Compressed Sensing

Practical Optimisation

Abhishek Shenoy

Abstract

This report investigates the use of convex optimisation to solve various norm approximation problems. We analyse various optimisation methods using the central path method after formulating them as linear programming problems.

1 Norm Approximation

The simplest norm approximation problem is an unconstrained problem of the following form.

$$\min ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||$$

$\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$ are given as problem data, $\boldsymbol{x} \in \mathbb{R}^n$ is the variable and $||\cdot||$ is a norm on $\mathbb{R}^m$. The dataset used for norm approximations below have dimension $m = 2n$.

1.1 Norm Approximation Problems

We investigate the norm approximation problem for the l_1, l_2 and l_∞ norm. The norm definitions for a vector $\boldsymbol{y} \in \mathbb{R}^m$ are given by the following:

$$\begin{aligned} l_1(\boldsymbol{y}) &= ||\boldsymbol{y}||_1 = \sum_{i=1}^{m} |y_i| \\ l_2(\boldsymbol{y}) &= ||\boldsymbol{y}||_2 = (\sum_{i=1}^{m} y_i^2)^{\frac{1}{2}} \\ l_\infty(\boldsymbol{y}) &= ||\boldsymbol{y}||_\infty = \max_{i \in (1,m)} |y_i| \end{aligned}$$

Both L_1 and L_∞ norm approximations have no closed-form solutions. In fact L_2 is the only case with an analytic solution.

In the below cases, $\tilde{\boldsymbol{A}} \in \mathbb{R}^{2m \times (n+m)}, \tilde{\boldsymbol{x}} \in \mathbb{R}^{n+m}, \tilde{\boldsymbol{c}} \in \mathbb{R}^{n+m}, \tilde{\boldsymbol{b}} \in \mathbb{R}^{2m}$.

1.1.1 L_1 Norm Approximation

$$\begin{aligned} \min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_1 &= \min_{\boldsymbol{x}} \sum_{i=1}^{m} |\boldsymbol{a}_i^T \boldsymbol{x} - b_i| \\ &= \min_{\boldsymbol{x}, t_i} \sum_{i=1}^{m} t_i \quad \{-t_i \leq \boldsymbol{a}_i^T \boldsymbol{x} - b_i \leq t_i\}_{i=1}^m \end{aligned}$$

This can be solved via the following linear program:

$$\begin{aligned} &\min_{\tilde{\boldsymbol{x}}} \tilde{\boldsymbol{c}}^T \tilde{\boldsymbol{x}} \\ &\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} \leq \tilde{\boldsymbol{b}} \end{aligned}$$

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

1.1.2 L_∞ Norm Approximation

$$\begin{aligned} \min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_\infty &= \min_{\boldsymbol{x}} \max_i |\boldsymbol{a}_i^T \boldsymbol{x} - b_i| \\ &= \min_{\boldsymbol{x}, t} \; t \quad \{-t \leq \boldsymbol{a}_i^T \boldsymbol{x} - b_i \leq t\}_{i=1}^m \end{aligned}$$

This can be solved via the following linear program:

$$\begin{aligned} &\min_{\tilde{\boldsymbol{x}}} \tilde{\boldsymbol{c}}^T \tilde{\boldsymbol{x}} \\ &\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} \leq \tilde{\boldsymbol{b}} \end{aligned}$$

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{t} \end{bmatrix} \quad \tilde{\boldsymbol{c}} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{1} \end{bmatrix} \quad \tilde{\boldsymbol{A}} = \begin{bmatrix} \boldsymbol{A} & -\boldsymbol{1} \\ -\boldsymbol{A} & -\boldsymbol{1} \end{bmatrix} \quad \tilde{\boldsymbol{b}} = \begin{bmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \end{bmatrix}$$

1.1.3 L_2 Norm Approximation

The L_2 norm approximation problem can be formulated using matrices and vectors leading to an analytic solution for the convex quadratic function.

$$\begin{aligned} \min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2 &= \min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 \\ &= \min_{\boldsymbol{x}} (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) \end{aligned}$$

By differentiating with respect to $\boldsymbol{x}$, we can find the least-squares solution.

$$\begin{aligned} \frac{d}{d\boldsymbol{x}} (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) &= 0 \\ \frac{d}{d\boldsymbol{x}} (\boldsymbol{x}^T \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{x}^T \boldsymbol{A}^T \boldsymbol{b} + \boldsymbol{b}^T \boldsymbol{b}) &= 0 \\ 2(\boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^T \boldsymbol{b}) &= 0 \\ \therefore \boldsymbol{A}^T \boldsymbol{A}\boldsymbol{x} &= \boldsymbol{A}^T \boldsymbol{b} \\ \boldsymbol{x} &= (\boldsymbol{A}^T \boldsymbol{A})^{-1} \boldsymbol{A}^T \boldsymbol{b} \end{aligned}$$

1.2 Norm Comparisons

$\boldsymbol{A}, \boldsymbol{b}$ Dataset	Size n	Minimised Values l_1	l_2	l_∞	Runtime (ms) l_1	l_2	l_∞
$(\boldsymbol{A}_1, \boldsymbol{b}_1)$	16	12.48	2.1	0.77	8.1	0.9	10.0
$(\boldsymbol{A}_2, \boldsymbol{b}_2)$	64	52.45	5.45	0.82	52.1	1.4	49.5
$(\boldsymbol{A}_3, \boldsymbol{b}_3)$	256	194.88	9.36	0.80	881.5	41.0	775.2
$(\boldsymbol{A}_4, \boldsymbol{b}_4)$	512	402.03	13.86	0.80	8288.3	129.0	6389.1
$(\boldsymbol{A}_5, \boldsymbol{b}_5)$	1024	808.43	18.71	0.79	56359.6	493.0	44112.0

Table 1. Minimisation of different norms of $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||$

The minimised values are of the correct magnitude as we would expect the l_1 norm to be the largest as it is the absolute sum, l_2 norm to be the next largest as it is the square-root of the sum of the squares and l_∞ norm to be the smallest as it is the maximum component of a vector.

The least-squares solution is by far the easiest and quickest to compute due to the analytical solution rather than taking an optimisation approach.

l_1 has a relatively faster runtime as compared to l_∞. It is also possible to see the non-linearity in the increase in runtime as n increases. For $n = 256, 512, 1024$, we see that for every factor 2 increase in n there is an approximate factor 4 increase in runtime for l_2 however for l_1 and l_∞ the increase in runtime is approximately by factor 8. We do however see a rough factor of $16(4^2)$ increase in runtime for all the 3 norms between $n = 64$ to $n = 256$ which is as expected. Below $n = 64$ all the norms can be computed relatively fast such that the speed difference is of the expected factor but the runtimes are so small that the differences are wide. l_2 is able to compute extremely fast even for $n = 64$ almost at the same speed as that of $n = 16$ possibly taking advantage of precompiled Numpy libraries for least-squares.

1.3 Norm Residuals

We plot the histogram of the residuals of the norm approximation problem for the data pair $(\boldsymbol{A}_5, \boldsymbol{b}_5)[n = 1024]$ for the l_1, l_2, l_∞-norms.

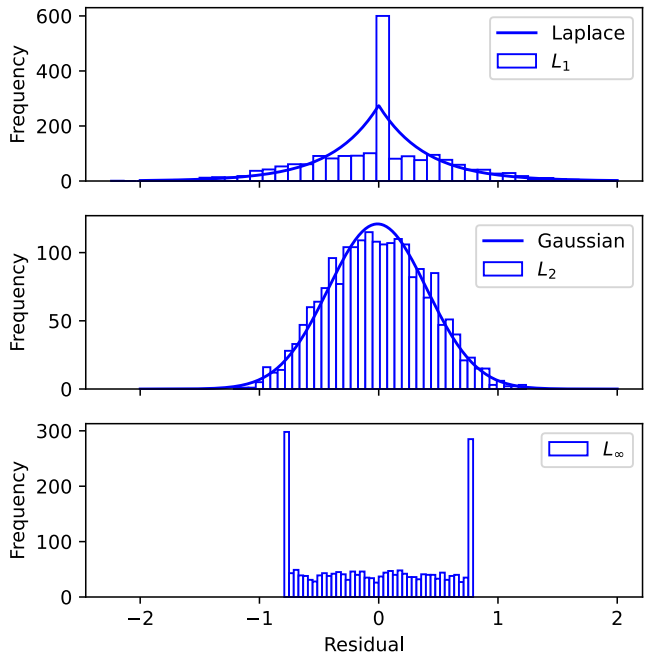


Figure 1. Histogram of the residuals of the norm approximation problem for $(\boldsymbol{A}_5, \boldsymbol{b}_5)$

The amplitude distribution of the optimal residual for the l_1-norm approximation problem will tend to have more zero and very small residuals, compared to the l_2-norm approximation solution. In contrast, the l_2-norm solution will tend to have relatively fewer large residuals (since large residuals incur a much larger penalty in l_2-norm approximation than in l_1-norm approximation). This is visibly clear in Figure 1 as the Laplace distribution for l_1 has a much higher density at 0 than the Gaussian distribution and the fat tails of the Laplace also means that there is a greater number of large residuals for l_1 than for l_2 due to the penalisation by the l_2-norm. The l_2-norm approximation has many modest residuals, and relatively few larger ones.

In the case of the l_∞-norm, we get a uniform distribution with heavy weightings on the largest outliers. This is explicable as the l_∞-norm returns the maximum component.

If you minimise the negative log-likelihood of the error distribution, you get the best estimator. For a Gaussian distribution, this results in a sum of squares minimization. For Laplace, this results in a sum of absolute value minimization. Hence a good fit for the data for a given norm occurs when the residual distribution matches the distribution corresponding to the norm used.

2 Central Path Formulation

Consider the following optimisation problem:

$$\begin{aligned} &\min_{\boldsymbol{x}} f_0(\boldsymbol{x}) \\ &\text{subject to } f_i(\boldsymbol{x}) \leq 0 \qquad i = 1, ..., m \end{aligned}$$

f_0, f_i are convex and twice continuously differentiable functions for $i = 1, ..., m$.

The central path is the solution of the unconstrained minimisation of the following problem:

$$\min_{\boldsymbol{x}} t f_0(\boldsymbol{x}) + \phi(\boldsymbol{x}) \qquad t \geq 0$$

ϕ is the logarithmic barrier function of the feasibility set $S = \{\boldsymbol{x} : f_i(\boldsymbol{x}) \leq 0\}$.

2.1 L_1 CP Formulation

We consider the central path LP formulation of the L_1 norm approximation problem expressed in Section 1.1.1 using the matrix $\tilde{\boldsymbol{A}}$ and vectors $\tilde{\boldsymbol{x}}$, $\tilde{\boldsymbol{b}}$, $\tilde{\boldsymbol{c}}$ previously defined.

$$\tilde{\boldsymbol{x}}^*(t) = \min_{\tilde{\boldsymbol{x}}} \; t\,(\tilde{\boldsymbol{c}}^T \tilde{\boldsymbol{x}}) \underbrace{- \sum_{i=1}^{m} \ln(\tilde{b}_i - \tilde{\boldsymbol{a}}_i^T \tilde{\boldsymbol{x}})}_{+\phi(\tilde{\boldsymbol{x}})}$$

A unique $\tilde{\boldsymbol{x}}^*(t)$ exists for all $t > 0$. This set of values form the central path from the centre of the polyhedron at $t = 0$ to the edges of the polyhedron at $t = \infty$. The barrier function corresponding to a constraint $\tilde{\boldsymbol{A}}\tilde{x} \leq \tilde{\boldsymbol{b}}$ forms the boundary of the polyhedron.

The gradient of the cost for a fixed $t \geq 0$ can be calculated by differentiating with respect to $\tilde{\boldsymbol{x}}$ as the following.

$$\text{Gradient} \quad t\tilde{\boldsymbol{c}} + \underbrace{\sum_{i=1}^{m} \frac{1}{\tilde{b}_i - \tilde{\boldsymbol{a}}_i^T \tilde{\boldsymbol{x}}} \tilde{\boldsymbol{a}}_i}_{\nabla\phi(\tilde{\boldsymbol{x}})}$$

Defining a vector $\boldsymbol{d}$ for which $d_i = \frac{1}{\tilde{b}_i - \tilde{\boldsymbol{a}}_i^T \tilde{\boldsymbol{x}}}$, the gradient can be simplified to the below.

$$\text{Gradient} \quad t\tilde{\boldsymbol{c}} + \tilde{\boldsymbol{A}}^T \boldsymbol{d}$$

The solution for the non-t term $\tilde{\boldsymbol{A}}^T \boldsymbol{d} = 0$ where $\boldsymbol{d}$ is a function of $\boldsymbol{x}$ gives the analytic centre $\boldsymbol{x}_{ac}$. The term with t essentially creates a path with the variation of t known as the central path. $\boldsymbol{x}_{ac}$ exists and is unique if and only if the polyhedron (Dikin ellipsoid) is non-empty and bounded (Vandenberghe, 2013).

As we move along the central path, the contour gets closer to the actual polyhedron. Because of this geometry, the log-barrier method is called an interior-point method, since we start at an interior point and move along the central path to get to the solution.

2.2 Backtracking Linesearch

We can solve for the solution by minimising the gradient (setting the gradient to 0). For this particular case, we consider $t = 1$ corresponding to the pair $(\boldsymbol{A}_3, \boldsymbol{b}_3)[n = 256]$. We can use many methods to solve the problem however here we specifically focus on using a first-order gradient method with backtracking linesearch to analyse the convergence properties.

Algorithm 1 Backtracking Linesearch

given a descent direction $\Delta \boldsymbol{x}$ for f at $\boldsymbol{x} \in \mathbf{dom} f$
$\alpha \in (0, 0.5), \beta \in (0, 1)$
$t \leftarrow 1$
while $f(\boldsymbol{x} + t\,\Delta \boldsymbol{x}) > f(\boldsymbol{x}) + \alpha t \nabla f(\boldsymbol{x})^T \Delta \boldsymbol{x}$ **do**
  $t \leftarrow \beta t$
end while

The method for the backtracking linesearch has been shown in Algorithm 1 for which there are several parameters that we need to pick carefully (Boyd and Vandenberghe, 2004).

This specific problem is convex and differentiable, and ∇f is Lipschitz continuous with constant $L > 0$. Therefore gradient descent can be used. Gradient descent with backtracking for a given β satisfies the following inequality:

$$f(\boldsymbol{x}_k) - \underbrace{f(\boldsymbol{x}^*)}_{p^*} \leq \frac{||\boldsymbol{x}_0 - \boldsymbol{x}^*||^2}{2 t_{min} k} \quad t_{min} = \min(1, \beta/L)$$

Gradient descent with a fixed step-size t (ie. $t_{min} = t$) has convergence rate $O(1/k)$ for k updates of $\boldsymbol{x}$. However with backtracking for $t_{min} = \min(1, \beta/L)$, this is only true if β is not too small so that we do not lose much compared to fixed step size (Gordon and Tibshirani, 2012).

2.2.1 Hyperparameters

α is typically chosen between 0.01 and 0.5, meaning that we accept a decrease in f between 1% and 50% of the prediction based on the linear extrapolation. β is often chosen to be between 0.1 (corresponding to a very crude search) and 0.8 (corresponding to a less crude search).

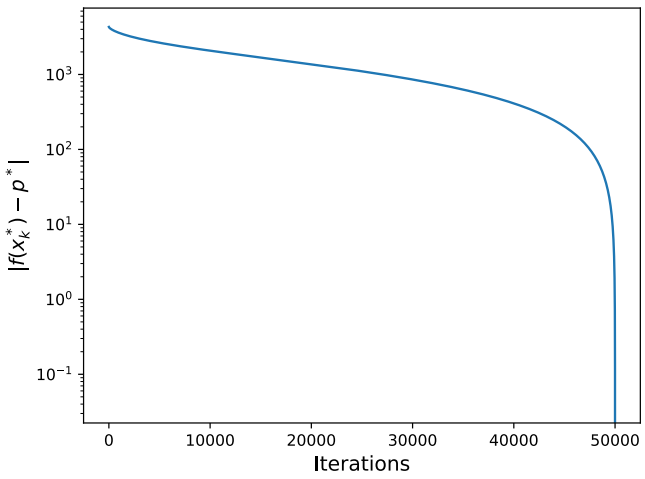


Figure 2. Linear convergence of backtracking linesearch on semi-log plot

2.3 Convergence Analysis

Figure 2 shows linearity up to the point of convergence using the gradient method with backtracking linesearch where $p*$ is the local optimum reached when the initialisation is $t = 1$.

As the number of iterations increases $k \to \infty$, the error must tend toward 0 as $|f(\boldsymbol{x}_k) - p*| \to 0$ and hence $f(\boldsymbol{x}_k) - p^* \leq d \cdot c^k$ for some constants c and d. More specifically this is given by the following:

$$\frac{f(\boldsymbol{x}_k) - p^*}{f(\boldsymbol{x}_0) - p^*} \leq c^k$$

Therefore the error $f(\boldsymbol{x}_k) - p^*$ converges to zero at least as fast as a geometric series. In the context of iterative numerical methods, this is called linear convergence since it is linear on a log-linear plot of error versus iteration number.

For the convex function f where $m\boldsymbol{I} \preccurlyeq \nabla^2 f \preccurlyeq M\boldsymbol{I}$ for the positive constants m and M, the bound for the condition number of the Hessian is given by M/m. For the case of gradient descent with exact linesearch, $c = 1 - m/M$. For gradient descent with backtracking linesearch, $c = 1 - \min(2m\alpha, 2\beta\alpha m/M)$. Therefore it is clear that if the condition number of the Hessian is too large, then the rate of convergence slows down. Nonetheless the convergence is at least linear (in the sense of iterative methods).

Note however that backtracking does not effectively find the global solution for the formulated problem as in Section 1.1.1 as t varies from 0 to ∞. By checking the condition number of the Hessian $\nabla^2 \phi(\tilde{\boldsymbol{x}}) = \boldsymbol{A}^T diag(\boldsymbol{d})^2 \boldsymbol{A}$, we find that it is extremely large hence explaining the slow convergence (Boyd and Vandenberghe (2004) Pg 475).

3 LASSO Problem

Consider the following l_1-regularised least squares problem where $\lambda > 0$ is the regularisation parameter and t varies from 0 to ∞:

$$\min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda ||\boldsymbol{x}||_1 \tag{1}$$

This problem is also known as the basis pursuit denosing problem (**BPDN**) / least absolute shrinkage and selection operator (**LASSO**) which always has a solution but is not necessarily unique.

Since there is no analytical solution, we need to express this either as a differentiable optimisation problem or using an iterative method to compute the solution.

3.1 Convex Quadratic Formulation

Equation 1 can be transformed to a convex quadratic problem with linear inequality constraints as shown in Equation 2.

By considering a new variable $u_i = |x_i|$ we define the inequality constraint as $-x_i < u_i < x_i$ which we can now use to define the logarithmic barrier function. Since there are two inequalities, there is a requirement of two barrier functions which we can combine into a single barrier function.

$$\begin{aligned}
\phi_t(\boldsymbol{x}, \boldsymbol{u}) &= t\Big(||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda \sum_{i=1}^{n} u_i\Big) + \Phi(\boldsymbol{x}, \boldsymbol{u}) \\
&= t\Big(||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda \boldsymbol{u}^T \boldsymbol{1}\Big) + \Phi(\boldsymbol{x}, \boldsymbol{u}) \\
\Phi(\boldsymbol{x}, \boldsymbol{u}) &= -\sum_{i=1}^{n} \ln(u_i + x_i) - \sum_{i=1}^{n} \ln(u_i - x_i) \\
&= -\sum_{i=1}^{n} \ln(u_i^2 - x_i^2)
\end{aligned} \tag{2}$$

Note that the equivalent mathematical notation for element-wise operations for $\Phi(\boldsymbol{x}, \boldsymbol{u})$ that can be converted into Python code is given by the following:
$\Phi(\boldsymbol{x}, \boldsymbol{u}) = -\sum_{i=1}^{n} (\ln \circ (\boldsymbol{u}^{\circ 2} - \boldsymbol{x}^{\circ 2}))_i$

3.2 Gradient and Hessian

We can now derive the gradient $\boldsymbol{g} = \nabla \phi_t$ and the Hessian $\boldsymbol{H} = \nabla^2 \phi_t$ similarly using Hadamard notation for element-wise operations where $\circ$ denotes element-wise function application, $\odot$ denotes element-wise multiplication and $\oslash$ denotes element-wise division.

$$\begin{aligned}
\boldsymbol{g} = \nabla \phi_t(\boldsymbol{x}, \boldsymbol{u}) &= \begin{bmatrix} \nabla_x \phi_t(\boldsymbol{x}, \boldsymbol{u}) \\ \nabla_u \phi_t(\boldsymbol{x}, \boldsymbol{u}) \end{bmatrix} \\
&= \begin{bmatrix} 2t\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + 2\boldsymbol{x} \oslash (\boldsymbol{u}^{\circ 2} - \boldsymbol{x}^{\circ 2}) \\ t\lambda \boldsymbol{1} - 2\boldsymbol{u} \oslash (\boldsymbol{u}^{\circ 2} - \boldsymbol{x}^{\circ 2}) \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{H} = \nabla^2 \phi_t(\boldsymbol{x}, \boldsymbol{u}) &= t\nabla^2 ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \nabla^2 \Phi(\boldsymbol{x}, \boldsymbol{u}) \\
&= t\nabla^2 (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + \nabla^2 \Phi(\boldsymbol{x}, \boldsymbol{u}) \\
&= \begin{bmatrix} 2t\boldsymbol{A}^T\boldsymbol{A} + \boldsymbol{D}_1 & \boldsymbol{D}_2 \\ \boldsymbol{D}_2 & \boldsymbol{D}_1 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{D}_1 &= \textbf{diag}(2(\boldsymbol{u}^{\circ 2} + \boldsymbol{x}^{\circ 2}) \oslash (\boldsymbol{u}^{\circ 2} - \boldsymbol{x}^{\circ 2})^{\circ 2}) \\
&= \textbf{diag}\Big(\frac{2(u_1^2 + x_1^2)}{(u_1^2 - x_1^2)^2}, ..., \frac{2(u_n^2 + x_n^2)}{(u_n^2 - x_n^2)^2}\Big) \\
\boldsymbol{D}_2 &= \textbf{diag}\Big(-4(\boldsymbol{u} \odot \boldsymbol{x}) \oslash (\boldsymbol{u}^{\circ 2} - \boldsymbol{x}^{\circ 2})^{\circ 2}\Big) \\
&= \textbf{diag}\Big(\frac{-4u_1 x_1}{(u_1^2 - x_1^2)^2}, ..., \frac{-4u_n x_n}{(u_n^2 - x_n^2)^2}\Big)
\end{aligned}$$

3.3 Sparse Signal Reconstruction

We consider a sparse signal recovery problem with a signal $\boldsymbol{x}_0 \in \mathbb{R}^{256}$ which consists of 10 spikes of amplitude ± 1. The measurement matrix $\boldsymbol{A} \in \mathbb{R}^{60 \times 256}$ and $\boldsymbol{x}_0$ and the vector of observations is $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}_0$.

We perform a sparse signal reconstruction by applying a Newton interior-point method to Equation 2 with regularisation parameter $\lambda = 0.01\lambda_{max}$ with $\lambda_{max} = |2\boldsymbol{A}^T\boldsymbol{b}|_\infty$ and plot the original and reconstructed signals (Fig 3).

This problem is an example of **compressed sensing** and solving this is extremely useful in signal processing for efficiently acquiring and reconstructing a signal by finding solutions to under-determined linear systems.

Note that we are explicitly investigating $\lambda = 0.01\lambda_{max}$ but if we wanted to determine the optimum λ, we could use the Morozov discrepancy principle. Since we have the desired signal, we can determine this optimum value of λ by plotting the variation of the reconstruction error $||\boldsymbol{x} - \boldsymbol{x}^*||$ with λ and picking λ which has the minimum error. Of course, in practice this method is not practically viable since we have to know the solution in advance and hence the Morozov discrepancy principle is used (Leykekhman, 2018).

3.3.1 Newton Interior-Point Method

Algorithm 2 Newton IPM for l_1-regularised LSPs

Given relative tolerance $\epsilon > 0$,$t \leftarrow \frac{1}{\lambda}$
$\boldsymbol{x} = \boldsymbol{0}, \boldsymbol{u} = \boldsymbol{1}$
while Stop Criterion $\leq \epsilon$ **do**
  Compute search direction $(\Delta\boldsymbol{x}, \Delta\boldsymbol{u})$ using Newton
  Compute step size τ by backtracking linesearch
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \tau\Delta\boldsymbol{x}$
  $\boldsymbol{u} \leftarrow \boldsymbol{u} + \tau\Delta\boldsymbol{u}$
  Update t
end while

When an iterative method is used to approximately solve the Newton system, the overall method is called a truncated Newton method.

To compute the search direction $\Delta\boldsymbol{d}$, we need to solve the following equation where $\boldsymbol{H}$ is the Hessian and $\boldsymbol{g}$ is the gradient.

$$\boldsymbol{H} \underbrace{\begin{bmatrix}\Delta \boldsymbol{x} \\ \Delta \boldsymbol{u}\end{bmatrix}}_{\Delta \boldsymbol{d}} = -\boldsymbol{g}$$

Instead of using the truncated Newton interior-point method (TNIPM) (approximating the search direction) for solving this task as done in Kim et al. (2007) using pre-conditioned gradient (PCG) methods (Huang et al., 2018), we directly compute the search-direction using the Hessian hence this is the full Newton interior-point method defined in Algorithm 2. This is only feasible in this problem because we know that the signal that needs to be reconstructed is relatively small and sparse specifically with only 10 non-zero elements out of the 256 components. Generally it is possible to approximate the Newton gradient and search direction by decomposing it into a matrix for which only the diagonal of the Hessian is required to be computed (Bickson and Dolev, 2010).

We explored 2 methods for using a NIPM specifically using the barrier method (BM) with Newton as well as the custom interior-point method (CIPM) defined in Kim et al. (2007).

3.3.2 t-Update Methods

Updating t is needed to traverse the central path. This needs to be chosen appropriately although it is very flexible.

We considered two methods of updating t specifically the use of the update rule $t = \mu t$ as well as the update rule $t = \max(\mu \min(\frac{2n}{\eta}, t), t)$ if $\tau \geq \tau_{min}$ defined in Kim et al. (2007) where η is the duality gap and n is the dimension of $\boldsymbol{x}$.

Barrier Method
The barrier method uses $t = \mu t$ with $\mu > 1$ (Boyd and Vandenberghe (2004) Pg 569) as the update rule for t.

The choice of the parameter μ involves a trade-off in the number of inner and outer iterations required. If μ is small (near 1) then at each outer iteration t increases by a small factor and therefore we require fewer Newton steps (inner iterations) to converge to the new centre but require larger number of outer iterations to move to the edge of the polyhedron along the central path. We also have to consider the initial value t_0 for the same reason. The value for μ and t_0 can therefore be determined by experimentation (Boyd and Vandenberghe (2004) Pg 570).

CIPM
The CIPM that we investigate uses the update rule $t = \max(\mu \min(\frac{2n}{\eta}, t), t)$ if $\tau \geq \tau_{min}$ defined in Kim et al. (2007) where η is the duality gap and n is the dimension of $\boldsymbol{x}$.

This update rule uses the step size τ as a crude measure of proximity to the central path. In particular, $\frac{2n}{\eta}$ is the value of t for which the associated central point has the same duality gap as the current point.

3.3.3 Termination Conditions

We considered multiple termination conditions during experimentation and although any one of these can be used, there is an additional cost associated with computing certain conditions and each condition also has to be calibrated appropriately so that convergence occurs even in the event of an inflection plateau.

Barrier Method
The barrier method uses a relative simple condition of $\frac{n}{t} < \epsilon$ where n is equal to the number of inequality conditions (in this case, also the dimension of $\boldsymbol{x}$). Note that $\frac{n}{t}$ is the duality gap and hence can be used as a termination condition.

CIPM
As a stopping criterion, the method uses the duality gap divided by the dual objective value. By weak duality, the ratio is an upper bound on the relative suboptimality where $p*$ is the optimal value of the l_1-regularized LSP and $f(\boldsymbol{x})$ is the primal objective computed with the point $\boldsymbol{x}$.

$$\frac{|f(\boldsymbol{x}) - p*|}{|p*|} \leq \frac{\eta}{G(\nu)}$$

Therefore we can use the duality term as the bound for termination:

$$\frac{\eta}{G(\nu)} < \epsilon$$

We can compute the dual feasible point ν, dual objective $G(\nu)$ and duality gap η as defined in (Kim et al., 2007).

$$\nu = 2\tau(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$$

The step size τ computed from the line search is then updated to the following:

$$\tau = \min(\lambda/(2|\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}))|)$$

We can now compute the dual objective $G(\nu)$ and the duality gap η:

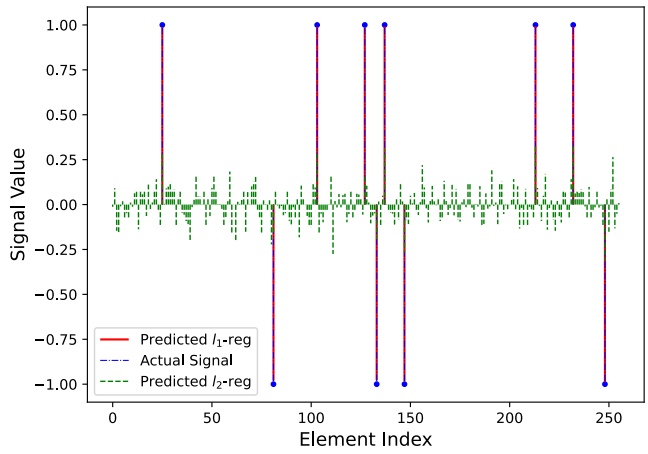
$$G(\nu) = -\frac{1}{4}\nu^T\nu - \nu^T\boldsymbol{b}$$
$$\eta = ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda||\boldsymbol{x}||_1 - G(\nu)$$

Other
We also experimented with other termination conditions to check if these worked better.

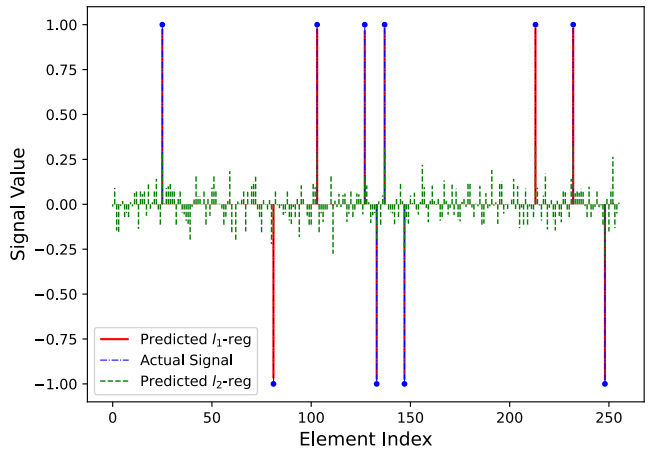
Similar to the derivation of CIPM, we can use a termination condition more commonly used for conjugate gradient methods which uses the gradients instead of the objective values.

$$\frac{|\nabla^2 f(\boldsymbol{x})\Delta\boldsymbol{x} + \nabla f(\boldsymbol{x})|}{|\nabla f(\boldsymbol{x})|} < \epsilon$$

We could also use the simplest condition usually used in gradient descent we could check if the magnitude of the gradient is less than some tolerance level which must be determined by experimentation.



(a) Barrier method (152 iterations)



(b) CIPM (573 iterations)

Figure 3. l_1 signal reconstruction using NIPMs (CIPM and barrier method) with l_2 analytical solution

$$|\boldsymbol{g}| < \epsilon$$

Note that both of these failed to provide any improvements and in fact performed worse. The LASSO problem is not particularly suited for gradient based approaches and we find that the optimal solution is amidst large gradients.

3.3.4 Results

When we attempt to reconstruct the signal, we find that both methods find the optimal solution however it is important to note that the barrier method is able to find the optimum in fewer iterations with a lower computational cost than the CIPM defined in Kim et al. (2007) (Fig 3). It is unclear what benefits the more expensive t-update rule and the termination condition provides. The CIPM termination condition seemingly does not provide guaranteed convergence unlike the barrier method. Moreover the proposed truncated version of the algorithm actually uses inexact search directions for which the convergence is not proved. The PCG if implemented however would likely provide a performance improvement over computing the Hessian directly, especially for dimensionally larger signals.

3.4 Minimum Energy Comparison

To evaluate the performance of the solution for the l_1-regularised problem, we compare the result to the minimum energy reconstruction for the solution that is closest to the origin (ie. the l_2-regularised problem). This is also known as **ridge regression**.

$$\min_{\boldsymbol{x}} ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda||\boldsymbol{x}||_2^2$$

The analytical solution for this problem can be similarly derived as done for the least-squares problem (Eq 3).

$$\begin{aligned} f_{l_2}(\boldsymbol{x}) &= ||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}||_2^2 + \lambda||\boldsymbol{x}||_2^2 \\ &= (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) + \lambda\boldsymbol{x}^T\boldsymbol{x} \\ &= \boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b} + \lambda\boldsymbol{x}^T\boldsymbol{x} \end{aligned}$$

$$\begin{aligned} \frac{d}{d\boldsymbol{x}} f_{l_2}(\boldsymbol{x}) = 2\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x} - 2\boldsymbol{A}^T\boldsymbol{b} + 2\lambda\boldsymbol{I}\boldsymbol{x} \qquad &= 0 \\ (\boldsymbol{A}^T\boldsymbol{A} + \lambda\boldsymbol{I})\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b} & \\ \therefore \boldsymbol{x}_{l_2} = (\boldsymbol{A}^T\boldsymbol{A} + \lambda\boldsymbol{I})^{-1}\boldsymbol{A}^T\boldsymbol{b} & \end{aligned} \tag{3}$$

Note that $\boldsymbol{A}^T\boldsymbol{A} + \lambda\boldsymbol{I}$ is always invertible for $\lambda > 0$ and hence this solution is directly computable.

Also note that an equivalent solution $\boldsymbol{x}_{l_2}$ via the use of Lagrange multipliers gives $\boldsymbol{x}_{l_2} = \boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{A}^T)^{-1}\boldsymbol{b}$ (Stankovic et al. (2019) Pg 21).

Objective \ Solution	$\boldsymbol{x}_{l_1}$ BM	$\boldsymbol{x}_{l_1}$ CIPM	$\boldsymbol{x}_{l_2}$	$\boldsymbol{x}_{opt}$
$f_{l_1}(\boldsymbol{x})$	0.06722	0.06805	0.13551	0.06767
$f_{l_2}(\boldsymbol{x})$	0.06624	0.06499	0.01797	0.06767

Table 2. Minimised values of the l_1 and l_2 regularised problem for the different solutions

From Table 2, we see that the l_2 solution is far from the optimal solution of the l_1 problem. We notice that the true l_1 solution $\boldsymbol{x}_{opt}$ has the same function value for both objective functions which is optimal for the l_1 problem but sub-optimal for the l_2 problem. Therefore it is clear that the global minimum of the l_2-regularised problem is not the same as the global minimum of the l_1-regularised problem as seen in Figure 3 and the corresponding Table 2.

This is because the standard ridge regression, based on the l_2-norm, minimizes the energy of solution $\boldsymbol{x}$ and not its sparsity and hence this is not useful for reconstructing the desired signal as the signal has very few non-zero components. The l_1 norm regularisation penalty puts the most weight on small residuals and the least weight on large values hence encouraging sparsity.

Table 2 shows that $\boldsymbol{x}_{l_1}$ BM is marginally better than $\boldsymbol{x}_{l_1}$ CIPM. Including the lower computational cost for BM, BM performs much better for sparse reconstruction in this specific case.

4 Discussion

The Newton method we used to perform signal reconstruction is in fact very inefficient and alternative methods such as iterative soft thresholding algorithm (**ISTA**) are computationally much cheaper. ISTA is specifically designed for reconstruction of sparse signals for the LASSO problem.

We create an implementation of ISTA and compare the results with the 2 NIPMs we investigated.

$$\text{soft}(x) = \text{sgn}(x)\max(0, |x| - \frac{\lambda}{2\alpha})$$
$$\boldsymbol{x}_{k+1} = \text{soft} \circ \left(\boldsymbol{x}_k + \frac{1}{\alpha}\boldsymbol{A}^T(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k)\right)$$

The parameter α is usually defined as $2\max(\text{eig}(\boldsymbol{A}^T\boldsymbol{A}))$. The soft function is the soft-thresholding rule and the iteration step can be used as many times as required to converge to a solution. Hence this method is known as the iterative soft-thresholding algorithm (ISTA). Note that this is just one of possible solutions of the minimization problem with the l_1-norm (Stankovic et al. (2019) Pg 20).

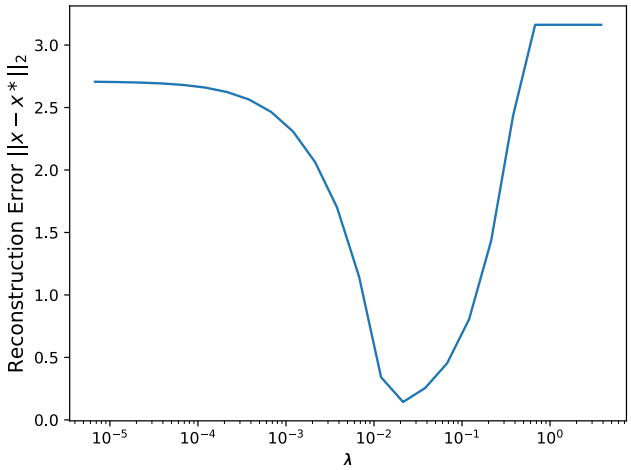


Figure 4. Investigating λ for reconstruction

At the point of minimum reconstruction error $\lambda \approx 0.02$ (Fig 4), we find that we require very few iterations for the algorithm to converge to the optimal solution in Figure 5. This is a good improvement over NIPMs both computationally as well as for accuracy.

Given by the decrease in error in Figure 5, we also see that the solution found is indeed optimal for our choice of λ. However this still requires more iterations than the barrier method for optimal convergence but it is important to note that each iteration is much more efficient than the barrier method. It is also important to note that we do not need to pick the λ that minimises the reconstruction error to find the optimal solution (as this requires knowledge of the optimal solution in the first place) but λ still needs to be chosen appropriately ($0.003 \leq \lambda \leq 0.3$).

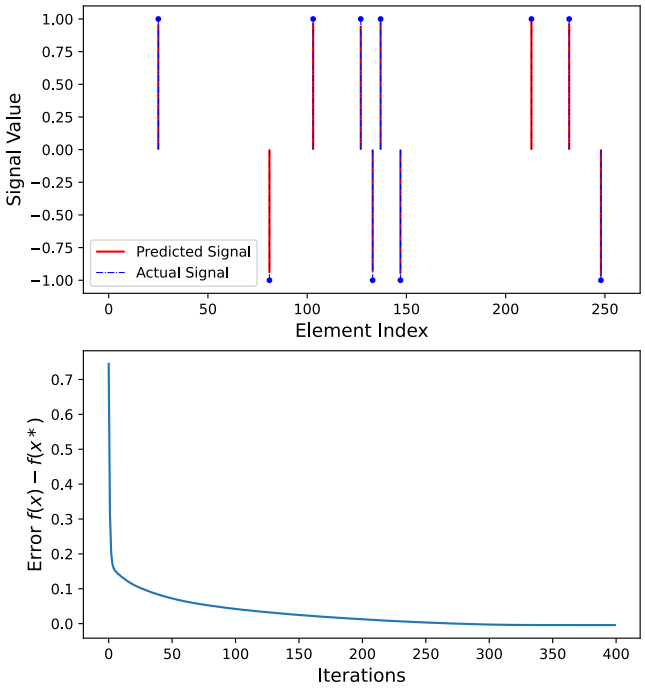


Figure 5. Reconstructed signal using ISTA for $\lambda \approx 0.02$ (400 iterations)

References

Lieven Vandenberghe. Central Path, 2013. URL http://www.seas.ucla.edu/~vandenbe/ee236a/lectures/cpath.pdf.

Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Geoff Gordon and Ryan Tibshirani. Gradient Descent Revisited, 2012. URL https://www.cs.cmu.edu/~ggordon/10725-F12/slides/05-gd-revisited.pdf.

Dmitriy Leykekhman. Regularized Linear Least Squares, 2018. URL https://www2.math.uconn.edu/~leykekhman/courses/MATH3795/Lectures/Lecture_10_Linear_least_squares_reg.pdf.

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.

Xiang Huang, Kuan He, Seunghwan Yoo, Oliver Cossairt, Aggelos Katsaggelos, Nicola Ferrier, and Mark Hereld. An interior point method for nonnegative sparse signal reconstruction. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1193–1197, 2018. doi: 10.1109/ICIP.2018.8451710.

Danny Bickson and Danny Dolev. Distributed sensor selection using a truncated newton method, 2010.

Ljubisa Stankovic, Ervin Sejdic, Srdjan Stankovic, Milos Dakovic, and Irena Orovic. A tutorial on sparse signal reconstruction and its applications in signal processing. *Circuits, Systems, and Signal Processing*, 38, 03 2019. doi: 10.1007/s00034-018-0909-2.

A Appendix

All Python files are available on Google Drive.

```python
import scipy.io as sio
import scipy.optimize
import numpy as np
import time

data_folder = "./data-q1/"

for iter in range(1, 6):
    A_mat = "A"+str(iter)
    b_arr = "b"+str(iter)
    A1 = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
    b1 = sio.loadmat(data_folder+b_arr+".mat")[b_arr].squeeze()

    M, N = A1.shape
    #print(A1.shape)

    I = np.eye(M)
    One = np.ones((M,M))

    c = np.concatenate((np.zeros(N), np.ones(M)))
    A_ub_1 = np.vstack(( np.hstack((A1, -I)), np.hstack((-A1, -I)) )) # L_1
    A_ub_inf = np.vstack(( np.hstack((A1, -One)), np.hstack((-A1, -One)) )) # L_inf
    b_ub = np.concatenate((b1, -b1))

    ###

    # L_1 minimisation
    start_1 = time.perf_counter()
    res_1 = scipy.optimize.linprog(c, A_ub=A_ub_1, b_ub=b_ub)
    end_1 = time.perf_counter() - start_1

    # L_inf minimisation
    start_inf = time.perf_counter()
    res_inf = scipy.optimize.linprog(c, A_ub=A_ub_inf, b_ub=b_ub)
    end_inf = time.perf_counter() - start_inf

    # L_2 minimisation
    start_2 = time.perf_counter()
    x, sse, rank, s = np.linalg.lstsq(A1, b1)
    end_2 = time.perf_counter() - start_2

    ###

    print(f"A{iter} b{iter}")
    # Min values
    print(f"Min Values")
    print(round(res_1.fun, 2), round(np.sqrt(sse[0]), 2), round(res_inf.fun, 2))

    # Runtime
    print(f"Runtime")
    print(round(end_1*1000, 1), round(end_2*1000, 1), round(end_inf*1000, 1))

    print("####################")
```

Listing 1. Norm Table - Generate Table 1

```python
import scipy.io as sio
import scipy.optimize
import scipy.stats
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q1/"
A_mat = "A5"
b_arr = "b5"
A1 = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
b1 = sio.loadmat(data_folder+b_arr+".mat")[b_arr].squeeze()
M, N = A1.shape
print(A1.shape)

I = np.eye(M)
One = np.ones((M,M))

c = np.concatenate((np.zeros(N), np.ones(M)))
A_ub_1 = np.vstack(( np.hstack((A1, -I)), np.hstack((-A1, -I)) )) # L_1
A_ub_inf = np.vstack(( np.hstack((A1, -One)), np.hstack((-A1, -One)) )) # L_inf
b_ub = np.concatenate((b1, -b1))

print(A_ub_1.shape) # 2m x (n+m)
print(b_ub.shape) # 2m
print(c.shape) # n+m

res_1 = scipy.optimize.linprog(c, A_ub=A_ub_1, b_ub=b_ub) # L_1 minimisation
res_inf = scipy.optimize.linprog(c, A_ub=A_ub_inf, b_ub=b_ub) # L_inf minimisation
x, sse, rank, s = np.linalg.lstsq(A1, b1) # L_2 minimisation

resid_1 = np.dot(A1, res_1.x[:N]) - b1
resid_inf = np.dot(A1, res_inf.x[:N]) - b1
resid_2 = np.dot(A1, x) - b1


xmin, xmax, xcount = -2, 2, 1000
lnspc = np.linspace(xmin, xmax, xcount)


##################

# Laplace (1-Norm)
ag,bg = scipy.stats.laplace.fit(resid_1)
pdf_laplace = scipy.stats.laplace.pdf(lnspc, ag, bg)

# Gaussian (2-Norm)
m, s = scipy.stats.norm.fit(resid_2)
pdf_g = scipy.stats.norm.pdf(lnspc, m, s)

##################

num_bins = 40
area_scale_1 = (max(resid_1)-min(resid_1))/num_bins * len(resid_1)
area_scale_2 = (max(resid_2)-min(resid_2))/num_bins * len(resid_2)
#area_scale_inf = (max(resid_inf)-min(resid_inf))/num_bins * len(resid_inf)

fig, axs = plt.subplots(3, 1, sharex=True)

axs[0].hist(resid_1, bins=num_bins, density=False, label=r"$L_1$", edgecolor='b', color='w')
axs[0].plot(lnspc, area_scale_1*pdf_laplace, label="Laplace", color='b')

axs[1].hist(resid_2, bins=num_bins, density=False, label=r"$L_2$", edgecolor='b', color='w')
axs[1].plot(lnspc, area_scale_2*pdf_g, label="Gaussian", color='b')

axs[2].hist(resid_inf, bins=num_bins, density=False, label=r"$L_{\infty}$", edgecolor='b', color='w')

axs[0].set_ylabel("Frequency")
axs[1].set_ylabel("Frequency")
axs[2].set_ylabel("Frequency")
axs[2].set_xlabel("Residual")

axs[0].legend()
axs[1].legend()
axs[2].legend()
plt.show()
```

Listing 2. Norm Residuals - Generate Figure 1

```python
import scipy.io as sio
import scipy.optimize
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q1/"
A_mat, b_arr = "A3", "b3"
A1 = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
b1 = sio.loadmat(data_folder+b_arr+".mat")[b_arr].squeeze()
M, N = A1.shape

I = np.eye(M)
One = np.ones((M,M))

c = np.concatenate((np.zeros(N), np.ones(M)))
A = np.vstack(( np.hstack((A1, -I)), np.hstack((-A1, -I)) ))
b = np.concatenate((b1, -b1))

def f(x):
    global A, b, c
    return np.dot(c, x) - np.sum(np.log(np.abs(b - A@x)))

def grad(x):
    global A, b, c
    d = 1/(b - A@x)
    #print("Condition Number", np.linalg.cond(A.T @ (np.diag(d)**2 @ A) ))
    return c + A.T @ d

# Parameters.
alpha = 0.3
beta = 0.8
t = 1

np.random.seed(0)
#x = np.zeros(N+M)
x = np.ones(N+M)*0.1
iter = 1

func_vals = []
while iter < 5e4:
    func_vals.append(f(x))

    g_vec = grad(x)
    # set descent direction to negative normalised gradient
    g = -g_vec/np.linalg.norm(g_vec)

    if iter % 1000 == 0:
        print(f'[{iter}] \t Grad: {np.linalg.norm(g_vec)} \t Step-size: {t}')

    # Backtracking linesearch implementation
    while f(x + t*g) > f(x) + (alpha * t * np.dot(g_vec, g)):
        t *= beta

    # Exact linesearch
    #res = scipy.optimize.line_search(f, grad, x, g)
    #t = res[0]

    x += t * g
    iter += 1

log_err = np.abs(np.array(func_vals) - func_vals[-1])
fig, axs = plt.subplots()
axs.plot(log_err)
#axs.plot(log_err[:40000])
#axs.set_ylim(1e2, 1e4)

axs.set_yscale('log')
axs.set_ylabel(r"$ |f(x_k^*) - p^*| $", fontsize="13")
axs.set_xlabel("Iterations", fontsize="13")
plt.show()
```

Listing 3. Backtracking Linesearch for l_1 Norm Approximation - Generate Fig 2

```python
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q3/"
A_mat, x_vec = "A", "x0"
A = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
xopt = sio.loadmat(data_folder+x_vec+".mat")[x_vec].squeeze()
M, N = A.shape
b = np.dot(A, xopt)
lam_max = np.linalg.norm(2*A.T@b, np.inf)
lam = 0.01 * lam_max

def objective_l2(x):
    global A, b, lam
    return np.linalg.norm(A@x - b)**2 + lam*np.linalg.norm(x, 2)**2

def objective(x):
    global A, b, lam
    return np.linalg.norm(A@x-b)**2+lam*np.linalg.norm(x,1)

def f(x, u, t):
    global A, b, lam
    return t*(np.linalg.norm(A@x - b)**2 + lam*np.sum(u)) - np.sum(np.log(np.abs(u**2-x**2)))

def grad_x(x,u,t):
    global A, b, lam
    return 2* (t* A.T @ (A@x-b) + x / (u**2 - x**2))

def grad_u(x,u,t):
    global A, b, lam
    return t*lam*np.ones(len(u)) - 2 * u/(u**2 - x**2)

def hessian(x,u,t):
    global A, b, lam
    den = (u**2-x**2)
    D1 = np.diag( 2* ((u**2+x**2) / den) / den  )
    D2 = np.diag( ((-4*x*u)  / den) / den )
    H = np.vstack(( np.hstack((2*t*A.T@A + D1, D2)), np.hstack((D2, D1)) ))
    return H

t = 1/lam
mu, epsilon = 1.1, 1e-6
alpha, beta, tau = 0.25, 0.4, 1
x, u = np.zeros(N), np.ones(N)
g_vec_x, g_vec_u = grad_x(x, u, t), grad_u(x, u, t)

iter = 1
while iter <= 200:
    g_vec_x, g_vec_u = grad_x(x, u, t), grad_u(x, u, t)
    g_vec = np.hstack(( g_vec_x, g_vec_u ))
    h = hessian(x, u, t)

    # Compute Newton search direction
    g = np.linalg.solve(h, -g_vec)
    g /= np.linalg.norm(g)
    g_x, g_u = g[:N], g[N:]

    # Backtracking linesearch implementation
    while f(x + tau*g_x, u+tau*g_u, t) > f(x, u, t) + (alpha * tau * (np.dot(g_vec_x, g_x) + np.dot(g_vec_u, g_u))):
        tau *= beta

    x += tau * g_x
    u += tau * g_u
    print(f"Iter {iter} \t Step-size {tau} \t t {t}")
    print("N/t", N/t)

    if N/t < epsilon:
        break

    t *= mu
    iter += 1

xnew = x

# Both solutions are equivalent
x_l2 = A.T@(np.linalg.inv(A@A.T)@b)
#x_l2 = np.linalg.inv(A.T@A + lam*np.eye(N))@(A.T@b)

fig, axs = plt.subplots(figsize=(7,5))
axs.vlines(range(N), ymin=np.zeros(N), ymax=xnew, colors=["r"]*N, label="Predicted "+r"$l_1$"+"-reg", lw=1.5)
axs.vlines(range(N), ymin=np.zeros(N), ymax=xopt, colors=["b"]*N, label="Actual Signal", linestyles='-.', lw=0.75)

nz_inds = np.argwhere(xopt != 0)
axs.scatter(nz_inds, xopt[nz_inds], c="b", marker=".")

# L2 Prediction
axs.vlines(range(N), ymin=np.zeros(N), ymax=x_l2, colors=["g"]*N, label="Predicted "+r"$l_2$"+"-reg", linestyles='--', lw=1)

axs.legend(loc="lower left")
axs.set_xlabel("Element Index", fontsize=13)
axs.set_ylabel("Signal Value", fontsize=13)
plt.show()
```

Listing 4. BM - Generate Fig 3a

```python
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q3/"
A_mat, x_vec = "A", "x0"
A = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
xopt = sio.loadmat(data_folder+x_vec+".mat")[x_vec].squeeze()
M, N = A.shape
b = np.dot(A, xopt)
lam_max = np.linalg.norm(2*A.T@b, np.inf)
lam = 0.01 * lam_max

def objective_l2(x):
    global A, b, lam
    return np.linalg.norm(A@x - b)**2 + lam*np.linalg.norm(x, 2)**2

def objective(x):
    global A, b, lam
    return np.linalg.norm(A@x-b)**2+lam*np.linalg.norm(x,1)

def dual_objective_G(v):
    global A, b, lam
    return -(1/4) * v@v - v@b

def f(x, u, t):
    global A, b, lam
    return t*(np.linalg.norm(A@x - b)**2 + lam*np.sum(u)) - np.sum(np.log(np.abs(u**2-x**2)))

def grad_x(x,u,t):
    global A, b, lam
    return 2* (t* A.T @ (A@x-b) + x / (u**2 - x**2))

def grad_u(x,u,t):
    global A, b, lam
    return t*lam*np.ones(len(u)) - 2 * u/(u**2 - x**2)

def hessian(x,u,t):
    global A, b, lam
    den = (u**2-x**2)
    D1 = np.diag( 2* ((u**2+x**2) / den) / den  )
    D2 = np.diag( ((-4*x*u)  / den) / den )
    H = np.vstack(( np.hstack((2*t*A.T@A + D1, D2)), np.hstack((D2, D1)) ))
    return H

t, mu, epsilon = 1/lam, 1.1, 0.2
alpha, beta, tau, tau_min = 0.01, 0.4, 1, 0.5
x, u = np.zeros(N), np.ones(N)
g_vec_x, g_vec_u = grad_x(x, u, t), grad_u(x, u, t)

iter = 1
while iter <= 1000:
    g_vec_x, g_vec_u = grad_x(x, u, t), grad_u(x, u, t)
    g_vec = np.hstack(( g_vec_x, g_vec_u ))
    h = hessian(x, u, t)
    # Compute Newton search direction
    g = np.linalg.solve(h, -g_vec)
    g /= np.linalg.norm(g)
    g_x, g_u = g[:N], g[N:]

    # Backtracking linesearch implementation
    while f(x+tau*g_x,u+tau*g_u,t) > f(x,u,t) + (alpha*tau*(np.dot(g_vec_x, g_x)+np.dot(g_vec_u, g_u))):
        tau *= beta

    x += tau * g_x
    u += tau * g_u
    v = 2*tau*(A@x - b) # dual feasible point v
    tau = min(lam/np.abs(2*A.T@(A@x - b)))
    G_v = dual_objective_G(v)
    eta = np.abs(objective(x) - G_v) # duality gap eta

    print(f"Iter {iter} \t Step-size {tau} \t t {t}")
    print("Term Cond", eta/G_v)
    if eta/G_v < epsilon:
        break
    if tau >= tau_min:
        t = max( mu * 2*min(N/eta, t), t )
    iter += 1

xnew = x
# Both solutions are equivalent
x_l2 = A.T@(np.linalg.inv(A@A.T)@b)
#x_l2 = np.linalg.inv(A.T@A + lam*np.eye(N))@(A.T@b)

fig, axs = plt.subplots(figsize=(7,5))
zs = np.zeros(N)
axs.vlines(range(N), ymin=zs, ymax=xnew, colors=["r"]*N, label="Predicted "+r"$l_1$"+"-reg", lw=1.5)
axs.vlines(range(N), ymin=zs, ymax=xopt, colors=["b"]*N, label="Actual Signal", linestyles='-.', lw=0.75)

nz_inds = np.argwhere(xopt != 0)
axs.scatter(nz_inds, xopt[nz_inds], c="b", marker=".")

axs.vlines(range(N),ymin=zs,ymax=x_l2,colors=["g"]*N, label="Predicted "+r"$l_2$"+"-reg",linestyles='--', lw=1)

axs.legend(loc="lower left")
axs.set_xlabel("Element Index", fontsize=13)
axs.set_ylabel("Signal Value", fontsize=13)
plt.show()
```

Listing 5. CIPM - Generate Fig 3b

```python
import scipy.io as sio
import scipy.optimize
import scipy.stats
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q3/"
A_mat, x_vec = "A", "x0"
A = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
xopt = sio.loadmat(data_folder+x_vec+".mat")[x_vec].squeeze()
M, N = A.shape
b = np.dot(A, xopt)
lam_max = np.linalg.norm(2*A.T@b, np.inf)

# primal objective value of x
def objective(x):
    global A, b, lam
    return np.linalg.norm(A@x - b)**2 + lam*np.linalg.norm(x, 1)

def soft(y, lamb):
    c = np.maximum(np.zeros(N), np.abs(y) - lamb)
    return np.sign(y) * c

np.random.seed(0)
# Note alpha is required to be greater than lam_max
alpha = 2*np.max(np.linalg.eigvals(A.T@A)).real # this is commonly used
#print("alpha", alpha, "\t lam max", lam_max)

x = (np.random.rand(N)*2-1)*0.01

xs = []
xopts = []
factors = np.array(range(-20, 4))
lams = (10**(factors/4) * lam_max)

for lam in lams:
    iter = 1
    while iter <= 200:
        y = 1/alpha * A.T@(b-A@x) + x
        x = soft(y, lam/(2*alpha))
        #print(lam)
        iter += 1
    xs.append(x)
    xopts.append(xopt)

fig, axs = plt.subplots()
axs.plot(lams, np.linalg.norm(np.array(xs)-np.array(xopts), axis=1))
#print(lams[np.argmin(np.linalg.norm(np.array(xs)-np.array(xopts), axis=1))])

axs.set_xscale('log')
axs.set_xlabel(r"$\lambda$", fontsize=13)
axs.set_ylabel("Reconstruction Error "+r"$||x-x*||_2$", fontsize=13)
plt.show()
```

Listing 6. ISTA λ-Variation - Generate Fig 4

```python
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt

data_folder = "./data-q3/"
A_mat, x_vec = "A", "x0"
A = sio.loadmat(data_folder+A_mat+".mat")[A_mat]
xopt = sio.loadmat(data_folder+x_vec+".mat")[x_vec].squeeze()
M, N = A.shape
b = np.dot(A, xopt)
lam_max = np.linalg.norm(2*A.T@b, np.inf)
lam = 0.03 * lam_max

# primal objective value of x
def objective(x):
    global A, b, lam
    return np.linalg.norm(A@x - b)**2 + lam*np.linalg.norm(x, 1)

def soft(y, lamb):
    c = np.maximum(np.zeros(N), np.abs(y) - lamb)
    return np.sign(y) * c

np.random.seed(0)
# Note alpha is required to be greater than lam_max
alpha =  2*np.max(np.linalg.eigvals(A.T@A)).real # this is commonly used
print("lam", lam, "\t alpha", alpha, "\t lam max", lam_max)

x = np.zeros(N)
#x = (np.random.rand(N)*2-1)*0.01

func_vals = []
iter = 1
while iter <= 400:
    y = 1/alpha * A.T@(b-A@x) + x
    x = soft(y, lam/(2*alpha))
    func_vals.append(objective(x))
    #print(f"Iter {iter}")
    iter += 1

xnew = x
print(objective(xnew), objective(xopt))

fig, axs = plt.subplots(2,1, figsize=(7,8))
# Plot components as vertical signal lines
axs[0].vlines(range(N), ymin=np.zeros(N), ymax=xnew, colors=["r"]*N, label="Predicted Signal", lw=1.5)
axs[0].vlines(range(N), ymin=np.zeros(N), ymax=xopt, colors=["b"]*N, label="Actual Signal", linestyles='-.', lw=0.75)

# For aesthetics and visibility of actual signal
non_zero_indices = np.argwhere(xopt != 0)
axs[0].scatter(non_zero_indices, xopt[non_zero_indices], c="b", marker=".")

axs[1].plot(np.array(func_vals)-objective(xopt))

axs[0].legend(loc="lower left")
axs[0].set_xlabel("Element Index", fontsize=13)
axs[0].set_ylabel("Signal Value", fontsize=13)

axs[1].set_xlabel("Iterations", fontsize=13)
axs[1].set_ylabel("Error "+r"$f(x) - f(x*)$", fontsize=13)

plt.show()
```

Listing 7. ISTA Method - Generate Fig 5