Project 2 - Optimization

Yoav Ellinson, Bar Atuar

January 2025

0.1 Project questions

Q1. The continuous integral along a ray path $\gamma(l)$ in X-ray tomography is given by:

$$y = \int_{\gamma(l)} \rho(\gamma(l))\, dl,$$

where:

- y is the measured attenuation along the ray,
- $\rho(\gamma(l))$ is the density at a point along the ray path,
- dl is the infinitesimal path length.

We can approximate this integral with the simple midpoint rule given by:

$$y_i \approx \sum_{j=1}^{n} \rho(x_j, y_j)\Delta l,$$

where:

- i indicates the ray number.
- (x_j, y_j) is the midpoint of the j-th segment,
- Δl is the length of the segment =

  $$\Delta x, \Delta y, \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

  for horizontal,vertical and diagonal paths accordingly.
- $\rho(x_j, y_j)$ is the density evaluated at the midpoint.

As mentioned, the paths lengths are given by the matrix $\mathbf{A}$, therfore:

$$y_i \approx \sum_{j=1}^{n} \rho(x_j, y_j)\mathbf{A}_{i,j} = \sum_{j=1}^{n} \mathbf{x}_j \mathbf{A}_{i,j}$$

Q2. If we stack the rays mesurmenst together we get:

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

where:

- m is the number of measurements.
- $\mathbf{X} \in \mathbb{R}^{M \times N}$, in our case $M = N = 5$
- $\mathbf{y} \in \mathbb{R}^{m}$
- $\mathbf{A} \in \mathbb{R}^{m \times (M \cdot N)}$

The relation between $\mathbf{y}$ and $\mathbf{x}$ is **linear**, thats because $\mathbf{A}$ is known and it only multiplies $\mathbf{x}$ to evaluate $\mathbf{y}$. Building $\mathbf{A}$ for the toy problem can be done buy constructing an 8X25 matrix corresponding to the 8 measurements and the 5X5 pixels we have. The rays will be numbered as in Figure 1.

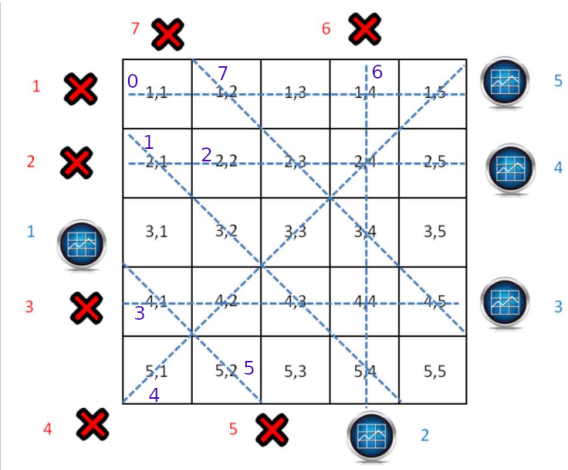


Figure 1: Ray numbering (in purple) scheme

In figure 2 you can see each ray's path (a row of $\mathbf{A}$ before column stack). To construct $\mathbf{A}$ all we need is to flatten each matrix and to put it in a row of $\mathbf{A}$.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) Ray 0

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} & 0 \end{bmatrix}$$

(b) Ray 1

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c) Ray 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(d) Ray 3

$$\begin{bmatrix} 0 & 0 & 0 & 0 & \sqrt{2} \\ 0 & 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 \\ \sqrt{2} & 0 & 0 & 0 & 0 \end{bmatrix}$$

(e) Ray 4

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 \end{bmatrix}$$

(f) Ray 5

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(g) Ray 6

$$\begin{bmatrix} 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(h) Ray 7

Figure 2: Rays path matrices

Q3. Given that we need to multiply the column-stacked representation of $\mathbf{X}$ from the left by D_x and D_y, the dimensions of these matrices are $\mathbb{R}^{MN \times MN}$. By simple matrix multiplication rules we can identify that each row of these matrices has two non-zero items or it has all zero items (where i= M,j=N).

- For the vertical derivative, we want to subtract each row from its greater. By using the column-stacked vector of the matrix we can see that by multiplying by the following matrix we will get the correct result.

$$D_y = \begin{bmatrix} -1 & ..N-1\ zeros.. & 1 & 0\cdots & 0 & 0 & 0 \\ 0 & -1 & ..N-1\ zeros.. & 1 & 0\cdots & 0 & 0 \\ 0 & 0 & -1 & ..N-1\ zeros.. & 1 & 0\cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & 0 & 0 & -1 & ..N-1\ zeros.. & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

  In simple words, each value in the i-th row is being multiplyed by -1 and then being summed with its corresponding item in the (i+1)-th row.

- For the horizontal derivative, we relay on the same principal, but we multiply each item in the i-th column by -1 and then sum it with its corresponding item in the (i+1)-th column, as

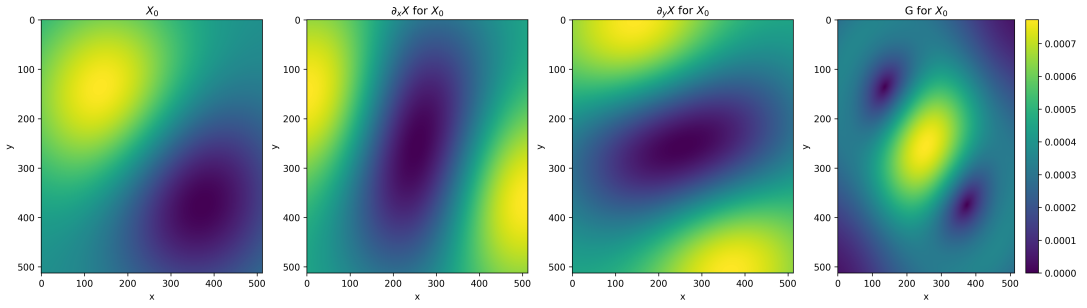
shown in the following matrix:

$$\begin{bmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ N-1\ times & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & -1 \end{bmatrix}$$

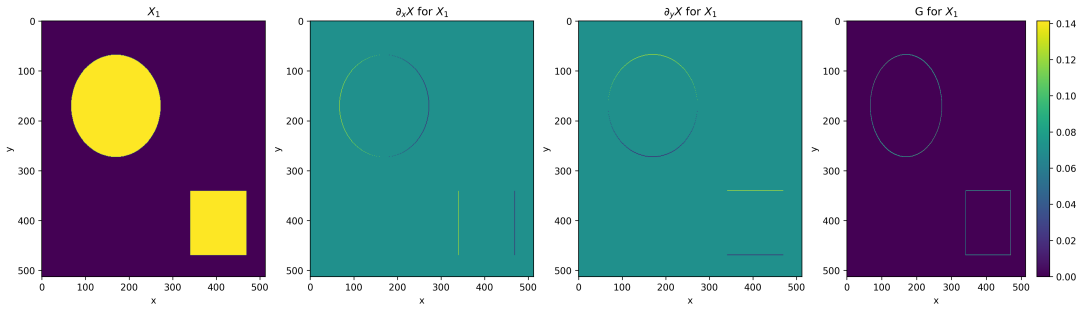
Every N-1 rows there is a zeros row because the of column-stacked representation, where every N items are one column and we want the last column of the result matrix to be zeros.

Code in Appendix A.a. This naive version of the code works for small dimensions of $\mathbf{X}$. But for larger sizes we need to sparcify the matrices (for a 513X513 image the D_x matrix size is 263169X263169 $\approx$ 516GB of memory). With some help online we got the code can be found in appendix A.b.

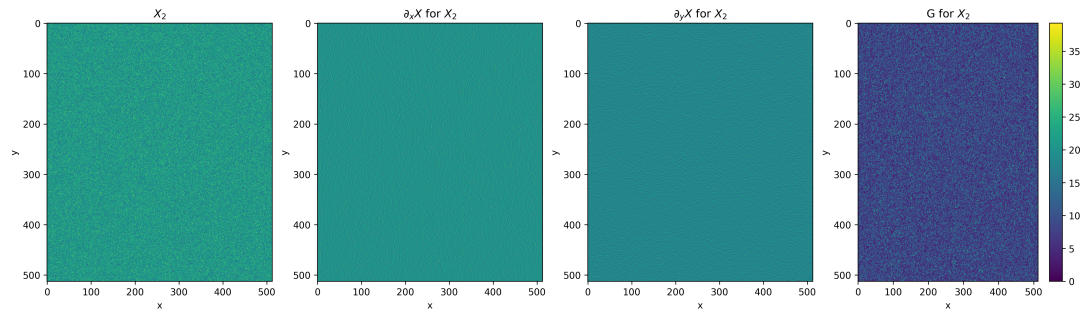
Q4. We ploted the results for the three included images:



(a) Results for X_0



(b) Results for X_1



(c) Results for X_2

Figure 3: The results of the three included files, in each plot we can see the original image (left),horizontal derivative, vertical derivative and the gradients magnitude

We can clearly see that the results are fair. especially in the second example where the combination of horizontal and vertical derivatives gives us the correct gradient - which is only present on the edges of the shapes.

Q5. Regarding Figure 2 (in the provided PDF, do not confuse with figure 1 in our document), There are **5X5=25** unknown parameters - the object density. The number of observations can be counted as the number of rays in the figure and its **8**. Given the objective:

$$\min_x \frac{1}{2}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2$$

We would like to minimize the function:

$$f(x)=\frac{1}{2}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2=(\mathbf{y}-\mathbf{A}\mathbf{x})^T(\mathbf{y}-\mathbf{A}\mathbf{x})=\frac{1}{2}\big(\mathbf{y}^T\mathbf{y}-2\mathbf{y}^T\mathbf{A}\mathbf{x}+\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\big)$$

Lets calc' the gradient and the Hessian:

$$\nabla f(x)=\mathbf{A}^T(\mathbf{Ax\text{-}y})$$

$$\nabla^2 f(x)=\mathbf{A}^T\mathbf{A}$$

We can easily show that $\nabla^2 f(x)$ is **PSD**:

$$\text{For any vector } \mathbf{z},\quad \mathbf{z}^T(\mathbf{A}^T\mathbf{A})\mathbf{z}=(\mathbf{A}\mathbf{z})^T(\mathbf{A}\mathbf{z})=\|\mathbf{A}\mathbf{z}\|_2^2\geq 0\Rightarrow\nabla^2 f(x)\succeq 0$$

Therefore the function is convex. To check for the number of possible solutions we need to check if the Hessian is **PD**, if so the function is strictly convex and then there is only one possible solution. In our case the Hessian is $\mathbf{A}^T\mathbf{A}$ where $\mathbf{A}\in\mathbb{R}^{8X25}$ so it doesn't have a full rank thus cannot be **PD**. So we can say that this problem has many possible solutions that satisfy the objective.

Q6. The problem we would like to solve is:

$$\min_x \frac{1}{2}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2+\frac{\lambda}{2}\|\mathbf{L}\mathbf{x}\|_2^2$$

Therfore the function we should minimize is:

$$f(x)=\frac{1}{2}\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2^2+\frac{\lambda}{2}\|\mathbf{L}\mathbf{x}\|_2^2=\frac{1}{2}(\mathbf{y}-\mathbf{A}\mathbf{x})^T(\mathbf{y}-\mathbf{A}\mathbf{x})+\frac{\lambda}{2}(\mathbf{L}\mathbf{x})^T(\mathbf{L}\mathbf{x})=\frac{1}{2}\big(\mathbf{y}^T\mathbf{y}-2\mathbf{y}^T\mathbf{A}\mathbf{x}+\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\big)+\frac{\lambda}{2}\mathbf{x}^T\mathbf{L}^T\mathbf{L}\mathbf{x}$$

The first-order optimality condition says that for the minimizer $\mathbf{x}^*$ the gradient of f(x) is zero.

$$\nabla f(\mathbf{x})=-\mathbf{A}^T\mathbf{y}+\mathbf{A}^T\mathbf{A}\mathbf{x}+\lambda L^T\mathbf{L}\mathbf{x}$$

Now we use the first order optimality condition:

$$\nabla f(\mathbf{x}^*)=-\mathbf{A}^T\mathbf{y}+\mathbf{A}^T\mathbf{A}\mathbf{x}^*+\lambda\mathbf{L}^T\mathbf{L}\mathbf{x}^*=0\Rightarrow\underline{(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{L}^T\mathbf{L})\mathbf{x}^*=\mathbf{A}^T\mathbf{y}}$$

<u>These</u> are the set called the *normal equations*

$$\mathbf{x}^*=(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{L}^T\mathbf{L})^{-1}\mathbf{A}^T\mathbf{y}.$$

This is the closed solution for the minimizer.

Q7. In the proposed quadric equation in x, the hessian will be **Q**, Therefore by forcing **Q** to be P.D we will get a strictly convex problem - so we will have one solution.

$$f(\mathbf{x})=\frac{1}{2}\big(\mathbf{y}^T\mathbf{y}-2\mathbf{y}^T\mathbf{A}\mathbf{x}+\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x}\big)+\frac{\lambda}{2}\mathbf{x}^T\mathbf{L}^T\mathbf{L}\mathbf{x}=\frac{1}{2}\mathbf{x}^T(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{L}^T\mathbf{L})\mathbf{x}-\mathbf{y}^T\mathbf{A}\mathbf{x}+\frac{1}{2}\mathbf{y}^T\mathbf{y}=$$

$$=\frac{1}{2}\mathbf{x}^T(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{L}^T\mathbf{L})\mathbf{x}-(\mathbf{A}^T\mathbf{y})^T\mathbf{x}+\frac{1}{2}\mathbf{y}^T\mathbf{y}$$

In this form we can identify that:

- $\mathbf{Q} = \mathbf{A}^T\mathbf{A} + \lambda\mathbf{L}^T\mathbf{L}$
- $\mathbf{b} = \mathbf{A}^T\mathbf{y}$
- $c = \frac{1}{2}\mathbf{y}^T\mathbf{y}$

Now we need to show that $\mathbf{A}$ is PD. We can easly say that $\mathbf{Q}$ is symmetric (sum of two symmetric matrices). In addition, for every vector $\mathbf{x}$ and any $\lambda > 0$:

$$\mathbf{x}^T\mathbf{Q}\mathbf{x} = ||\mathbf{A}\mathbf{x}||^2 + \lambda||\mathbf{L}\mathbf{x}||^2$$

Therfore only if $\mathbf{x} \in Null(A, L)$ we will get a value that is 0, otherwise $\mathbf{Q} \succ 0$.

Q8. First we will write the gradient for the function:

$$f(x) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$$

$$\nabla f(x) = \mathbf{Q}\mathbf{x} + \mathbf{b} = (\mathbf{A}^T\mathbf{A} + \lambda\mathbf{L}^T\mathbf{L})\mathbf{x} - \mathbf{A}^T\mathbf{y}$$

The GD algorithm , with step size α:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\nabla f(\mathbf{x}_k) = \mathbf{x}_k - \alpha\big((\mathbf{A}^T\mathbf{A} + \lambda\mathbf{L}^T\mathbf{L})\mathbf{x}_k - \mathbf{A}^T\mathbf{y}\big)$$

Q9. To find the range of the step size we will need to compute the condition number of $\mathbf{Q}$. To do so we need to build $\mathbf{Q}$ from $\mathbf{A}$ and $\mathbf{L}$ as mentioned - we used numpy, Code can be found in appendix B. The results:

- $\kappa_Q = 1245092.865$
- $\lambda_{MAX} = 12.57$
- $\lambda_{MIN} = 1.009 \times 10^{-5}$

The range of the step size in this case ('12.57'-smooth) is: $\alpha <= \frac{1}{12.57} = 0.07955$.
To calculate in how many steps the objective will reduce by a factor of 10 using a constant step size of $\alpha = \frac{1}{12.57}$ we use this inequality from the lecture:

$$\|\mathbf{x}_k - \mathbf{x}^*\|^2 \leq \left(1 - \frac{\mu}{L}\right)^k \|\mathbf{x}_0 - \mathbf{x}^*\|^2.$$

It means that the reducing factor of the objective from $\mathbf{x}_0$ to $\mathbf{x}_k$ is equivalent to $(1 - \frac{1}{\kappa_Q})^k$ where k is the number of iterations.

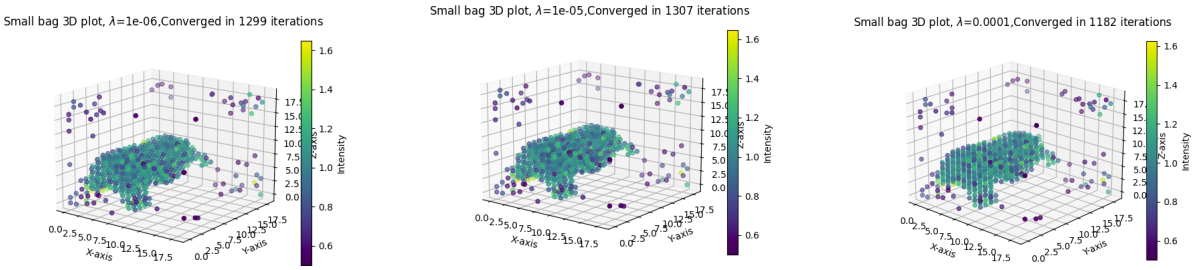
$$0.1 = (1 - \frac{1}{\kappa_Q})^k \Rightarrow k = \frac{1}{\log_{10}(1 - \frac{1}{\kappa_Q})} = 2.86 \times 10^6 \text{ iterations}$$

Q10. We implemented the CGLS algorithm in python, can be found in appendix C.a:

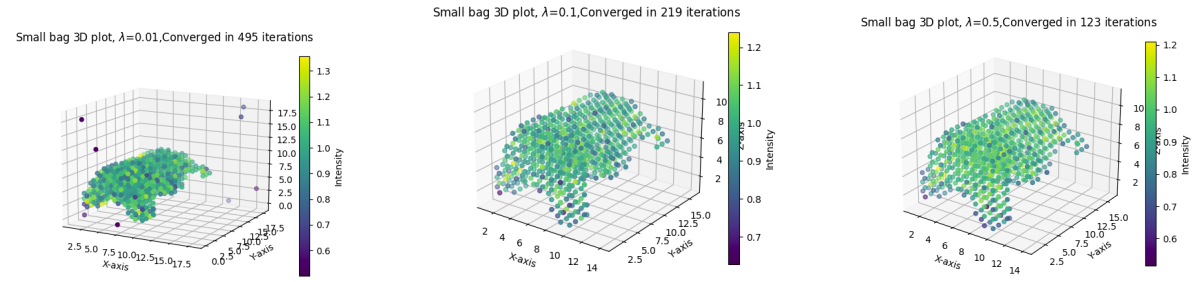
Then we tested it on the toy problem in in converged in 22 iterations with a tolerance of $1 \times 10^{-}7$. The conjugate gradients algorithem convergance rate is n iterations for $x \in \mathbb{R}^n$ - in the toy problem $x \in \mathbb{R}^{25}$ which confirms our estimation results.
We created the 3D derivatives with this function, Code in appendix 3.b.

Q11. We computed x for the small bag for several λs and plotted the reconstructed X in Figure 4, the plots includes only values above 0.5 resulting in a background-less picture of the object.

We can clearly see that the results are similar, and that the convergence rate is greater when the regularization is stricter (larger λ). But for a very large λ we start losing data, This is caused by the objective not being met. We plotted the objective value as a function of lambda in figure 5.



(a) λ=1e-6 (b) λ=1e-5 (c) λ=1e-4



(d) λ=1e-2 (e) λ=1e-1 (f) λ=0.5

(g) λ=2 (h) λ=20

Figure 4: The reconstruction of $\mathbf{X}$ for different values of λ

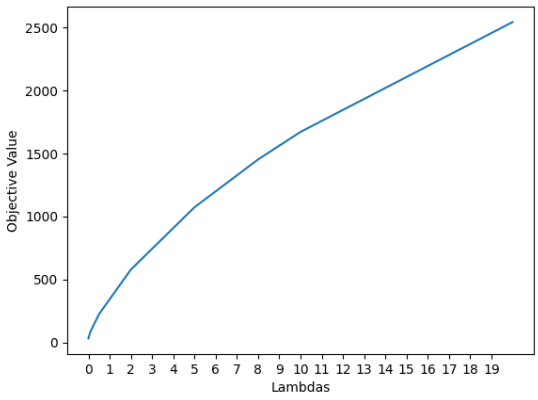


Figure 5: The objective value as a function of λ

Q12. We computed the norms using python:

- $||\mathbf{D}_x f_1(\mathbf{x})||_1 = 1$
- $||\mathbf{D}_x f_1(\mathbf{x})||_2 = 0.7071$

- $||\mathbf{D}_x f_2(\mathbf{x})||_1 = 1$
- $||\mathbf{D}_x f_2(\mathbf{x})||_2 = 0.55766$

In terms of l_1 norm neither interpolation method is preferable because they both has the same l_1 norm. But in terms of l_2 norm, $f_1(x)$ varies less smoothly then $f_2(x)$ therefore it has a higher l_2 norm - the preferable in terms of l_2 norm is $f_2(x)$.

Q13. We are minimizing the objective:

$$J(x) = \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \alpha\|\mathbf{Lx}\|_1$$

Convexity
The problem consists of two terms:

(a) $\|\mathbf{Ax} - \mathbf{b}\|_2^2$: A quadratic function in $\mathbf{x}$, which is convex and smooth.

(b) $\|\mathbf{Lx}\|_1$: The L_1-norm of $\mathbf{Lx}$, which is convex but **non-smooth** - l_1 is non smooth at 0.

The sum of two convex functions (even if one is non-smooth) is **convex**. Therefore, $J(x)$ is a convex function.

Strict Convexity

- Strict convexity depends on whether $\mathbf{A}^T\mathbf{A}$ (the hessian matrix of $J(x)$) is positive definite. If $\mathbf{A}$ has full column rank, $\mathbf{A}^T\mathbf{A}$ is positive definite, making $\|\mathbf{Ax} - \mathbf{b}\|_2^2$ strictly convex.
- The l_1 norm $\|\mathbf{Lx}\|_1$ does not affect strict convexity because it is not a part of the hessian.

Conclusion: The problem is strictly convex if $\mathbf{A}^T\mathbf{A}$ is positive definite, meaning $\mathbf{A}$ has full column rank. As seen in previous questions, this matrix is PD - therefore the objective is strictly convex.

Smoothness
In the lectures we described a function as smooth if its first derivative exists.

- $\|\mathbf{Ax} - \mathbf{b}\|_2^2$ is smooth because it has continuous first and second derivatives.
- $\|\mathbf{Lx}\|_1$ is non-smooth because the derivative is not defined at $\mathbf{x} = 0$ due to the absolute value function.

Therefore the objective is **strictly convex but not smooth**.

Q14. 1. By definition:

$$||\mathbf{Lx}||_1 = \sum_{i=1}^{m} |\gamma_i|$$

And the derivitive of each γ_i:

$$\frac{\partial|\gamma_i|}{\partial x_k} = \frac{\gamma_i}{|\gamma_i|} \cdot \frac{\partial\gamma_i}{\partial x_k}, \quad \forall\gamma_i \neq 0.$$

Therfore (chain rule):

$$\frac{\partial||\mathbf{Lx}||_1}{\partial x_k} = \sum_{i=1}^{m} \frac{\gamma_i}{|\gamma_i|} \cdot \mathbf{L}_{i,k}$$

2. Each element in the gradient is:

$$\nabla(||\mathbf{Lx}||_1)_i = \mathbf{L}^T w_i, \text{where:} w_i = \frac{\gamma_i}{|\gamma_i|}$$

We can define a diagonal matrix $\mathbf{W}$ s.t:

$$\mathbf{W}_{i,i} = \begin{cases} \frac{1}{|\gamma_i|}, & \text{if } |\gamma_i| \geq \epsilon, \\ \frac{1}{\epsilon}, & \text{if } |\gamma_i| < \epsilon. \end{cases}$$

Then:

$$\nabla(||\mathbf{Lx}||_1) = \mathbf{L}^T\mathbf{WLx}$$

3. The frist order optimality condition for the objective $J(x) = \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \alpha\|\mathbf{Lx}\|_1$ Is:

$$\nabla \mathbf{J}(\mathbf{x}^*) = 2\mathbf{A}^T(\mathbf{Ax}^* - \mathbf{b}) + \alpha\mathbf{L}^T\mathbf{WLx}^* = 0$$

$$(\mathbf{A}^T\mathbf{A} + \frac{\alpha}{2}\mathbf{L}^T\mathbf{WL})\mathbf{x}^* = \mathbf{A}^T\mathbf{b}$$

4. Similarly to (4), we can write the objective as the following LS problem:

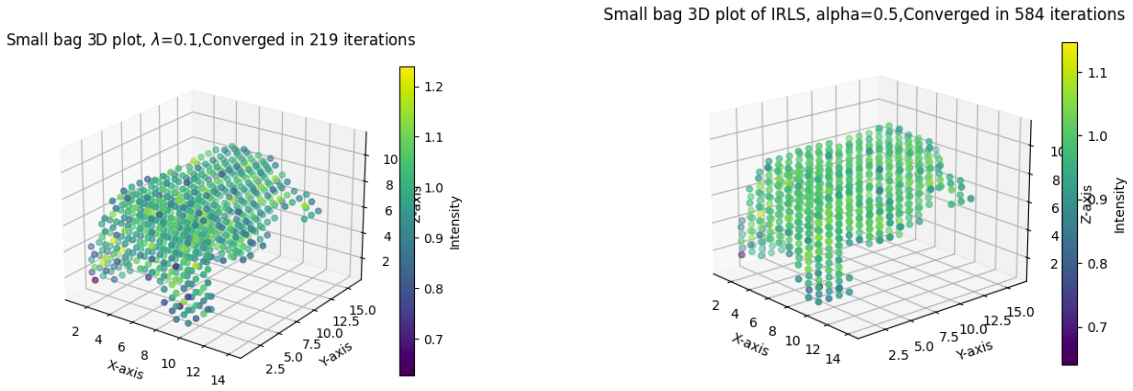
$$\min_{\mathbf{x}} \frac{1}{2} \left\| \begin{bmatrix} \mathbf{A} \\ \sqrt{\frac{\alpha}{2}}\mathbf{W}^{\frac{1}{2}}\mathbf{L} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} \right\|_2^2$$

Thats because:

$$\frac{\alpha}{2}\|\mathbf{W}^{1/2}\mathbf{Lx}\|_2^2 = \frac{\alpha}{2}\mathbf{x}^T\mathbf{L}^T\mathbf{WLx}$$

But when differentiating with respect to $\mathbf{x}$ we get the same gradient as the gradient of the l_1 norm.

Q15. We did as described and plotted the results in figure 6: We can clearly see that the result of the



(a) λ=0.1 (b) λ=0.1 and α=0.5

Figure 6: CGLS (left) VS IRLS of the Small bag

IRLS is smoother within the object! (The object seems to be a teddy bear).

Q16. We ran the IRLS algorithm on the Large bag and found a gun! (Figure 7).

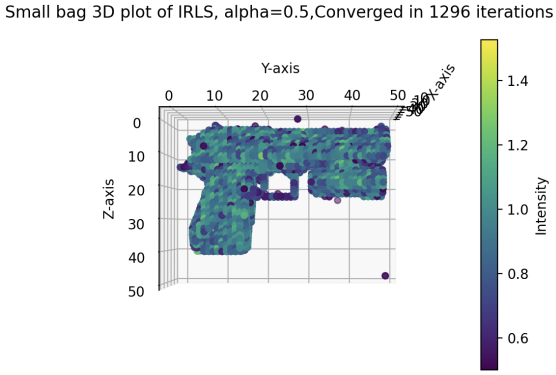


Figure 7: 3D reconstruction of the Large bag

.1 Appendix A

a) Non sparse implementation of the derivative matrix:

```
    def construct_D_matrices(M, N):
        MN = M*N
        #D_x
        D_x = (np.eye(MN)*-1)+ np.eye(MN,k=1)
        for i in range(N-1,MN,N):
            D_x[i] = np.zeros(MN)
        #Dy
        D_y = np.zeros((MN,MN))
        for i in range(MN-N):
            D_y[i,i] = -1
            D_y[i,i+N] = 1
        return D_x,D_y
```

b) Sparse implementation of the derivative matrix:

```
    def construct_D_matrices_sparse(M, N):
        MN = M * N
        # Construct sparse D_x
        diagonals_x = [-np.ones(MN), np.ones(MN - 1)]
        offsets_x = [0, 1]
        D_x = diags(diagonals_x, offsets_x, shape=(MN, MN), format='lil')

        # Remove connections across row boundaries
        for i in range(N - 1, MN, N):
            D_x[i,i] = 0
            if i+1==MN:
                break
            D_x[i,i+1] = 0
        D_x = D_x.tocsr()  # Convert to CSR format for efficient operations

        # Construct sparse D_y
        D_y = lil_matrix((MN, MN))
        for i in range(MN - N):
            D_y[i, i] = -1
            D_y[i, i + N] = 1
        D_y = D_y.tocsr()  # Convert to CSR format for efficient operations

        return D_x, D_y
```

.2 Appendix B

The condition number calculation using Python:

```
M,N = (5,5)
D_x,D_y = construct_D_matrices(M,N)
L = np.concatenate((D_x,D_y))
lambda_reg = 1e-5
Q = A.T@A +lambda_reg*(L.T@L)
u,v = np.linalg.eigh(Q)
kappa = max(u)/min(u)
```

.3 Appendix C

a) CGLS implementation:

```
    def cgls(A, L, y, max_iter=100, eps=1e-6, lambda_reg=1e-5):
        m, n = A.shape
        x = np.zeros(n)  # x_0 is zeros

        sqrt_lambda = np.sqrt(lambda_reg)
        A = sparse.vstack([A, sqrt_lambda * L])
        if len(y.shape) ==1:
            y = np.expand_dims(y,1)
        y = np.vstack([y, np.zeros((L.shape[0],1))])
        s_k= np.expand_dims(A@x,1) -y
        g_k = A.T @ s_k
        g_0 = g_k.copy()
        d = -g_k  # d_0 = -g_0
        residuals = [np.linalg.norm(g_k)]  # initial gradient norm

        for k in range(max_iter):
            #compute Ad_k to simplify for the future
            Ad = A @ d
            #compute step size alpha_k
            alpha = (g_k.T @ g_k) / (Ad.T @ Ad)
            #update the solution
            x += (alpha.squeeze() * d).reshape(x.shape)
            s_k += alpha * Ad  # s_{k+1} = s_k + alpha_k A d_k
            g_k_plus_1 = A.T @ s_k

            # check for convergence - gradient norm
            gradient_norm = np.linalg.norm(g_k_plus_1)
            residuals.append(gradient_norm)
            if gradient_norm < eps:
                break
            # compute beta_k for the new search direction and update
            beta = ((g_k_plus_1.T@ g_k_plus_1) / (g_k.T@ g_k)).squeeze()
            d = -g_k_plus_1 + beta * d
            #for next iter
            g_k = g_k_plus_1
        return x, residuals, k
```

b) 3D derivative function:

```
        def construct_D_matrices_3D_sparse(M, N, P):
            MN = M * N  # Number of cells in each 2D slice
            MNP = M * N * P  # Total number of cells in the 3D grid

            # Construct D_x (finite differences along x-axis)
            diagonals_x = [-np.ones(MNP), np.ones(MNP - 1)]
            offsets_x = [0, 1]
            D_x = diags(diagonals_x, offsets_x, shape=(MNP, MNP), format='lil')

            for z in range(P):
                for i in range(N - 1, MN + z * MN, N):
                    D_x[i, i] = 0
                    if i + 1 < MNP:
                        D_x[i, i + 1] = 0
            D_x = D_x.tocsr()

            D_y = lil_matrix((MNP, MNP))
            for z in range(P):
                for i in range((M - 1) * N + z * MN):
                    D_y[i, i] = -1
                    if i + N < MNP:
                        D_y[i, i + N] = 1
            D_y = D_y.tocsr()

            D_z = lil_matrix((MNP, MNP))
            for i in range(MN * (P - 1)):
                D_z[i, i] = -1
                D_z[i, i + MN] = 1
            D_z = D_z.tocsr()

            return D_x, D_y, D_z
```