantees that the method solves the problem to a given relative accuracy ϵ [4].

The duality gap also enables for an update rule for the logarithmic barrier accuracy parameter t:

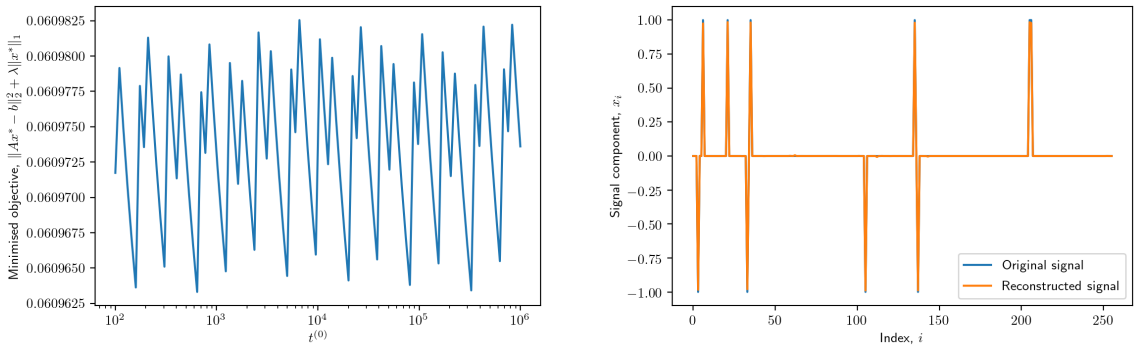
$$t := \begin{cases}\max\{\mu\min\{\hat{t}, t\}t\}, & s \geq s_{\min}\\ t, & s < s_{\min}\end{cases} \tag{60}$$

where $\hat{t} = 2n/\eta$, $\mu > 1$ and $s_{\min} \in (0, 0.5]$. An interpretation of this update rule is that if t were held constant at $t = \hat{t}$, then (x,u,ν) would converge to $(x^*(\hat{t}), u^*(\hat{t}), \nu^*(\hat{t}))$, at which point the duality gap would be exactly η [6].

The step length s is used as an approximate measure of the closeness to the central path; when the current point (x,u,ν) is in near proximity to the central path, then ϕ_t is nearly minimised, and $s = 1$. On the contrary, when far from the central path, $s \ll 1$. Thus, when the current point is near the central path, as judged by $s \geq s_{\min}$ and $\hat{t} \approx t$, then t is increased by a factor μ, else it is kept at its current value. An informal justification of the convergence of this modified interior-point algorithm is given in [6].

The TNIPM is implemented in two stages: the first step converts the primal problem in (34) to the dual problem in (54) and incorporates the update rule for t, and the second replaces the explicit calculation of the Newton step by the PCG approximation. In doing so, the intermediate results obtained using the duality gap convergence criterion can be analysed, thus confirming whether or not the duality gap has been correctly applied.

Figure 5(a) plots the variation of the minimised l_1-regularised least squares objective $\|Ax^* - b\|_2^2 + \lambda\|x^*\|_1$ for the dual Newton interior-point method as a function of the initial logarithmic barrier parameter $t^{(0)}$,



(a) **(b)**

Figure 5: Panel (a) plots the minimised objective of the l_1-regularised least squares problem for the dual Newton interior-point method with iterative update rule for t as a function of $t^{(0)}$. Panel (b) plots the original and reconstructed signals for an initial parameter setting $t^{(0)} = 1/\lambda$.

for a positive tolerance $\epsilon = 10^{-3}$. The same configuration of backtracking line search parameters as the primal Newton interior-point method were used: $\alpha = 0.1$ and $\beta = 0.4$. Compared to the corresponding plot for the primal method, shown in Figure 4(a), in which the objective function decreased monotonically with increasing t, the objective function for the dual method remained in an incredibly tight range over all $t^{(0)}$. Introducing the update rule for t drastically improves the convergence characteristics of the interior-point method: the algorithm converges to an optimal point irrespective of the starting value of the parameter t.

Figure 5(b) plots the original and reconstructed signals using an initial parameter value $t^{(0)} = 1/\lambda$, as suggested in [4] and [6]. The algorithm terminated after 59 iterations, and the minimised objective of the l_1-regularised least squares problem was 0.06093 (4 s.f.). This is comparable to the primal Newton interior-point method with fixed $t = 10^8$, which too returned a minimised objective of 0.06093 (4 s.f.).

Confident in the correctness of the implementation of the dual method, attention was turned to applying PCG steps to approximate the solution to the Newton system in eq. (53). The TNIPM code used in [4] is readily available online[3]. It uses the MATLAB solver `pcg`[4] with a function handle that computes $\nabla^2\phi_t(x,u)\begin{bmatrix}\Delta x_{\text{nt}}\\ \Delta u_{\text{nt}}\end{bmatrix}$ in place of the Hessian $\nabla^2\phi_t(x,u)$, and a preconditioner matrix M which approximates the inverse of the Hessian. Effective preconditioning dramatically improves the rate of convergence, resulting in fewer required iterations to attain a given error tolerance. The equivalent SciPy function `scipy.sparse.linalg.cg`[5] takes the same arguments as `pcg`, and thus was implemented using code which mirrors its MATLAB counterpart as close as possible.

With the same stopping threshold as before ($\epsilon = 10^{-3}$), the Python/SciPy TNIPM showed extremely slow convergence; even after 100000 iterations, $\eta/G(\nu) = 0.01804$ (4 s.f.), and so the algorithm had not converged. However, the objective appeared to be moving in the right direction, as shown in Figure 6.

A debugging attempt was made to try resolve the code, but no clear mistakes were found. Due to time constraints, issues with the TNIPM implementation were left unresolved.

[3] Simple MATLAB Solver for l_1-regularized Least Squares Problems: `https://web.stanford.edu/~boyd/l1_ls/`

[4] MATLAB `pcg`: `https://uk.mathworks.com/help/matlab/ref/pcg.html`

[5] SciPy `scipy.sparse.linalg.cg`: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html`

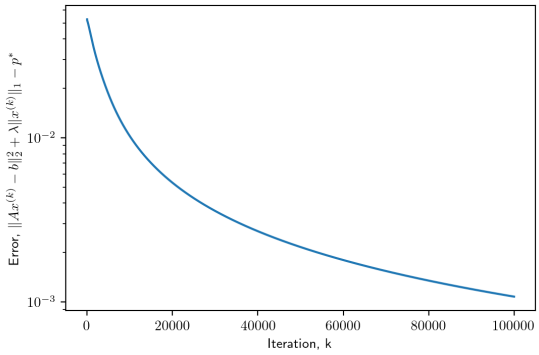


Figure 6: Variation of the error $(\|Ax^{(k)} - b\|_2^2 + \lambda\|x^{(k)}\|_1 - p^*)$ with TNIPM iteration k. The plot demonstrates the quadratic convergence of the algorithm, albeit at an extremely slow rate, with an average error reduction by a factor of around $10^{-2/100000} \approx 0.99995$ per iteration.

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441.

[2] Stephen Boyd and Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018. doi: 10.1017/9781108583664.

[3] Jonathan A. Kelner and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595931341.

[4] Seung-Jean Kim, K. Koh, M. Lustig, Stephen Boyd, and Dimitry Gorinevsky. An Interior-Point Method for Large-Scale ℓ_1-Regularized Least Squares. *IEEE Journal of Selected Topics in Signal Processing*, 1(4):606–617, 2007. doi: 10.1109/JSTSP.2007.910971.

[5] Victor Klee and G. J. Minty. How good is the simplex algorithm? In *Inequalities: III*, New York, NY, USA, 1972. Academic Press. ISBN 0126403031.

[6] Koh Kwangmoo, Kim Seung-Jean, and Stephen Boyd. An Interior-Point Method for Large-Scale Logistic Regression. *Journal of Machine Learning Research*, 8:1519–1555, 2007. ISSN 03772217.

[7] Jorge Nocedal. *Numerical optimization*. Springer, New York, 2006. ISBN 9780387400655.

Appendix

Section 1 Source Code

```
%% load problem data
A = load('A5.mat');
b = load('b5.mat');

temp = fieldnames(A);
A = A.(temp{1});
temp = fieldnames(b);
b = b.(temp{1});

dim = size(A);
m = dim(1); n = dim(2);
I = eye(m);

%% l1-norm
```

```
% Construct l1 LP matrices, as per eq. (10)
A1_til = [[A -I];[-A -I]];
b1_til = [b -b];
c1_til = [zeros(n,1) ; ones(m,1)];

% MATLAB linprog LP solver
tic
[x1, f1] = linprog(c1_til, A1_til, b1_til);
t1 = toc;

%% l2-norm

% Least-squares solver
tic
x2 = lsqminnorm(A,b);
t2 = toc;

% l2 residuals
r2 = (A * x2(1:n) - b);
% Evaluate minimised l2-norm residuals
f2 = norm(r2, 2);

%% linfty-norm

% Construct linfty LP matrices, as per eq. (13)
Ainfty_til = [[A -ones(m,1)];[-A -ones(m,1)]];
binfty_til = [b -b];
cinfty_til = [zeros(n,1) ; 1];

% MATLAB linprog LP solver
tic
[xinfty, finfty] = linprog(cinfty_til, Ainfty_til, binfty_til);
tinfty = toc;
```

Section 2 Source Code

The gradient and Hessian of the logarithmic barrier function from are computed compactly using the representations:

$$\nabla\phi(\tilde{x}) = \tilde{A}^T\tilde{d}, \quad \nabla^2\phi(\tilde{x}) = \tilde{A}^T \operatorname{diag}(\tilde{d})^2\tilde{A},$$

where the elements of $\tilde{d} \in \mathbb{R}^{2m}$ are given by $\tilde{d}_i = 1/(\tilde{b}_i - \tilde{a}_i^T\tilde{x})$.

```
import numpy as np
import scipy.io as sio
from numpy.linalg import norm, inv, cond

# Load problem data
A, b = sio.loadmat('A3.mat'), sio.loadmat('b3.mat')
A, b = A['A3'], b['b3']

m, n = A.shape[0], A.shape[1]
I = np.identity(m)
# Construct l1 LP matrices, as per eq. (10)
A_til = np.concatenate((np.concatenate((A, -I), axis=1), np.concatenate((-A, -I),
axis=1)), axis=0)
b_til = np.concatenate((b, -b), axis=0)[:, 0]
c_til = np.concatenate((np.zeros(n), np.ones(m)), axis=0)

```

```
16
17  def phi(x):
18      """Evaluate the log-barrier function"""
19      r_til = b_til - A_til @ x
20      # dom phi = {x | A_til x < b_til}, so replace negative value r_til[i] by 0
21      r_til = np.where(r_til > 0, r_til, 0)
22      return -np.sum(np.log(r_til))
23
24
25  def grad_phi(x):
26      """Evaluate the gradient of the log-barrier function"""
27      # From compact notation, grad_phi = A_til^T  d_til
28      d_til = 1 / (b_til - A_til @ x)
29      return A_til.T @ d_til
30
31
32  def hes_phi(x):
33      """Evaluate the Hessian of the log-barrier function"""
34      # From compact notation, hes_phi = A_til^T  diag(d_til^2) A_til^T
35      d = 1 / (b_til - A_til @ x)
36      return A_til.T @ np.diag(d_til ** 2) @ A_til
37
38
39  def J(x, t=1):
40      """Evaluate the objective function"""
41      return t * np.dot(c_til, x) + phi(x)
42
43
44  def grad_J(x, t=1):
45      """Evaluate the gradient of objective function"""
46      return t * c_til + grad_phi(x)
47
48
49  def hes_J(x, t=1):
50      """Evaluate the Hessian of objective function"""
51      return hes_phi(x)
52
53
54  def backtrack_desc(x0, alpha=0.1, beta=0.4, eta=1e-3):
55      """Gradient descent with backtracking line search.
56
57      Keyword arguments:
58      x0 -- 2n vector; starting point
59      alpha -- scalar in (0, 0.5); backtrack l.s. constant (default 0.1)
60      beta -- scalar in (0, 1); backtrack l.s. constant (default 0.4)
61      eta -- positive scalar; stopping threshold (default 1e-3)
62
63      Returns:
64      x -- 2n vector; minimised point
65      J_hist -- list; history data of objective function
66      count -- scalar; number of iterations to convergence
67      """
68
69      assert phi(x0) < np.inf, 'x0 not in feasible region'
70      x = x0
71      count = 0
72      J_hist = []
73      dx = - grad_J(x)
74      # Evaluate stopping criterion
75      while norm(dx, ord=2) > eta:
```

```
        J_hist.append(J(x))
        s = 1
        # Backtrack l.s.
        while J(x + s * dx) > J(x) - alpha * s * dx.T @ dx:
            s *= beta
        x = x + s * dx
        dx = - grad_J(x)
        count += 1
    return x, J_hist, count


def exact_desc(x0, eta=1e-3):
    """Gradient descent with exact line search.

    Keyword arguments:
    x0 -- 2n vector; starting point
    eta -- positive scalar; stopping threshold (default 1e-3)

    Returns:
    x -- 2n vector; minimised point
    J_hist -- list; history data of objective function
    count -- scalar; number of iterations to convergence
    """

    assert phi(x0) < np.inf, 'x0 not in feasible region'
    x = x0
    count = 0
    J_eval = []
    dx = - grad_J(x)
    # Evaluate stopping criterion
    while norm(dx, ord=2) > eta:
        J_eval.append(J(x))
        # Exact l.s.
        s = (np.dot(dx, dx)) / (dx.T @ hes_J(x) @ dx)
        x = x + s * dx
        dx = - grad_J(x)
        count += 1
    return x, J_eval, count


# Initial point in feasible region
x0 = (inv(A_til.T @ A_til) @ A_til.T) @ (b_til - 3)

# Perform gradient descent
x_bt, J_eval_bt, count_bt = backtrack_desc(x0)
x_e, J_eval_e, count_e = exact_desc(x0)

# optimised objective (backtrack l.s, exact l.s.)
bt_min_obj = norm((A @ x_bt[:n] - b.flatten()), ord=1)
e_min_obj = norm((A @ x_e[:n] - b.flatten()), ord=1)

# Condition number of Hessian
condition_number = cond(hes_J(x_bt))

# Investigation into the effect of alpha on backtracking l.s
alpha_array = np.arange(0.05, 0.55, 0.05)
alpha_counts = []
for alpha in alpha_array:
    x_bt, J_eval_bt, count_bt = backtrack_desc(x0, alpha=alpha, beta=0.4)
    alpha_counts.append(count_bt)
```

```
136
137 # Investigation into the effect of beta on backtracking l.s
138 beta_array = np.arange(0.05, 1, 0.05)
139 beta_counts = []
140 for beta in beta_array:
141     x_bt, J_eval_bt, count_bt = backtrack_desc(x0, alpha=0.1, beta=beta)
142     beta_counts.append(count_bt)
```

Section 3 Source Code

```
1  # Load problem data
2  A, x0 = sio.loadmat('A.mat'), sio.loadmat('x0.mat')
3  A, x0 = A['A'], x0['x']
4  b = (A @ x0).flatten()
5
6  m, n = A.shape[0], A.shape[1]
7
8
9  def objective(x):
10     """Evaluate the original (l1-regularised least squares problem) objective
       function"""
11     return norm(A @ x - b, ord=2) ** 2 + lambd * norm(x, ord=1)
12
13
14 def Phi(x, u):
15     """Evaluate the log-barrier function"""
16     Phi_1 = np.where(u - x > 0, u - x, 0)
17     Phi_2 = np.where(u + x > 0, u + x, 0)
18     return - sum(np.log(Phi_1) + np.log(Phi_2))
19
20
21 def phi(x, u):
22     """Evaluate the central-path formulation"""
23     return t * norm(A @ x - b, ord=2) ** 2 + t * lambd * sum(u) + Phi(x, u)
24
25
26 def grad_phi(x, u):
27     """Evaluate the gradient of the central-path formulation"""
28     gradx_phi = 2 * t * A.T @ (A @ x - b) + 1 / (u - x) - 1 / (u + x)
29     gradu_phi = t * lambd - 1 / (u - x) - 1 / (u + x)
30     grad_phi = np.concatenate((gradx_phi, gradu_phi))
31     return grad_phi
32
33
34 def hes_phi(x, u):
35     """Evaluate the Hessian of the central-path formulation"""
36     C = np.diagflat(1 / (u + x) ** 2) + np.diagflat(1 / (u - x) ** 2)
37     E = np.diagflat(1 / (u + x) ** 2) - np.diagflat(1 / (u - x) ** 2)
38     hes_phi = np.block([[2 * t * A.T @ A + C, E], [E, C]])
39     return hes_phi
40
41
42 def backtrack_ls(x, u, dxu, alpha=0.1, beta=0.4):
43     """Backtracking linesearch. Finds the step length to approximately minimise
44     phi along the ray {(x + step_size * dx, u + step_size * du) | step_size > 0}
45
46     Keyword arguments:
47     x, u -- n vectors; point at which to evaluate step size
```

```
    dxu -- 2n vector; descent direction
    alpha -- scalar in (0, 0.5); backtrack l.s. constant (default 0.1)
    beta -- scalar in (0, 1); backtrack l.s. constant (default 0.4)

    Returns:
    step_size -- scalar in (0,1); step size by which to move in descent direction dxu
    """

    step_size = 1
    while phi(x + step_size * dxu[:n], u + step_size * dxu[n:]) > phi(x, u) - alpha *
    step_size * dxu.T @ dxu:
        step_size *= beta
    return step_size


def int_point(x0, u0, epsilon=0.001):
    """Newton interior point method (NIPM) with backtracking line search.

    Keyword arguments:
    x0, u0 -- n vectors; starting point
    epsilon -- positive scalar; stopping threshold (default 0.001)

    Returns:
    x -- n vector; minimised point
    """

    assert phi(x0, u0) < np.inf, '(x0, u0) not in feasible region'
    x, u = x0, u0
    count = 0
    s = grad_phi(x, u).T @ inv(hes_phi(x, u)) @ grad_phi(x, u)
    # Evaluate stopping criterion
    while s / 2 > epsilon:
        # Newton descent direction
        dxu = - inv(hes_phi(x, u)) @ grad_phi(x, u)
        step_size = backtrack_ls(x, u, dxu)
        x = x + step_size * dxu[:n]
        u = u + step_size * dxu[n:]
        s = grad_phi(x, u).T @ inv(hes_phi(x, u)) @ grad_phi(x, u)
        count += 1
        print('Iteration: {} | Objective: {} | Stopping Crit: {}'.format(count,
        objective(x), s / 2))
        print('-----------------------------------------------------------------')
    return x


# Set regularisation parameter
lambd_max = max(abs(2 * A.T @ b).flatten())
lambd = 0.01 * lambd_max

# Set log-barrier accuracy parameter
t = 10 ** 6

# Initialise feasible starting point
x0, u0 = np.zeros(n), np.ones(n)

# NIPM
x = int_point(x0, u0, epsilon=0.001)

# Investigation into the effect of parameter t
t_array = np.logspace(2, 6, 20)
```

```
t_objective = []
for t in t_array:
    x = int_point(x0, u0, epsilon=0.001)
    t_objective.append(objective(x))
```

Dual Newton Interior-Point Method (Extension) Source Code

```
def dual_point(x):
    """Evaluate the dual point"""
    z = A @ x - b
    v = 2 * z
    maxA_v = norm(A.T @ v, ord=np.inf)
    if maxA_v > lambd:
        v = v * lambd / maxA_v
    return v


def G(v):
    """Evaluate the dual objective"""
    return -0.25 * v.T @ v - v.T @ b


def duality_gap(x, G):
    """Evaluate the duality gap"""
    return objective(x) - G


def update_t(t, nu, s, mu=2, s_min=0.5):
    """Update rule for log-barrier parameter t.

    Keyword arguments:
    t -- current value of t
    nu --  duality gap
    s -- step size
    mu -- update parameter (G.P. ratio), > 0 (default 2)
    s_min -- update parameter in (0,1] (default 0.5)

    Returns:
    t -- updated t
    """

    if s > s_min:
        return max(mu * min(2 * n / nu, t), t)
    else:
        return t


def dual_int_point(x0, u0, epsilon=0.001):
    """Dual Newton interior point method (DNIPM) with backtracking line search.

    Keyword arguments:
    x0, u0 -- n vectors; starting point
    nu -- positive scalar; stopping threshold (default 0.001)

    Returns:
    x -- n vector; minimised point
    """

```

```
    global t

    assert phi(x0, u0) < np.inf, '(x0, u0) not in feasible region'

    x, u = x0, u0
    count = 0
    v = dual_point(x)
    nu = duality_gap(x, G(v))

    # Evaluate stopping criterion
    while (nu / G(v)) > epsilon:
        # Newton descent direction
        dphi = - inv(hes_phi(x, u)) @ grad_phi(x, u)
        # Step size through backtracking l.s
        step_size = backtrack_ls(x, u, dphi)

        # Update (x,u) -> (x+dx,u+du)
        x = x + step_size * dphi[:n]
        u = u + step_size * dphi[n:]

        # Evaluate the duality gap
        v = dual_point(x)
        nu = duality_gap(x, G(v))

        # Update t
        t = update_t(t, nu, step_size, mu=2)

        count += 1
        print('Iteration: {} | Objective: {} | Stopping Crit: {}'.format(count,
        objective(x), nu / G(v)))
        print('----------------------------------------------------------------------')
    return x


# Set initial log-barrier accuracy parameter
t = 1 / lambd

# DNIPM
x = dual_int_point(x0, u0, epsilon=0.001)
```

Truncated Newton Interior-Point Method (Extension) Source Code

```
def Axfunc_l1(x, A, At, d1, d2, p1, p2, p3):
    """Compute AX (PCG)

    Returns:
    y = hessphi*[x1;x2], where hessphi = [A.T * A * 2 + D1, D2; D2, D1]
    """

    x1 = x[:n]
    x2 = x[n:]
    y1 = (At @ ((A @ x1) * 2)) + np.multiply(d1, x1) + np.multiply(d2, x2)
    y2 = np.multiply(d2, x1) + np.multiply(d1, x2)
    y = np.concatenate((y1, y2))
    return y


def Mfunc_l1(x, A, At, d1, d2, p1, p2, p3):
```

```
    """Compute P^{-1}X (PCG):

    Returns:
    y = P^{-1} * x
    """

    x1 = x[:n]
    x2 = x[n:]
    y1 = np.multiply(p1, x1) - np.multiply(p2, x2)
    y2 = -np.multiply(p2, x1) + np.multiply(p3, x2)
    y = np.concatenate((y1, y2))
    return y


def tnipm(x0, u0, epsilon=0.001):
    """Truncated Newton interior point method (TNIPM) with backtracking line search.

    Keyword arguments:
    x0, u0 -- n vectors; starting point
    nu -- positive scalar; stopping threshold (default 0.001)

    Returns:
    x -- n vector; minimised point
    """

    global t

    assert phi(x0, u0) < np.inf, '(x0, u0) not in feasible region'

    x, u = x0, u0
    e_pcg = epsilon
    count = 0

    v = dual_point(x)
    nu = duality_gap(x, G(v))

    # Evaluate stopping criterion
    while (nu / G(v)) > epsilon:

        # Compute the search direction (notation consistent with MATLAB script)
        q1 = 1 / (u + x)
        q2 = 1 / (u - x)

        d1 = t * (q1 ** 2 + q2 ** 2)
        d2 = t * (q1 ** 2 - q2 ** 2)

        diagxtx = np.diag(At @ A)

        prb = diagxtx + d1
        prs = np.multiply(prb, d1) - (d2 ** 2)
        p1, p2, p3 = np.divide(d1, prs), np.divide(d2, prs), np.divide(prb, prs)

        x_to_Ax = lambda x: Axfunc_l1(x, A, At, d1, d2, p1, p2, p3)
        x_to_Mx = lambda x: Mfunc_l1(x, A, At, d1, d2, p1, p2, p3)

        Ax = LinearOperator((2 * n, 2 * n), matvec=x_to_Ax)
        M = LinearOperator((2 * n, 2 * n), matvec=x_to_Mx)

        dxu = cg(Ax, -grad_phi(x, u), tol=e_pcg, M=M)[0]  # PCG approximation to
        Newton system
```

```
277
278         # Compute step size by backtracking l.s.
279         step_size = backtrack_ls(x, u, dxu)
280
281         # Update (x,u) -> (x+dx,u+du)
282         x = x + step_size * dxu[:n]
283         u = u + step_size * dxu[n:]
284
285         # Evaluate the duality gap
286         v = dual_point(x)
287         nu = duality_gap(x, G(v))
288
289         # Update t, pcg tolerance
290         t = update_t(t, nu, step_size, mu=2)
291         e_pcg = min(0.1, epsilon * nu / norm(dxu, ord=2))
292
293         count += 1
294         if count % 10000 == 0:
295             print('Iteration: {} | Objective: {} | Stopping Crit: {}'.format(count,
                objective(x), nu / G(v)))
296             print('----------------------------------------------------------------------')
297     return x
```