Prepared for submission to DIS Image Analysis Coursework; Word Count: 2999

Data Intensive Science Image Analysis Coursework

dw661

University of Cambridge

E-mail: dw661@cam.ac.uk

Contents

1 Question 1

1.1 Segmentation of lungs from a CT scan

In this subsection, lungs are segmented out of a CT scan. The main steps taken were:

1. Segmentation via thresholding.

2. Postprocessing using morphological operations.

Relevant code can be found in `Q1-CT.py`; preprocessing was deemed not necessary as original image's quality hasn't been compromised.

Segmentation

In figure 1, distribution of pixel values in original image is visualised as a histogram:

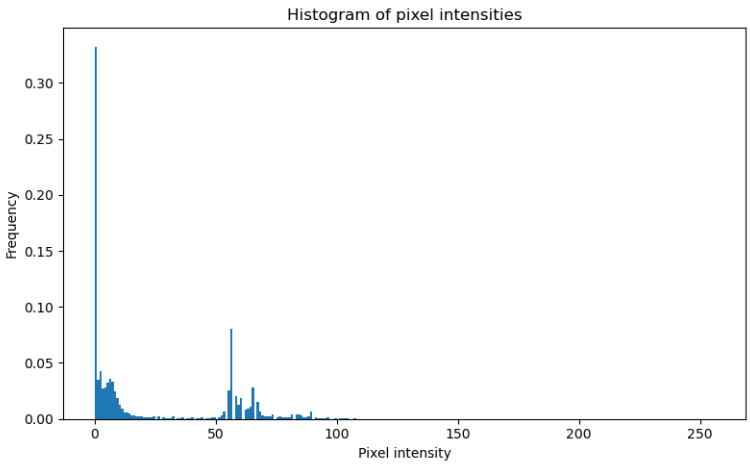

Figure 1: Histogram of pixel intensities in the CT image

The intensities roughly follow a bimodal distribution with peaks near 0 (background and lungs) and 60 (adjacent tissues and veins). The well-separability in peaks makes thresholding effective for segmentation.

Otsu's method [1] was used to determine the separation threshold. This method chooses a threshold t such that the intra-class variance $\sigma^2_{intra}(t)$ is minimised. $\sigma^2_{intra}(t)$ is defined as:

$$\sigma^2_{intra}(t) = w_1(t)\sigma^2_1(t) + w_2(t)\sigma^2_2(t) \tag{1.1}$$

$w_1(t)$ denotes the proportion of pixels below t, and $\sigma^2_1(t)$ denotes the sample variance amongst those pixels; $w_2(t)$ and $\sigma^2_2(t)$ are defined similarly. A custom implementation of Otsu's method can be found under `Q1-CT.py`; its pseudocode is detailed in algorithm 1.

Algorithm 1 Otsu's method

$var\_min \leftarrow \infty$, $best\_bin \leftarrow 0$
for $t = 0, 1, 2, 3, \cdots, 255$ **do**
  $w_1(t), \sigma_1^2(t), w_2(t), \sigma_2^2(t) \leftarrow$ Compute weights and variances
  $\sigma_{intra}^2(t) = w_1(t)\sigma_1^2(t) + w_2(t)\sigma_2^2(t)$
  if $\sigma_{intra}^2(t) < var\_min$ **then**
    $var\_min \leftarrow \sigma_{intra}^2(t)$
    $best\_bin \leftarrow t$
  end if
end for
Return $best\_bin$

Postprocessing

In the leftmost plot of figure 2, the result after thresholding is visualised; despite some success, there are still imperfections: notably, the tissues/nodules insides lung were excluded in the segmentation. To include them without compromising fidelity, opening and closing were first applied to smooth out the internal structures; the output afterwards is showcased in the middle plot of 2.

This is almost what we want, but small artefacts in the form of holes are still present in the image. Thus, `remove_small_holes` from `skimage` was called to remove them, resulting in the final segmentation in the rightmost plot of 2.

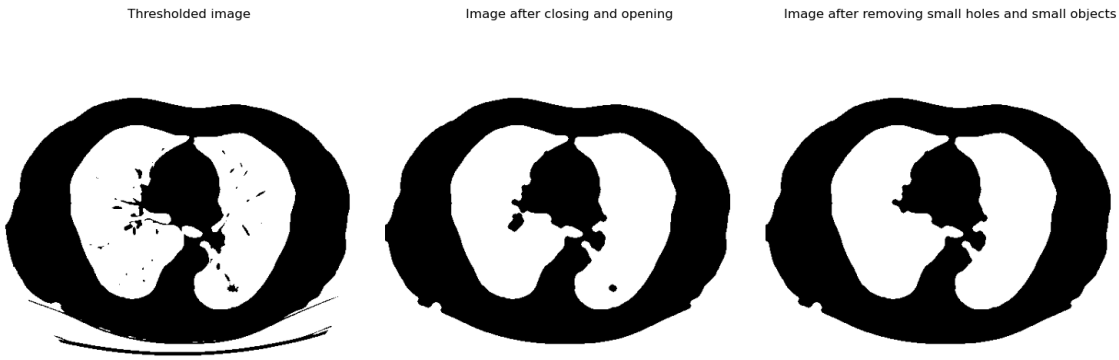

Figure 2: Figure showing a), initial result after thresholding; b) result after opening and closing; c) final segmentation after removal of small holes and objects

Lastly, masks of the two connected components corresponding to the lungs were obtained using `label` in `skimage`; placing them over the original image, figure 3 is obtained as the final segmentation.

Figure 3: Final segmentation of lungs

1.2 Segmentation of purple tulips from a noisy image

In this subsection, we detail the steps taken to segment purple tulips out of a noisy image. The main steps were:

- Preprocessing by denoising the image.
- Segmentation using KMeans.
- Postprocessing using morphological operations.

Relevant code can be found in `Q1-flowers.py`.

Preprocessing

Preprocessing was deemed necessary, as it improves the quality of clustering substantially. Total variation was used as the denoising algorithm, due to its strong ability of preserving structural fidelity (especially edges) whilst removing noise. The regularisation strength parameter, λ, was chosen to be 0.1, as empirical experimentation demonstrated it strikes a good balance between noise removal and retaining sharpness. Figure 4 shows comparison between the original image and the denoised image. The importance of preprocessing will be further highlighted in the below subsection.

Segmentation

Clustering was chosen as the segmentation method for this task, as there are 5 main colours in the image (green in leaves/stem, orange, red, white, purple in flowers); it'd be intuitive to

Figure 4: Comparison of original image and denoised image

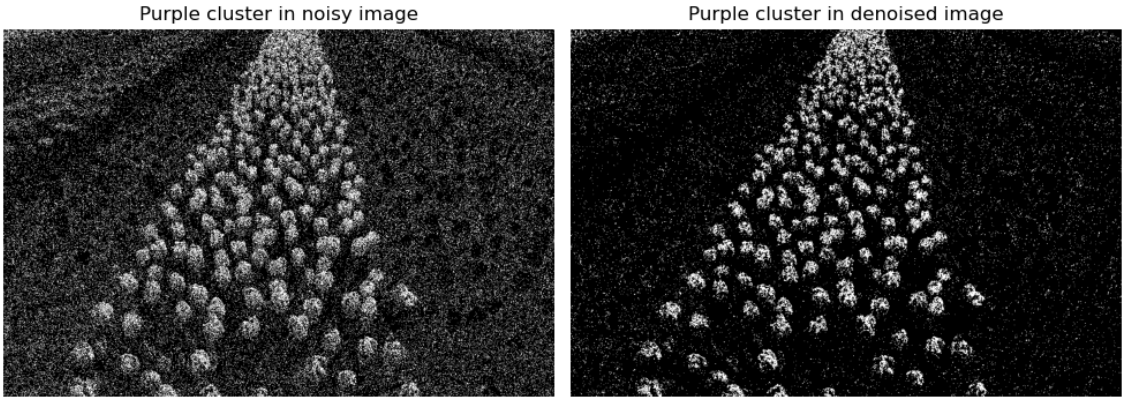

Figure 5: Illustration of selected purple clusters after clustering, denoised and undenoised

group pixels based on their colours. Moreover, edge based methods may struggle in this task, as boundaries around objects aren't particularly pronounced and it's difficult to differentiate edges that are solely associated with purple tulips. Before clustering, the image was also converted from RGB space to LAB space [2] to allow better distinction between different colours.

During clustering, we take number of clusters to be 5, and the cluster whose center is closest to purple's LAB values [3] is chosen to be the 'purple' cluster. In figure 4, the two plots illustrate the resultant 'purple' clusters obtained with and without denoising. It can be seen that the quality after denoising is much better, evidenced by its less pixelated background, thus providing more reason on why denoising was necessary.

Postprocessing

Although the cluster obtained in figure 5 largely aligns with the locations of purple tulips, there are still imperfections, evidenced by occasional infidel background pixels. Hence, for further refinement, morphological operations are performed to eliminate background artefacts. Specifically, binary closing was called to fill in the small gaps between petals, then

`remove_small_objects` from `skimage` was used to eliminate isolated background pixels. The outputs of each step are highlighted in figure 6.

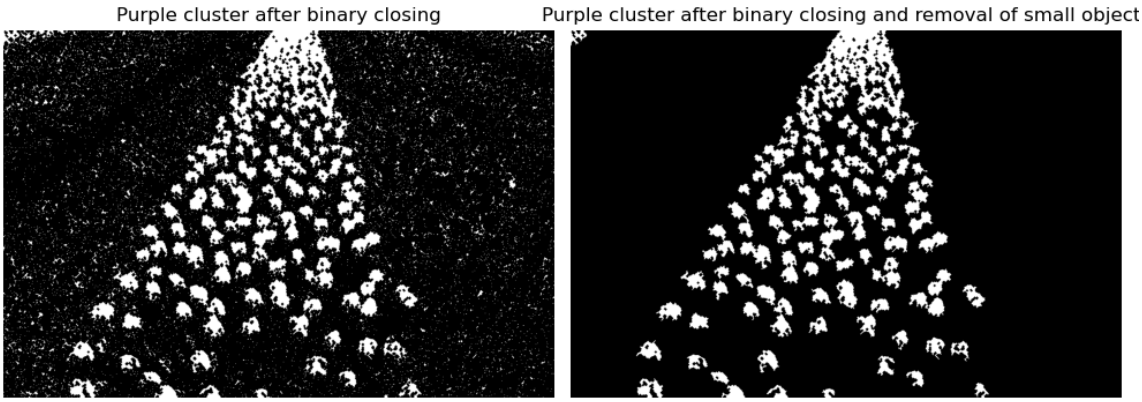

Figure 6: Outputs after each morphological operation

Placing this binary mask over the original image to highlight the segmented purple tulips, we obtain figure 7. This final segmentation is approximately on par with expectation, but it can

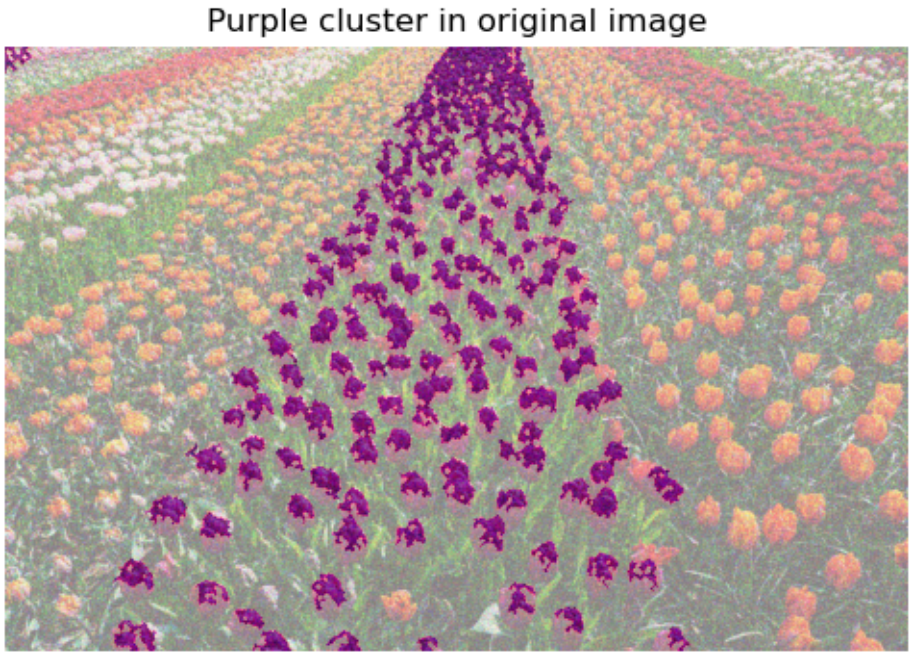

Figure 7: Highlighted segmented purple tulips

be noticed that some parts of petals are missed out in the segmentation. This likely arose because some purple pixels were lost during the denoising and postprocessing procedures.

1.3 Segmenting coins out of image corrupted by black lines

In this task, we aim to segment four specified coins (first coin in first row, second coin in second row, etc.) out of an image with 24 coins. The main steps taken were:

- Preprocessing by removing contaminating black lines.
- Segmentation using thresholding + closing + selecting connected components.
- Postprocessing by removing unwanted regions and labelling regions according to their positions in space.

Preprocessing

On the leftmost plot of figure 8, the original corrupted image is visualised. The vertical black lines present could pose significant challenges during segmentation, as it promotes disconnectivity, thus making identifying connected components difficult. To overcome this shortcoming, these lines are removed through inpainting using `inpaint_biharmonic` from `skimage` [4]. The inpainted image is highlighted on the rightmost plot of 8.

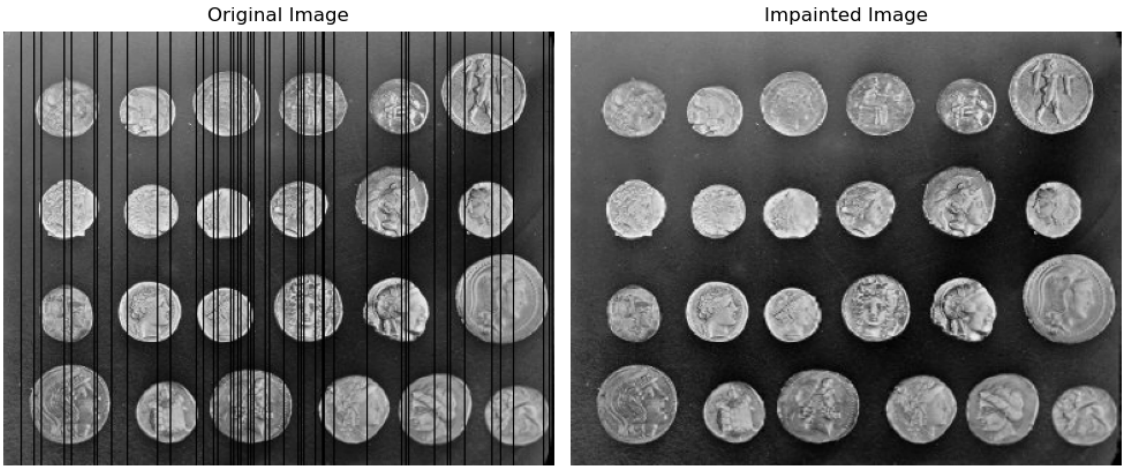

Figure 8: Comparison of original corrupted image and inpainted image

Segmentation

Watershedding was chosen as the segmentation method, complemented by additional morphological operations afterwards for better distinction of the regions. Watershedding was deemed suitable here, as there is a clear distinction between objects of interest (coins) and the background, making local minima/maxima easy to determine, thereby making the watershed algorithm straight forward to use.

Before implementation, 'elevation' and 'markers' need to be determined; 'elevation' acts the boundaries of the rasins to be flooded, and 'markers' determine the starting points of the flood. To compute 'elevation', we leveraged the Sobel kernel to detect the edges of coins,

using them as the 'rasin boundaries' for flooding. As for 'markers', all pixels below 0.1 are labelled as 1 and all above 0.6 as labelled as 2. These were chosen as they are representative of the different portions in the image (background and coins). Ideally, floods starting at 1 should fill up background and similarly, floods starting at 2 should fill up the coins.

Postprocessing

In the leftmost plot of figure 9, the result from watershedding is visualised. Although the segmentation is roughly on par with our expectations, there are still two small caveats present: (1) the second last coin in row 3 appear to have small holes present in its internal texture, and (2) a tiny detail from the background of the picture is mistakenly classified as a region. Both these caveats are highlighted in the leftmost plot in figure 9. To overcome (1), closing was used to remove holes; for (2), all identified regions are iterated through, and ones whose area is below 200 are removed. The final classified coins are highlighted on right of figure 9.

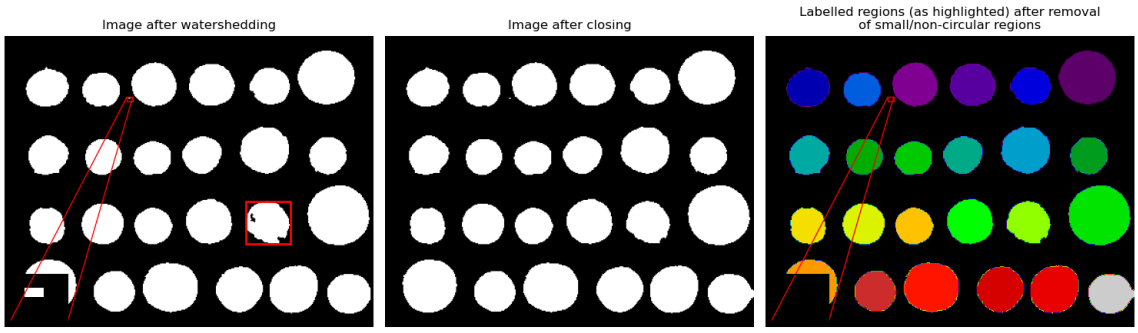

Figure 9: Figure showing a), initial result after watershedding, with specific highlights showing visible caveats; b) result after applying closing; c) final segmentation of coins after closing and removal of small regions

Lastly, to specifically select first coin in first row, second coin in second row (and etc.), centroids of segmented coins are ordered from left to right, top to bottom, thereby allowing us to access coins at specific positions via indexing. The 4 selected coins are highlighted in figure 10.

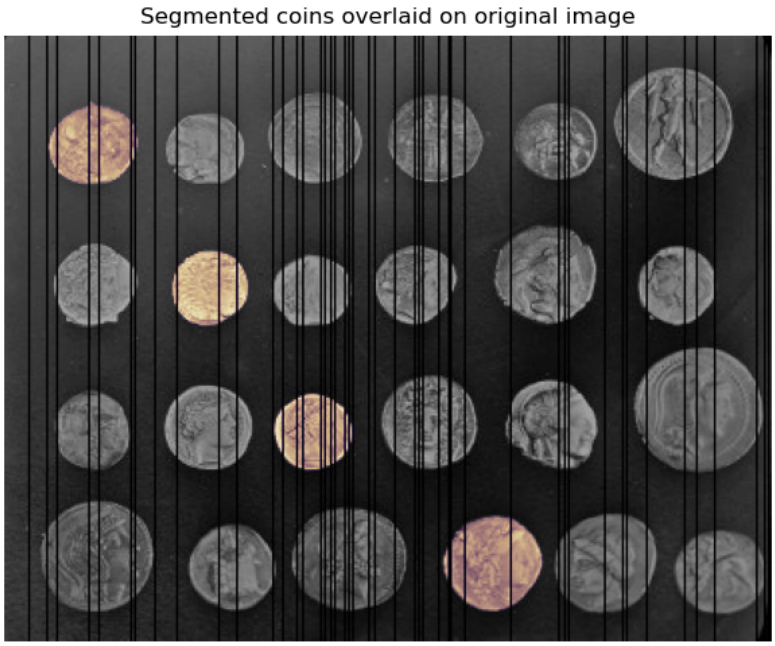

Figure 10: Final segmentation of first coin in first row, second coin in second row, etc.

2 Question 2

2.1 Question 2.1

For this question, we aim to find the straight lines of best fits to sets of (x, y) data through solving the following minimisation problem:

$$\min_{a,b} \|a\mathbf{x} + b - \mathbf{y}\|_k \tag{2.1}$$

with norms $k = 1, 2$, on two sets of data (one with only noisy measurements, one with noisy measurements + an outlier). The minimisation problems were solved using `scipy.optimize`'s `minimize` function, and the sets of fitted coefficients are as follows:

Norm/Data	a	b
L1, noisy data	0.066	0.309
L1, outlier data	0.066	0.312
L2, noisy data	0.067	0.284
L2, outlier data	0.046	0.627

Table 1: Table showing coefficient values under different norms and different data

Evidently, the fitted values changed minimally when $L1$ norm was used, but deviated substantially under $L2$ norm when outlier was introduced. This can also be seen in figure 11.

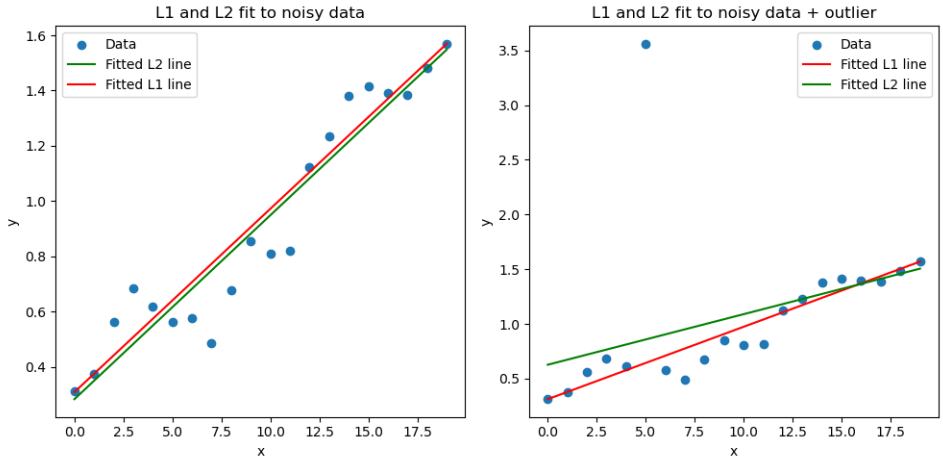

Figure 11: Plots of fitted lines minimised under different norms

$L1$ minimisation is more robust to extreme outlier, because its objective function places equal emphasis on all residual components. On the other hand, $L2$ norm is specifically penalising against residual components of larger magnitudes as it calculates the sum of residual squares. This causes fitted solution to disproportionately compensate towards outlier values, which is undesired.

However, if the data is only noisy but contains no outliers, (especially if the noise is Gaussian distributed), $L2$ may be more suitable, as computing its solution is equivalent to finding the maximum likelihood for observed data points, assuming they are Gaussian distributed.

Moreover, $L2$ solution is computationally easier to compute and guaranteed to have one unique solution, which is a property $L1$ norm doesn't possess.

2.2 Question 2.2

In this subsection, we explore the effect of sampling on performance of Compressed Sensing. A length 100 vector is first generated, with 10 of its components set to random samples from Uniform Distribution in range $[0, 4]$ and the rest set to 0. Then, Gaussian noise with $\sigma = 0.05$ are added to the signal. The original and noisy signal are both visualised in figure 12.

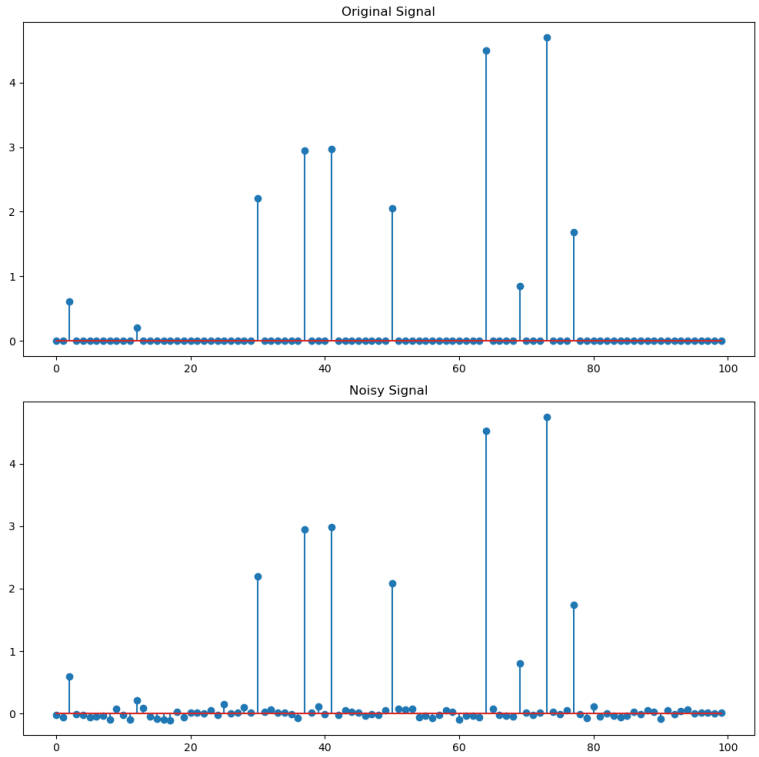

Figure 12: Figure showing the original signal and the signal with noise

Next, we take the Fourier transform on the noisy signal. Before transforming, the signal was zero-padded to length 128 (next biggest power of 2) as per usual conventions of Fourier Transform [5]. Then, the coefficients are undersampled in Fourier domain, retaining only 32 coefficients. Specifically, two sampling approaches are experimented - random sampling and equidistant sampling. These sampled coefficients are highlighted in figure 13.

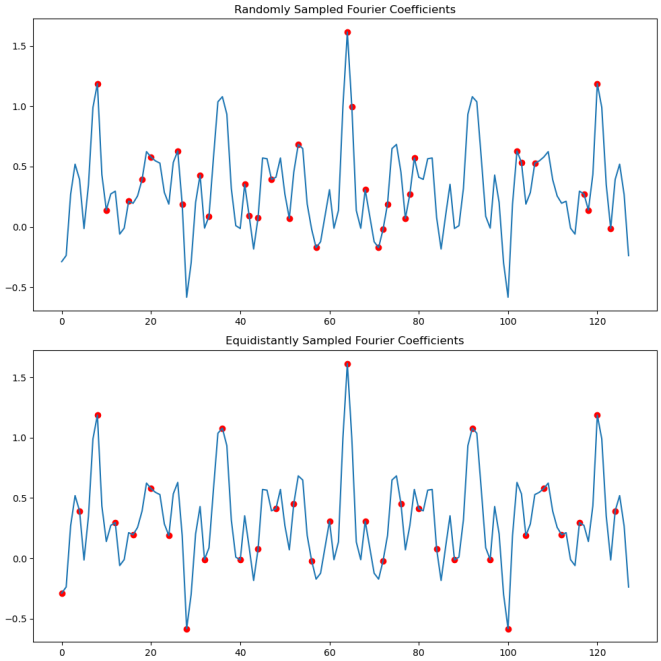

Figure 13: Figure highlighting sampled Fourier coefficients

Taking the inverse Fourier Transforms of both sets of coefficients and multiplying by 4 to account for energy lost during undersampling, the reconstructed signals are visualised in figure 14.

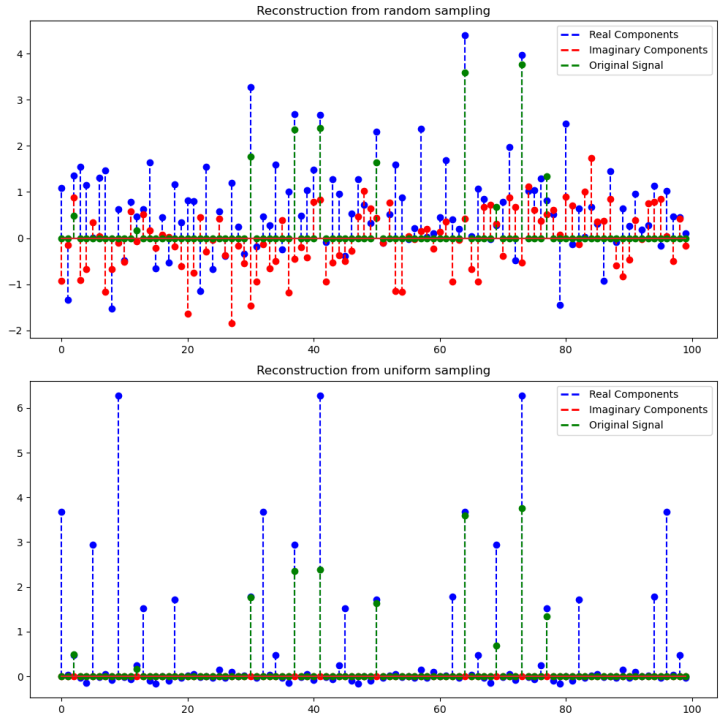

Figure 14: IFT of undersampled signals, under different sampling schemes

Both reconstructed signal suffer from aliasing. However, the aliasing artefacts differ in nature. During equidistant sampling, the aliased signal appears periodic and resembles the original signal superimposed on itself. This complicates the recovery of the original signal, as distinguishing it from the aliased components is challenging. On the other hand, under random sampling, the aliased signal demonstrates noise-like artefacts, but large components of the original signal which are non-zero still stick out above the 'noise'. Since the original signal is sparse, it can be recovered through removing aliasing noise via sparsifying regularisation approaches. Specifically, the following minimisation problem is solved:

$$\arg\min_{\hat{x}} \frac{1}{2}\|F_\Omega \hat{x} - y_\Omega\|_2^2 + \underbrace{\lambda\|\hat{x}\|_1}_{\text{Sparsifying term}} \tag{2.2}$$

where F_Ω correspond to the undersampled DFT matrix and y_Ω being the undersampled observed Fourier coefficients. Since there is no closed-form solution for 2.2, this problem is iteratively solved using complex soft-thresholding with data consistency check at each iteration. λ is a hyperparameter which governs the strength of regularisation. The procedure of solving 2.2 is detailed in algorithm below.

Algorithm 2 Iterative solution for 2.2

$y \leftarrow$ Undersampled Fourier Coefficients
$\hat{X}_0 \leftarrow y$
for $k = 1, 2, 3, \cdots 100$ **do**
  $\hat{x}_k \leftarrow \texttt{ifft}(\hat{X}_{k-1})$
  $\hat{x}_k \leftarrow \texttt{ComplexSoftThresh}(\hat{x}_k)$
  $\hat{X}_k^* \leftarrow \texttt{fft}(\hat{x}_k)$
  $\hat{X}_k \leftarrow \texttt{ConsistencyCheck}(\hat{X}_k^*)$
end for
Return x_{100}

We experimented with iteratively reconstructing using $\lambda = 0.01, 0.05, 0.1$. The results are showcased in figure 15 (errors are calculated using `np.linalg.norm`). Random undersampling showed promising results and significance of λ is apparent; when we take $\lambda = 0.01$, aliasing noise isn't removed completely, as evidenced by the low magnitude oscillations; on the other hand, when we take $\lambda = 0.1$, almost all noise are removed, but at the expense of losing magnitude on the non-zero components in the original signal. $\lambda = 0.05$ appears to strike a good balance between the two. As for the uniformly undersampled case, almost no improvement was made upon the original approximate reconstruction from 14. This is likely due to aforementioned speculations: it's difficult to distinguish the original signal out of a periodically aliased signal. Moreover, since success of compressive sensing is dependent on incoherence during sampling [6], the coherent nature of equidistant sampling contradicts this condition, hence resulting in very limited success.

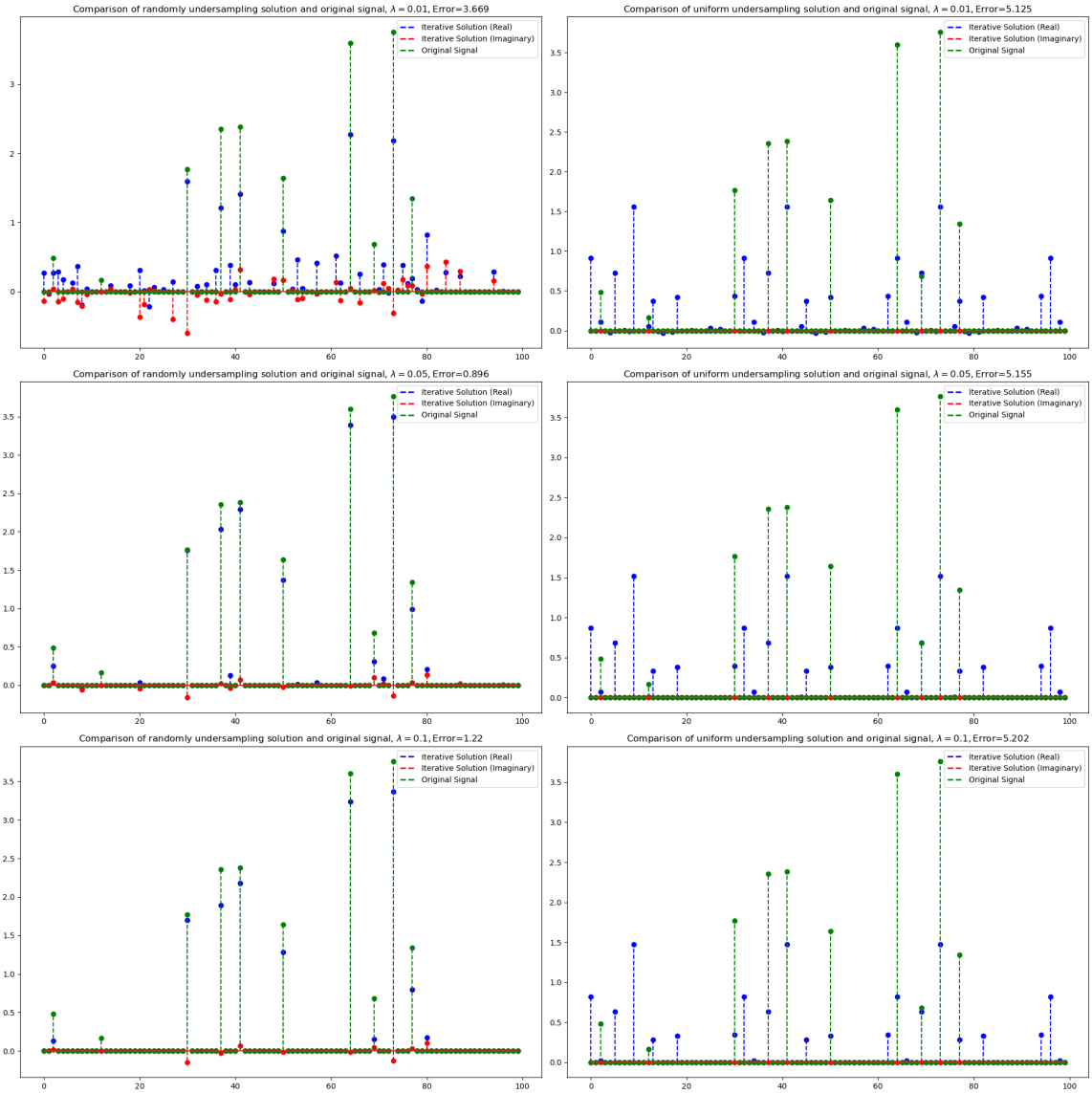

Figure 15: Comparison of iterative solutions for undersampled FT coefficients, using different sampling schemes and λ's

2.3 Question 2.3

2.3.1 Part (a)

In this question, we investigate discrete wavelet transform (DWT) and its use for compressed reconstruction. During DWT, the signal gets hierarchically decomposed into multiple levels; at each level, 'approximation' and 'detailed' coefficients are calculated; the former correspond to low-frequency components of the signal, and is computed through applying low-pass filters; similarly, the latter correspond to high-frequency components and are computed via applying high-pass filters. Once approximation coefficients are calculated, they are fed into the next level to undergo the same approximation-detailed decomposition again, until desired depth is reached.

We start by investigating the wavelet transform process of the Riverside image using Daubechies wavelets. Before computing the transforms, the image is cropped to a $(624, 832)$ image, removing surrounding blank space. Then, using `dwt2` from `helper.py`, the 4-level wavelet decomposition is computed. The wavelet coefficients are visualised in figure 16. Reconstruction is obtained through calling `idwt2`. As seen in figure 17, the reconstruction is almost perfect, with errors are only in orders 10^{-15}. These small errors likely arose since only finite number of wavelet coefficients are obtained; achieving perfect reconstruction requires knowledge of all coefficient values on a continuous spectrum of scales and positions.

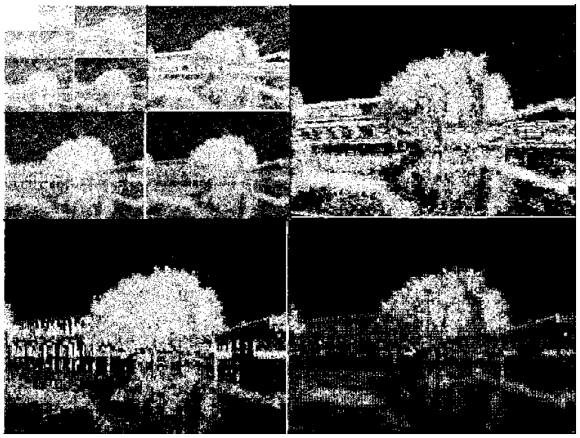

Figure 16: Display of Wavelet Coefficients after Daubechies transform

Figure 17: Figure showing (a) original image (with white space cropped); (b) reconstructed image; (c) difference image

2.3.2 Part (b)

In this subsection, the wavelet coefficients are thresholded such that only the largest 15% are retained. In figure 18, the wavelet coefficients whose magnitude are above 0.01 are highlighted before and after thresholding. Evidently, the low-frequency coefficients in the top left corner of the plot are least affected by thresholding, indicating that majority of signal's energy is concentrated near these low frequencies.

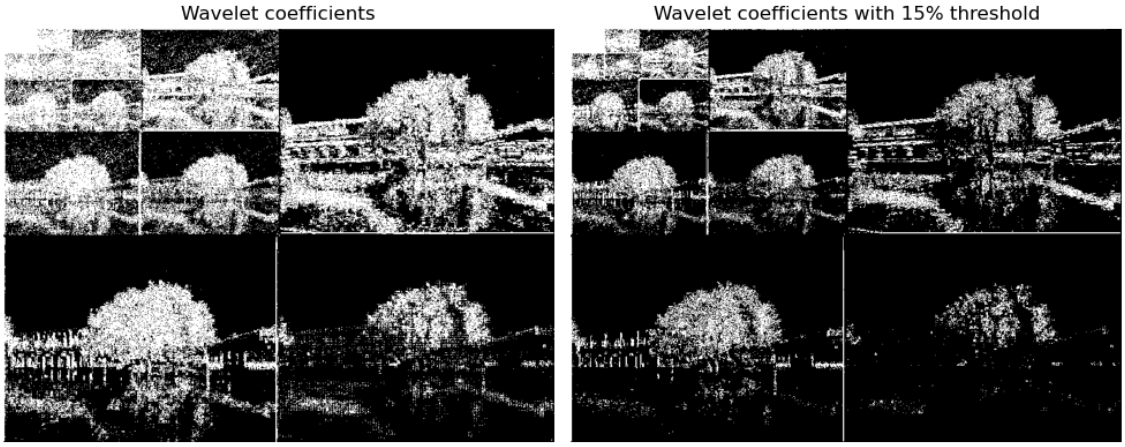

Figure 18: Comparison of wavelet coefficients (a) before thresholding; (b) after thresholding 15% largest coefficient

2.3.3 Part (c)

We retain wavelet coefficients at 4 different threshold levels $(20\%, 10\%, 5\%, 2.5\%)$ and apply `idwt2` to them respectively. The reconstructions and difference images are highlighted in figure 19. The value range in the difference images were set to be $[0, 0.1]$ for fair comparison between different thresholds.

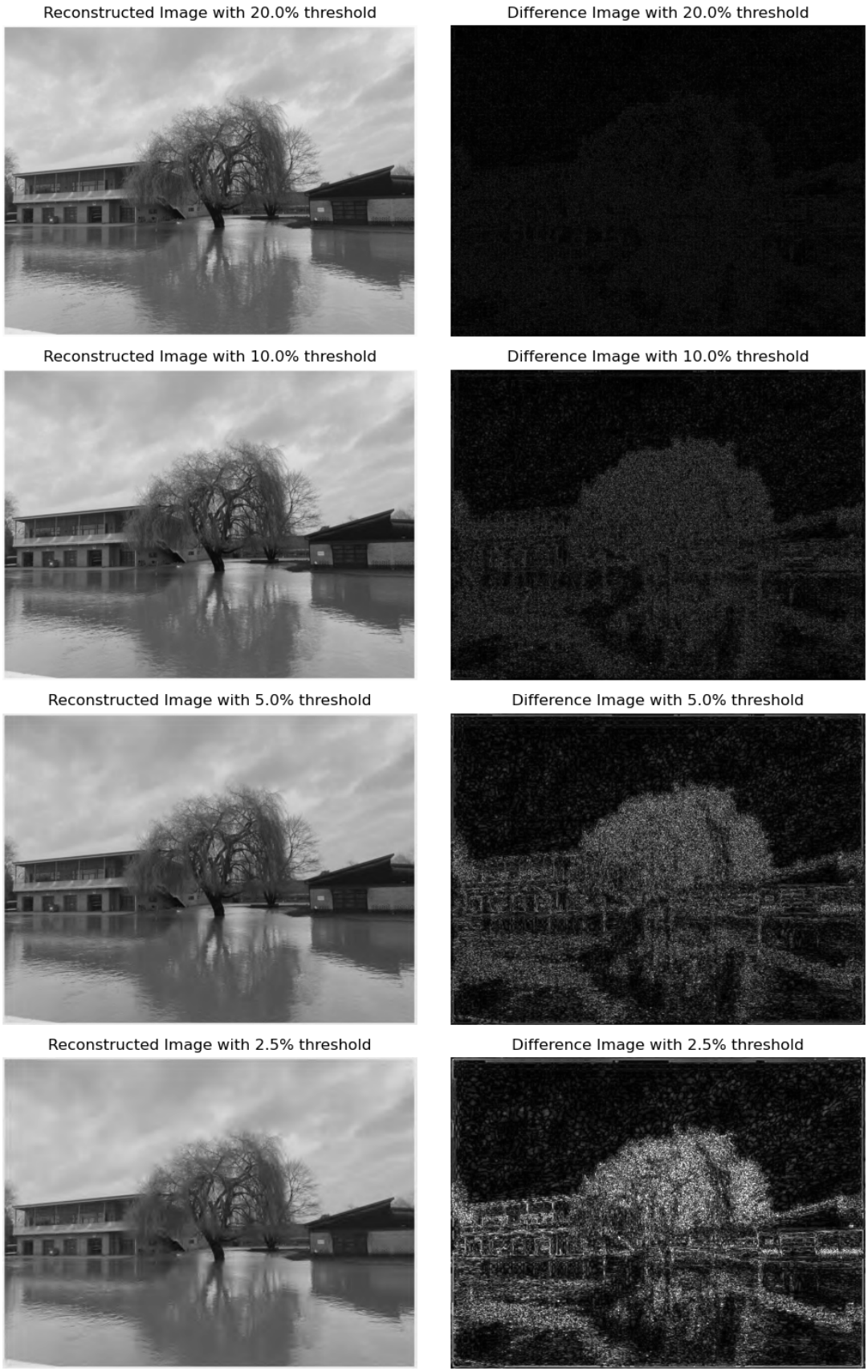

Figure 19: Visualisations of reconstructed image and difference image at different threshold levels

Observing the difference image in figure 19, the quality of reconstructions worsened somewhat as more wavelet coefficients are discarded; this is also in accordance with results from table 2, where reconstructions were evaluated quantitatively using standard image metrics such as MSE, PSNR and SSIM. Nevertheless, despite the slight decline in quality, considering that only 2.5% of the coefficients were retained in the final reconstruction, the achieved results are still remarkable.

Threshold	MSE	PSNR	SSIM
20%	2.75×10^{-5}	45.61	0.987
10%	1.36×10^{-4}	38.66	0.954
5%	3.73×10^{-4}	34.27	0.905
2.5%	7.49×10^{-4}	31.26	0.846

Table 2: Table showing evaluated image metrics at different retainment thresholds

To understand why DWT excelled at image compression, we examine the distribution of coefficient values in figure 20:

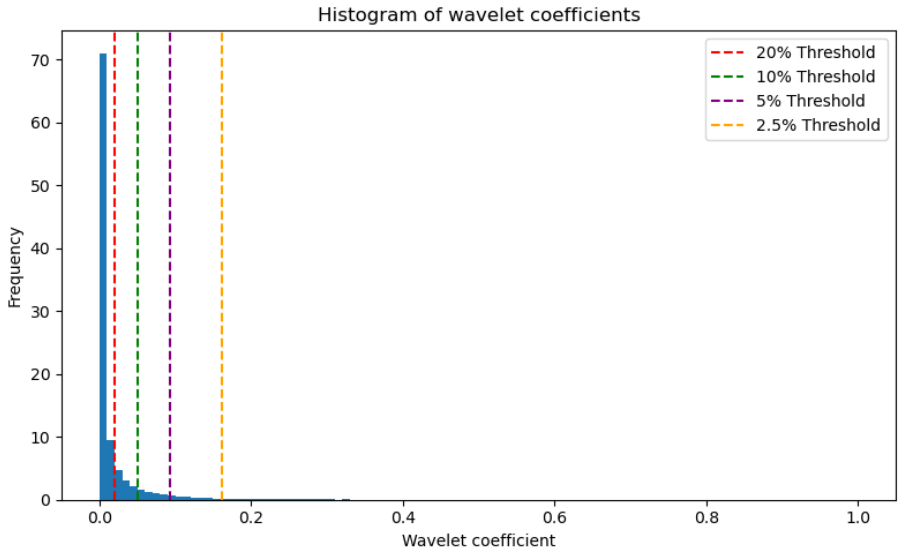

Figure 20: Histogram of wavelet coefficient values

$\approx 70\%$ of coefficient values are in the interval $[0, 0.01]$, showcasing that the wavelet representation of the image is very sparse. These near 0 coefficients contribute very little during reconstruction, thus discarding them causes relatively minimal compromise to the final images.

Another interesting observation is that, the main divergence between reconstructed image and ground truth occurs near object boundaries. Referring back to figure 18, much of the discarded coefficients are associated with high frequency values. In images, edges are often associated with high frequencies, as they represent sudden changes in values in short intervals. Therefore, thresholding wavelet coefficients can lead to some sharpness around the edges being lost, although this caveat is almost negligible to human eye.

3 Question 3

3.1 Question 3.1

The function which we aim to minimise is: $f(\mathbf{x}) = \frac{1}{2}x_1^2 + x_2^2$. ($x_1$ refers to the first component of the length-2 vector $\mathbf{x}$, and x_2 refers to the second component). Recall that:

- If f is convex and L-smooth, then:

$$f(\mathbf{x}_K) - f(\mathbf{x}^*) \leq \frac{L\|\mathbf{x}_0 - \mathbf{x}^*\|_2^2}{2K} \tag{3.1}$$

Thus, to employ 3.1 to use, we start by proving convexity of f. The Hessian Matrix of f is given by:

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \tag{3.2}$$

This matrix is positive definite for all values of (x_1, x_2), thus it's a convex function.

Next, following the hint from the question, we prove that f is 2-smooth. Recall the definition of an L-smooth function:

- A function f is L-smooth if its derivative, ∇f, is Lipschitz-continuous with constant L:

$$\|\nabla(\mathbf{x}) - \nabla(\mathbf{y})\|_2 \leq L\|\mathbf{x} - \mathbf{y}\|_2 \tag{3.3}$$

The derivative of f is given by:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} x_1 \\ 2x_2 \end{pmatrix} \tag{3.4}$$

Let $\mathbf{x} = (x_1, x_2), \mathbf{y} = (y_1, y_2)$ be 2 arbitrary points in $\mathbb{R}^2$. LHS of 3.3 reads:

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\|_2 = \sqrt{(x_1 - y_1)^2 + 4(x_2 - y_2)^2} \tag{3.5}$$

Using $L = 2$, RHS becomes:

$$2\|\mathbf{x} - \mathbf{y}\|_2 = 2\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{3.6}$$

Since both 3.5 and 3.6 are positive, we can square them both and compare their magnitudes. Squaring 3.5, we get:

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\|_2^2 = (x_1 - y_1)^2 + 4(x_2 - y_2)^2 \tag{3.7}$$

Similarly, squaring 3.6, we get:

$$(2\|\mathbf{x} - \mathbf{y}\|_2)^2 = 4(x_1 - y_1)^2 + 4(x_2 - y_2)^2 \tag{3.8}$$

Evidently, $(x_1 - y_1)^2 + 4(x_2 - y_2)^2 \leq 4(x_1 - y_1)^2 + 4(x_2 - y_2)^2$, thus confirming ∇f is Lipschitz continuous with respect to 2 and f is therefore 2-smooth.

It can also be shown that 2 is the smallest L such that f is L-smooth. To prove this, suppose on the contrary that $\exists\lambda$ such that f is λ-smooth and $\lambda < 2$. Considering the inequality from 3.3 again, the LHS stays the same:

$$\sqrt{(x_1 - y_1)^2 + 4(x_2 - y_2)^2}$$

But RHS now becomes:

$$\lambda\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$$

Choosing $\mathbf{x}=(x_1,x_2),\mathbf{y}=(y_1,y_2)\in\mathbb{R}^2$ such that $x_1=y_1$, then LHS becomes:

$$\sqrt{(x_1-y_1)^2+4(x_2-y_2)^2}=\sqrt{4(x_2-y_2)^2}=2|x_2-y_2|$$

and RHS becomes:

$$\lambda\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}=\sqrt{(x_2-y_2)^2}=\lambda|x_2-y_2|$$

Evidently:

$$2|x_2-y_2|\nleq\lambda|x_2-y_2| \tag{3.9}$$

which is a contradiction.

We can now put 3.1 to use. The global minimal point of the function is given by $(0,0)^T$, as it's the point which achieves the minimum admissible value for the sum of two squared terms (0). Substituting $L=2,\mathbf{x}_0=(1,1)^T,\mathbf{x}^*=(0,0)^T,\epsilon=f(\mathbf{x}_K)-f(\mathbf{x}^*)=0.01$, we have:

$$0.01\leq\frac{2\|\mathbf{x}_0-\mathbf{x}^*\|_2^2}{2K} \tag{3.10}$$

$$=\frac{2\times2}{2K} \tag{3.11}$$

$$=\frac{2}{K} \tag{3.12}$$

Upon rearranging we can obtain:

$$K\leq200 \tag{3.13}$$

Thus, 200 is the upper bound on the number of iterations for convergence to accuracy of 0.01.

In `Q3-1.py`, Gradient Descent was implemented to minimise f with stepsize $\frac{1}{L}=\frac{1}{2}$. The number of iterations to arrive at accuracy of 0.01 was 3, a lot lower than the proposed upper bound of 200, which may seem surprising. However, it can be proved that f is 1-strongly convex. A function f is μ-strongly convex if its Hessian ∇^2 satisfies [7]:

$$\nabla^2 f(\mathbf{x})\succeq\mu I \tag{3.14}$$

In other words, $\nabla^2 f(\mathbf{x})-\mu I$ is positive semidefinite. Referring back to equation 3.2, taking $\mu=1$, we have:

$$\nabla^2 f(\mathbf{x})-I=\begin{pmatrix}0&0\\0&1\end{pmatrix} \tag{3.15}$$

and it's evident that 1 is the largest constant such that $\nabla^2 f(\mathbf{x})-\mu I$ remains positive semi-definite. Thus, we conclude that f is 1-strongly convex.

For a μ-strongly convex function, the following statement holds regarding the convergence rate of Gradient Descent:

$$\|\mathbf{x}_K-\mathbf{x}^*\|_2^2\leq\left(1-\frac{\mu}{L}\right)^K\|\mathbf{x_0}-\mathbf{x}^*\|_2^2 \tag{3.16}$$

This gives a convergence rate of $\mathcal{O}(\log(\frac{1}{\epsilon}))$ ($\log(100)\approx4.6$ using $\epsilon=0.01$), whereas the original bound is of order $\mathcal{O}(\frac{1}{\epsilon})=100$. This new improved order of convergence rate aligns better with the observed number of iterations.

3.2 Question 3.2

In this subsection, we discuss the completed implementations and the theoretical foundations behind them. All relevant code can be found under the Python file `Q3-2.py` and `LGD_models.py`.

3.2.1 LGD Net

In CT imaging, minimisation problems of the following form arise often:

$$\min_x \underbrace{\|Ax - y\|_2^2}_{\text{Data fidelity term}} + \underbrace{g(x)}_{\text{Regularisation term}} \tag{3.17}$$

where x refers to the unknown ground truth image, A denotes the forward Ray Transform (usually discretised as multiplication by a matrix A), y refers to the observed sinogram. Under proximal Gradient Descent algorithm, each step involves updating parameters as follows:

$$x_{k+1} = prox_g(x_k - \tau \nabla f(x_k)) \tag{3.18}$$

where f is the data fidelity term ($\|Ax - y\|_2^2$) in equation 3.17 and its derivative ∇f is given by $2A^T(Ax_k - y)$. Under Learned Gradient Descent, we let the network learn how to combine x_k and $\nabla f(x_k) = 2A^T(Ax_k - y)$ during each unrolled iteration, as well as the update stepsizes τ_k, thus replacing the proximal operators with learned neural networks.

In implementation, the forward transform A is done by `fwd_op_odl`, the adjoint transformation A^T by `adj_op_odl`. `odl_torch` was utilised to make operators into `OperatorModule` objects, allowing them to be passed into `torch` networks and enjoy auto-differentiation. The completed algorithm in the forward pass of `LGD_net` is as follows:

Algorithm 3 LGD

for $k = 0, 1, 2, 3, 4$ **do**
- $u \leftarrow Ax_k - y$ ▷ Calculate the current residual
- $grad \leftarrow A^T u$ ▷ Calculate ∇f through multiplying by A^T
- $dx \leftarrow h_{\theta_k}(x_k, grad)$ ▷ Calculate the learned update using networks h_{θ_k}
- $x_{k+1} \leftarrow x_k + \tau_k dx$ ▷ Update x

end for

Return x_5

3.2.2 `prox_net`

The class `prox_net` contains implementation for the networks h_{θ_k} from algorithm 3. The specific structure was chosen to be 3-layer CNN with `PReLU` activations between layers; each layer consists of 32 input and output channels, barring the original input (containing 2 channels for $x_k, grad$) and final output (consisting of only 1 channel for output).

3.2.3 Training

During training, we initialise x_0 from 3 to be the filtered backprojection of noisy measurement y, as it's an approximate pseudoinverse, thus providing a good starting point for the algorithm; training was ran for 2000 iterations, with MSE used as the loss function. In figure 21, the obtained reconstructions from FBP, TV and LGD are visualised:

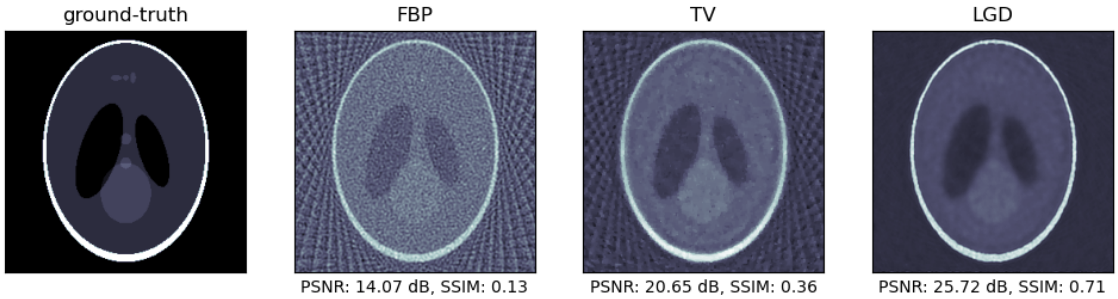

Figure 21: Comparison of reconstructions from different algorithms

Out of the three methods, FBP resulted in worst performance, evidenced both by the noisy background and suboptimal image metrics; this may be attributed to its unstability, as its filtering term in Fourier domain ($|\omega|$) magnifies high frequency components of the image which are often associated with noise.

As for Total Variance Regularisation, the following optimisation problem is solved:

$$\min_x \underbrace{\|Ax - y\|_2^2}_{\text{Data fidelity term}} + \underbrace{\lambda \|\nabla x\|_1}_{\text{Regularisation term}} \tag{3.19}$$

Since $\lambda\|\nabla x\|_1$ is non-smooth, ADMM algorithm is used to iteratively solve this. The reconstruction from TV resulted in substantial improvement compared to FBP, however visible artefacts in the forms of background lines are still present.

Learned Gradient Descent resulted in the best reconstruction, beating TV and FBP by substantial margins. It generalises classical regularisation based reconstruction methods by combining their iterative update structure with the representative power of neural networks. Instead of enforcing same update structure and stepsize at each iteration, the large paramterisation of neural networks admits relaxation on these constraints by making them learnable, thus making finding optimal updates easier. Moreover, once trained, LGD takes ≈ 1 second to generate a reconstruction, whereas TV takes substantially longer; this makes LGD more suitable for use in clinical settings, as immediate reconstructions are more preferred.

References

[1] Otsu's method, Wikipedia. Available at: https://en.wikipedia.org/wiki/Otsu%27s_method.

[2] CIELAB color space, Wikipedia. Available at:
https://en.wikipedia.org/wiki/CIELAB_color_space

[3] Purple CIELAB values, Converting Colors. Available at:
https://convertingcolors.com/cielab-color-45.36_78.75_-77.41.html

[4] Inpaint Biharmonic, Scikit-Images. Available at:
https://scikit-image.org/docs/stable/auto_examples/filters/plot_inpaint.html

[5] FFT Zero-Padding, Bitweenie. Available at:
https://www.bitweenie.com/listings/fft-zero-padding/

[6] Sparsity and Incoherence in Compressive Sampling, Arxiv. Available at:
https://arxiv.org/pdf/math/0611957

[7] Lecture Notes, Princeton University. Available at:
https://www.princeton.edu/ aaa/Public/Teaching/ORF523/ORF523_Lec7.pdf

A Use of ChatGPT and Generative AI tools

During the completion of coursework, generative tools such as ChatGPT and CoPilot were used supportively and minimally. All code involving any algorithms or calculations were entirely produced by myself; Copilot was only partially used for Docstring and plotting, and ChatGPT was only used for latex syntax queries. Examples of prompts include:

"How to align plots properly in LaTex document?"