RADAR SIGNAL PROCESSING USING COMPRESSED SENSING

Internship Report

AUTHORS

Abhinav M Balakrishnan

Arun Ramesh

INSTITUTE

School of Engineering, CUSAT

ACKNOWLEDGEMENT

This acknowledgment is a testament to the intensive drive and technical competence of many individuals who have contributed to the success of our project.

Special thanks to Shri B.S. Teza, Scientist 'E', ASL DRDO, for not only selecting us for this internship, but also for his consistent guidance, encouragement, and valuable insights throughout the course of our project.

We also thank our professors for their constant support and inspiration. Special thanks to our HoD, Dr. Deepa Shankar and our class co-ordinator and department faculty, Dr. Mridula S.

ABSTRACT

This project explores the application of compressed sensing techniques to radar signal processing, aiming to overcome the limitations imposed by traditional Nyquist sampling. It also presents the theoretical foundations of compressed sensing, details key reconstruction algorithms such as Orthogonal Matching Pursuit (OMP), Iterative Shrinkage Thresholding Algorithm (ISTA), and Coordinate Descent (CoD), and evaluates their performance through simulations and Monte Carlo trials. The results demonstrate the effectiveness and trade-offs of these algorithms in both noiseless and noisy environments, highlighting their potential for efficient and robust radar signal acquisition and processing. The project also explores the future possibilities and modifications for these algorithms for executing real time RADAR processing.

The GitHub repository for all the code and results can be accessed: CS Algorithms

TABLE OF CONTENTS

CHAPTER 1: NYQUIST CRITERIA AND ITS LIMITATIONS

1.1 INTRODUCTION

The Nyquist–Shannon sampling theorem is a theorem in the field of signal processing which serves as a fundamental bridge between continuous-time signals and discrete-time signals. It establishes a sufficient condition for a sample rate that permits a discrete sequence of samples to capture all the information from a continuous-time signal of finite bandwidth.

For a signal of frequency f_{signal}, the minimum sampling rate required to avoid aliasing, according to the Nyquist criterion is,

Nyquist-Shannon Sampling Criteria

$$f_s \geq 2f_{\text{signal}} \tag{1}$$

This means that the sampling frequency must be at least twice the highest frequency present in the signal to ensure perfect reconstruction from its samples.

1.2 LIMITATIONS

One of the main limitations of the Nyquist sampling theorem is the requirement for high sampling rates when dealing with signals that contain high-frequency components, which can be challenging to achieve in practice due to several reasons:

- **Hardware Limitations:** Analog-to-digital converters (ADCs), capable of very high sampling rates are expensive and may not be readily available. The speed and resolution of ADCs are often limited by current technology.
- **Data Storage and Processing:** High sampling rates generate large volumes of data, which require significant storage capacity and processing power. This can make real-time processing and analysis difficult or costly.
- **Power Consumption:** Systems operating at high sampling rates typically consume more power, which is a critical concern in portable or battery-powered devices.
- **Noise Sensitivity:** At higher frequencies, electronic components are more susceptible to noise and interference, which can degrade the quality of the sampled signal.

These limitations motivate the development of alternative sampling techniques, such as **Compressed Sensing**, which aim to reconstruct signals accurately from fewer samples than required by the traditional Nyquist criterion, especially when the signal is sparse or compressible in some domain.

CHAPTER 2: COMPRESSED SENSING

2.1 INTRODUCTION

The limitations of the Nyquist criterion, especially in applications requiring high data rates or operating under hardware constraints, have led to the exploration of new signal acquisition paradigms. Compressed Sensing (CS) is one such approach that leverages the sparsity of signals in some domain to enable accurate reconstruction from far fewer samples than traditionally required.

2.2 MOTIVATIONS FOR COMPRESSED SENSING

Key motivations for using compressed sensing include:

- **Efficient Data Acquisition:** CS allows for the collection of only the most informative measurements, reducing the burden on data acquisition systems.
- **Reduced Storage and Transmission Costs:** By acquiring fewer samples, CS minimizes the amount of data that needs to be stored or transmitted, which is particularly beneficial in bandwidth-limited or remote sensing scenarios.
- **Lower Power Consumption:** Fewer samples mean less processing and lower power requirements, which is advantageous for battery-powered and embedded systems.
- **Enabling New Applications:** CS opens up possibilities for applications where traditional sampling is impractical, such as medical imaging, wireless communications, and radar signal processing.

In the following chapters, we explore the principles of compressed sensing and its application to radar signal processing.

2.3 FUNDAMENTAL TERMS

Before delving deeper into compressed sensing, it is important to understand some fundamental terms:

- **Sparsity:** A signal is said to be sparse if most of its coefficients are zero or close to zero. Sparsity is a key assumption in compressed sensing.

- **Basis:** In compressed sensing, a basis is a set of vectors (such as Fourier, wavelet, or DCT bases) in which the signal can be represented as a linear combination. The choice of basis is crucial, as it determines the sparsity and thus the effectiveness of compressed sensing for a given signal. It is also called the **dictionary matrix**.
- **Measurement Matrix:** In compressed sensing, the measurement matrix is used to acquire linear projections of the original signal. It is also known as the dictionary matrix or sampling matrix.
- **Reconstruction Algorithm:** Algorithms such as Basis Pursuit, Orthogonal Matching Pursuit (OMP), and LASSO are used to recover the original sparse signal from the compressed measurements.

Understanding these terms is essential for grasping the principles and practical implementation of compressed sensing.

2.4 MATHEMATICAL MODEL

In compressed sensing, the measurement process can be mathematically modeled as:

$$\mathbf{y} = \phi\mathbf{x} \tag{2}$$

where:

- $\mathbf{x} \in \mathbb{R}^n$ is the **original signal** (which is assumed to be sparse or compressible in some basis)
- $\phi \in \mathbb{R}^{m \times n}$ is the **measurement matrix** (with $m < n$)
- $\mathbf{y} \in \mathbb{R}^m$ is the **compressed (measurement) vector**.

If the signal $\mathbf{x}$ is not sparse in its original domain but is sparse in some transform domain (e.g., DCT, DFT, or wavelet), we can write $\mathbf{x} = \Psi\mathbf{s}$, where Ψ is the **basis matrix** and $\mathbf{s}$ is the **sparse coefficient vector**. The measurement model then becomes:

$$\mathbf{y} = \phi\Psi\mathbf{s} = \Theta\mathbf{s} \tag{3}$$

where $\Theta = \phi\Psi$ is the **sensing matrix**.

The goal of compressed sensing is to recover $\mathbf{x}$ (or $\mathbf{s}$) from the measurements $\mathbf{y}$, given knowledge of ϕ (and Ψ if applicable), by exploiting the sparsity of the signal.

CHAPTER 3: RECONSTRUCTION ALGORITHMS

The various algorithms are used for reconstructing back the original signal that was initially compressed by the process as shown previously.

3.1 ORTHOGONAL MATCHING PURSUIT (OMP)

The OMP algorithm is an iterative greedy algorithm used to recover sparse signals from compressed measurements. At each iteration, it selects the column of the measurement matrix that is most correlated with the current residual and updates the solution accordingly. The process continues until a sufficiently small residual is met. The steps are listed below, as shown below

Algorithm 3: OMP($\mathbf{A}, \mathbf{b}$)

Input: $\mathbf{A}, \mathbf{b}$

Result: $\mathbf{x}_k$

Initialization $\mathbf{r}_0 = \mathbf{b}$, $\Lambda_0 = \varnothing$;

\- Normalize all columns of $\mathbf{A}$ to unit L_2 norm;

\- Remove duplicated columns in $\mathbf{A}$ (make $\mathbf{A}$ full rank);

for $k = 1, 2, \ldots$ **do**

  Step-1-2. $\Lambda_k = \Lambda_{k-1} \cup \left\{ \underset{j \notin \Lambda_{k-1}}{\operatorname{argmax}} \left| \mathbf{A}^\top \mathbf{r}_{k-1} \right| \right\}$;

  Step-3. $\mathbf{x}_k(i \in \Lambda_k) = \underset{\mathbf{x}}{\operatorname{argmin}} \|\mathbf{A}_{\Lambda_k}\mathbf{x} - \mathbf{b}\|_2, \quad \mathbf{x}_k(i \notin \Lambda_k) = 0$;

  Step-4-5. $\mathbf{r}_k \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_k$;

end

Figure 1: OMP Algorithm[1]

This algorithm can be implemented in MATLAB and Python with necessary toolboxes and libraries.

3.1.1 Algorithm Implementation

- **MATLAB**

Here, The code is divided into two main parts: the OMP function definition and the signal generation/reconstruction workflow. The omp function implements the OMP algorithm. It takes as input a measurement matrix A, a measurement vector b, and the sparsity level K. The function normalizes the columns of A and initializes the residual r as the measurement vector. It iteratively selects the column of A most correlated with the current residual,

adds its index to the support set Lambda, and solves a least-squares problem to update the estimated sparse vector x. The residual is updated accordingly. This process repeats for K iterations, corresponding to the assumed sparsity of the original signal. For ideal omp implementation, n,m,k are defined and a random gaussian sensing matrix is created. then a sparse signal is created in frequency domain and is multiplied with the sensing matrix to create the measurements(output). Now, the omp algorithm is used to recover the frequency domain signal back and the rreconstructed and original signals are plotted together to check for errors. For noisy implementation, an additional gaussian noise is generated and added to the output measurement vector and this noisy output is given to the omp algorithm. Then a sum of random time domain sinusoidal input was given with Ψ as a random gaussian matrix and ϕ as a inbuilt function for dft, dftmtx(). dft was taken instead of idft since the omp was calculated in the frequency domain. the ouput was then converted to time domain using ifft() and was plotted.

- **Python**

  Libraries like **numpy** and **matplotlib** are imported for mathematical operations and plotting results respectively.

  - **Stage 1:** The basic implementation was done by taking length of signal (n), number of measurements (m) and non-zero values or sparsity (k) as input. The sensing matrix was assumed to be filled with random gaussian values.
  - **Stage 2:** The next stage involved taking a sum of three sinusoidal signals as input signal (k = 3) and it is converted to a more sparser domain with **Discrete Cosine Transform (DCT)**. The function is used by importing the **scipy** library. While initially k was fixed, it is then taken as an input from user. DCT was initially tested for a single sine wave as well as for sum of sine waves of different frequencies, as shown in the figure below.
  - **Stage 3:** In the above stages, reconstruction was observed for pure signals. So, a noise (in dB) was introduced before the reconstruction process.

All these stages were plotted and the error was calculated and observed.

3.1.2 Monte-Carlo Trials

Monte Carlo trials are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The is used to understand the behaviour of the algorithm

under change in parameters, including sparsity, noise and number of measurements taken. It is useful for analysis, understanding its performance for various values of multiple inputs. In other words, this acts like a testbench for the algorithm.

The input values were stored in a list and these were fed to the trial algorithm. The error was calculated for a number of trials for the same input values, and only the average error is plotted to prevent unwanted variations in reconstruction.

3.1.3 Observations & Results

- **MATLAB**

The algorithm was initially tested directly in frequncy domain. In its ideal form(ie. without noise), for a low enough sparsity, the algorithm perfectly reconstructed the frequency and the amplitude values of compressed signal. Then, two values of noise was given(SNR=0dB and SNR=20dB). The Algorithm was able to reconstruct the signal near-perfectly for an SNR of 20 dB. For an SNR of 0 dB(signal power=noise power), the results were more inaccurate, both in terms of position on the graph(frequency) and the amplitude values.The algorithm was able to reconstruct some parts of the signal with a fair amount of accuracy.

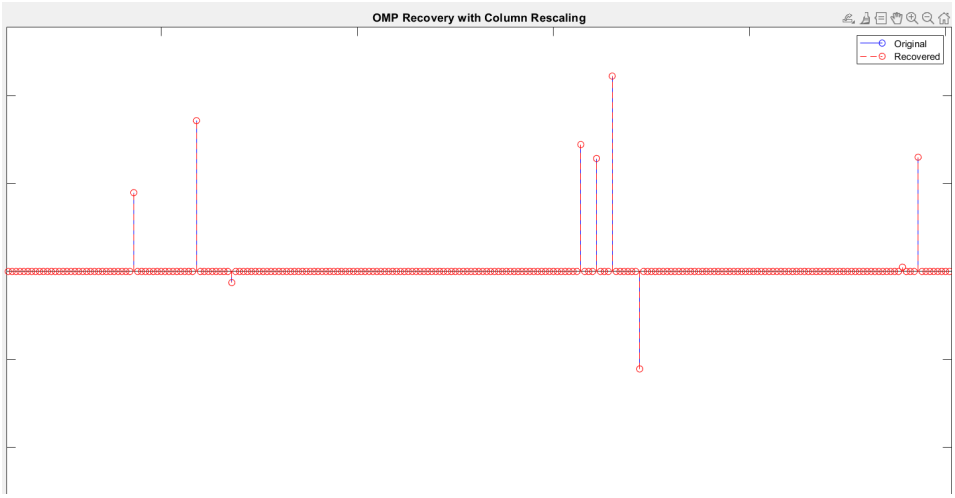

Figure 2: OMP Signal Reconstruction:Ideal (No noise added)

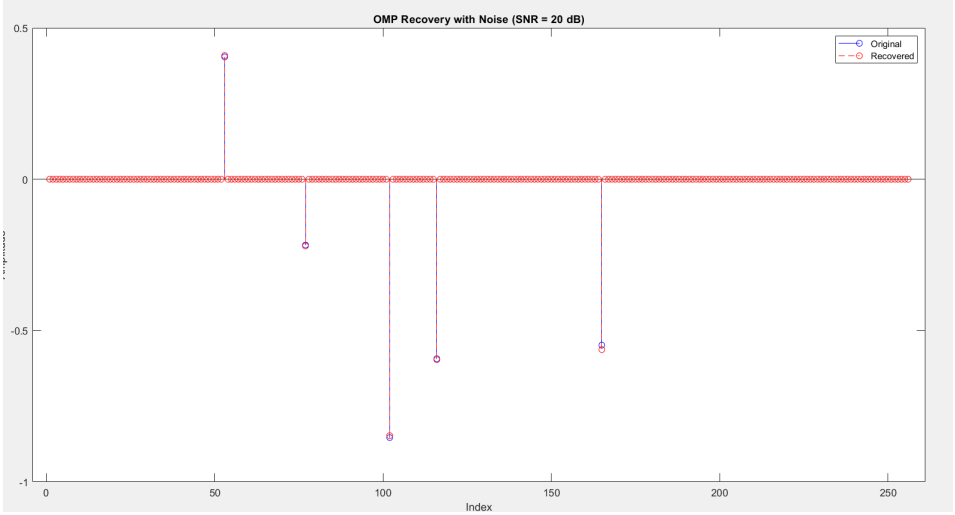

Figure 3: OMP Signal Reconstruction: 20dB

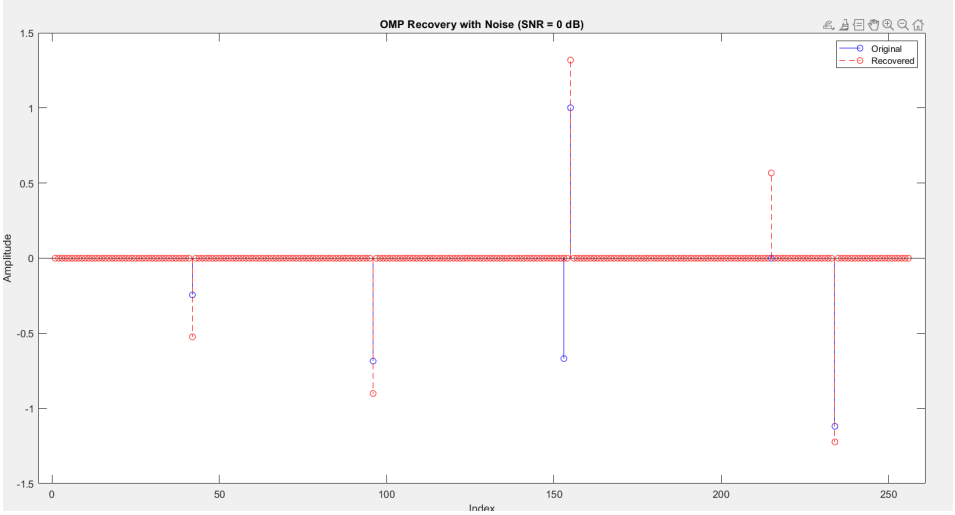

Figure 4: OMP Signal Reconstruction: 0dB

Next, the code was used to implement sinusoids in time domain. A sum of 5 real sinusoids was given as input to the algorithm. Then the output is plotted along with the original signal to compare them. Initially, the algorithm gave an output which had its amplitude greatly decreased w.r.t the original signal.

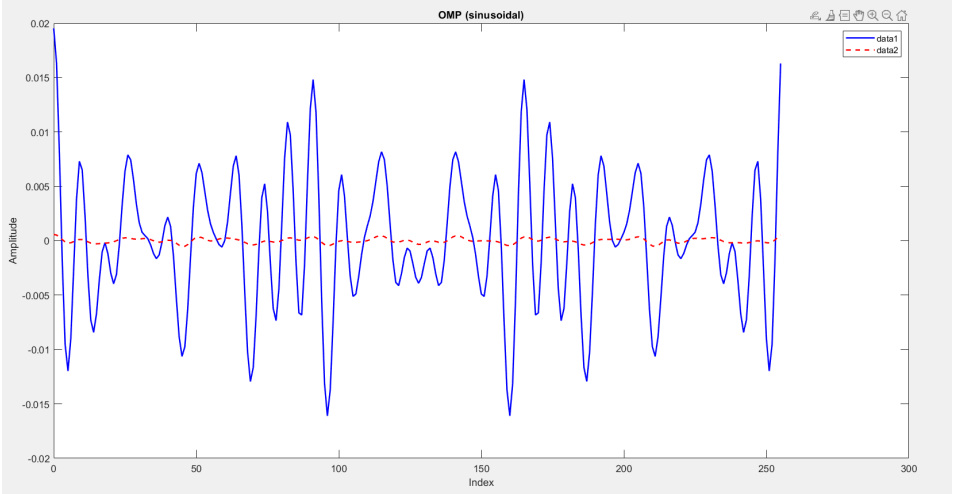

Figure 5: OMP Signal Reconstruction: 0dB

Further testing of the code showed the importance of normalising the theta matrix. The normalised theta matrix showed the correct output. After this was resolved, the algorithm was able to reconstruct the signal fairly accurately. Smaller peaks of the input signal was harder to reconstruct for the algorithm, and also, there was a reduction in the amplitude of the reconstructed signal w.r.t the original signal.A slight phase shift was observed in some outputs when the code is run for different random inputs.

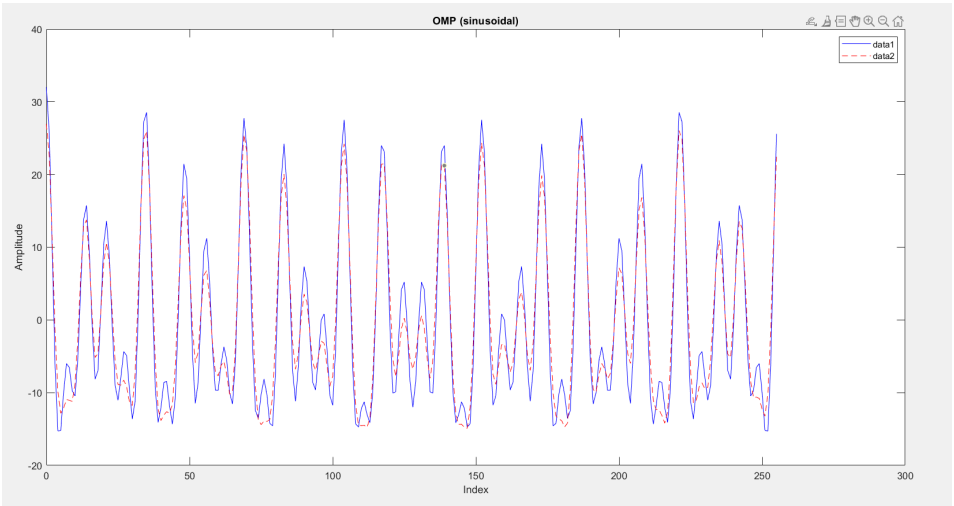

Figure 6: OMP Signal Reconstruction: sinusoidal input

- **Python**

For **Stage 1** implementation, the sparse matrix is already created by specifying k. So, the compressed matrix (y) is generated by just multiplying sensing matrix (Θ) and the generated sparse matrix (s). The results are plotted as shown below,

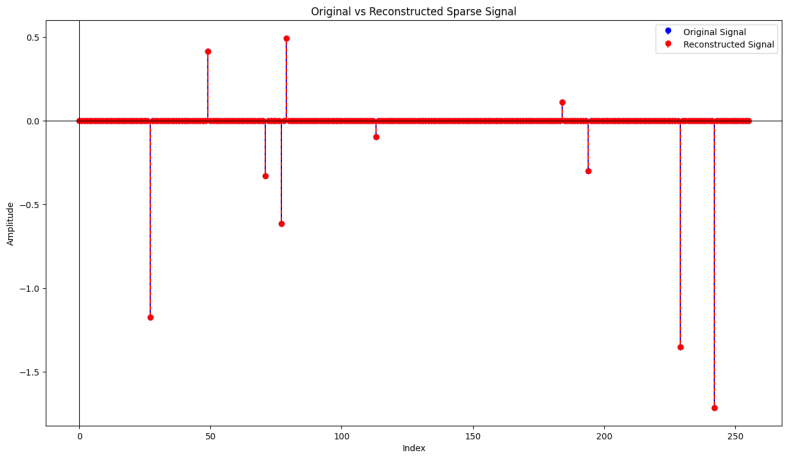

Figure 7: OMP Algorithm Stage 1 Implementation: Perfect Reconstruction

While the reconstruction as shown above is very accurate, it is not always the case. As sparsity increases, the measurements to be taken also increases. Hence, there are some necessary conditions for perfect recovery of a signal. As mentioned in [3], the relation between n, m and k is:-

$$\boxed{m \geq C \cdot k \cdot \log\left(\frac{n}{k}\right)} \tag{4}$$

where **C** is a constant almost equal to 2.

Hence, if the above equation is not satisfied, then reconstruction is very difficult. The failed reconstruction is shown below.

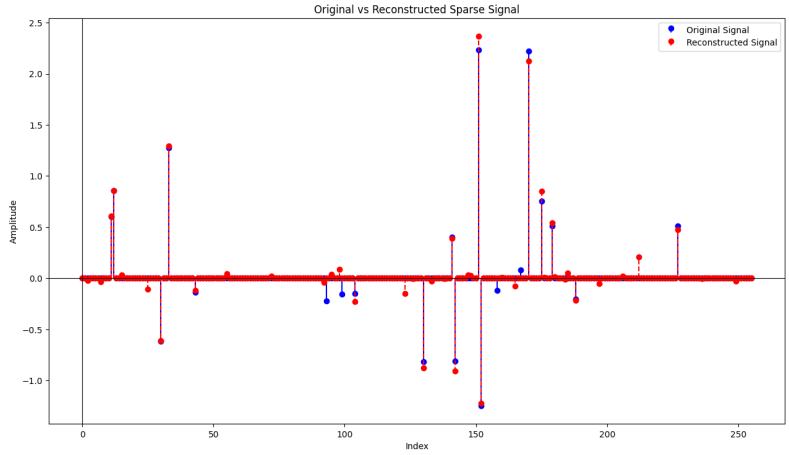

Figure 8: OMP Algorithm Stage 1 Implementation: Failed Reconstruction

When it comes to **Stage 2** implementation, the sensing matrix is divided into a basis matrix and measurement matrix. The basis matrix is used to convert our input signal to a sparser signal. For sinusoidal inputs, it is best to represent the signals in its frequency domain. So, FFT or DCT can be used. Since, all sinusoids are real signals, DCT was possible. The sum of sinusoids were converted to DCT and the results are being plotted to check its sparsity.

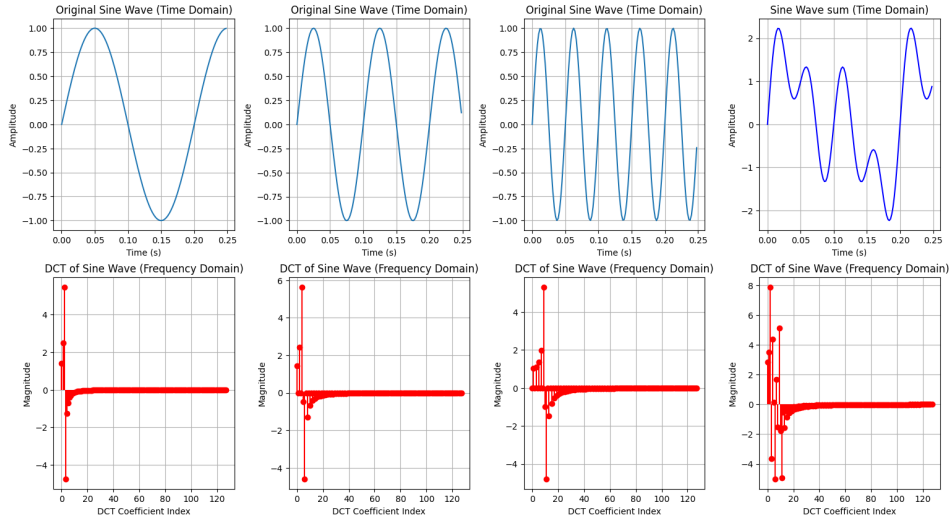

Figure 9: OMP Algorithm Stage 2 Implementation: DCT Basis on Sinusoidal signals

The sinusoidal signal is initially tested for various values of n and m, keeping k = 3. Some of the results are plotted as shown,

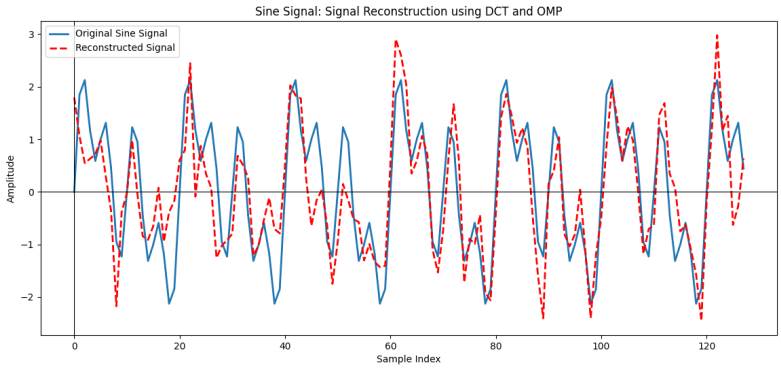

Figure 10: OMP Algorithm Stage 2 Implementation: For n = 128, m = 60

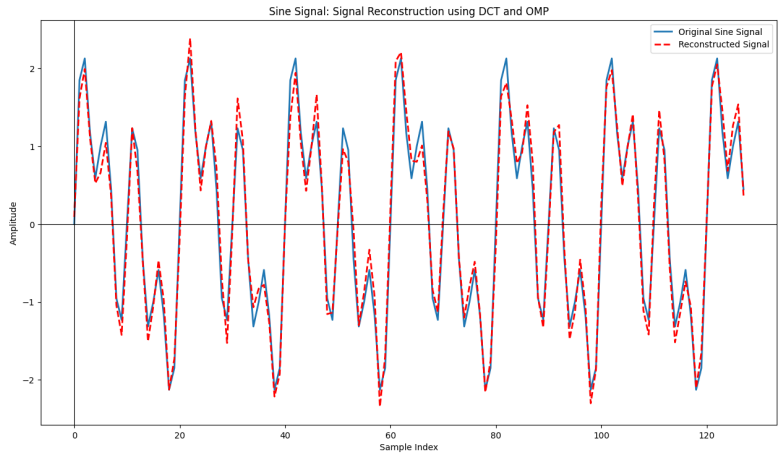

Figure 11: OMP Algorithm Stage 2 Implementation: For n = 128, m = 100

So, generally we can say as **number of measurements increases, the reconstruction error decreases**. Till now, no noise has been considered during the reconstruction. To analyse the algorithm for each value of n, m, k and even noise, it is difficult for us to understand the trend of error. So, a Monte Carlo trial has been implemented on the OMP algorithm for three variable parameters, **measurements**, **sparsity** and **noise**. So, all three parameters are compared and the results are plotted.

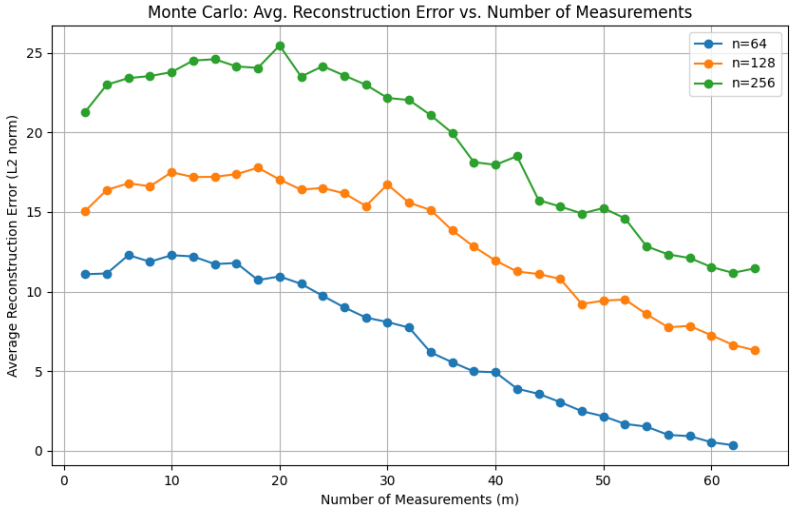

Figure 12: Monte Carlo Trial: Measurements (m)

The analysis above is for noiseless, fixed sparsity (k = 3) reconstruction.

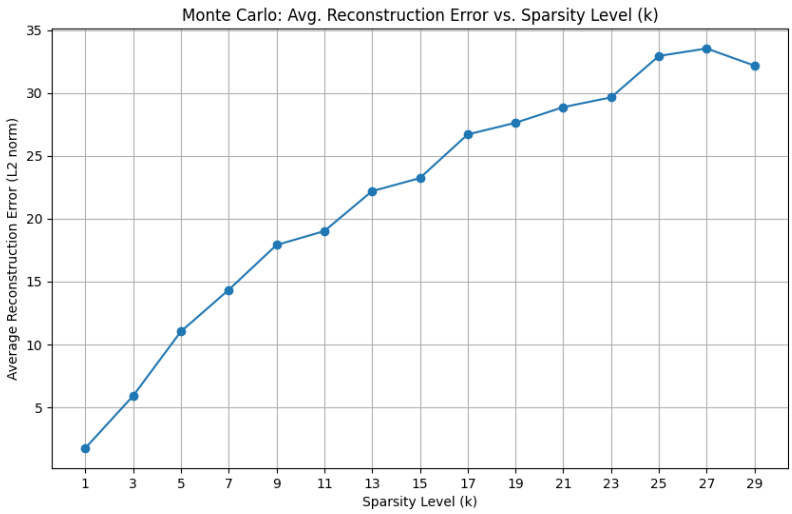

Figure 13: Monte Carlo Trial: Sparsity (k)

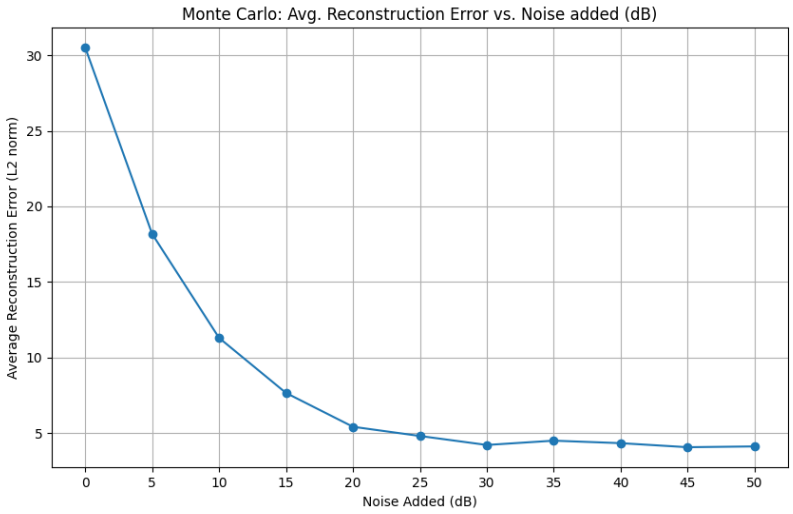

Figure 14: Monte Carlo Trial: Noise (in dB)

In summary, OMP is a robust and efficient algorithm for compressed sensing when the signal is sparse and the measurement conditions are favorable. However, its sensitivity to noise and the need for sufficient measurements must be considered in practical applications.

3.2 ITERATIVE SHRINKAGE THRESHOLDING ALGORITHM (ISTA)

The ISTA is an iterative, convex optimisation method for solving sparse signal recovery problems, particularly those formulated as LASSO or basis pursuit denoising. ISTA iteratively updates the solution by applying a gradient descent step followed by a soft-thresholding (shrinkage) operation to promote sparsity. The general steps are:

1. Initialize the sparse coefficient vector.
2. At each iteration, perform a gradient descent step to minimize the data fidelity term.
3. Apply the soft-thresholding operator to enforce sparsity.
4. Repeat until convergence.

These steps are as shown, from

Algorithm 1 ISTA

function ISTA(X, Z, W_d, α, L)
  Require: $L >$ largest eigenvalue of $W_d^T W_d$.
  Initialize: $Z = 0$,
  repeat
    $Z = h_{(\alpha/L)}(Z - \frac{1}{L} W_d^T (W_d Z - X))$
  until change in Z below a threshold
end function

Figure 15: ISTA Algorithm[2]

3.2.1 Algorithm Implementation & Monte Carlo Trial

Just like in OMP implemntation, the basic libraries were imported and the sinusoidal input is converted to its sparser domain using DCT. The soft thresholding function plays a role in enforcing sparsity by shrinking very small values to zero. Since convergence is very slow in ISTA, the number of iterations are higher than that of OMP.

Monte Carlo has been implemented in a very similar manner as that of OMP and it has been checked for all the 3 paramters. Moreover, both the algorithms have been compared for these parameters, and their performance has been observed and analysed.

3.2.2 Observations & Results

Implementing ISTA for a sinusoidal input had some similarities with that of the OMP algorithm. The trends in the major 3 parameters are same for both the algorithms. Initially ISTA was checked for pure reconstruction, that is no noise interfernce. The result is as plotted,

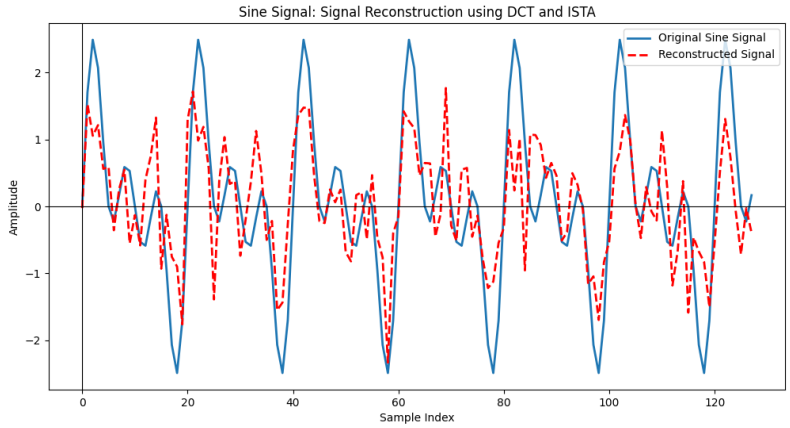

Figure 16: ISTA Algorithm Implementation (Ideal)

It is observed that as the number of iterations for ISTA increases, the error decreases.

Now, when noise is added during the process, its performance is also as shown,

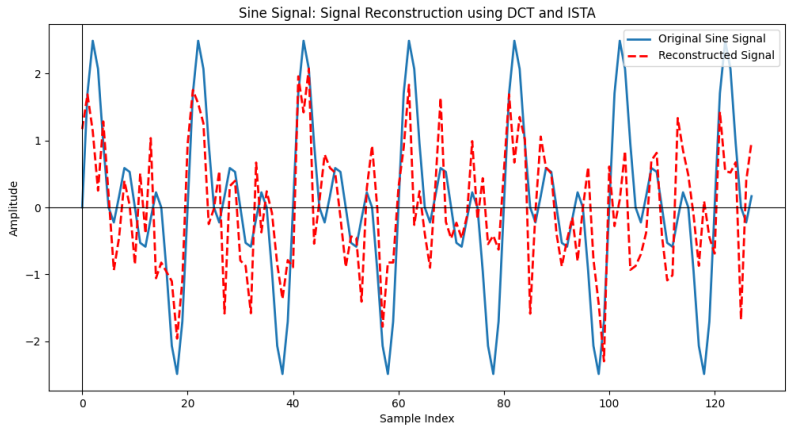

Figure 17: ISTA Algorithm Implementation (With Noise)

Here, unlike OMP, ISTA is very resistant to noise variation and hence explains its advantage over OMP. That is because of the shrinkage function, which shrinks small coefficients to zero and its regularisation term penalises the high variance solutions. In contrary, OMP being a greedy algorithm, tends to select the noisy atoms causing to succumb to the effect of noise. Hence, ISTA is more robust to noise than OMP.

The Monte Carlo for ISTA has been implemented with varying measurements for every value of n. The results are to some extent similar to that of the OMP, that is the reconstruction error follows an inverse relationship with the number of measurements, which is as shown below,

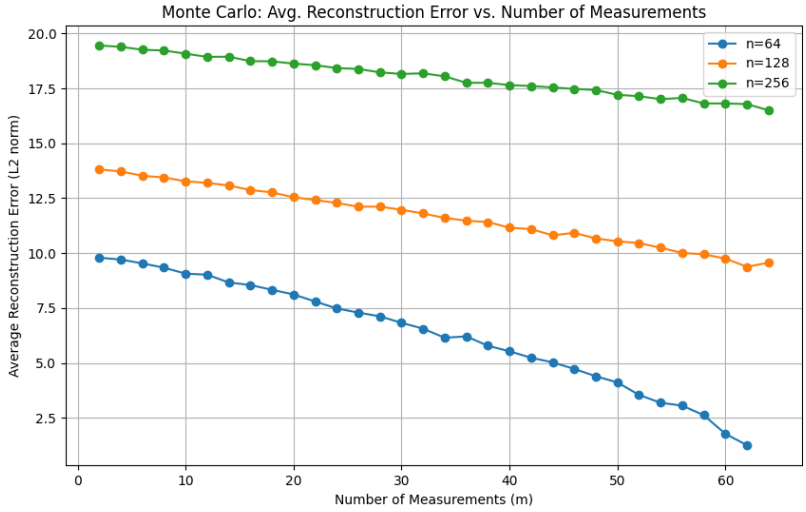

Figure 18: Monte Carlo Trial: Measurements (m)

With these 3 parameters used for the trial, it can be done to compare both ISTA and OMP. This is done so as to assess and understand the scope of the algorithm for future work, etc. The comparisons are shown below

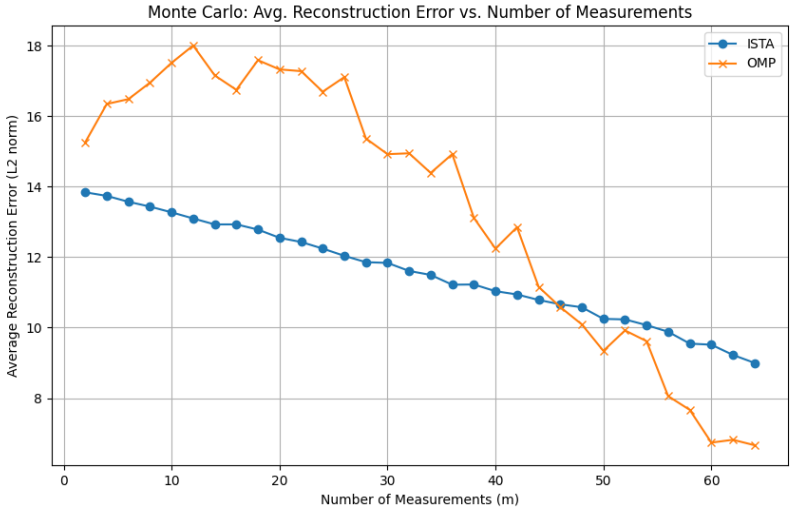

Figure 19: OMP v/s ISTA (m)

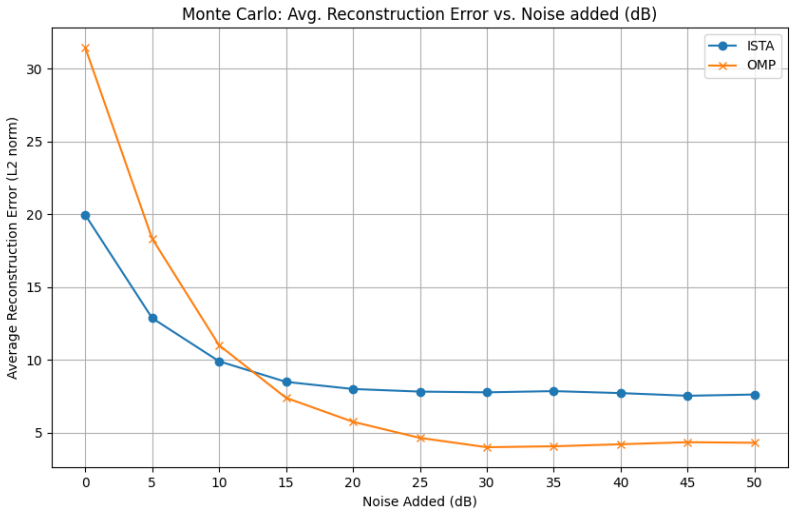

Figure 20: OMP v/s ISTA (noise)

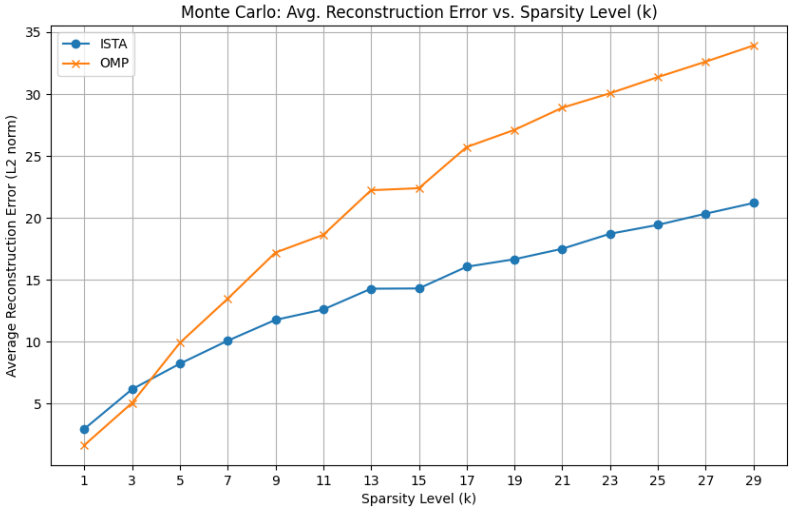

Figure 21: OMP v/s ISTA (sparsity)

In summary, both OMP and ISTA have their own strengths and are suitable for different scenarios in compressed sensing-based radar signal processing:

- **OMP** excels when the signal is highly sparse and the number of measurements is sufficient. It is computationally efficient and provides accurate reconstruction in low-noise environments. However, its performance degrades with increased noise or when the sparsity assumption is violated.
- **ISTA** is more robust to noise due to its regularization and shrinkage steps. It can handle less sparse signals and noisy measurements better than OMP, albeit at the cost of slower convergence and higher computational complexity. It is best in handling undersampled and noisy data.

The choice between OMP and ISTA depends on the specific requirements of the application, such as the expected sparsity of the signal, noise levels, and computational resources. In practice, a trade-off must be made between reconstruction accuracy, noise robustness, and computational efficiency.

3.3 COORDINATE DESCENT (CoD)

Coordinate Descent is a simple yet powerful optimization algorithm that is widely used in compressed sensing applications, especially for solving large-scale sparse recovery problems such as LASSO (Least Absolute Shrinkage and Selection Operator). It works by minimizing (or maximizing) a function by solving for one variable at a time while keeping the others fixed. The process repeats, cycling through each variable (or "coordinate") in turn, updating its value to reduce the objective function. The general steps are: 1. Initialize the sparse coefficient vector. 2. For each coordinate (variable), update its value by minimizing the objective function with respect to that coordinate, keeping all other variables fixed. 3. Apply the soft-thresholding operator to the updated coordinate to enforce sparsity. 4. Repeat steps 2 and 3 for all coordinates, cycling through them until convergence.

Algorithm 2 Coordinate Descent (Li & Osher, 2009)

function CoD(X, Z, W_d, S, α)
  Require: $S = I - W_d^T W_d$
  Initialize: $Z = 0;\ B = W_d^T X$
  repeat
    $\bar{Z} = h_\alpha(B)$
    $k =$ index of largest component of $|Z - \bar{Z}|$
    $\forall j \in [1, m]: \quad B_j = B_j + S_{jk}(\bar{Z}_k - Z_k)$
    $Z_k = \bar{Z}_k$
  until change in Z is below a threshold
  $Z = h_\alpha(B)$
end function

3.3.1 Algorithm Implementation

In the MATLAB implementation of CoD algorithm, The process begins by generating a sparse signal (z_true) and its measurement (y) using a random sensing matrix (Phi). The main loop iteratively updates each coordinate of the sparse coefficient vector (z) by applying the soft-thresholding operator, which enforces sparsity. At each iteration, the coordinate with the largest change is selected and updated to minimize the objective function. The reconstructed

signal is then compared to the original, and the mean squared error (MSE) is tracked over iterations to monitor convergence. The code concludes by plotting both the original and reconstructed signals, as well as the MSE progression, illustrating the effectiveness of the CoD algorithm in recovering sparse signals.

3.3.2 Monte Carlo Trials

A python script was created for the algorithm that closely matches the MATLAB implementation and Monte Carlo Trials were conducted on the algorithm. The input values were stored in a list and these were fed to the trial algorithm. The error was calculated for a number of trials for the same input values, and only the average error is plotted to prevent unwanted variations in reconstruction.

3.3.3 Observations & Results

Two types of implementations were considered- an ideal noiseless input signal and an input signal with 10dB of noise. For the ideal condition, multple values of m were considered(32,64 and 128) for n = 128 and the original vs reconstructed signal graphs were plotted.

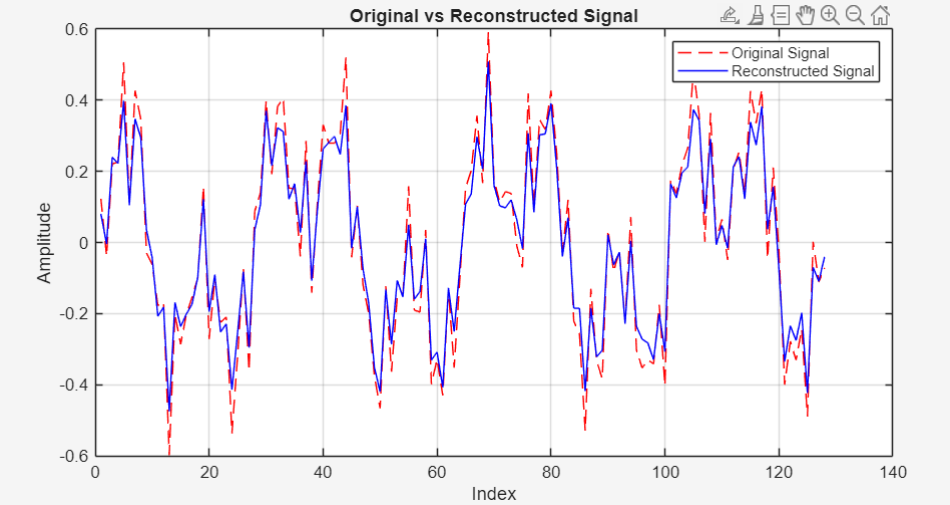

Figure 22: Ideal CoD at m = 32

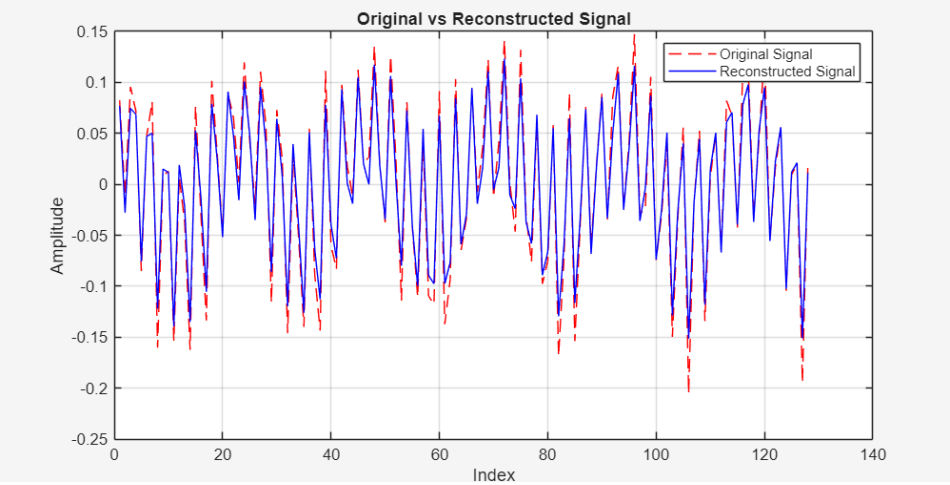

Figure 23: Ideal CoD at m = 64

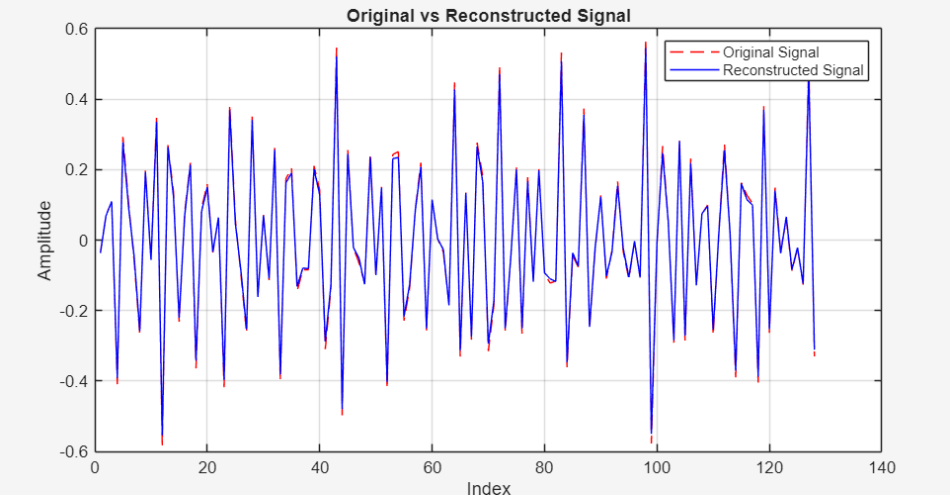

Figure 24: Ideal CoD at m = 128

Also, the plot between the Mean Squared Error(MSE) and the number of iterations gives us the convergence of the algorithm.

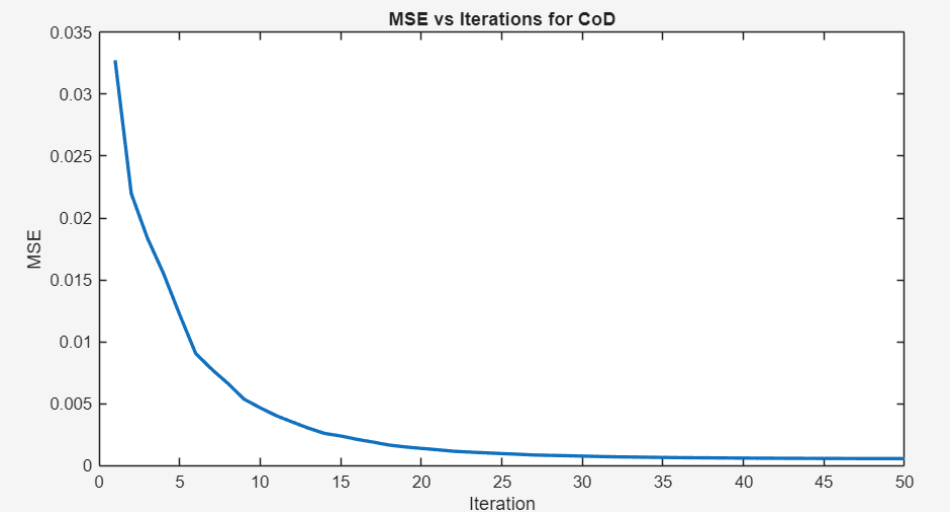

Figure 25: Iterations vs MSE plot for Ideal CoD

it is observed that increase in resolution, ie. increasing the value of m increases the accuracy of the reconstructed signal. The accuracy is more than enough for m = 32, which is only one fourth of the total samples. at m = 64, the signals is almost entirely reconstructed except for a slight decrease in Amplitude.

Now, for the noisy implementation m = 32 and 64 were considered for n = 128 and the corresponding graphs were plotted.

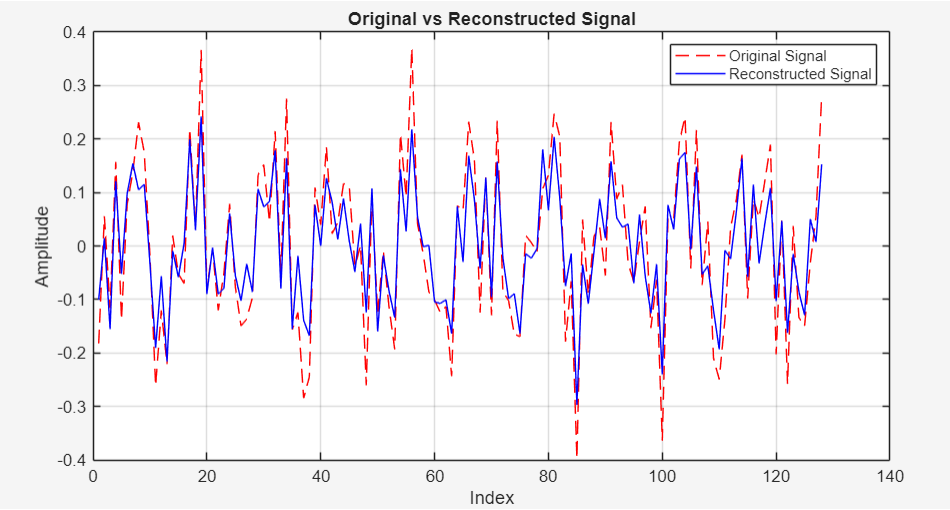

Figure 26: Original vs Reconstructed signal: m=32

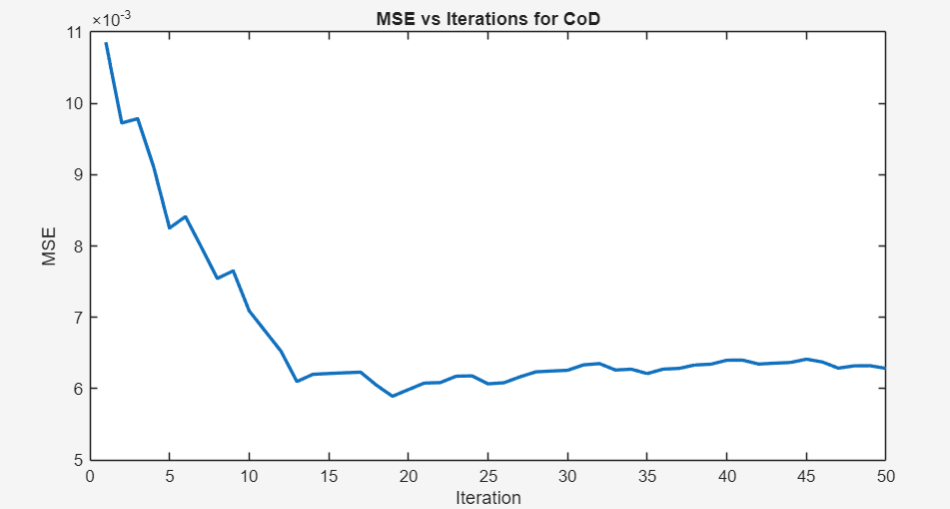

Figure 27: Iterations vs MSE graph: m = 32

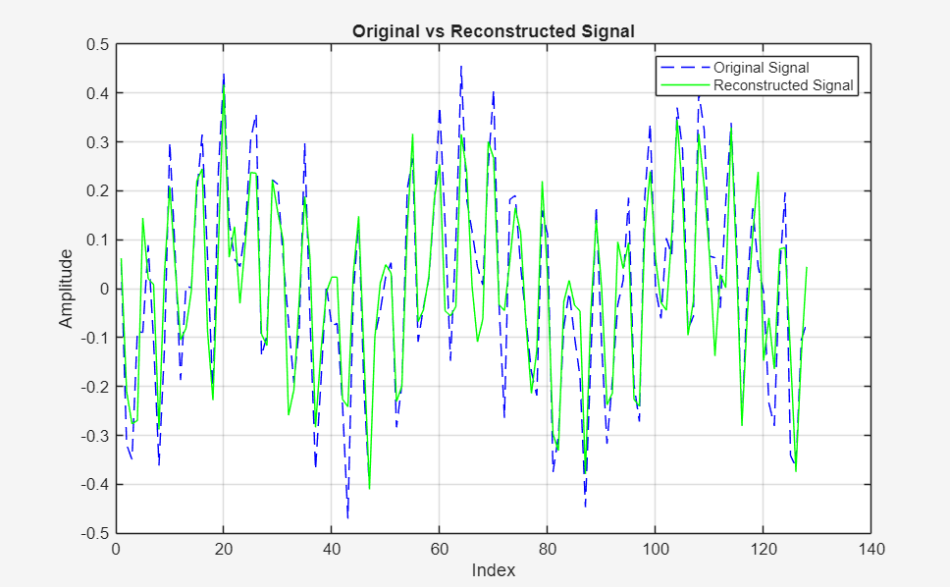

Figure 28: Original vs Reconstructed signal: m=64

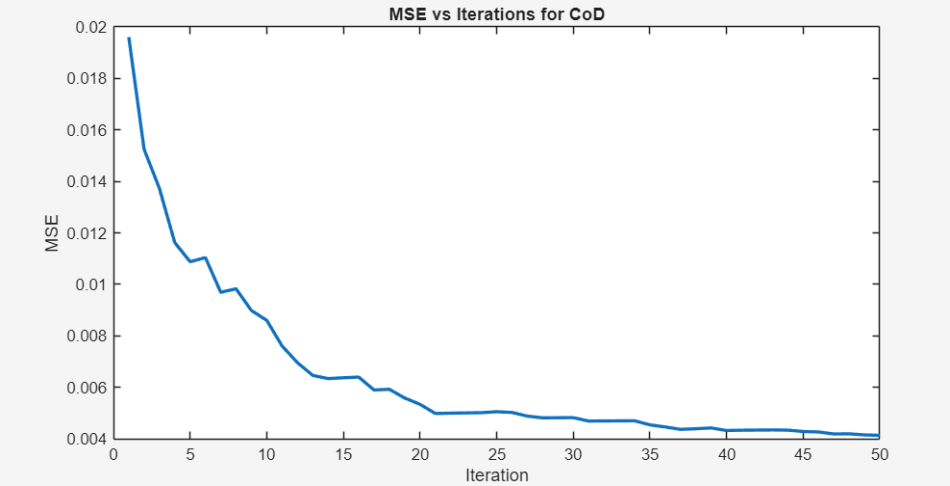

Figure 29: Iterations vs MSE graph: m = 64

for noisy implementation, n = 32 shows considerable error in amplitudes. Some of the smaller peaks are ignored by the algorithm. Its Iteration vs MSE graph is spiky but still shows a clear trend of convergence. n = 64 shows even better performance. There are still some amplitude errors, but this is very good performance for only half the samples. The Iteration vs MSE graph is much more smooth. Overall, the algorithm is more than capable of reconstructing a noisy signal of low resolution with managable errors. The Monte Carlo for CoD has been implemented in the same way as for OMP and ISTA, with varying measurements for every value of n.

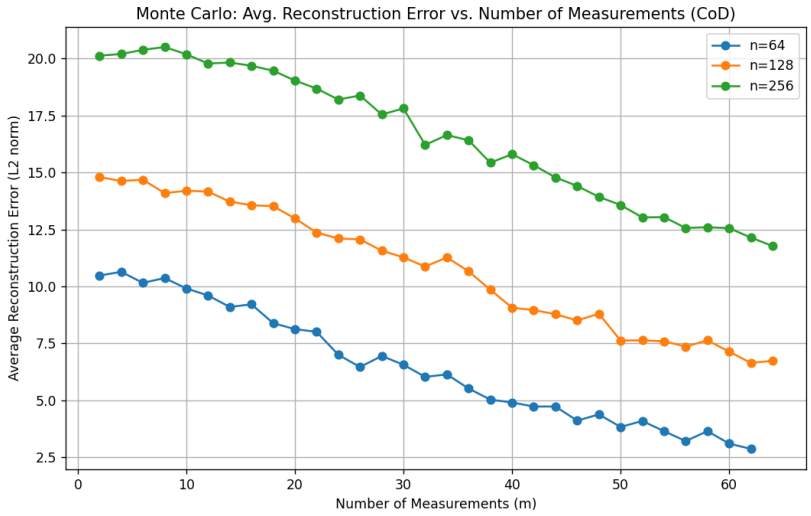

Figure 30: Monte Carlo Trials - CoD

The results show that the reconstruction error is inversely proportional to the number of measurements m and proportional to the length n.

CHAPTER 4: REAL TIME PROCESSING OF RADAR SIGNALS

The algorithms discussed above, such as OMP, ISTA, and CoD, generally have computational complexities that scale quadratically or cubically with the problem size. For example, OMP has a worst-case time complexity of cubic terms per signal. ISTA and CoD, while sometimes more efficient per iteration, may require a large number of iterations to converge, leading to overall quadratic or higher complexity.

So, hence, while these algorithms are effective for offline or simulated environments, they are often not sufficient for real-time radar signal processing due to several reasons:

- **Computational Complexity:** Algorithms like OMP, ISTA, and CoD can be computationally intensive, especially for large-scale problems or high-dimensional signals. Real-time radar applications require fast processing to meet strict latency requirements, which may not be achievable with these iterative algorithms on standard hardware.
- **Latency Constraints:** Real-time systems demand immediate or near-instantaneous responses. The iterative nature of these algorithms can introduce unacceptable delays, making them unsuitable for time-critical radar applications.
- **Resource Limitations:** Embedded radar systems often have limited memory and processing power. The memory and computational requirements of these algorithms may exceed the capabilities of such systems.
- **Robustness to Dynamic Environments:** Real-time radar must handle rapidly changing environments, interference, and noise. The algorithms discussed may not adapt quickly enough to such variations or may require parameter tuning that is impractical in real time.
- **Scalability:** As the number of targets or the dimensionality of the data increases, the performance of these algorithms can degrade, further limiting their applicability in real-time scenarios.

To address these challenges, we are introducing two modifications in our model,

1. **Augmented Dictionary for Blind Reconstruction**
2. **Reconstruction Algorithm Unrolling**

CHAPTER 5: BLIND RECONSTRUCTION USING AUGMENTED DICTIONARY

5.1 INTRODUCTION

In traditional compressed sensing, the measurement process assumes prior knowledge of the basis (or dictionary) in which the signal is sparse. However, in many real-world radar applications, the exact basis or the parameters of the signal (such as frequency, phase, or waveform) may not be known in advance. This scenario is referred to as **blind reconstruction**.

Blind reconstruction aims to recover both the sparse signal and the underlying dictionary (or its parameters) from the compressed measurements. One effective approach to address this challenge is to use an **augmented dictionary**— a collection of candidate basis vectors that span a wider range of possible signal structures.

So, the upcoming sections will discuss the implementation and the observations drawn from it.

5.2 IMPLEMENTATION

In the previous implementations, the input signal type is known and hence, the basis is also known and the compression is done with ease. But in real life, the the radar can receive any type of signals, and some of them might not be sparse enough for the signal to be taken for reconstruction.

For now, only 3 types of signals are accepted as inputs, that is **Single tone Sinusoids**, **LFM(Linear Frequency Modulated)** a.k.a **Chirp signals** and **BPSK (Binary Phase Shifted Key)** signals. The input signals are generated and is sent through a mixer. The suitable basis is analysed and selected for making the augmented basis by concatenating each of the individual dictionaries.

$$\Psi_{aug} = \left[\Psi_{sinusoid} \,\middle|\, \Psi_{LFM} \,\middle|\, \Psi_{BPSK} \right] \tag{5}$$

where $\Psi_{sinusoid}$, Ψ_{LFM}, and Ψ_{BPSK} are the basis matrices for Single-tone sinusoids, LFM signals, and BPSK signals, respectively, and Ψ_{aug} is the augmented dictionary formed by concatenating these bases.

A gaussian noise is also added to check its performance in noisy environments.

5.3 OBSERVATIONS & RESULTS

Initially the DCT matrix is used as the basis for the above mentioned input signals and their results are as shown,

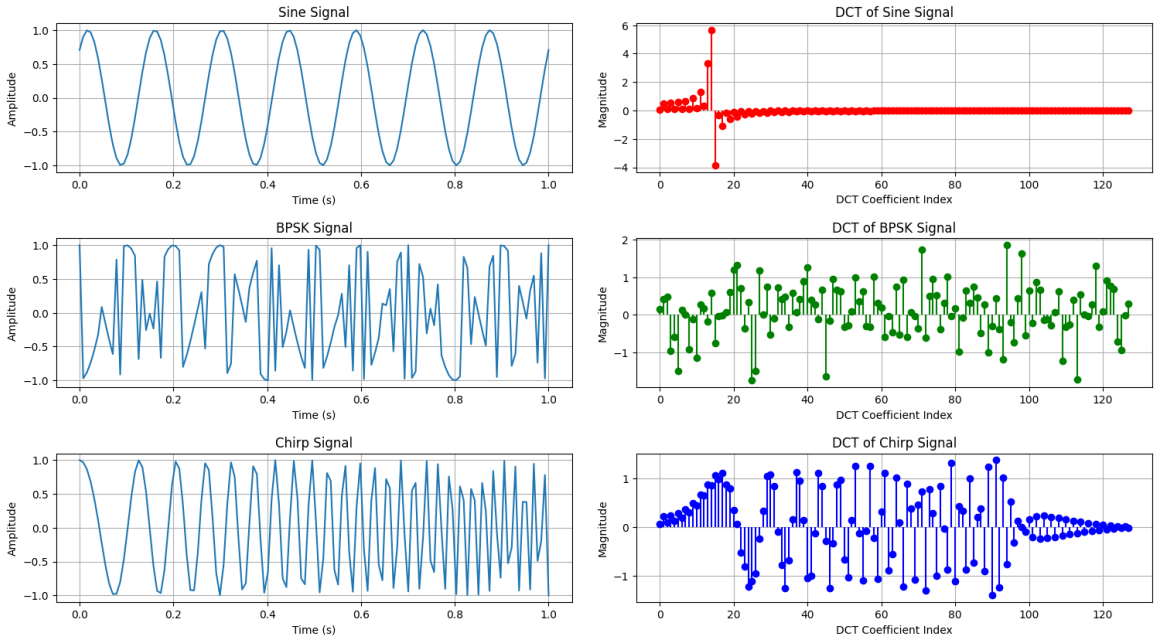

Figure 31: DCT on different radar signals

As shown above, other than the single tone sinusoids, the other two matrices are not as sparse as expected. Even though, chirp signals are expected to be not sparse, the BPSK signal is not sparse enough for reconstruction.

So, for easy implementation of the dictionaries, the respective atoms are created and is used as a dictionaries, that is, the bpsk and chirp signal values are generated and filled to their respective matrix as their dictionaries.

The mixer initially just added the 3 signals, but later on coefficients were added for better understanding of its sparsity. The results, however, are not really as satisfactory as expected. The signal reconstruction is as shown

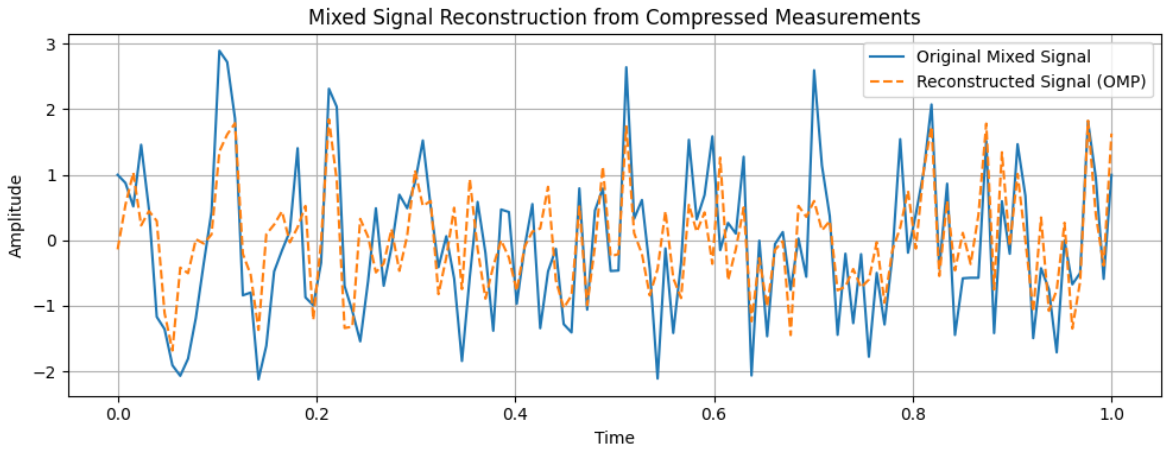

Figure 32: Augmented Dictionary Implementation (Ideal): n=128 m=64

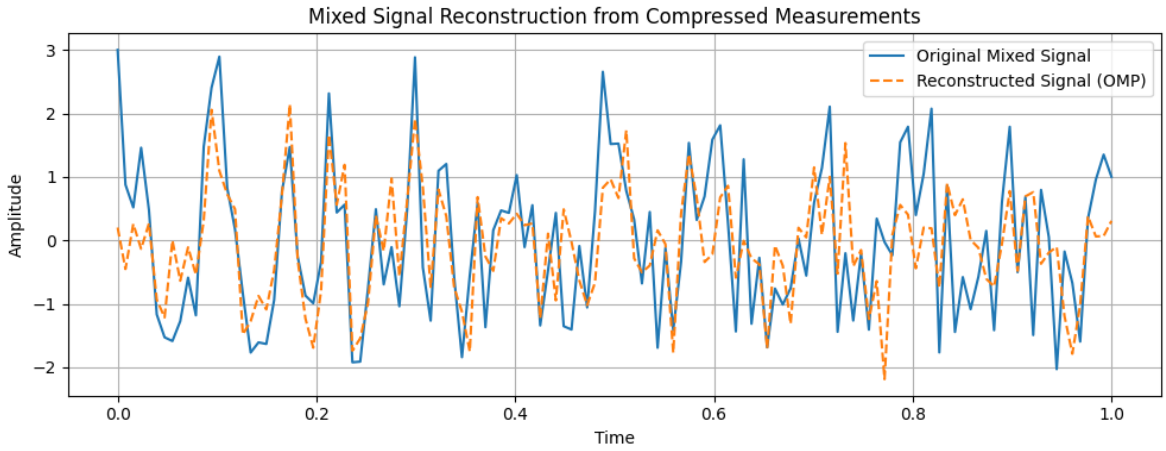

Figure 33: Augmented Dictionary Implementation (Noisy): n=128 m=64

so, now instead of just simply adding the signals as it is, coefficients were added (a, b, c) such that each component of the input signal can be tuned for observation.

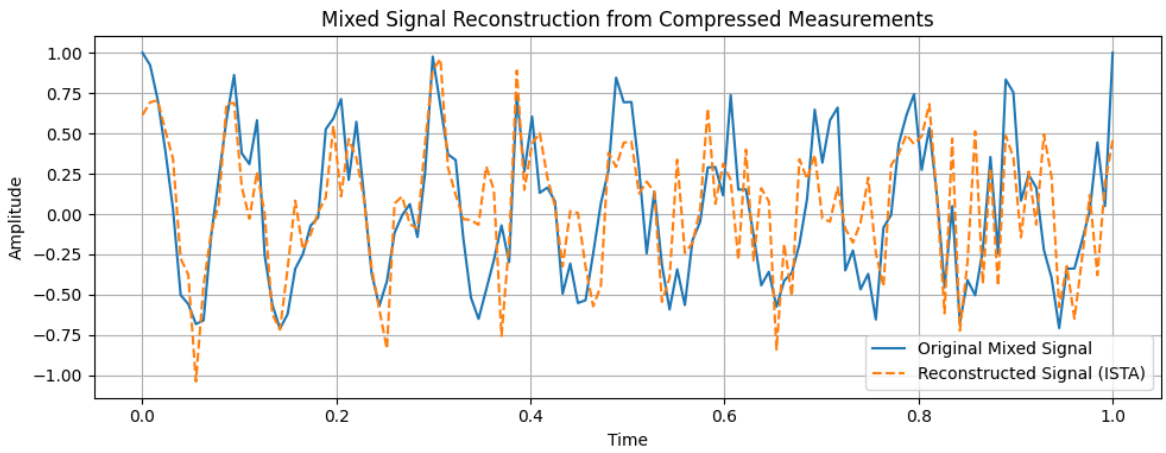

Figure 34: Augmented Dictionary Implementation (Ideal) with mixer: n=128 m=64

It is observed that the reconstruction error is pretty significant compared to the overall amplitude of the input signal.

From the above implementation, we can say that the augmeneted dictionary is a good attempt in assessing multiple radar input signals. But despite that, the dictionary still needs to be pre-defined.

Moreover, most of the other radar signals are non-linear and some of these signals can't have a basis for sparsity.

The best option is to adopt a learned method, or autoencoder to learn a non-linear based basis for all the possible trained signals.

CHAPTER 6: RECONSTRUCTION ALGORITHM UNROLLING

6.1 INTRODUCTION

As explained in the sections above, the current reconstruction algorithms are iterative, hence require a long computational period before an output is generated, which hampers the real time processing of signal. One way to avoid this is improving the Reconstruction algorithms to work real time.For such a case, machine learning techniques could help. One promising approach is **algorithm unrolling** (also known as unfolding), where the iterative steps of a reconstruction algorithm are mapped onto the layers of a neural network[2]. This technique enables the integration of domain knowledge from classical algorithms with the learning capacity of neural networks, resulting in models that are both interpretable and highly effective.

Algorithm unrolling involves representing each iteration of an algorithm—such as ISTA or OMP—as a layer in a neural network. The parameters of these layers (e.g., thresholds, step sizes, or transforms) can then be learned from data, allowing the network to adapt to the specific characteristics of the signals and measurement process. This approach not only accelerates convergence compared to traditional iterative methods but also improves reconstruction quality, especially in challenging scenarios with noise or model mismatch.

In this section, we discuss its advantages for real-time radar signal processing, and introduce two unrolled algorithms - **Learned ISTA(LISTA)** and **Learned CoD(LCoD)**.

6.2 ADVANTAGES

Algorithm unfolding is a technique that transforms iterative algorithms (like those used in optimization or signal processing) into a sequence of layers, similar to neural networks. This approach offers several advantages:

Interpretability: Each layer corresponds to a step in the original algorithm, making the model's operations easier to understand and analyze. **Efficiency**: By learning parameters for each layer, unfolded algorithms can converge faster and require fewer iterations than traditional methods. **Adaptability**: The unfolded structure can be trained end-to-end, allowing it to adapt to specific data distributions or tasks. **Performance**: Often achieves better accuracy and robustness compared to standard iterative algorithms, especially when combined with data-driven learning.

6.3 LEARNED ISTA (LISTA)

Learned ISTA (LISTA) is a neural network-based approach that unrolls the traditional ISTA algorithm into a fixed number of layers, where each layer mimics one iteration of ISTA. Unlike standard ISTA, where parameters such as step size and thresholds are fixed, LISTA learns these parameters from data during training. This enables faster convergence and improved reconstruction accuracy. Each layer of the LISTA network consists of a linear transformation followed by a non-linear shrinkage (soft-thresholding) operation, and the parameters of these operations are optimized by successive iterations. As a result, LISTA combines the interpretability of classical algorithms with the adaptability and efficiency of deep learning, making it well-suited for real-time and large-scale compressed sensing applications.

6.3.1 Algorithm Implementation

The algorithm implemented here has been improvised from the Python implementation of LISTA from [4]. All the required packages were imported and the seed was fixed for reproducibility. A simulated dataset was generated with a random dictionary and sparse signal. This was plotted.

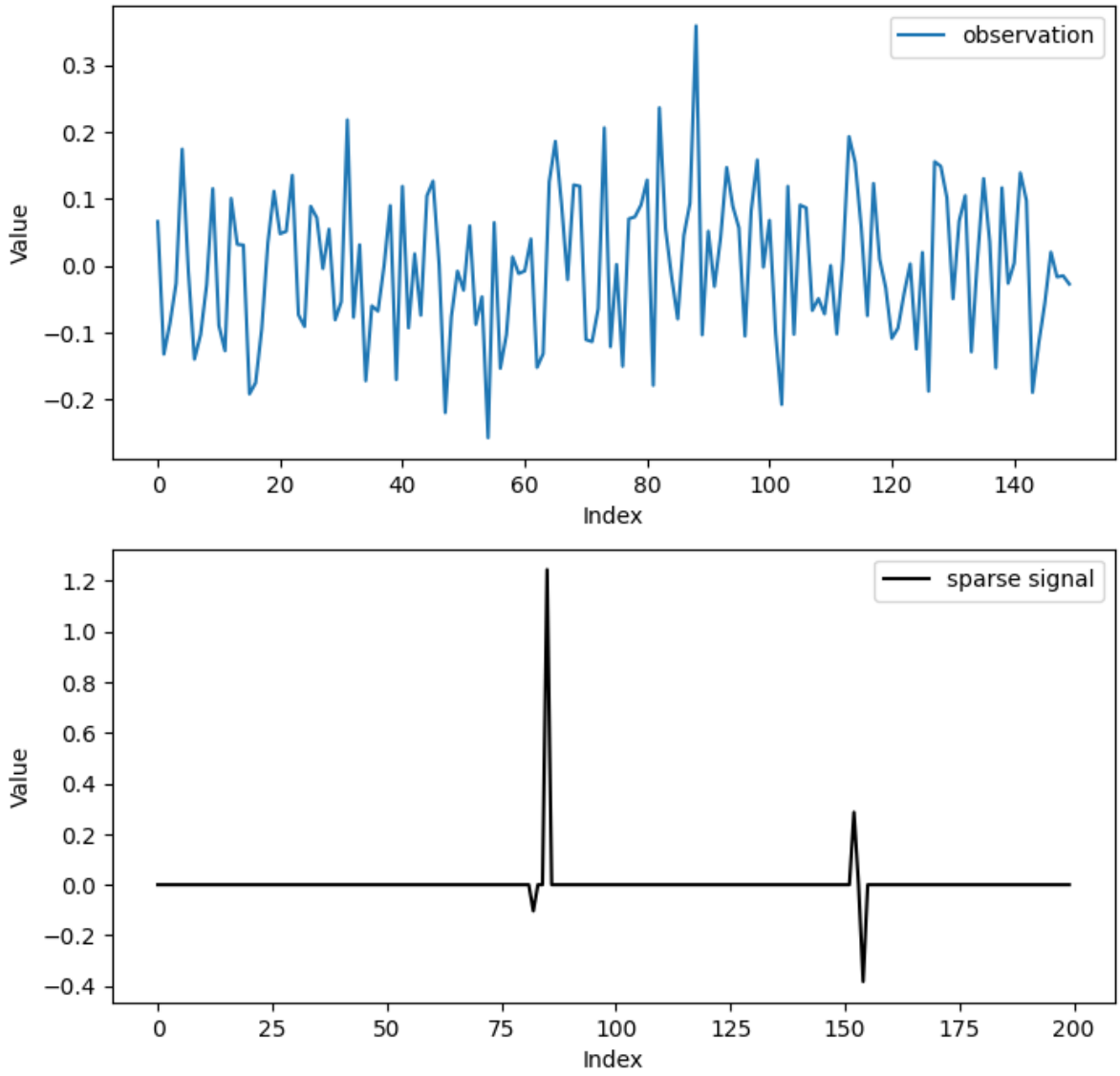

Figure 35: Plot of Simulated Data

First the ISTA algorithm was applied to the test set. The untrained LISTA algorithm was then applied to the dataset, and the results were compared with the standard ISTA algorithm. Now, the learned LISTA algorithm was applied to the dataset, and the results were compared with the standard ISTA algorithm.

6.3.2 Observations and Conclusion

The Iterations vs MSE plot of ISTA and Untrained LISTA shows clear convergence. The untrained LISTA and classic ISTA algorithms should be mathematically equivalent,but the starting point of both algorithms are different, indicating issues with equivalence. But both algorithms converge to the same reconstruction error.

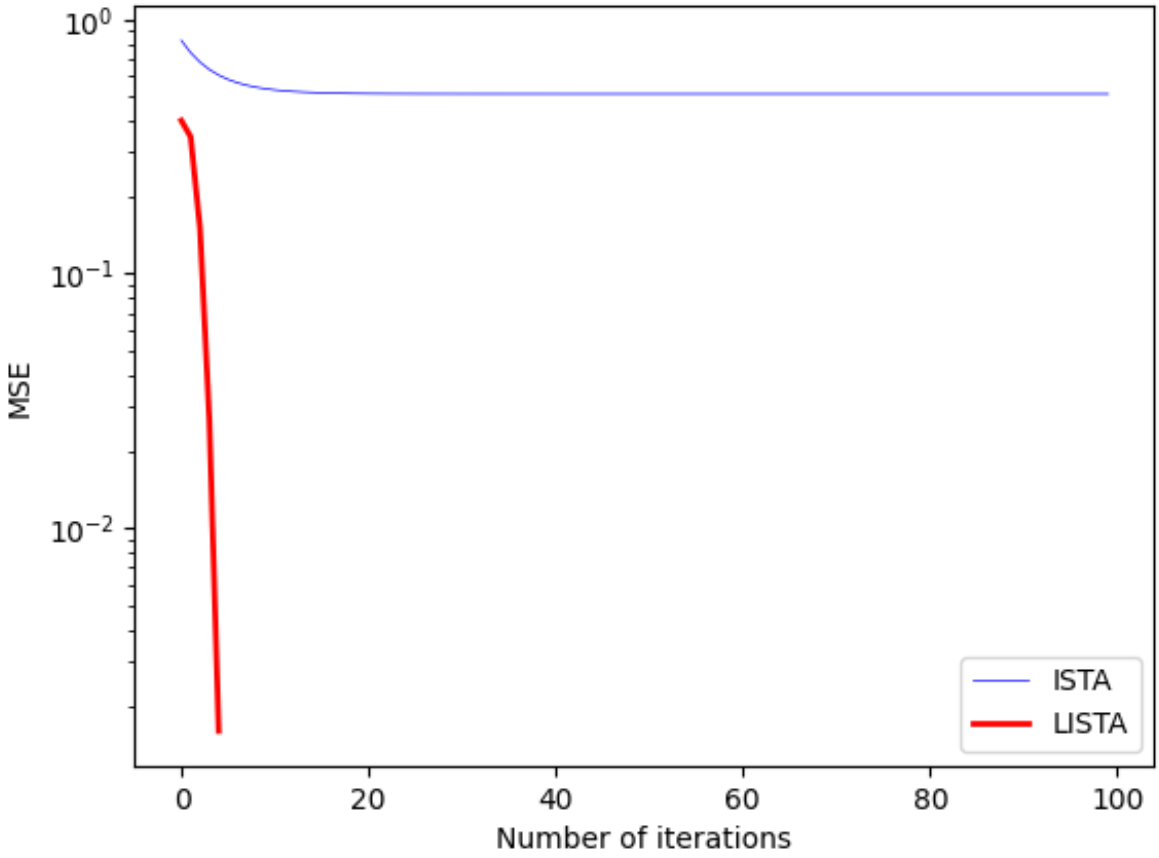

Figure 36: ISTA vs Untrained LISTA

The iterations vs MSE plot of ISTA vs Trained LISTA is given below. The trained LISTA algorithm converges to a better reconstruction error than the untrained LISTA algorithm, using very low numbers of iterations. The starting point of both algorithms is different, indicating the same issues with equivalence as that of the above plot.

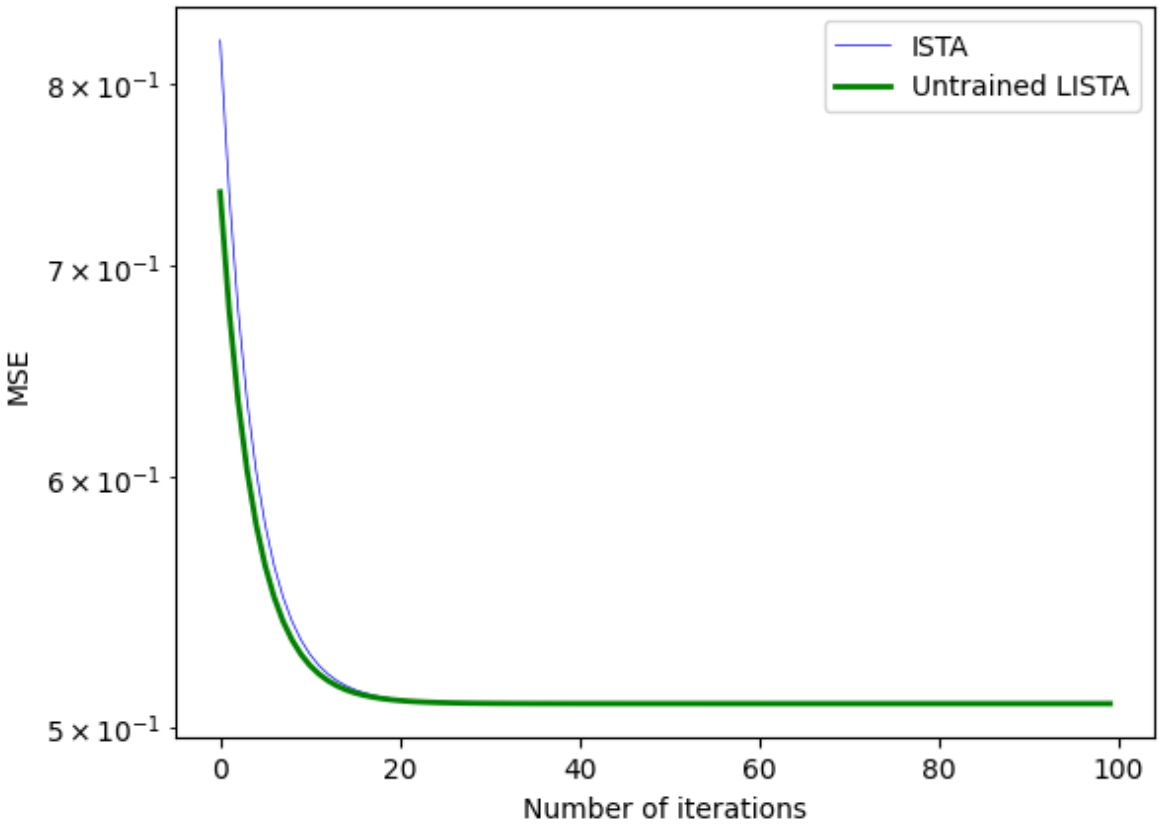

Figure 37: ISTA vs trained LISTA

The conclusion is that Learned ISTA (LISTA) method is more than capable of replacing ISTA, combining the interpretability of classic ISTA with the adaptability and efficiency of neural networks.

6.4 LEARNED COORDINATE DESCENT (LCoD)

The Learned Coordinate Descent (LCoD) algorithm is an optimization technique that iteratively updates one coordinate (variable) at a time to minimize a target function. Unlike standard coordinate descent, LCoD leverages machine learning to adaptively select which coordinate to update and how much to change it, based on patterns learned from data. This approach can lead to faster convergence and improved performance, especially in high-dimensional or structured problems.

6.4.1 Algorithm Implementation

The code follows the same algorithmic flow as the previous Python implementation of LISTA. All the required packages were imported and the seed was fixed for reproducibility. A simulated dataset was generated with a random dictionary and sparse signal.

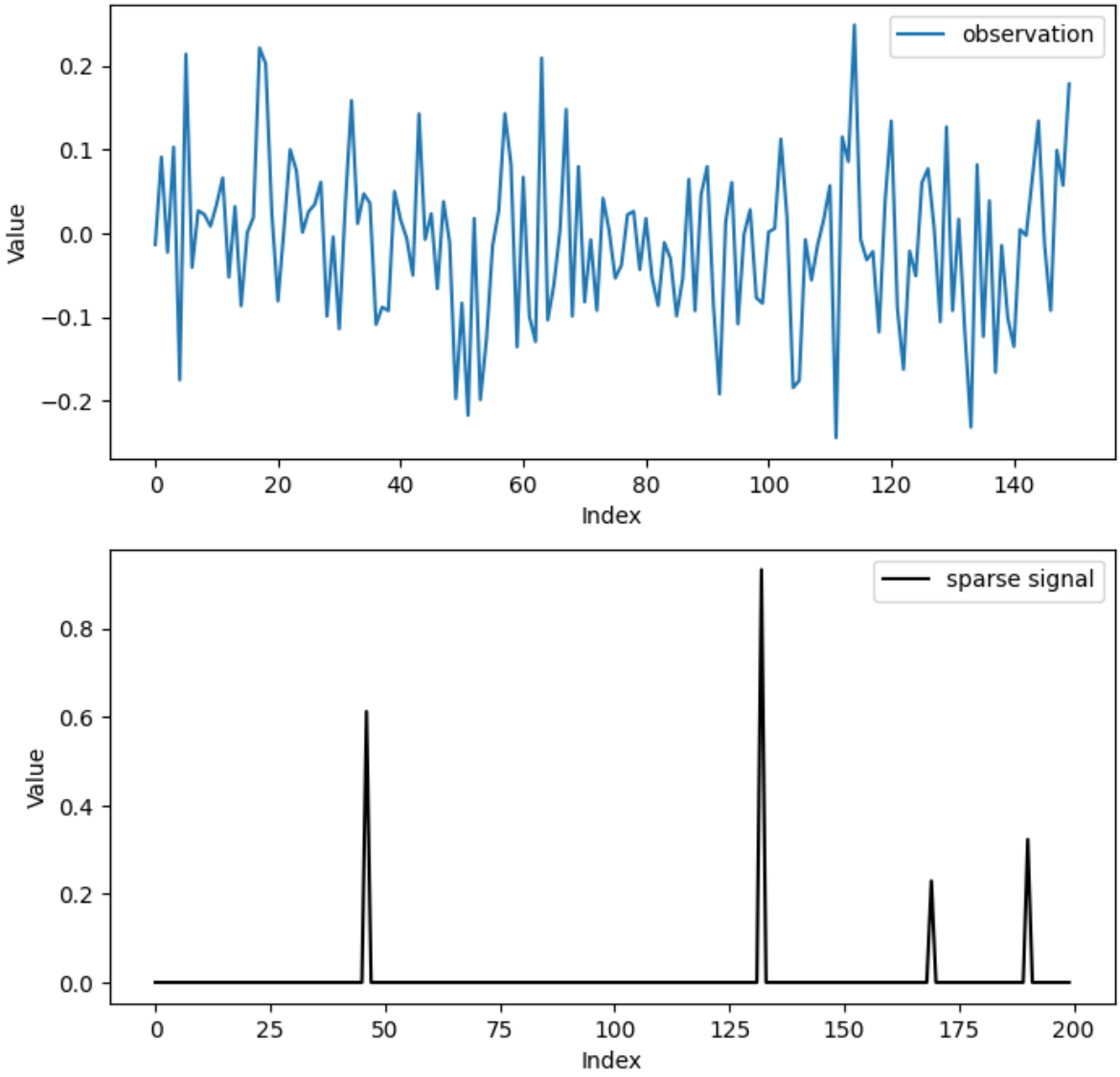

Figure 38: Plot of Simulated Data

First the CoD algorithm was applied to the test set. The untrained LCoD algorithm was then

applied to the dataset, and the results were compared with the standard CoD algorithm. Now, the learned LCoD algorithm was applied to the dataset, and the results were compared with the standard CoD algorithm.

6.4.2 Observations

The Iterations vs MSE plot of CoD and Untrained LCoD shows that both algorithms are mathematically equivalent. This is illustated in the plot below. This is so, since no optimisations or changes were made to the standard iterative step of CoD to create unrolled iterative step of LCoD.

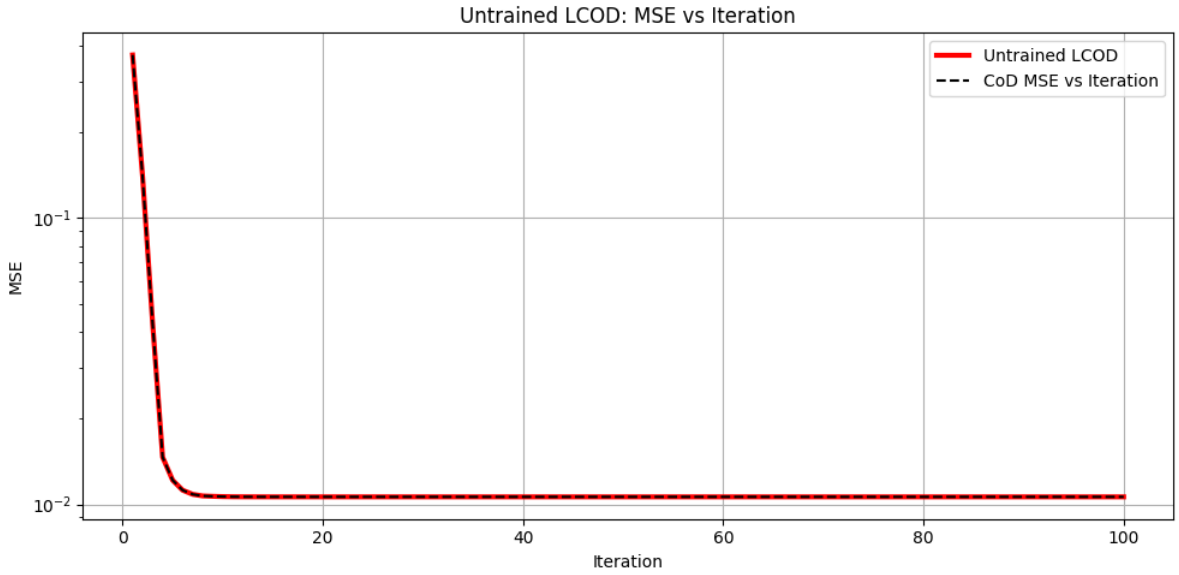

Figure 39: COD vs Untrained LCoD

The iterations vs MSE plot of CoD vs Trained lCoD is given below. The trained LISTA algorithm converges to a better reconstruction error than the untrained LISTA algorithm, using lesser numbers of iterations.

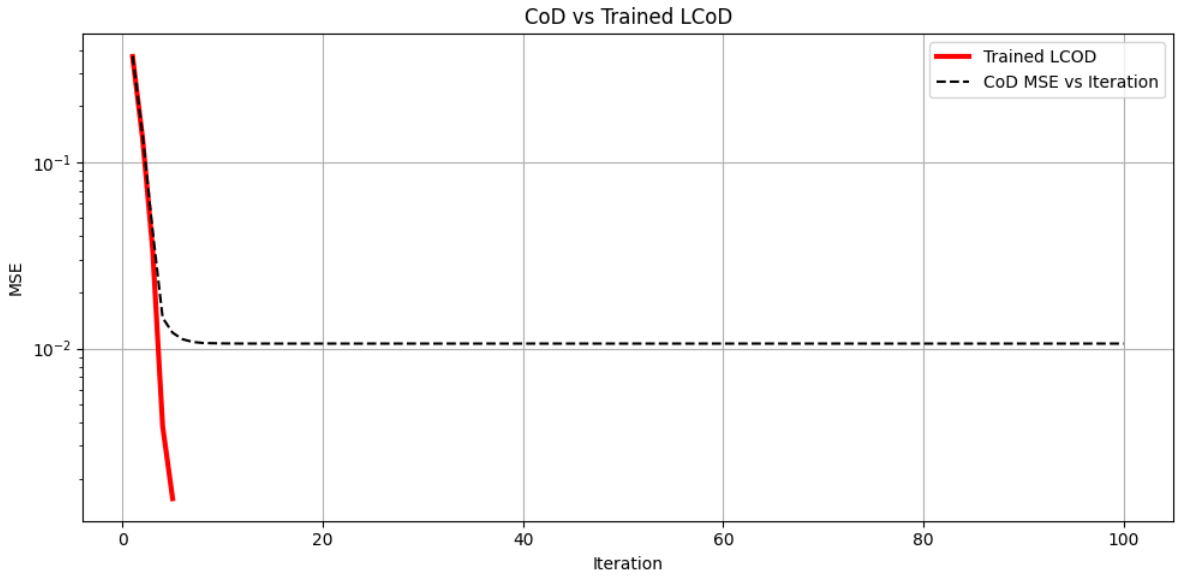

Figure 40: COD vs trained LCoD

A similar conclusion is made to the previous section.

6.5 COMPARING BOTH ALGORITHMS

A Python implementation comparing the performance of Learned ISTA (LISTA) and Learned CoD (LCoD) algorithms on a simulated dataset is provided.

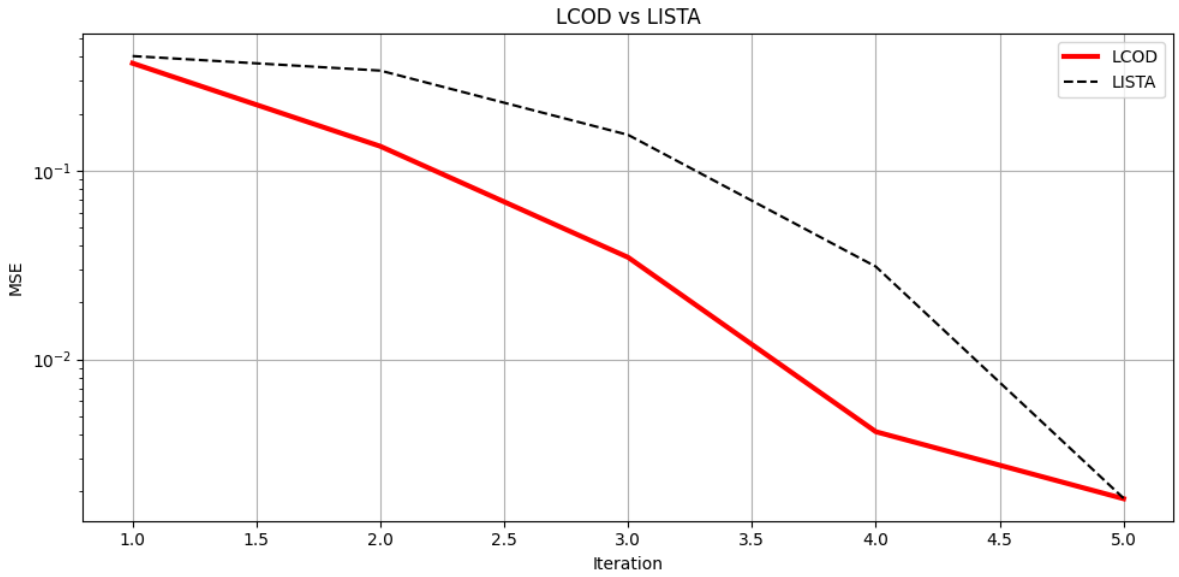

Figure 41: Comparison of LISTA and LCoD

The plot above compares the MSE vs Iterations graph of Learned ISTA (LISTA) and Learned CoD (LCoD) algorithms. Both algorithms converge to the same reconstruction error. However, two major observations were made:

1. The iterations of LISTA is more inefficient compared to LCoD.

This is because CoD is an iterative algorithm that selects and updates one coordinate at a time greedily, hence it is guaranteed that it would converge to a lesser MSE per iteration. ISTA does not have such guarantees, hence LISTA may have higher MSE than LCoD at the same number of iterations.

2. LCoD have higher computation time per iteration compared to LISTA.

As explained above, CoD selects and updates coordinates one at a time per iteration, hence it is more computationally expensive. Also, The iterative step of CoD in the current implementation is not optimised for torch operations, therefore steps involving torch operations take longer to compute. The algorithm is serial by definition, thus training involving batch operations needs workarounds to make it happen. Hence, the current implementation takes longer time to compute than ISTA or LISTA for the given problem.

The faster computation time of LISTA is beneficial for real-time signal processing applications involving small-scale and parallelised computing. But for larger scale LASSO problems,

LCoD can be more efficient, since the convergence guarentee of CoD means even though individual iterations can be time-consuming, the number of iterations required is much lower, which means the total computation time compared to LISTA is less.

6.6 FINAL REMARKS

The above observations show that Algorithm unrolling can significantly improve the performance of iterative algorithms for a fraction of the number of iterations. This method combines low sampling requirement of compressed sensing algorithms with lower time complexity of learned ML models, both of which are important requirements for real-time signal processing applications.

CHAPTER 7: FUTURE SCOPE

This report lays a solid foundation for advanced research and practical applications in compresses sensing for radar signal processing.

1. In this project, the input signals are purely real, hence the use of DCT as basis. But in real life applications, the phase (complex) part is also applicable.
2. We have explored different reconstruction algorithms (like OMP, ISTA, CoD). The reconstruction can be further enhanced by other advanced algorithms like Fast ISTA (FISTA) or Alternating Direction Method of Multipliers (ADMM).
3. Instead of augmented dictionary, a learned dictionary (Autoencoder) can be used for predicting the non-linear radar signals.
4. Optimise torch operations for LCoD.
5. Trying to implement the real time processing into a hardware (FPGAs/Embedded GPUs) by reducing model size and complexity for low latency power.

REFERENCES

References

[1] *Orthogonal Matching Pursuit Algorithm: A brief introduction*. 2022.

[2] Karol Gregor and Yann LeCun. "Learning fast approximations of sparse coding". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Omnipress, 2010, pp. 399–406. ISBN: 9781605589077.

[3] S B Dhok M Rani and R B Deshmukh. *A systematic review of Compressed Sensing: Concepts, Implementations and Applications*. 2018.

[4] Nir Shlezinger et al. *Model-Based Deep Learning*. 2022. arXiv: 2012.08405 [eess.SP]. URL: https://arxiv.org/abs/2012.08405.

APPENDICES

FUNCTION:-

- **Sine Wave Generator**

```
def generate_sine_signal(n, k, freq=5, fs=100):
    t = np.arange(n) / fs
    sum_signal = np.zeros(n)
    for i in range(1, k + 1):
        freq = i * 5
        sum_signal += np.sin(2 * np.pi * freq * t)
    return sum_signal
```

- **Noise Generator**

```
def add_noise(y, snr_db):
    signal_power = np.mean(np.abs(y)**2)
    snr_linear = 10**(snr_db / 10)
    noise_power = signal_power / snr_linear
    noise = np.sqrt(noise_power) * np.random.randn(*y.shape)
    return y + noise
```

- **OMP Function (Python)**

```
import numpy as np

def omp(y, A, tol=1e-6):
    m, n = A.shape
    r = y.copy()
    idx_set = []
    x_hat = np.zeros(n)

    for _ in range(m):
        correlations = A.T @ r
        idx = np.argmax(np.abs(correlations))
        idx_set.append(idx)
```

```
        A_selected = A[:, idx_set]
        x_ls, _, _, _ = np.linalg.lstsq(A_selected, y, rcond=None)
        r = y - A_selected @ x_ls
        if np.linalg.norm(r) < tol:
            break

    x_hat[idx_set] = x_ls
    return x_hat
```

- **OMP Function (MATLAB)**

```
clc; close all; clear all;
function x = omp(A, b, K)
    originalA = A;               % Store the original A
    norms = vecnorm(A);
    A = A ./ norms;
    r = b;
    Lambda = [];
    N = size(A, 2);
    x = zeros(N, 1);

    for k = 1:K
        h_k = abs(A' * r);
        h_k(Lambda) = 0;
        [~, l_k] = max(h_k);

        Lambda = [Lambda, l_k];
        Asub = A(:, Lambda);
        x_sub = Asub \ b;

        x = zeros(N, 1);
        x(Lambda) = x_sub ./ norms(Lambda)';
        r = b - originalA(:, Lambda) * x(Lambda);   % Corrected
    end
end
```

- **Monte-Carlo Trial**

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import dct, idct
from numpy.linalg import norm

def monte_carlo_trial(n, m, sampling_rate):
    x_time = generate_sine_signal(n, sampling_rate)
    x_sparse = dct(x_time, norm='ortho')
    A = measurement(m, n)
    y = A @ x_sparse
    x_sparse_rec = omp(y, A)
    x_time_rec = idct(x_sparse_rec, norm='ortho')
    error = norm(x_time - x_time_rec, ord=2)
    return error

# ---- Monte Carlo Simulation and Plotting ----
num_trials = 50
n_values = [64, 128, 256]
m_values = np.arange(2, 65, 2)  # Number of measurements
noise_val = np.arange(0, 51, 5)  # Noise levels in dB
k_values = np.arange(1, 11, 1)  # Sparsity levels
sampling_rate = 100

plt.figure(figsize=(10, 6))

for n in n_values:
    avg_errors = []
    for m in m_values:
        if m >= n:
            avg_errors.append(np.nan)
            continue
        errors = []
        for _ in range(num_trials):
            errors.append(monte_carlo_trial(n, m, sampling_rate))
        avg_errors.append(np.mean(errors))
    plt.plot(m_values, avg_errors, marker='o', label=f'n={n}')
```

- **ISTA Function (Python)**

```python
def soft_thresholding(x, threshold):
    """Soft thresholding operator h_{alpha/L}"""
    return np.sign(x) * np.maximum(np.abs(x) - threshold, 0.0)

def ista(X, W_d, max_iter=200, tol=1e-6):
    alpha = 0.1
    L = 1.1 * np.linalg.norm(W_d.T @ W_d, 2)
    # L > max eigenvalue of W_d.T @ W_d
    m, n = W_d.shape
    Z = np.zeros(n)
    for _ in range(max_iter):
        Z_old = Z.copy()
        gradient = W_d.T @ (W_d @ Z - X)
        Z -= gradient / L
        Z = soft_thresholding(Z - (1.0 / L) * gradient, alpha / L)
        if np.linalg.norm(Z - Z_old, ord = 2) < tol:
            break
    return Z
```

- **CoD Implementation (Python)**

```python
# Coordinate Descent Algorithm
z = np.zeros((n, 1))
B = theta.T @ y  # Initial coefficients
S = np.eye(n) - theta.T @ theta  # Residual matrix

mse_history = np.zeros(num_iter)

for t in range(num_iter):
    z_bar = np.sign(B) * np.maximum(np.abs(B) - alpha, 0)
    k = np.argmax(np.abs(z - z_bar))
    delta = z_bar[k, 0] - z[k, 0]
    B = B + S[:, [k]] * delta
    z[k, 0] = z_bar[k, 0]
    x_rec_iter = Psi @ z
```

```
    mse_history[t] = np.mean((x - x_rec_iter) ** 2)
x_rec = Psi @ z
```

- **RADAR signal Generator**

```
import numpy as np
import torch
from scipy.signal import chirp

def generate_sine_signals(batch_size, length, freq_range=(1, 10)):
    t = np.linspace(0, 1, length)
    signals = []
    for _ in range(batch_size):
        freq = np.random.uniform(*freq_range)
        phase = np.random.uniform(0, 2*np.pi)
        signal = np.sin(2 * np.pi * freq * t + phase)
        signals.append(signal)
    return torch.tensor(signals, dtype=torch.float32)

def generate_bpsk_signals(batch_size, length):
    symbols = 2 * torch.randint(0, 2, (batch_size, length)) - 1
    return symbols.float()

def generate_chirp_signals(batch_size, length, f0=5, f1=50):
    t = np.linspace(0, 1, length)
    signals = []
    for _ in range(batch_size):
        signal = chirp(t, f0=f0, f1=f1, t1=1, method='linear')
        signals.append(signal)
    return torch.tensor(signals, dtype=torch.float32)

batch_size = 1
length = 200
sine_signals = generate_sine_signals(batch_size, length)
bpsk_signals = generate_bpsk_signals(batch_size, length)
chirp_signals = generate_chirp_signals(batch_size, length)
```

- **Augmented Dictionary**

```
import numpy as np
from scipy.fftpack import dct, idct
import matplotlib.pyplot as plt

def soft_thresholding(x, threshold):
    """Soft thresholding operator h_{alpha/L}"""
    return np.sign(x) * np.maximum(np.abs(x) - threshold, 0.0)

def ista(X, W_d, max_iter=200, tol=1e-6):
    alpha = 0.1
    L = 1.1 * np.linalg.norm(W_d.T @ W_d, 2)
    # L > max eigenvalue of W_d.T @ W_d
    m, n = W_d.shape
    Z = np.zeros(n)
    for _ in range(max_iter):
        Z_old = Z.copy()
        gradient = W_d.T @ (W_d @ Z - X)
        Z -= gradient / L
        Z = soft_thresholding(Z - (1.0 / L) * gradient, alpha / L)
        if np.linalg.norm(Z - Z_old, ord = 2) < tol:
            break
    return Z

# 1. Signal Setup
n = int(input("Enter length of signal (n): "))  # Signal length
t = np.linspace(0, 1, n)

# Sinusoidal component (tone)
tone = np.cos(2 * np.pi * 10 * t)

# BPSK component
bits = np.random.randint(0, 2, n)
bpsk = (2 * bits - 1) * np.cos(2 * np.pi * 5 * t)
```

```
# Chirp component
f0, k = 5, 80
chirp = np.cos(2 * np.pi * (f0 * t + 0.5 * k * t**2))

# Mixed signal
a, b, c = 1, 1, 1  # Coefficients for mixing
signal = a*tone + b*bpsk + c*chirp

# Sub-dictionary 1: DCT
D_dct = np.eye(n)
D_dct = idct(D_dct, norm='ortho').T

# Sub-dictionary 2: Chirp-like atoms
def generate_chirp_atoms(n, num_atoms):
    t = np.linspace(0, 1, n)
    D_chirp = np.zeros((n, num_atoms))
    for i in range(num_atoms):
        f0 = np.random.uniform(5, 20)
        k = np.random.uniform(10, 100)
        D_chirp[:, i] = np.cos(2 * np.pi * (f0 * t + 0.5 * k * t**2))
    D_chirp /= np.linalg.norm(D_chirp, axis=0)
    return D_chirp

D_chirp = generate_chirp_atoms(n, n)

# Sub-dictionary 3: BPSK atoms
def generate_bpsk_atoms(n, num_atoms, fc=5, fs=100):
    t = np.arange(n) / fs
    D_bpsk = np.zeros((n, num_atoms))
    for i in range(num_atoms):
        bits = np.random.randint(0, 2, n)
        symbols = 2 * bits - 1
        D_bpsk[:, i] = symbols * np.cos(2 * np.pi * fc * t)
    # Normalize atoms
    D_bpsk /= np.linalg.norm(D_bpsk, axis=0)
    return D_bpsk
```

```
# Usage:
D_bpsk = generate_bpsk_atoms(n, n)
D_total = np.concatenate([D_dct, D_dct, D_chirp], axis=1)

# Compressed Measurement
m = int(input("Enter number of compressed samples (m < n): "))  # Compressed samples
Phi = np.random.randn(m, n)# / np.sqrt(k)  # Measurement matrix
y = Phi @ signal

# Add noise to the measurements
snr_db = int(input("Enter noise to be added (dB): "))  # Signal-to-noise ratio in dB
y_noisy = add_noise(y, snr_db)

# Construct sensing matrix (Theta)
A = Phi @ D_total

# Sparse Recovery using ISTA
#z = ista(y_noisy, A)
z = omp(y, A)
signal_hat = D_total @ z

# Calculate reconstruction error
error = np.linalg.norm(signal - signal_hat)
print(f"Reconstruction error (L2 norm): {error:.4f}")

# 9. Plot original and reconstructed
plt.figure(figsize=(10, 4))
plt.plot(t, signal, label="Original Mixed Signal")
plt.plot(t, signal_hat, '--', label="Reconstructed Signal (ISTA)")
plt.title("Mixed Signal Reconstruction from Compressed Measurements")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend()
plt.grid(True)
plt.tight_layout()
```

```
plt.show()
```

- **Creating Simulated Dataset**

```
class SimulatedData(Data.Dataset): #Creates tuple (x, H, s) for each sample
    def __init__(self, x, H, s):
        self.x = x
        self.s = s
        self.H = H

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        x = self.x[idx, :]
        H = self.H
        s = self.s[idx, :]
        return x, H, s

def create_data_set(H, n, m, k, N=1000, batch_size=512, signal_dev=0.5, noise_dev=0.01
    # Initialization
    x = torch.zeros(N, n)
    s = torch.zeros(N, m)

    # Create signals
    for i in range(N):
        # Create a k-sparsed signal s
        index_k = np.random.choice(m, k, replace=False)
        peaks = signal_dev * np.random.randn(k)

        s[i, index_k] = torch.from_numpy(peaks).to(s)

        # X = Hs+w
        x[i, :] = H @ s[i, :] + noise_dev * torch.randn(n)

    simulated = SimulatedData(x=x, H=H, s=s)
```

```
    data_loader = Data.DataLoader(dataset=simulated, batch_size=batch_size, shuffle=True)
    return data_loader

N = 1000 # number of samples
n = 150 # dim(x)
m = 200 # dim(s)
k = 4 # k-sparse signal
T_t = 5  # Number of iterations

# Measurement matrix
H = torch.randn(n, m)
H /= torch.norm(H, dim=0)

# Generate datasets
train_loader = create_data_set(H, n=n, m=m, k=k, N=N)
test_loader = create_data_set(H, n=n, m=m, k=k, N=N, batch_size=N)
```

- **LISTA Class**

```
class LISTA_Model(nn.Module):
    def __init__(self, n, m, L, T=6, rho=1.0, H=None):
        super(LISTA_Model, self).__init__()
        self.n, self.m = n, m
        self.H = H
        self.L = L
        self.T = T  # ISTA Iterations
        self.rho = rho  # Lagrangian Multiplier
        self.A = nn.Linear(n, m, bias=False)  # Weight Matrix
        self.B = nn.Linear(m, m, bias=False)  # Weight Matrix
        # ISTA Stepsizes eta
        self.beta = nn.Parameter(torch.ones(T + 1, 1, 1)/L, requires_grad=True)
        self.mu = nn.Parameter(torch.ones(T + 1, 1, 1)/L, requires_grad=True)
        # Initialization
        if H is not None:
            self.A.weight.data = H.t()
            self.B.weight.data = H.t() @ H
```

```
        '''for param in self.A.parameters(): # A needs no_grad (should not be trained)
            param.requires_grad = False
        for param in self.B.parameters():# B needs no_grad (should not be trained)
            param.requires_grad = False'''

    def _shrink(self, s, beta):
        return beta * F.softshrink(s / beta, lambd=self.rho)

    def forward(self, x, s_gt=None):
        mse_vs_itr = []

        s_hat = self._shrink(self.mu[0, :, :] * self.A(x), self.beta[0, :, :])
        for i in range(1, self.T + 1):
            s_hat = self._shrink(s_hat - self.mu[i, :, :] * self.B(s_hat) + self.mu[i,
                                 self.beta[i, :, :], )

            # Aggregate each iteration's MSE loss
            if s_gt is not None:
                mse_vs_itr.append(F.mse_loss(s_hat.detach(), s_gt.detach(), reduction=

        return s_hat, mse_vs_itr
```

- **LCoD Class**

```
class LearnedCoD(nn.Module):
    def __init__(self, H, T=10, learn_alpha=True):
        super(LearnedCoD, self).__init__()
        self.T = T
        self.H = H
        self.m = H.shape[1]

        # Fixed matrices (you can make them learnable if needed)
        self.B_mat = nn.Parameter(H.T.clone(), requires_grad=False)
        self.S_mat = nn.Parameter(torch.eye(self.m, dtype=torch.float64) - H.T @ H, re

        # Learnable thresholds
```

```
        if learn_alpha:
            self.alpha_list = nn.ParameterList([nn.Parameter(torch.tensor(0.05, dtype=
        else:
            self.alpha_list = [torch.tensor(0.05, dtype=torch.float64) for _ in range(

    def soft_threshold(self, B, alpha):
        return torch.sign(B) * torch.maximum(torch.abs(B) - alpha, torch.zeros_like(B)

    def forward(self, x, s_gt=None):
        if x.dim() == 2 and x.shape[1] == self.H.shape[0]:
            batch_size = x.shape[0]
            z_out = []
            mse_list = []

            for i in range(batch_size):
                z_i, mse_i = self.forward(x[i].unsqueeze(1), s_gt[i].unsqueeze(1) if s
                z_out.append(z_i.squeeze(1))  # (m, 1) -> (m,)
                if s_gt is not None:
                    mse_list.append(mse_i)

            z_out = torch.stack(z_out, dim=0)  # shape: (batch_size, m)
            if s_gt is not None:
                mse_avg = torch.stack(mse_list, dim=0).mean(dim=0)
                return z_out, mse_avg
            else:
                return z_out, None

    # Single sample mode (x shape = [n, 1])
        B = self.B_mat @ x  # shape: (m, 1)
        z = torch.zeros((self.m, 1), dtype=torch.float64, device=x.device)
        mse_vs_iter = []

        for t in range(self.T):
            z_bar = self.soft_threshold(B, self.alpha_list[t])
            k = torch.argmax(torch.abs(z - z_bar))
            delta = z_bar[k, 0] - z[k, 0]
```

```
            B = B + self.S_mat[:, [k]] * delta
            z[k, 0] = z_bar[k, 0]

            if s_gt is not None:
                mse = F.mse_loss(z.detach(), s_gt.detach(), reduction="sum").item()
                mse_vs_iter.append(mse)

        if s_gt is not None:
            return z, torch.tensor(mse_vs_iter, dtype=torch.float64)
        else:
            return z, None
```

- **Training Function**

def train(model, train_loader, valid_loader, num_epochs=50): “““Train a network. Returns: loss_test {numpy} – loss function values on test set”“” # Initialization optimizer = torch.optim.SGD(model.parameters(), lr=5e-05, momentum=0.9, weight_decay=0,) scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1) loss_train = np.zeros((num_epochs,)) loss_test = np.zeros((num_epochs,)) # Main loop for epoch in range(num_epochs): model.train() train_loss = 0 for step, (b_x, b_H, b_s) in enumerate(train_loader): s_hat, _ = model.forward(b_x) loss = F.mse_loss(s_hat, b_s, reduction=“sum”) optimizer.zero_grad() loss.backward() optimizer.step() model.zero_grad() train_loss += loss.data.item() loss_train[epoch] = train_loss / len(train_loader.dataset) scheduler.step()

```
    # validation
    model.eval()
    test_loss = 0
    for step, (b_x, b_H, b_s) in enumerate(valid_loader):
        # b_x, b_H, b_x = b_x.cuda(), b_H.cuda(), b_s.cuda()
        s_hat, _ = model.forward(b_x)
        test_loss += F.mse_loss(s_hat, b_s, reduction="sum").data.item()
    loss_test[epoch] = test_loss / len(valid_loader.dataset)
    # Print
    if epoch % 10 == 0:
        print(
```

```
            "Epoch %d, Train loss %.8f, Validation loss %.8f"
            % (epoch, loss_train[epoch], loss_test[epoch])
        )

return loss_test
```