Block Compressed Sensing with Soft Thresholding: Exploring Transfer Domains and Block Sizes for Performance Optimization

RAAHUL L S - CB.EN.U4ECE22143
DHAYADHARSH S M - CB.EN.U4ECE22147
SHRIMATHAN - CB.EN.U4ECE22149
VISHAL SRIVATSAN - CB.EN.U4ECE22156
Department of Electronics and Communication Engineering,
Amrita School of Engineering, Coimbatore, Amrita Vishwa Vidyapeetham, India

***Abstract*—In this paper, we propose a GUI-based compressed sensing algorithm with a divide and conquer approach popularly known as block processing along with an efficient post-processing technique, that can be used to reduce the storage space required to store document images, which are the most widely generated images worldwide.We also provide here with a detailed investigation of several methods such as wavelet, DCT, fourier transforms, and so on with varied parameters in order to determine the best combination for the reconstruction of compressed sensed picture signals.**

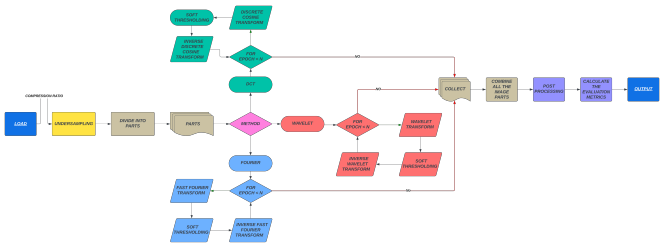

Fig. 1. Graphical Abstract

***Index Terms*—GUI-based compressed sensing,Document image compression, Divide and conquer approach,Post-processing techniques, Storage space reduction,Wavelet transform, Discrete Cosine Transform (DCT),Fourier transform, Reconstruction algorithms,Parameter optimization**

I. Introduction

In recent years, it has become clear that we as humans generate enormous amounts of data as a result of advancements in communication and signal processing, as well as easy access to the internet, resulting in a massive amount of demand for storage such as cloud storage and data centers, which not only consume electronic resources but also indirectly influence various issues such as global warming. As a result, we require a new method of storing our knowledge in a more effective and efficient manner. This search begs us the question: Do we need to collect all this data in its entirety?

Historically, Galileo Galilei's principle, "Measure what can be measured," suggested comprehensive data collection. However, Albert Einstein countered this view, saying, "Not everything that can be counted counts." Modern understanding aligns more with Einstein, recognizing that the informational content of data is often far less than its sheer volume suggests [1]. The image space is enormous and contains an uncountable amount of combinations. However, in that enormous expanse, only a small number of combinations have visual meaning; the rest are simply pepper salt noise. In this vast domain the Nyquist theorem state that a signal can be properly reconstructed from its samples if the sampling rate is more than twice its highest frequency [2] and this is a well-known statement in the signal processing domain. However this is true only if no prior information about the signal is available; nevertheless, it has been demonstrated that given knowledge of the signal's sparsity, the signal can be reconstructed with fewer samples than the Nyquist theorem indicates. Sparsity is defined as the proportion of zero elements in a matrix [3].Furthermore, most real-world signals are very sparse [4]. Using this we could easily recreate most of our real world signals. And this is the core idea behind compressed sensing also called as compressive sensing [3]. It challenges the classic Nyquist-Shannon theorem by allowing for signal reconstruction with much less samples than the theorem's limitations.This translates into a number of possible advantages, such as:

- Shorter Acquisition Times: CS can greatly used to reduce the signal acquisition times in applications like real-time signal processing or medical imaging (MRI, for example), resulting in faster scans [5].
- Increased Bandwidth Efficiency: In situations where our bandwidth is limited we could use compressive sensing to represent our signal in fewer number samples reducing the number of transmissions required and hence increasing the bandwidth effciency [6]
- Better hardware Design: The CS's lower sampling needs which opens the door to the invention of sensors that is more energy and space efficient [7].

II. Literature Survey

The pioneers in the field of compressed sensing are David Donoho (Stanford University) ,Emmanuel Candes (Cal-

tech University), Richard Banuik(RICE University), Terrence Tao(University of California).Their work was the first to establish that it was possible to recover a vector x0(e.g. a digital signal or image) from incomplete and contaminated observations y = Ax0 + e; A is a n by m matrix with far fewer rows than columns and e is an error term. This paper highlights the importance of sparsity in reconstruction of under-sampled images [8].

After its conception, David Donoho, Michael Lustig, John Pauly in their work described how MRI scans can be made less harmful by reducing acquisition time and amount of exposure by using CS algorithms by taking lesser number of samples.

Then their paper titled "Robust Uncertainty Principles:Exact Signal Reconstruction from Highly Incomplete Frequency Information" [9]explains the necessary conditions that should be satisfied by the data to ensure robustness and stability of the reconstruction procedure.

Gandhiraj.R, Soman.K.P in their work has given a comprehensive outline all possible reconstruction algorithms, measurement matrices, transform domain for compressed sensing and have determined OMP-based reconstruction was efficient. [10] The convex optimisation technique proposed by H.Anupama, S.Shantala demonstrates the usage of Orthogonal Matching Pursuit(OMP) which is a greedy algorithm that could be used for sparse reconstruction. The paper shows that OMP could reduce the amount of Random linear measurements required for the reconstruction procedure [11].

Robert Vanderbei, Han Liu in their work has focused on using Kroneckar optimisation instead of standard convex optimisation techniques which is noted to provide a dramatic improvement in computation time.The only drawback of this method is that it requires stronger conditions for perfect recovery which decreases its generalization quality [12].

When it comes to implementation of CS the main challenge is to produce a random matrix for subsampling the data. Thus to reduce computations for producing the random matrix mask used for sampling the data Thong.T, Lu Gan have proposed a standard Random Matrix which pre-randomize a sensing signal by scrambling the signal and then sub-sampling the signal in frequency domain [13]. This method proves to be efficient for block based CS. Lu Gan also published paper on Block compressed Sensing which divides the image into various blocks for which compressed sensing algorithm is separately implemented [14]. Since each block is reconstructed individually, the over-all process can be done faster using Block-wise Compressed Sensing .

Yusuke Oike(Sony) and El Gamal has designed a custom CMOS chip which could reduce the amount of energy consumption by a factor of 15 by implementing CS algorithms [15]. Even though conventional image compression could reduce the readout power consumption, it could not reduce the Analog to Digital conversion power.

Dimitris Bertsimas and Johnson proposed a method which uses a custom branch-and-bound algorithm with second order cone relaxation to improve solution quality, producing sparser solutions with lower reconstruction errors compared to existing methods [16]. Tested on synthetic and real-world ECG data, this approach shows significant improvements in both sparsity and accuracy within minutes.

P Raj, T Malathi Nair, R Uma demonstrates compressive sensing with Deep Neural Network (CS-DNN) approach [17]. They have compared their model with existing models and proved their model to be more efficient and versatile Machidon, Pejović and Veljko explores integration of Deep-learning and CS. Their paper explains how this could lead to higher reconstruction efficiency but would require significant amount of resources which would not be suitable for on-device training. They also note it could be solved by using transferred learning.

III. Mathematical formulations

The central problem in compressed sensing is to solve highly under-determined systems of linear equations for sparse solutions. Formally, for a sparse vector $x \in \mathbb{R}^n$ and a matrix $A \in \mathbb{R}^{m \times n}$ (with $m \ll n$), we aim to reconstruct x from $y \in \mathbb{R}^m$ where:

$$y = Ax$$

In the general case, where x is represented sparsely using a matrix Φ, we solve for the coefficient vector c:

$$y = A\Phi c$$

1) Solution Strategies: First, we must determine whether the sparsest solution to $y = Ax$ is unique, formulated as the minimization problem:

$$\min \|x\|_0 \quad \text{subject to} \quad y = Ax$$

This problem is generally NP-hard due to the combinatorial nature of the search. However, replacing $\|\cdot\|_0$ with the ℓ_1-norm leads to the problem:

$$\min \|x\|_1 \quad \text{subject to} \quad y = Ax$$

This can be efficiently solved using linear programming. Proving that the minimization of ℓ_0 and ℓ_1 yields the same solution under certain conditions has been a significant breakthrough.

2) Success Guarantees: Key conditions for successful compressed sensing include the sparsity of x and the incoherence of A. The mutual coherence of A, defined by:

$$\mu(A) = \max_{i \neq j} \frac{|\langle a_i, a_j \rangle|}{\|a_i\|_2 \|a_j\|_2}$$

where a_i are the column vectors of A, provides an upper bound on the number of non-zero entries x can have for accurate reconstruction.

Another crucial property is the Restricted Isometry Property (RIP), which measures how well A preserves the Euclidean length of sparse vectors. If A satisfies RIP of order k with a small constant δ_{2k}, exact reconstruction is possible, even in the presence of noise.

IV. Proposed Methodology

The main objective of this project is to optimize the simple projection onto convex sets(POCS) iterative algorithm by fine-tuning the parameters involved in the process.

A. Overview

1) Load grayscale images from a specified folder.
2) Perform image under-sampling.
3) Reconstruct the under-sampled images using compressed sensing while varying the parameters.
4) Calculate performance metrics (MSE, PSNR, SSIM, time taken) for the reconstructed images.
5) Save these metrics to a CSV file.
6) Analyze and visualize the distribution of the metrics.

Now the main aim is to form the reconstruction algorithm

B. Problem Formulation

The main condition that should be fulfilled by the image for accurate reconstruction is to have high sparsity.If it is satisfied then the goal is to reconstruct the sparse signal $\hat{x}$ by solving the following optimization problem which is nothing but the l1 norm:

$$\hat{x} = \arg\min_x \frac{1}{2}\|x - y\|_2^2 + \lambda\|x\|_1$$

Here:

- $\hat{x}$ is the estimated sparse signal.
- x is the undersampled Fourier transform of the estimate.
- y are the observed samples of the Fourier transform (of the original sparse signal).
- λ is a regularization parameter that controls the sparsity of the solution.

Now, we will consider three cases:

- When $y_i > \lambda$:

$$\frac{d}{dx_i}\left(\frac{1}{2}(x_i - y_i)^2 + \lambda|x_i|\right) = 0$$

$$x_i - y_i + \lambda = 0$$

$$x_i = y_i - \lambda$$

- When $y_i < -\lambda$:

$$\frac{d}{dx_i}\left(\frac{1}{2}(x_i - y_i)^2 + \lambda|x_i|\right) = 0$$

$$x_i - y_i - \lambda = 0$$

$$x_i = y_i + \lambda$$

- When $-\lambda \leq y_i \leq \lambda$:

$$\frac{d}{dx_i}\left(\frac{1}{2}(x_i - y_i)^2 + \lambda|x_i|\right) = 0$$

$$x_i - y_i + \lambda = 0 \quad (\text{for } x_i > 0)$$

$$x_i - y_i - \lambda = 0 \quad (\text{for } x_i < 0)$$

$$x_i = 0$$

So:

- When $y_i > \lambda$:

$$x_i^* = y_i - \lambda$$

- When $y_i < -\lambda$:

$$x_i^* = y_i + \lambda$$

- When $-\lambda \leq y_i \leq \lambda$:

$$x_i^* = 0$$

The following cases when applied is the necessary soft thresholding function.Soft-thresholding is a crucial technique used in signal processing, particularly in sparse signal recovery and denoising. It is a form of shrinkage or regularization that promotes sparsity by reducing the magnitude of small coefficients to zero while preserving larger coefficients. This technique is especially useful in compressed sensing, wavelet-based denoising, and other contexts where signal sparsity is desired.

Now given a signal x and a threshold λ, the soft-thresholding operation `SoftThresh`(x, λ) is defined as:

$$\texttt{SoftThresh}(x, \lambda) = \text{sign}(x) \cdot \max(|x| - \lambda, 0)$$

In other words:

- If the absolute value of a coefficient $|x|$ is less than or equal to λ, the coefficient is set to zero.
- If the absolute value of a coefficient $|x|$ is greater than λ, the coefficient is shrunk towards zero by λ.

This can be broken down into the following steps:

1) **Magnitude Reduction**: For each coefficient x, reduce its magnitude by λ.
2) **Thresholding**: If the magnitude of the coefficient is less than λ, set it to zero.
3) **Sign Preservation**: The sign of the original coefficient is preserved in the resulting coefficient.

Graphically, soft-thresholding can be visualized as follows:

- For coefficients x in the range $[-\lambda, \lambda]$, the output is zero.
- For coefficients x greater than λ, the output is $x - \lambda$.
- For coefficients x less than $-\lambda$, the output is $x + \lambda$.

C. Iterative Reconstruction Algorithm

To solve the optimization problem, the algorithm iteratively updates the estimate of the signal by alternating between enforcing sparsity and data consistency. Here's the step-by-step process:

1) **Initialize:**
   - Set $X^0 = y$, the initial estimate of the Fourier-transform of the signal, where y includes zeros for non-acquired data.
2) **Iterate until Convergence:** For the i-th iteration:
   a) **Inverse Fourier Transform:** Compute the inverse Fourier transform to get an estimate of the signal in the time domain: $\hat{x}^i = F^* X^i$
   b) **Soft-Thresholding:** Apply soft-thresholding to promote sparsity in the time domain: $\hat{x}^i = \text{SoftThresh}(\hat{x}^i, \lambda)$

c) **Fourier Transform:** Compute the Fourier transform of the thresholded signal: $X^i = F\hat{x}^i$

d) **Data Consistency:** Enforce data consistency for the measured observations in the frequency domain:

$$X^{(i+1)}[j] = \begin{cases} X^i[j] & \text{if } y[j] = 0 \\ y[j] & \text{if } y[j] \neq 0 \end{cases}$$

3) **Convergence Check:** Repeat the iteration until the difference between successive estimates is below a specified threshold ϵ:

$$\|\hat{x}^{(i+1)} - \hat{x}^i\|_2 < \epsilon$$

D. Why Split Image into Blocks

- **Parallel Processing:** By splitting the image into smaller blocks, we can leverage multithreading to perform operations on each block in parallel, significantly reducing computation time.
- **Memory Management:** Smaller blocks require less memory to process, which is beneficial when working with high-resolution images.
- **Localized Features:** Image features are often localized, and processing smaller blocks can capture these features more effectively, improving reconstruction quality.
- **Error Containment:** Errors in reconstruction are contained within blocks, which can be easier to manage and correct compared to errors spread across the whole image.

V. Implementation

A. Load images from a specified folder

B. Under-sample an image by a specified rate.

C. Split an image into n number of parts.

D. Reconstruct the image using different domains (wavelet, Fourier, DCT).

E. Perform image reconstruction using a specified domain.

F. Visualize an image using Matplotlib.

G. Calculate performance metrics for the reconstructed image.s

H. Perform the complete reconstruction process and evaluate the performance of each technique.

VI. Results and discussion

A. Dataset

The dataset used was of University of Granada which contained 50 standard black and white images used for Image processing of size 512x512 [20].

B. Performance Metrics

1) Mean Squared Error (MSE): MSE measures the average squared difference between the original and reconstructed signals or images [18]. It provides a measure of the overall distortion introduced by the reconstruction algorithm. Lower MSE values indicate better reconstruction quality. It is calculated as:

$$\text{MSE} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \left(I(i,j) - K(i,j)\right)^2 \tag{1}$$

where I and K are the original and reconstructed images, respectively, and m and n are the dimensions of the images.

Acceptable Range:

- MSE values range from 0 to infinity.
- Lower values, closer to 0, are better. Typical acceptable values depend on the image content and the specific application but are generally small.

2) Peak Signal-to-Noise Ratio (PSNR): PSNR is a metric used to measure the quality of reconstructed images [19]. It compares the maximum possible signal power to the power of distorting noise that affects the fidelity of its representation. Higher PSNR indicates better quality of the reconstructed image. Mathematically, it is expressed as:

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{Max(x_i)^2}{\text{MSE}} \right) \tag{2}$$

where $Max(x_i)$ is the maximum possible pixel value of the image and MSE is the Mean Squared Error between the original and the reconstructed image.

Acceptable Range [19]:

- PSNR values typically range from 20 to 40 dB for most applications.
- Higher values, usually above 30 dB, indicate high quality.
- Values below 20 dB often indicate poor quality.

3) Structural Similarity Index Measure (SSIM): SSIM measures the similarity between two images, taking into account luminance, contrast, and structure. It evaluates both local and global image quality. SSIM values range from -1 to 1, with 1 indicating perfect similarity. SSIM is often preferred over PSNR for assessing perceived image quality [19]. At last we also measured time taken by each image to find the complexity of the overall algorithm. SSIM values range from -1 to 1, where 1 indicates perfect similarity. It is defined as:

$$\text{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{3}$$

where μ_x and μ_y are the average of x and y (the original and reconstructed images), σ_x^2 and σ_y^2 are the variances of x and y, σ_{xy} is the covariance of x and y, and C_1 and C_2 are small constants to stabilize the division.

Acceptable Range:

- SSIM values typically range from 0 to 1.
- Higher values, closer to 1, indicate higher similarity and thus better quality.
- Values below 0.5 generally indicate significant quality degradation.

4) Normalized Cross-Correlation (NCC): Normalized Cross-Correlation (NCC) is a measure of the similarity between two signals [18]. It is widely used in image processing for tasks such as template matching and stereo vision. The NCC between two signals, f and g, can be computed as follows:

$$\text{NCC}(f,g) = \frac{\sum_{i,j}(f(i,j)-\bar{f})(g(i,j)-\bar{g})}{\sqrt{\sum_{i,j}(f(i,j)-\bar{f})^2 \sum_{i,j}(g(i,j)-\bar{g})^2}}$$

where:

- $f(i,j)$ and $g(i,j)$ are the values of the pixels at the position (i,j) in the images f and g, respectively.
- $\bar{f}$ and $\bar{g}$ are the mean values of the images f and g, respectively.

The NCC value ranges from -1 to 1, where:

- NCC = 1 indicates a perfect positive correlation between the images.
- NCC = -1 indicates a perfect negative correlation between the images.
- NCC = 0 indicates no correlation between the images.

The normalization in NCC makes it robust to variations in lighting and contrast, as it takes into account the mean and standard deviation of the images.

C. Results

We have optimized the l1-norm based reconstruction algorithm by mainly focusing on MSE,PSNR,SSIM and time taken for reconstruction. The parameters which we tweaked are:

- Number of iterations
- Transfer domains
- Block size and number
- Post-processing

1) Number of iterations: To find the optimal number of iterations, the algorithm was run for 1000 iterations and error vs iterations were plotted. The error is calculated as the sum of distance between each pixel in original image and reconstructed image. The optimal number of iterations was found to be 600.

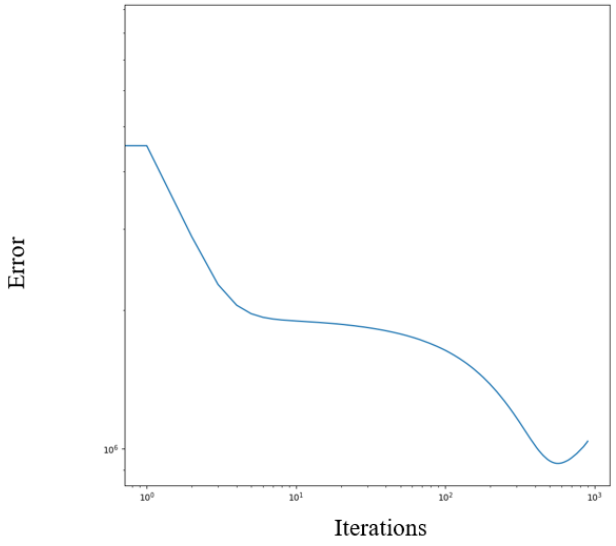

Fig. 2. Graph plotted between error and number of iterations .

The optimal number of iterations was found to be 600.

2) Transfer domain: The domains we chose for investigation were :

- Fourier Transform
- Discrete Cosine Transform
- Wavelet transform

The algorithm was run for compression rates 0.25,0.5,0.75 for 50 images with different domains. The results obtained:

TABLE I
COMPARISON OF COMPRESSION METRICS WITH TRANSFER DOMAINS

Compression Rate	Metrics	Fourier	Wavelet	DCT
0.25	MSE	3349.87	200.415	108.657
	PSNR	12.459	24.581	27.188
	SSIM	0.1549	0.687	0.7488
	Time taken (s)	17.4913	71.734	14.1677
0.5	MSE	2659.86	91.307	38.183
	PSNR	13.27	27.957	32.022
	SSIM	0.1607	0.8436	0.90127
	Time taken (s)	11.988	87.24	24.2426
0.75	MSE	1114.6	34.3928	13.259
	PSNR	16.931	32.2054	36.06
	SSIM	0.289	0.936	0.9589
	Time taken (s)	15.0246	77.125	13.7593

From the results DCT was found to be more effective.

3) Number of blocks: From using DCT and 600 iterations for 50 images, the algorithm was run using variable number of blocks.

TABLE II
COMPARISON OF COMPRESSION METRICS WITH NUMBER OF BLOCKS

Number of Blocks	MSE	PSNR	SSIM	Time taken
1	39.639	31.858	0.8788	57.624
4	39.1007	31.925	0.8883	47.736
16	38.399	31.995	0.8961	32.20281
64	38.183	32.0222	0.9012	24.24267
256	39.161	31.9147	0.9029	49.5709

From the results, the optimal number of blocks was found to be 64.

4) Post-processing: The main noticeable difference between the reconstructed image and the original image was that it lacked contrast. Thus contrasted was boosted in the post-processing.The results after it are:

TABLE III
AVERAGE VALUES OF METRICS AFTER POST-PROCESSING

Metrics	Average Values
MSE	159.63
PSNR (dB)	25.93
SSIM	0.8406
Time taken (s)	33.422

Fig. 3. Example output

D. Inference

- Generally increasing the number of iteration leads to higher reconstruction accuracy. This is because more iterations allow the algorithm to converge closely to the original sparse signal. While higher number of iterations are not only computation-wise expensive but also may result in formation of artifacts and the reconstructed image may start converging to also the noise in the data. This is reflected in the results.
- The DCT is effective for compressed sensing because it has strong energy compaction properties, meaning it can represent a signal with a relatively small number of significant coefficients. This allows the DCT to capture the essential features of a signal with fewer coefficients, making it highly suitable for compressed sensing where the goal is to reconstruct the signal accurately from a reduced number of samples.
- The main problem associated with the algorithm was the time taken. By separating the image into blocks and applying CS algorithm separately, overall size of computation reduce for each block. Also smaller blocks often contain less complex data, making them more sparse. Higher sparsity generally leads to faster and more efficient reconstruction. But as we increase the number of blocks the overall information in each may be lower thus it could take more time to converge. Thus a optimal number of 64 blocks was to chosen.
- Even though the image looked visually better after post-processing the metrics were little degraded.Thus post-processing could be used for compressing written documents which would benefit from being enhanced.

VII. FUTURE RECOMMENDATION

VIII. CONCLUSION

In conclusion, this paper investigated the trade-off between reconstruction accuracy, computational cost, and information preservation in compressed sensing algorithms. The findings demonstrate that increasing the number of iterations in the reconstruction process improves accuracy but can lead to artifacts and noise amplification. Additionally, dividing the signal into smaller blocks for compressed sensing can improve efficiency, especially for sparse data. However, a balance must be struck between block size and information content to ensure convergence. Finally, while post-processing can enhance the visual quality of reconstructed images, it may slightly decrease quantitative metrics. From the results obtained applying soft-thresholding in DCT domain while separating the image into 64 blocks provided best results. Overall, these results suggest that compressed sensing techniques offer a valuable approach for signal reconstruction, but careful consideration of parameters and potential drawbacks is necessary for optimal performance.

REFERENCES

[1] Y. C. Eldar and G. Kutyniok, Compressed Sensing: Theory and Applications. Cambridge, U.K.: Cambridge Univ. Press, 2012.

[2] C. E. Shannon, "Communication in the presence of noise," Proc. IRE, vol. 37, no. 1, pp. 10-21, Jan. 1949, doi: 10.1109/JRPROC.1949.232969.

[3] D. L. Donoho, "Compressed sensing," IEEE Trans. Inf. Theory, vol. 52, no. 4, pp. 1289-1306, Apr. 2006.

[4] L. Gan, "Block Compressed Sensing of Natural Images," in *Proc. 2007 15th Int. Conf. Digit. Signal Process.*, pp. 403-406, 2007.

[5] M. Lustig, D. L. Donoho, J. M. Santos, and J. M. Pauly, "Compressed Sensing MRI," IEEE Signal Process. Mag., vol. 25, no. 2, pp. 72-82, Mar. 2008, doi: 10.1109/MSP.2007.914728.

[6] Y. Li, G. Shi, X. Xie, and C. Chen, "Compressive modulation in digital communication," in Proc. 2013 IEEE Int. Symp. Circuits Syst. (ISCAS), Beijing, China, 2013, pp. 1966-1969, doi: 10.1109/ISCAS.2013.6572254.

[7] S. Leitner, H. Wang, and S. Tragoudas, "Design of scalable hardware-efficient compressive sensing image sensors," IEEE Sens. J., vol. 18, no. 2, pp. 641-651, Jan. 15, 2018, doi: 10.1109/JSEN.2017.2766040.

[8] Candes, Emmanuel, Romberg, Justin, and Tao, Terence. Stable Signal Recovery from Incomplete and Inaccurate Measurements. *arXiv preprint arXiv:math/0503066*, 2005. Primary class: Numerical Analysis.

[9] Candes, E.J., Romberg, J., and Tao, T. "Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information," *IEEE Transactions on Information Theory*, vol. 52, no. 2, pp. 489-509, 2006. doi: 10.1109/TIT.2005.862083.

[10] S. Ravindranath, S. R. N. Ram, S. Subhashini, A. V. S. Reddy, M. Janarth, R. AswathVignesh, R. Gandhiraj, and K. P. Soman, "Compressive sensing based image acquisition and reconstruction analysis," in Proc. 2014 Int. Conf. Green Comput. Commun. Electr. Eng. (ICGCCEE), pp. 1-6, 2014.

[11] Anupama, H., Shanthala, S., Mahadevaswamy, H.R., and Awasthi, Nitin. "Convex - Optimization for Reconstructing Compressed Signal using Orthogonal Matching Pursuit," in *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, 2018, pp. 554-557. doi: 10.1109/CESYS.2018.8724095.

[12] Vanderbei, Robert, Liu, Han, Wang, Lie, and Lin, Kevin. "Optimization for Compressed Sensing: the Simplex Method and Kronecker Sparsification," 2013.

[13] Do, Thong T., Gan, Lu, Nguyen, Nam H., and Tran, Trac D. "Fast and Efficient Compressive Sensing Using Structurally Random Matrices," *IEEE Transactions on Signal Processing*, vol. 60, no. 1, pp. 139-154, 2012. doi: 10.1109/TSP.2011.2170977.

[14] Lu Gan, "Block Compressed Sensing of Natural Images," in *2007 15th International Conference on Digital Signal Processing*, 2007, pp. 403-406. doi: 10.1109/ICDSP.2007.4288604.

[15] Oike, Yusuke, and Gamal, Abbas El. "A 256×256 CMOS image sensor with -based single-shot compressed sensing," in *2012 IEEE International Solid-State Circuits Conference*, 2012, pp. 386-388. doi: 10.1109/ISSCC.2012.6177057.

[16] Bertsimas, Dimitris, and Johnson, Nicholas A. G. "Compressed Sensing: A Discrete Optimization Approach," 2023. arXiv:2306.04647.

[17] P, G., Raj, T. S., Malathi, G., Nair, R. A., and Uma, S. "Compressive Sensing for Image Reconstruction: A Deep Neural Network Approach," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 9s, pp. 112-118, 2023.

[18] A. M. Eskicioglu and P. S. Fisher, "Image quality measures and their performance," *IEEE Transactions on Communications*, vol. 43, no. 12, pp. 2959-2965, 1995, doi: 10.1109/26.477498.

[19] A. Horé and D. Ziou, "Image quality metrics: PSNR vs. SSIM," in *Proceedings of the 20th International Conference on Pattern Recognition (ICPR)*, Aug. 2010, pp. 2366-2369, doi: 10.1109/ICPR.2010.579. url

[20] University of Granada Computer Vision Group, "Granada Computer Vision Group - Dataset," 2024. [Online]. Available: https://ccia.ugr.es/cvg/CG/base.htm. [Accessed: 2024-04-23].

APPENDIX: IMAGE RECONSTRUCTION CODE

```
import tkinter as tk
from tkinter import filedialog, ttk
from PIL import Image, ImageTk
import pywt
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import dct, idct
from sklearn.metrics import mean_squared_error
from skimage.metrics
import structural_similarity as ssim
import pandas as pd
import threading
import queue

# Define necessary functions
def undersampler(undersample_rate, original1):
    n = original1.shape[0] * original1.shape[1]
    original_undersampled = (original1.reshape(-1) * np.random.permutation(
        np.concatenate((np.ones(int(n * undersample_rate)),
        np.zeros(int(n * (1-undersample_rate))))))
    ).reshape(original1.shape)
    return original_undersampled

def flat_wavelet_transform2(x, method='bior1.3'):
    coeffs = pywt.wavedec2(x, method)
    output = coeffs[0].reshape(-1)
    for tups in coeffs[1:]:
        for c in tups:
            output = np.concatenate((output, c.reshape(-1)))
    return output

def inverse_flat_wavelet_transform2(X, shape, method='bior1.3'):
    shapes = pywt.wavedecn_shapes(shape, method)
    nx = shapes[0][0]
    ny = shapes[0][1]
    n = nx * ny
    coeffs = [X[:n].reshape(nx, ny)]
    for i, d in enumerate(shapes[1:]):
        vals = list(d.values())
        nx = vals[0][0]
        ny = vals[0][1]
        coeffs.append((X[n: n + nx * ny].reshape(nx, ny),
                       X[n + nx * ny: n + 2 * nx * ny].reshape(nx, ny),
                       X[n + 2 * nx * ny: n + 3 * nx * ny].reshape(nx, ny)))
        n += 3 * nx * ny
    return pywt.waverec2(coeffs, method)

def soft_thresh(x, lam):
    if not isinstance(x[0], complex):
        return np.zeros(x.shape) + (x + lam) * (x < -lam) + (x - lam) * (x > lam)
    else:
        return np.zeros(x.shape) + (abs(x) - lam) / abs(x) * x * (abs(x) > lam)

def dct2(a):
    return dct(dct(a.T, norm='ortho').T, norm='ortho')

def idct2(a):
    return idct(idct(a.T, norm='ortho').T, norm='ortho')

def distance(x, y):
    return sum(abs(x.reshape(-1) - y.reshape(-1)))

def reconstructor(ind ,lam, y, domain, original, epoch,output_queue):
    eps = 1e-2
    lam_decay = 0.995
    minlam = 1
    err2 = []

    xhat = y.copy()
    if domain == "wavelet":
        for i in range(epoch):
            method = 'haar'
            xhat_old = xhat
            Xhat_old = flat_wavelet_transform2(xhat, method)

            Xhat = soft_thresh(Xhat_old, lam)
            xhat = inverse_flat_wavelet_transform2(Xhat, (128,128), method)

            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    elif domain == "fourier":
        for i in range(epoch):
            xhat_old = xhat
            Xhat_old = np.fft.fft2(xhat)
            Xhat = soft_thresh(Xhat_old, lam)
            xhat = np.fft.ifft2(Xhat).real
            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    elif domain == "dct":
        for i in range(epoch):
            xhat_old = xhat
            Xhat_old = dct2(xhat)
            Xhat = soft_thresh(Xhat_old, lam)
            xhat = idct2(Xhat)
            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    xhat.reshape(original.shape)

    output_queue.put((ind,xhat,err2))
    return

def reconstructor1(lam, y, domain, original, epoch,output_queue):
    eps = 1e-2
    lam_decay = 0.995
    minlam = 1
    err2 = []

    xhat = y.copy()
    if domain == "wavelet":
        for i in range(epoch):
            method = 'haar'
            xhat_old = xhat
            Xhat_old = flat_wavelet_transform2(xhat, method)
            Xhat = soft_thresh(Xhat_old, lam)
            xhat = inverse_flat_wavelet_transform2(Xhat, (1024, 1024), method)
            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    elif domain == "fourier":
        for i in range(epoch):
            xhat_old = xhat
            Xhat_old = np.fft.fft2(xhat)
            Xhat = soft_thresh(Xhat_old, lam)
            xhat = np.fft.ifft2(Xhat).real
            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    elif domain == "dct":
        for i in range(epoch):
            xhat_old = xhat
            Xhat_old = dct2(xhat)
            Xhat = soft_thresh(Xhat_old, lam)
            xhat = idct2(Xhat)
            xhat[y != 0] = y[y != 0]

            xhat = xhat.astype(int)
            xhat[xhat < 0] = 0
            xhat[xhat > 255] = 255
            err2.append(distance(original, xhat))
            lam *= lam_decay
    xhat.reshape(original.shape)

    return xhat, err2

def calculate_ncc(original, reconstructed):
    # Ensure both original and reconstructed images are float64 for precision
    original = original.astype(np.float64)
    reconstructed = reconstructed.astype(np.float64)

    # Calculate mean of original and reconstructed images
    mean_original = np.mean(original)
    mean_reconstructed = np.mean(reconstructed)

    # Calculate cross-correlation term
    cross_corr = np.sum((original - mean_original) * (reconstructed - mean_reconstructed))

    # Calculate RMS (Root Mean Square) term
```

```
    rms_original = np.sqrt(np.sum((original - mean_original)**2))
    rms_reconstructed = np.sqrt(np.sum((reconstructed - mean_reconstructed)**2))

    # Calculate NCC (Normalized Cross-Correlation)
    ncc_value = cross_corr / (rms_original * rms_reconstructed)

    return ncc_value

def calculate_metrics(original, reconstructed):
    mse_value = mean_squared_error(original, reconstructed)
    psnr_value = 20 * np.log10(np.max(original) / np.sqrt(mse_value))
    data_range = original.max() - original.min()
    ssim_value, _ = ssim(original, reconstructed, data_range=data_range, full=True)
    ncc_value = calculate_ncc(original, reconstructed)
    return mse_value, psnr_value, ssim_value, ncc_value

def update_metrics_table(metrics_df):
    for i in metrics_table.get_children():
        metrics_table.delete(i)
    for _, row in metrics_df.iterrows():
        metrics_table.insert('', 'end', values=list(row))

# GUI Functions
def load_image():
    global original_image,file_path
    file_path = filedialog.askopenfilename()
    if file_path:
        image = Image.open(file_path)
        image = image.convert('L')
        image = image.resize((1024,1024), Image.LANCZOS)
        original_image = np.array(image, dtype=np.float32)

        img_tk = ImageTk.PhotoImage(image)
        original_img_label.configure(image=img_tk)
        original_img_label.image = img_tk


def perform_undersampling():
    global undersampled_image
    try:
        compression_ratio = float(compression_ratio_entry.get())
    except ValueError:
        tk.messagebox.showerror("Invalid Input",
        "Please enter a valid compression ratio.")
        return

    undersampled_image = undersampler(compression_ratio, original_image)

    undersampled_image_pil = Image.fromarray(undersampled_image).convert('L')
    img_tk = ImageTk.PhotoImage(undersampled_image_pil)
    undersampled_img_label.configure(image=img_tk)
    undersampled_img_label.image = img_tk
def split_image(image):
    height, width = image.shape[:2]
    part_height, part_width = height // 8, width // 8
    parts = []
    for i in range(8):
        for j in range(8):
            parts.append(image[i * part_height:(i + 1) *
            part_height, j * part_width:(j + 1) * part_width])
    return parts


# Define a function to combine four parts into a single image
def combine_parts(parts):
    rows = []
    for i in range(8):
        row = np.concatenate(parts[i * 8:(i + 1) * 8], axis=1)
        rows.append(row)
    combined_image = np.concatenate(rows, axis=0)
    return combined_image
import os
from tkinter import messagebox
def save_reconstructed_image():
    if reconstructed_image is not None:
        # Get the original file name and add "_CS" before the file extension
        file_path = filedialog.asksaveasfilename(defaultextension=".png",
        filetypes=[("PNG files", "*.png"), ("All files", "*.*")],
        if file_path:
            Image.fromarray(reconstructed_image).convert('L').save(file_path)
            messagebox.showinfo("Image Saved",
            f"Reconstructed image saved as {file_path}")
    else:
        messagebox.showerror("Error", "No reconstructed image to save.")



import time
# Initialize an empty list to store the time taken for each divided image
divided_image_times = []
# Create and start threads
threads = []

# Create a queue to store the output from threads
output_queue = queue.Queue()


def perform_reconstruction():
    global reconstructed_image, reconstructed_image_full
    method = method_var.get()
    lam = 100  # initial lambda value
    epoch = 600  # number of epochs
    undersampled_parts = split_image(undersampled_image)

    # Split original image into four parts
    original_parts = split_image(original_image)

    # Initialize an empty list to store reconstructed parts
    reconstructed_parts = [None]*64

    # Iterate over each pair of undersampled and original parts
    start_time = time.time()
    for i in range(64):
        thread = threading.Thread(target=reconstructor, args=(i,lam,undersampled_parts[i], method,
        original_parts[i], epoch,output_queue))
        thread.start()
        threads.append(thread)

    # Join all threads to wait for them to finish
    for thread in threads:
        thread.join()
    while not output_queue.empty():
        co,a, _ = output_queue.get()
        reconstructed_parts[co] = a
    # Combine reconstructed parts into a single image
    reconstructed_image = combine_parts(reconstructed_parts)
    end_time = time.time()  # Record end time
    # Calculate time taken
    time_taken = end_time - start_time
    print("Time taken for a divided image:", time_taken, "seconds")

    # Convert reconstructed image to PIL format
    reconstructed_image_pil = Image.fromarray(reconstructed_image).convert('L')
    # Display the reconstructed image
    img_tk = ImageTk.PhotoImage(reconstructed_image_pil)
    reconstructed_img_label.configure(image=img_tk)
    reconstructed_img_label.image = img_tk

    start_time_full = time.time()  # Record start time
    # Full image reconstruction
    reconstructed_image_full, _ = reconstructor1(lam, undersampled_image, method
    , original_image, epoch,output_queue)
    end_time_full = time.time()  # Record end time
    # Calculate time taken for full image reconstruction
    time_taken_full = end_time_full - start_time_full
    print("Time taken for the full image reconstruction:", time_taken_full, "seconds")



    # Calculate metrics for the entire reconstructed image
    # Calculate metrics for the entire reconstructed image
    mse_segmented, psnr_segmented, ssim_segmented, ncc_segmented =
        calculate_metrics(original_image, reconstructed_image)
    mse_full, psnr_full, ssim_full, ncc_full =
        calculate_metrics(original_image, reconstructed_image_full)

    # Create a DataFrame to store metrics for segmented and full image reconstructions
    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PSNR', 'SSIM', 'NCC'],
        'Segmented Image': [mse_segmented, psnr_segmented, ssim_segmented, ncc_segmented],
        'Full Image': [mse_full, psnr_full, ssim_full, ncc_full]
    })

    # Print the comparison table
    print("Metrics Comparison:")
    print(metrics_df)

    # Update metrics table
    update_metrics_table(metrics_df)

# Create the main window
root = tk.Tk()
root.title("Image Reconstruction GUI")
screen_width = root.winfo_screenwidth()
screen_height = root.winfo_screenheight()

# Calculate the x and y coordinates for the Tk root window
x_coordinate = (screen_width / 2) - (1600 / 2)  # Assuming the width of the window is 1600
y_coordinate = (screen_height / 2) - (1000 / 2)  # Assuming the height of the window is 1000

# Set the geometry of the root window to center it on the screen
root.geometry(f"1600x1000+{int(x_coordinate)}+{int(y_coordinate)}")
root.configure(bg="#333333")

# Create and place widgets
load_img_button = tk.Button(root, text="Load Image", command=load_image,bg="blue", fg="white")
load_img_button.grid(row=0, column=0, padx=10, pady=10)

method_var = tk.StringVar(value="reconstructed_CS.png")
method_var.set("dct")
method_frame = tk.Frame(root, bg="blue")
method_frame.grid(row=0, column=2, padx=10, pady=10)

# Create the OptionMenu inside the frame
method_menu = tk.OptionMenu(method_frame, method_var, "wavelet", "fourier", "dct")
method_menu.config(bg="blue",fg='white')  # Set background color of the dropdown menu
method_menu.grid(row=0, column=0, padx=10, pady=10)

compression_ratio_label = tk.Label(root, text="Compression Ratio:",bg="blue", fg="white")
compression_ratio_label.grid(row=0, column=4, padx=10, pady=10)
compression_ratio_entry = tk.Entry(root)
compression_ratio_entry.grid(row=0, column=4, padx=10, pady=10)
compression_ratio_entry.insert(0, "0.5")

def draw_photo_frame():
    # Create a frame for photo display
    photo_frame = tk.Frame(root, bg="white", width=1024, height=1024)
    photo_frame.grid(row=1, column=1, columnspan=3, padx=10, pady=10)
    photo_frame1 = tk.Frame(root, bg="white", width=1024, height=1024)
    photo_frame1.grid(row=1, column=3, columnspan=3, padx=10, pady=10)
    photo_frame3 = tk.Frame(root, bg="white", width=1024, height=1024)
    photo_frame3.grid(row=1, column=5, columnspan=3, padx=10, pady=10)
# Call the function to draw the frame initially
draw_photo_frame()

original_img_label = tk.Label(root)
original_img_label.grid(row=1, column=2, padx=10, pady=10)

undersampled_img_label = tk.Label(root)
undersampled_img_label.grid(row=1, column=4, padx=10, pady=10)

reconstructed_img_label = tk.Label(root)
```

```
reconstructed_img_label.grid(row=1, column=6, padx=10, pady=10)

undersample_button = tk.Button(root, text="Perform Undersampling", \
command=perform_undersampling,bg="blue", fg="white")
undersample_button.grid(row=2, column=2, padx=10, pady=10)

reconstruct_button = tk.Button(root, text="Perform Reconstruction",
command=perform_reconstruction,bg="blue", fg="white")
reconstruct_button.grid(row=2, column=4, padx=10, pady=10)

save_button = tk.Button(root, text="Save Reconstructed Image",
command=save_reconstructed_image, bg="blue", fg="white")
save_button.grid(row=2, column=6, padx=10, pady=10)


metrics_table = ttk.Treeview(root, columns=('Metric',
'Value'), show='headings', height=10)
metrics_table.heading('Metric', text='Metric')
metrics_table.heading('Value', text='Value')
metrics_table.column('Metric', width=400)
metrics_table.column('Value', width=400)
style = ttk.Style()
style.configure("Treeview.Heading", font=("Arial", 16))
style.configure("Treeview", font=("Arial",14), rowheight=120)
metrics_table.grid(row=3, column=2, columnspan=3, padx=10, pady=10)

# Start the main loop
root.mainloop()
```