4M17 CW1

1 Question 1

1.1 1a

The norms on $\mathbb{R}^m$ are given by:

$$\ell_1 = \sum_{i=1}^{m} |x_i|, \quad \ell_2 = \left(\sum_{i=1}^{m} |x_i|^2\right)^{1/2}, \quad \ell_\infty = \max_i |x_i| \tag{1}$$

where the ℓ_∞ norm is derived from the ℓ_p norm as $p \to \infty$. Only the component of x with largest value remains once the elementwise power is taken.

The norm approximation problem generally minimises the residual error of a variable $x \in \mathbb{R}^n$ given data $A \in \mathbb{R}^{mxn}, \quad b \in \mathbb{R}^m$:

$$\begin{aligned} \text{minimise} \quad & ||\mathbf{y}|| \\ \text{subject to} \quad & \mathbf{y} = \mathbf{Ax} - \mathbf{b} \end{aligned} \tag{2}$$

The norm approximation problem is an example of the generalised penalty function approximation problem on componentwise residuals:

$$\begin{aligned} \text{minimise} \quad & \phi(r_1) + ... + \phi(r_m) \\ \text{subject to} \quad & \mathbf{r} = \mathbf{Ax} - \mathbf{b} \end{aligned} \tag{3}$$

The definition of a convex function on two points (Jensen's inequality) is:

$$\text{For} f : \mathbb{R}^{>} \to \mathbb{R}, \quad x, y \in \text{domain}(f), \quad 0 \le \theta \le 1 :$$
$$f(\theta x + (1-\theta)y) \le \theta f(x) + (1-\theta) f(y)$$

Every norm on $\mathbb{R}^m$ is convex. To prove this, write the ℓ_2 norm of the residual as a function f(x) and use the triangle inequality and homogeneity of norm [1]:

$$f(x) = ||Ax - b||_2 :$$
$$f(\theta x + (1-\theta)y) \le f(\theta x) + f((1-\theta)y) = \theta f(x) + (1-\theta) f(y)$$

Composition of a convex and nondecreasing function (here $g(x) = x^2$) preserves convesity. Given r_i is the ith residual, the ℓ_2 minimisation problem is equivalent to the uncontrained QP of minimising sum of squared residuals:

$$\min_x ||Ax - b||_2^2 = \sum_i^m r_i^2 \tag{4}$$

Hence, expanding the squared norm of the residual as a quadratic convex function gives (all terms are vectors/ matrices, so bold will be dropped from here):

$$f(x) = x^T A^T A x - 2A^T b^T x + b^T b \tag{5}$$

Since the linear Taylor expansion at any point in a convex function is a global underestimator (first-order condition for convexity), the global minimum is obtained when:

$$\begin{aligned}\nabla f(\tilde{x}) &= 0\\ 2A^TA\tilde{x} &= A^Tb\\ \tilde{x} &= (A^TA)^{-1}A^Tb\end{aligned}$$

The middle equation is a linear system of equations with analytic solution $\tilde{x}$. $\tilde{x}$ is the projection of the target vector b onto the column space of A (psuedoinverse maps image of b on A back to rowspace).

1.2 1b

1.2.1 Optimising maximum of affine functions

The pointwise maximum of a family of affine functions is a convex function.

In general the problem where a function is approximated as a piece-wise linear envelope:

$$\begin{aligned}\min_x \quad & \max_{i=1,\dots,m}(\mathbf{c_i^T x} + \mathbf{d})\\ \text{subject to} \quad & \mathbf{Ax} \preceq \mathbf{b}\end{aligned} \tag{6}$$

can be rewritten as

$$\begin{aligned}\min_{x,t} \quad & t\\ \text{subject to} \quad & \mathbf{c_i^T x} + \mathbf{d_i} \preceq \mathbf{t} \quad i = 1, ..., m\\ & \mathbf{Ax} \preceq \mathbf{b}\end{aligned} \tag{7}$$

Rather than minimising the original function in an unconstrained manner, these equivalent problems minimise one variable, and moves the function into the constraint, with a new feasible polygon in $\mathbb{R}^{2m}$.

1.2.2 ℓ_1 norm

The sum of absolute residuals is equivalent to this norm i.e. for $\mathbf{a_i}$ as rows of $\mathbf{A}$ (i.e. each data entry):

$$\min_x \sum_{i=1}^{m} |\mathbf{a_i}^T\mathbf{x} - b_i|$$

Observe an equivalent expression for the modulus of a scalar z_i [2]:

$$|z_i| = \min(t_i \mid -t_i \leq z_i \leq t_i) \tag{8}$$

Applying this rule to every component of the residual, 1-norm minimisation is equivalent to

$$\begin{aligned}\min \quad & \mathbf{t}\\ \text{subject to} \quad & \mathbf{Ax} - \mathbf{b} \preceq \mathbf{t}^T\\ & \mathbf{b} - \mathbf{Ax} \preceq \mathbf{t}^T\end{aligned} \tag{9}$$

Which can be written in the standard LP form:

$$\begin{aligned}\min_x \quad & \mathbf{\tilde{c}^T\tilde{x}}\\ \text{subject to} \quad & \mathbf{\tilde{A}\tilde{x}} \leq \mathbf{\tilde{b}}\end{aligned} \tag{10}$$

if using the definitions below. There are 2m constraints and n+m variable components. $\mathbf{I}_m$ is a mxm identity matrix:

$$\mathbf{\tilde{c}} = \begin{bmatrix}\mathbf{0}_n\\ \mathbf{1}_m\end{bmatrix}, \quad \mathbf{\tilde{x}} = \begin{bmatrix}\mathbf{x}\\ \mathbf{t}\end{bmatrix}, \quad \mathbf{\tilde{A}} = \begin{bmatrix}\mathbf{A} & -\mathbf{I_m}\\ -\mathbf{A} & -\mathbf{I_m}\end{bmatrix} \quad \mathbf{\tilde{b}} = \begin{bmatrix}\mathbf{b}\\ -\mathbf{b}\end{bmatrix}, \tag{11}$$

1.2.3 ℓ_∞ norm

Minimising in the ℓ_∞ norm is Chebychev approximation, minimising the maximum element of the residual.

Observe an equivalent expression for the maximum of a set of m scalars z_i [2]:

$$\max_i |z_i| = \min(t \mid -t \leq z_i \leq t \text{ for } i = i, ..., m) \tag{12}$$

Using the form in equation 9, the Chebychev minimisation is equivalent to

$$\begin{aligned} \min_x \quad & \tilde{c}^T \tilde{x} \\ \text{subject to} \quad & \tilde{A}\tilde{x} \leq \tilde{b} \end{aligned} \tag{13}$$

if using the definitions below. There are 2m constraints and n+1 variable components. $\mathbf{I_m}$ is a vector of m 1s:

$$\tilde{c} = \begin{bmatrix} \mathbf{0}_n \\ 1 \end{bmatrix}, \quad \tilde{x} = \begin{bmatrix} \mathbf{x} \\ t \end{bmatrix}, \quad \tilde{A} = \begin{bmatrix} \mathbf{A} & -\mathbf{1_m} \\ -\mathbf{A} & -\mathbf{1_m} \end{bmatrix} \quad \tilde{b} = \begin{bmatrix} \mathbf{b} \\ -\mathbf{b} \end{bmatrix}, \tag{14}$$

1.3 1c

Table 1: Norm of residuals (4 s.f.) of solutions from scipy linprog LP solver and scipy least squares solver for the ℓ_2 norm for m constraints = 2n data points and varying values of n. t is runtime in milliseconds.

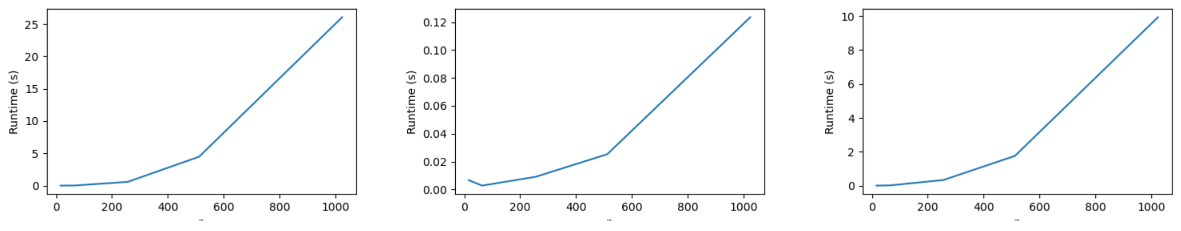
Norm	n=16	t/ms	n=64	t/ms	n=256	t/ms	n=512	t/ms	n=1024	t/ms
$\|\|Ax - b\|\|_1$	12.48	14.91	52.45	25.10	194.9	585.9	402.0	4478	808.4	26,060
$\|\|Ax - b\|\|_2$	2.010	6.989	5.449	2.379	9.357	9.399	12.86	26.17	18.71	125.9
$\|\|Ax - b\|\|_\infty$	0.7719	2.148	0.8200	14.02	0.8024	332.3	0.8033	1754	0.7915	9939

The three norm values of residuals returned from the LP or least squares optimisation algorithms using scipy are shown in table 1, along with runtimes. HiGHS interior point method was chosen for the LP solver; the simplex method provided by the same package was found to be far less efficient.

All norms display a polynomial runtime dependency on size of array. This is expected as the LP uses an interior point method which has a polynomial worst-case complexity (relevant for 1-norm and infinity-norm). The 2-norm is solved analytically using the pseudoinverse, with Cholesky factorisation. This has $O(N^2)$ runtime with problem size also.

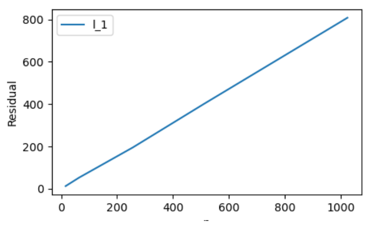
The 1-norm scales linearly with size, as expected since it is the sum of absolute values of the residual vector. The 2-norm scales with the square root of the size, which is also expected as it involves square rooting the square of the element-wise residual values. The infinity-norm does not significantly change once n becomes large enough, which is expected as it measures the maximum value in the residual.

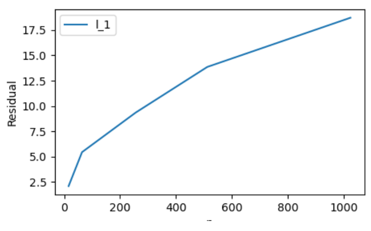
Optimising the 1-norm takes the longest because it involves the largest datasets. The infinity-norm vector only has one t-value, whereas the 1-norm has m of these. Optimising the 2-norm takes significantly less time than the others as it uses an analytical solution finder rather than iterative algorithm.



(a) ℓ_1 (b) ℓ_2 (c) ℓ_∞

Figure 1: Runtime for HiGHS interior point method against dataset size (dimension of each datapoint n = number of datapoints/2).


(a) ℓ_1


(b) ℓ_2

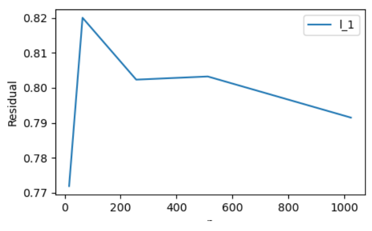
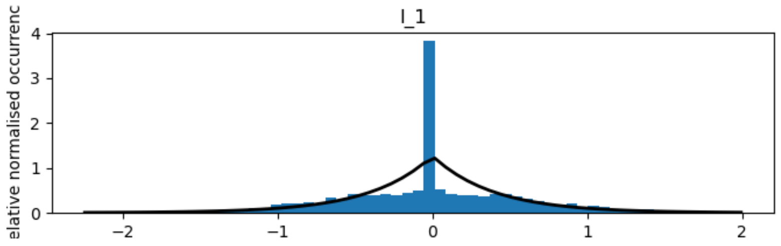
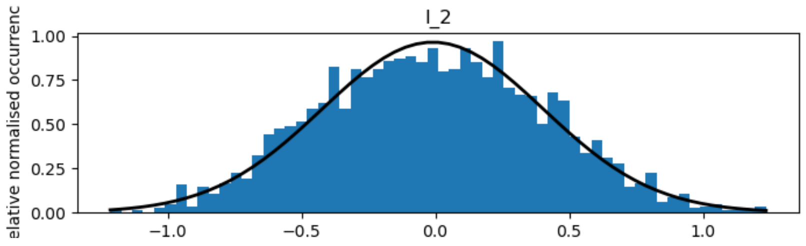

(c) ℓ_∞

Figure 2: Norm of residual $||Ax - b||$ against dataset size.

1.4 1d


(a) ℓ_1 Laplace, μ = 0.00, b = 0.395.


(b) ℓ_2 Gaussian, $\mu = -0.0998$, $\sigma = 0.413$.

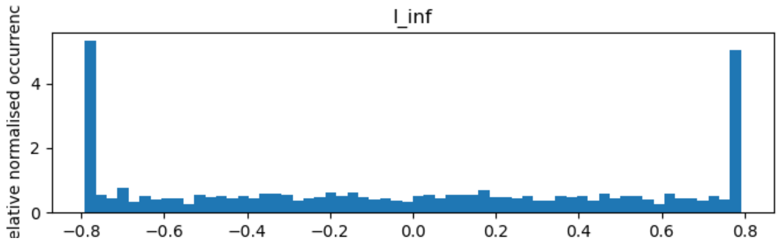

(c) ℓ_∞ approximation.

Figure 3: Residual histograms with 60 bins for the 3 algorithmic/analytically tractable norm approximation problems described above. The y axis has been scaled such that the area under the bins sums to 1.

Residuals calculated as $\mathbf{Ax} - \mathbf{b}$ where x is the solution vector are shown in figure 3. Increasing the norm shifts the distribution away from having most probability mass at small values.

The ℓ_1 norm minimisation problem produces sparse residual vectors, with norms with a Laplacian distribution. This norm is robust to outliers, hence is called a robust estimator. Hence it has many small residuals and fewer larger ones.

The ℓ_2 norm is sensitive to large residuals and not so much to modest ones. It has a Gaussian residual distribution, which relates to the interpretation of least squares minimisation as the maximum likelihood value of a linear variable in x with Gaussian additive noise (e.g. measurement error). This is not always good, as it weights outliers heavily - a few methods are developed to mitigate this, including Huber's penalty function, which is a convex approximation to a fixed penalty above a certain threshold residual value.

The ℓ_∞ norm only looks at the largest element, so no residuals lie beyond a certain value (here ± 0.8). Inspection of the data shows the values in A never exceed ± 1 and values in b never exceed ± 0.8, so minimisation of the residual may at times, if x is sparse, produce residuals with value up to ± 0.8.

As mentioned in section 1.1, these are all specific instances of the generalised penalty function approximation problem. The maximum likelihood interpretation of this is a MLE problem [1]:

$$\min_x \sum_{i=1}^{m} \phi(b_i - \mathbf{a}_i^T \mathbf{x}) \tag{15}$$

The noise density on measurements b is given by [1]:

$$p(z) = \frac{e^{-\phi(x)}}{\int e^{-\phi(u) du}} \tag{16}$$

This lends an explanation for the Laplacian distribution of residuals for the 1-norm: the Laplace PDF is given by:

$$\frac{e^{-|x-\mu|}}{2b} \tag{17}$$

which arises from equation 16 if $\phi(x)$ is the ℓ_1 norm.

2 Question 2

2.1 2a

Each of the 2m constraints in equation 11 can be written as:

$$\tilde{\mathbf{a}}_i^T \tilde{\mathbf{x}} \leq \tilde{b}_i \tag{18}$$

where $\tilde{a}_i$ is the ith row of $\tilde{\mathbf{A}}$ and $\tilde{b}_i$ is the ith component of $\tilde{\mathbf{b}}$. The form in equation 18 provides $f_i(x)$ in this optimisation problem as:

$$f_i(x) = \tilde{\mathbf{a}}_i^T \tilde{\mathbf{x}} - b_i \tag{19}$$

The logarithmic barrier function acts as a convex and smooth approximation on the ideal penalty (indicator) function for traversing outside the feasible set, where $I_-(f_i(x))$ is nonzero only when $f_i(x) < 0$. In contrast to Q1, where the objective function was moved into the constraints, this turns a constrained problem into an unconstrained problem by moving the constraints into the objective.

$$\text{minimise} \quad f_o(x) + \sum_{i=1}^{m} I_-(f_i(x)) \tag{20}$$

The log barrier function on the feasible set is:

$$\begin{aligned} \phi(\mathbf{x}) &= \begin{cases} -\sum_{i=1}^{m} \log(-f_i(\mathbf{x})) & \text{if} f_i(\mathbf{x}) < 0 \\ \infty & \text{if} f_i(\mathbf{x}) \geq 0 \end{cases} \\ \text{domain}(\phi) &= \{\mathbf{x} | \mathbf{A}\mathbf{x} \preceq \mathbf{b}\} \end{aligned} \tag{21}$$

Its domain is the feasible set. One reason logarithms are used as barrier functions is the ease of calculation of the gradient. The closed-form expression (defined only on the feasible set) is given by:

$$\nabla \left(t\tilde{\mathbf{c}}^{\mathbf{T}} \tilde{\mathbf{x}} - \sum_{i=1}^{m} \log(b_i - \tilde{\mathbf{a}}_i^T \tilde{\mathbf{x}}) \right) = t\tilde{\mathbf{c}} + \sum_{i=1}^{m} \frac{a_i}{b_i - \tilde{\mathbf{a}}_i^T \tilde{\mathbf{x}}} \tag{22}$$

Algorithm 1 Backtracking line search with gradient descent.

```
Require: initial $\mathbf{x}_0$ such that $\tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}} \le \tilde{b}_i \forall i$
Ensure: $y = x^n$
  $\alpha \leftarrow 0.3$
  $\beta \leftarrow 0.5$
  $\nabla f(x) \leftarrow t\tilde{\mathbf{c}} + \sum_{i=1}^m \frac{a_i}{b_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}}}$
  $f(x) \leftarrow t\tilde{\mathbf{c}}^{\mathbf{T}}\tilde{\mathbf{x}} - \sum_{i=1}^m \log(b_i - \tilde{\mathbf{a}}_i^T\tilde{\mathbf{x}})$
  while $||\nabla f(x)||_2^2 > \epsilon$ do
      t = 1
    while $x + t\nabla f(x)$ not in domain(f) do
      t = $\beta t$
    end while
    while $f(x + t\nabla f(x)) > f(x) - \alpha t \nabla f(x)^T \Delta x$ do
      t = $\beta t$
    end while
      $x = x - t\nabla f(x)$
      Recalculate $f(x), \nabla f(x)$
  end while
```

2.2 2b

The tolerance on $||\nabla f(x)||_2^2$ was decreased from 0.1 to 0.0001. The final value of the function is 335.0659 (4 d.p.). Parameters used and psuedocode are shown in algorithm 1. Initialisation is important as it must be done in the feasible region. Setting $\tilde{\mathbf{x}}_0 = \tilde{\mathbf{c}}$ works as $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = [-\mathbf{1}_{2m}]$ and all $|b_i| < 1$.

2.2.1 Duality and Slater's Constraint

The Lagrange dual problem with inequality constraints $f_i(x) < 0$ and equality constraints $h_i(x) = 0$ finds the highest lower bound on the primal problem. In the feasible region, for $\lambda \succeq 0$, the term $\sum_{i=1}^m \lambda_i f_i(x) < 0$ - the Lagrangian is a lower bound. Geometrically, with equality constraints, the Lagrangian includes the optimality condition $\nabla f(x) = -\sum_{i=1}^m \lambda_i \nabla h_i(x)$: i.e. the direction of steepest descent is parallel to a linear combination of gradient vectors of constraints. If this were not true, then movement along the constraints would allow the objective function to be minimised further. The dual problem is given by:

$$\begin{aligned}&\max_{\lambda,\nu}\{\inf_{x\in D}\{L(x,\lambda,\nu)\}\} = \max_{\lambda,\nu}\{\inf_{x}\{f(x) + \sum_{i=1}^{m}\lambda_i f_i(x) + \sum_{i=1}^{p}\nu_i h_i(x)\}\} = d^*\\ &\text{subject to} \quad \lambda \succeq 0\end{aligned} \tag{23}$$

Strong duality is where the primal and dual solutions are the same. It is usually true when the primal is a convex optimisation problem (convex objective and inequalities, and affine equalities). Slater's condition for strict feasible of the primal solution (or relaxed to allow weak feasibility if the conditions are affine) is sufficient for strong duality.

2.2.2 Duality Gap and t

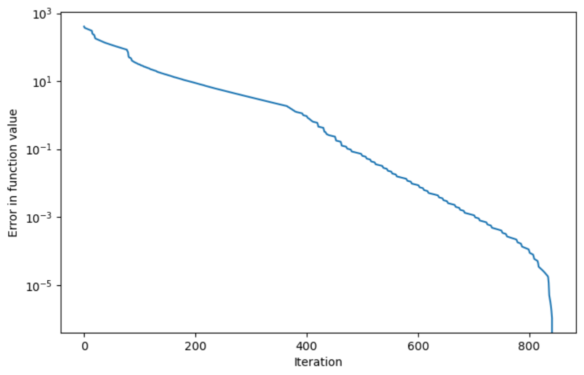
Large values of t improves the objective function solution approximation, but setting t too large results in difficulty with minimisation since the gradient varies more rapidly near the edge of the feasible set. This implies it is best to incrementally increase t and start each iteration at the previous solution.

Formally, the stopping criterion is elucidated with duality theory. Consider the Lagrangian $f_0(x) + \sum_{i=1}^m \lambda_i^* f_i(x)$ at the optimal point found by the central path. The criteria to minimise this is equivalent to the central path gradient for log barrier function if $\lambda_i^*(t) = -\frac{1}{tf_i(x^*)}$. So, the dual solution has lower bound

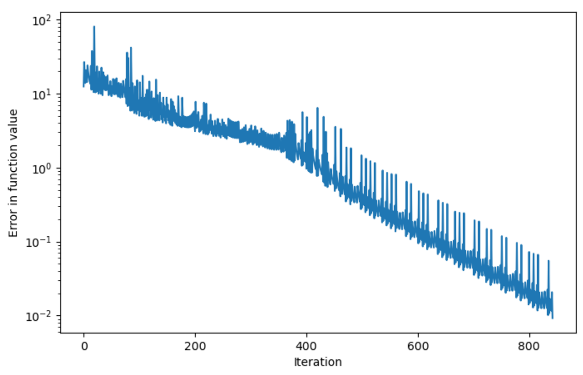
$$\tilde{\mathbf{c}}^{\mathbf{T}}\tilde{\mathbf{x}}^* - \frac{m}{t} \tag{24}$$

which shows the duality gap is m/t. The stopping criterion is $m/t < \epsilon$ for some set value of precision, and approximation gets better as $t \to \infty$.

2.3 2c



(a) Absolute convergence error in function



(b) 2-norm of gradient

Figure 4: Log-linear relationship between absolute convergence error in function and gradient, and iteration count.

The image above shows how absolute error in the function and the magnitude of the gradient changes with iteration count for gradient descent. Both graphs are approximately linear in log-scale, showing exponential decrease. The exact convergence on p^* with backtracking line search for gradient descent confirms this [1]:

$$f(x^k) - p^* \leq c^k(f(x^0) - p^*) \tag{25}$$

for

$$c = 1 - \min(2m\alpha,\ 2\alpha\beta m/M) < 1 \tag{26}$$

and m, M bound strong convexity on f(x) with the following condition on the eigenvalues of the Hessian:

$$m\mathbf{I} \preceq \nabla^2 f(\mathbf{x}) \preceq M\mathbf{I} \tag{27}$$

The gradient graph is spiky - this suggests motion along the landscape in a 'staircase' fashion, due to some directions having higher gradient than others.

3 Question 3

3.1 3a

This is a convex quadratic problem:

$$\min_x t||Ax - b||_2^2 + t\lambda \sum_{i=1}^{m} u_i \\ \text{subject to} - u_i \leq x_i \leq u_i \tag{28}$$

A suitable log barrier function has a domain that approaches 0 when x gets close to $\pm u_i$:

$$\phi(x, u) = -\sum_{i=1}^{m} \log(u_i - x_i) - \sum_{i=1}^{m} \log(u_i + x_i) \tag{29}$$

3.2 3b

The gradients with respect to each of the parameters are:

$$[\nabla_x \phi(x,u)]_i = \frac{\partial \phi(x,u)}{\partial x_i} = 2t([\mathbf{A^T A x}]_\mathbf{i} - \mathbf{A^T b}) + \frac{1}{u_i - x_i} - \frac{1}{u_i + x_i} \tag{30}$$

$$[\nabla_u \phi(x,u)]_i = \frac{\partial \phi(x,u)}{\partial u_i} = t\lambda - \frac{1}{u_i - x_i} - \frac{1}{u_i + x_i} \tag{31}$$

The Hessian matrix, where rows are derivatives w.r.t. x and columns w.r.t. u, is given by:

$$\begin{bmatrix} H_{11} = \frac{\partial^2 \phi(x,u)}{\partial x^2} & H_{12} = \frac{\partial^2 \phi(x,u)}{\partial x \partial u} \\ H_{21} = \frac{\partial^2 \phi(x,u)}{\partial u \partial x} & H_{22} = \frac{\partial^2 \phi(x,u)}{\partial u^2} \end{bmatrix} \tag{32}$$

x, u are vectors so each entry is a vector derivative of a vector, i.e. matrix:

$$\mathbf{H_{11}} = 2t(\mathbf{A^T A}) + \operatorname{diag} \begin{bmatrix} \frac{1}{(u_1+x_1)^2} + \frac{1}{(u_1-x_1)^2} \\ \vdots \\ \frac{1}{(u_1+x_n)^2} + \frac{1}{(u_1-x_n)^2} \end{bmatrix} \tag{33}$$

$$\mathbf{H_{22}} = \operatorname{diag} \begin{bmatrix} \frac{1}{(u_1-x_1)^2} + \frac{1}{(u_1+x_1)^2} \\ \vdots \\ \frac{1}{(u_n-x_1)^2} + \frac{1}{(u_n+x_1)^2} \end{bmatrix} \tag{34}$$

$$\mathbf{H_{21}} = \mathbf{H_{12}} = \operatorname{diag} \begin{bmatrix} \frac{1}{(u_1+x_1)^2} - \frac{1}{(u_1-x_1)^2} \\ \vdots \\ \frac{1}{(u_1+x_n)^2} - \frac{1}{(u_1-x_n)^2} \end{bmatrix} \tag{35}$$

3.3 3c

First-order gradient descent with backtracking line search was applied. Newton's method must simultaneously optimise x and u, by concatenating an optimisation vector u in the code. x is kept in the feasible region through a backtracking for loop. This is also more efficient than exact line search.

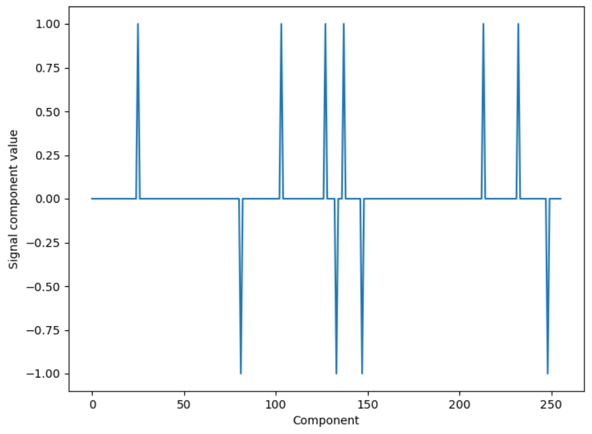
$$\mathbf{X} = \begin{bmatrix} \mathbf{x} \\ \mathbf{u} \end{bmatrix}, \quad \Delta \mathbf{X} = -\mathbf{H}^{-1} \nabla \mathbf{X} \tag{36}$$

The Tikhonov regularised least-squares problem can be interpreted as cardinality minimalisation, or robust least-squares. This is because minimising the 1-norm of x avoids errors in A being amplified.

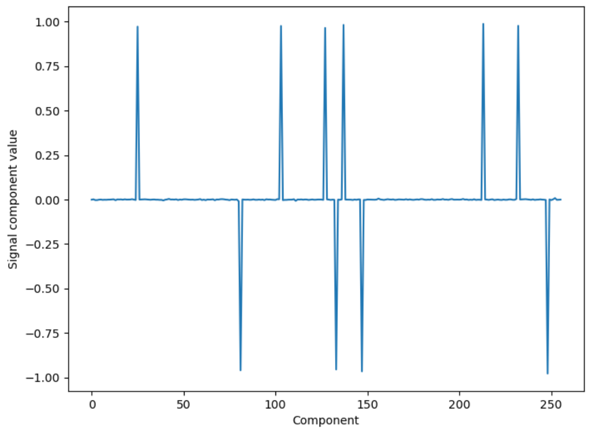
The stopping criterion used is partially derived from the Newton decrement through minimisation of the local second order Taylor expansion::

$$f(x) - \inf_y \tilde{f}(y) = \frac{1}{2} \mathbf{J^T H^{-1} J} \preceq \epsilon \tag{37}$$

where $\tilde{f}(y)$ is a local second order approximation about point x, with Jacobian J and Hessian H. A second condition checks if the 2-norm of the gradient is less than the threshold, and stops if either condition holds.



(a) Original signal.



(b) Reconstructed signal.

Figure 5: Original and reconstructed signals.

Starting value of t was set to be 0.1, episilon 0.1. In practice with this code, there is currently poor convergence for the central path method, but Newton's method produces a good reconstruction directly if t is set to 1e8.

4 Appendix

4.1 References

1. Boyd, S.P., Vandenberghe, L. *Convex Optimisation.* Cambridge University Press, 2004.

2. Sepulchre, R. *4M17 Notes, Linear Programming*, CUED 2022-2023.

4.2 Code Q1

```python
from scipy.optimize import linprog
from scipy.linalg import lstsq
import numpy as np
import timeit
import os
import matplotlib.pyplot as plt
from scipy import stats

script_dir = os.path.dirname(__file__)
rel_path = "Q1/"
abs_file_path = os.path.join(script_dir, rel_path)
output_dir = "Q1_results/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

def read_files(A, b):
    # Read csv file into numpy array
    A_filepath = os.path.join(abs_file_path, A+'.csv')
    b_filepath = os.path.join(abs_file_path, b+'.csv')
    A = np.genfromtxt(A_filepath, delimiter=',')
    b = np.genfromtxt(b_filepath, delimiter=',')
    return A, b

def solve_l1():
    """l1 norm approximation"""
    results = []
    runtimes = []
    ns = []
    for i in range(1, 6):
        a = 'A'+str(i)
        b = 'b'+str(i)
        A, b = read_files(a, b)
        m = len(b)
        n = len(A[0])
        ns.append(n)
        # Two stacked identity matrices of size m x m
        A_tilde_r = np.vstack([-np.identity(m), -np.identity(m)])
        # Stack A on top of its negative
        A_tilde_l = np.vstack([A, -A])
        A_tilde = np.hstack([A_tilde_l, A_tilde_r])
        c_tilde = np.concatenate((np.zeros((n)), np.ones(m)), axis=0)
        b_tilde = np.concatenate((b, -b), axis=0)

        start_time = timeit.default_timer()
        x = linprog(c_tilde, A_tilde, b_tilde).x
        runtimes.append(timeit.default_timer() - start_time)
        residual = np.dot(A, x[:n]) - b
        results.append(np.sum(np.abs(residual)))
        if i == 5:
            plot_residual(residual, 'l_1', dist=True, type=1)
    plot_runtime(ns, runtimes, 'l_1')
    plot_results(ns, results, 'l_1')
    print(results)
    print(runtimes)

def solve_l2():
    """least squares approximation"""
    results = []
    runtimes = []
    ns = []
    for i in range(1, 6):
        a = 'A'+str(i)
        b = 'b'+str(i)
        A, b = read_files(a, b)
        ns.append(len(A[0]))
        start_time = timeit.default_timer()
        x = lstsq(A, b)[0] # first return element is x array (see scipy docs)
        runtimes.append(timeit.default_timer() - start_time)
        residual = np.dot(A, x) - b
        results.append(np.linalg.norm(residual, 2))
        if i == 5:
            plot_residual(residual, 'l_2', dist=True, type=2)
    plot_runtime(ns, runtimes, 'l_2')
    plot_results(ns, results, 'l_2')
    print(results)
    print(runtimes)

def solve_linf():
    """l_inf norm approximation"""
    results = []
    runtimes = []
    ns = []
    for i in range(1, 6):
        a = 'A'+str(i)
        b = 'b'+str(i)
        A, b = read_files(a, b)
        m = len(b)
        n = len(A[0])
        ns.append(n)
        # Stack A on top of its negative
        A_tilde_l = np.vstack([A, -A])
        # Add a column of -1s
        A_tilde = np.hstack([A_tilde_l, -np.ones((2*m, 1))])
        c_tilde = np.concatenate((np.zeros((n)), np.ones(1)), axis=0)
        b_tilde = np.concatenate((b, -b), axis=0)

        start_time = timeit.default_timer()
        x = linprog(c_tilde, A_tilde, b_tilde).x
        runtimes.append(timeit.default_timer() - start_time)
        residual = np.dot(A, x[:n]) - b
        results.append(np.max(np.abs(residual)))
        if i == 5:
            plot_residual(residual, 'l_inf')
    plot_runtime(ns, runtimes, 'l_inf')
    plot_results(ns, results, 'l_inf')
    print(results)
    print(runtimes)

def plot_residual(x, title, dist=False, type=0):
    """Plot residual distribution histogram"""
    fig = plt.figure(figsize=(8,2))
    ax  = fig.add_subplot(111)
    n, bins, _ = ax.hist(x, bins=60, density=1)
    if dist == True and type == 2:
        mean, std = stats.norm.fit(x)
        p = stats.norm.pdf(bins, mean, std)
        ax.plot(bins, p, 'k', linewidth=2)
        print(mean, std)
    if dist == True and type == 1:
        # Fit Laplacian distribution to data
        mu, b = stats.laplace.fit(x)
        p = stats.laplace.pdf(bins, mu, b)
        ax.plot(bins, p, 'k', linewidth=2)
        print(mu, b)
    ax.set_title(title)
    ax.set_xlabel('Residual')
    ax.set_ylabel('Relative normalised occurrence')
    plt.savefig(output_dir+title+'.png')

def plot_runtime(n, runtimes, title):
    """Plot runtime vs n"""
    fig = plt.figure(figsize=(5,5))
    ax  = fig.add_subplot(111)
    ax.plot(n, runtimes)
    ax.set_xlabel('n')
    ax.set_ylabel('Runtime (s)')
    plt.savefig(output_dir+title+'_runtime.png')

def plot_results(n, results, title):
    """Plot residual vs n for all three norms"""
    fig = plt.figure(figsize=(5,5))
    ax  = fig.add_subplot(111)
    ax.plot(n, results, label='l_1')
    ax.set_xlabel('n')
    ax.set_ylabel('Residual')
    ax.legend()
    plt.savefig(output_dir+title+'_results.png')
```

4.3 Code Q2

```
import numpy as np
import os
import matplotlib.pyplot as plt
from Q1 import read_files

script_dir = os.path.dirname(__file__)
rel_path = "Q1/"
abs_file_path = os.path.join(script_dir, rel_path)
output_dir = "Q2_results/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# def f(x):
#     """Function to minimize. Test with Rosenbrock function"""
#     return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2
```

```python

# def f_grad(x):
#     """Gradient of function to minimize. Test with Rosenbrock function."""
#     return np.array([-2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0]**2), 200 * (x[1] - x[0]**2)])

def f(A, b, c, x, t=1):
    """Function to minimize. Returns a scalar."""
    if (b-np.matmul(A, x) > 0).all():
        return t*np.dot(c, x) - np.sum(np.log(-np.matmul(A, x) + b))
    else:
        return np.inf

def f_grad(A, b, c, x, t=1):
    """Gradient of function to minimize. Returns a vector."""
    return t*c + np.dot(A.T, 1 / (-np.dot(A, x) + b))

def get_data():
    A, b = read_files("A3", "b3")
    m = len(b)
    n = len(A[0])
    # Two stacked identity matrices of size m x m
    A_tilde_r = np.vstack([-np.identity(m), -np.identity(m)])
    # Stack A on top of its negative
    A_tilde_l = np.vstack([A, -A])
    A_tilde = np.hstack([A_tilde_l, A_tilde_r])
    c_tilde = np.concatenate((np.zeros((n)), np.ones(m)), axis=0)
    b_tilde = np.concatenate((b, -b), axis=0)
    return A_tilde, b_tilde, c_tilde

def backtrack(epsilon):
    """Backtracking line search"""
    func_history = []
    grad_history = []
    alpha = 0.3
    beta = 0.5
    A_tilde, b_tilde, c_tilde = get_data()
    # Initialise x to value inside feasible region
    x = c_tilde
    grad = f_grad(A_tilde, b_tilde, c_tilde, x)
    func = f(A_tilde, b_tilde, c_tilde, x)
    iteration = 0

    while np.linalg.norm(grad, ord=2) > epsilon:
        t = 1   # Start t=1 for line search
        # First ensure in feasible region
        while f(A_tilde, b_tilde, c_tilde, x-t*grad) == np.inf:
            t = beta*t
        # Now satisfy backtracking condition
        while f(A_tilde, b_tilde, c_tilde, x-t*grad) > func - alpha*t*np.dot(grad, grad):
            t = beta*t
        x = x - t*grad
        iteration += 1
        # Next step size t
        grad = f_grad(A_tilde, b_tilde, c_tilde, x)
        # Next function value
        func = f(A_tilde, b_tilde, c_tilde, x)
        func_history.append(func)
        grad_history.append(np.linalg.norm(grad, ord=2))
        if i % 100 == 0:
            print(func)
    func_history = np.array(func_history)-func_history[-1]
    print(func)
    plot_func_history(func_history, "function")
    plot_func_history(grad_history, "gradient")

def plot_func_history(f, title):
    """Plot function history"""
    fig = plt.figure(figsize=(8,5))
    ax  = fig.add_subplot(111)
    ax.plot(f)
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Error in function value")
    ax.set_yscale('log')
    plt.savefig(output_dir+title+"_history.png")
```

4.4 Code Q3

```
import numpy as np
import os
import matplotlib.pyplot as plt

script_dir = os.path.dirname(__file__)
rel_path = "Q3/"
abs_file_path = os.path.join(script_dir, rel_path)
output_dir = "Q3_results/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
# Read data
A_filepath = os.path.join(abs_file_path, 'A.csv')
x_filepath = os.path.join(abs_file_path, 'x0.csv')
A = np.genfromtxt(A_filepath, delimiter=',')
x0 = np.genfromtxt(x_filepath, delimiter=',')
b = np.matmul(A, x0)
m = len(b)
n = len(A[0])
# Lambda constant
lammy = 0.01*np.linalg.norm(2*np.matmul(A.T,b), ord=np.inf)

def f_original(X):
    """Barrier phi function to minimize. Returns a scalar."""
    x = X[:n]
    u = X[n:]
    if (u-x < 0).any() or (u+x < 0).any():
        return np.inf
    return np.linalg.norm(np.matmul(A, x) - b, ord=2) + lammy*np.sum(u)

def barrier(X):
    x = X[:n]
    u = X[n:]
    return - np.sum(np.log(u-x)) - np.sum(np.log(u+x))

def f(X, t):
    return t*f_original(X) + barrier(X)

def f_grad(X, t):
    """Gradient of function. Returns a vector where first m entries are wrt x and last m entries
    are wrt u."""
    x = X[:n]
    u = X[n:]
    f_grad_x = 2*t*np.matmul(A.T, np.matmul(A,x)-b) + 1/(u-x) - 1/(u+x)
    f_grad_u = t*lammy - 1/(u-x) - 1/(u+x)
    return np.concatenate((f_grad_x, f_grad_u))

def f_hess(X, t):
    """H_11. Returns a vector, which is the diagonals of the formal matrix."""
    x = X[:n]
    u = X[n:]
    # Vectors of diagonals of matrices
    H_11 = 2*t*np.matmul(A.T, A) + np.diag(1/np.square(u-x) + 1/np.square(u+x))
    H_12 = np.diag(-1/np.square(u-x) + 1/np.square(u+x))
    H_22 = np.diag(1/np.square(u-x) + 1/np.square(u+x))
    return np.block([[H_11, H_12],[H_12, H_22]])

def Newton(epsilon, t, X_0):
    """Newton's method for central path barrier.

    :param epsilon: tolerance for stopping condition
    :param t: central path parameter
    :param X_0: inital feasible point for X
    """
    # Inital feasible point for X
    X = X_0
    func = f(X, t)
    grad = f_grad(X, t)
    hess = f_hess(X, t)
    inv_hess = np.linalg.inv(hess)
    step_dir = -np.matmul(inv_hess, grad)
    nabla_f_grad = np.dot(grad.T, step_dir)
    alpha = 0.3 # defines comparison in backtracking line search
    beta = 0.9 # defines rate of decrease of tau, step size in each Newton iteration
    iteration = 0

    # Two-way stopping condition
    while 0.5*abs(np.dot(grad.T, step_dir)) > epsilon and np.linalg.norm(grad) > epsilon:
        tau = 1 # Initial Newton step size, tuned with line search

        #  Check feasbility
        u = X[n:]
        x = X[:n]
        backtrack_it = 0
        while (u-x <= 0).any() or (u+x <= 0).any():
            Xdash = X + tau*step_dir
            u = Xdash[n:]
            x = Xdash[:n]
            tau = beta*tau
            backtrack_it += 1
            if backtrack_it > 1000:
                print('Feasibility backtracking failed')
                break
        # Now satisfy backtracking condition
        backtrack_it = 0
        while f_original(X+tau*step_dir) > f_original(X) - alpha*tau*nabla_f_grad:
            tau = beta*tau
            backtrack_it += 1
            if backtrack_it > 500:
                print('Linesearch backtracking failed')
                break
        X = X + tau*step_dir
        func = f(X, t)
        grad = f_grad(X, t)
        hess = f_hess(X, t)
        inv_hess = np.linalg.inv(hess)
        step_dir = -np.matmul(inv_hess, grad)
        iteration += 1
        print(func)

    return X

def central_path(epsilon, t_init, t_max):
    t = t_init
    mu = 10 # defines rate of increase of t, central path parameter
    # Inital feasible point for X
    X = np.concatenate((np.ones(n)/2, np.ones(n)))
    while t < t_max:
        X = Newton(epsilon, t, X)
        t = mu*t
    return X

# Plot original and reconstructed signals
def plot(X, title):
    fig = plt.figure(figsize=(8,6))
    ax  = fig.add_subplot(111)
    ax.plot(X[:n])
    ax.set_ylabel('Signal component value')
    ax.set_xlabel('Component')
    plt.savefig(output_dir+title+"_signal.png")

X_0 = np.concatenate((np.ones(n)/2, np.ones(n)))
```

```
X = central_path(0.01, 0.1, 1e5)
print(X)
plot(X, "Reconstructed")
plot(x0, "Original")
```