RADAR SIGNAL PROCESSING USING COMPRESSED SENSING

Internship Report

Abhinav M Balakrishnan Arun Ramesh

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.

First and foremost, we express our sincere gratitude to Mrs. Usha P. Verma, Associate Director of ASL, DRDO, for the opportunity to intern at this esteemed organization.

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

The abstract goes here.

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. A signal is considered sparse if it has only a few nonzero coefficients when expressed in this basis. The choice of basis is crucial, as it determines the sparsity and thus the effectiveness of compressed sensing for a given signal.
- **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 vector of compressed measurements.

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.

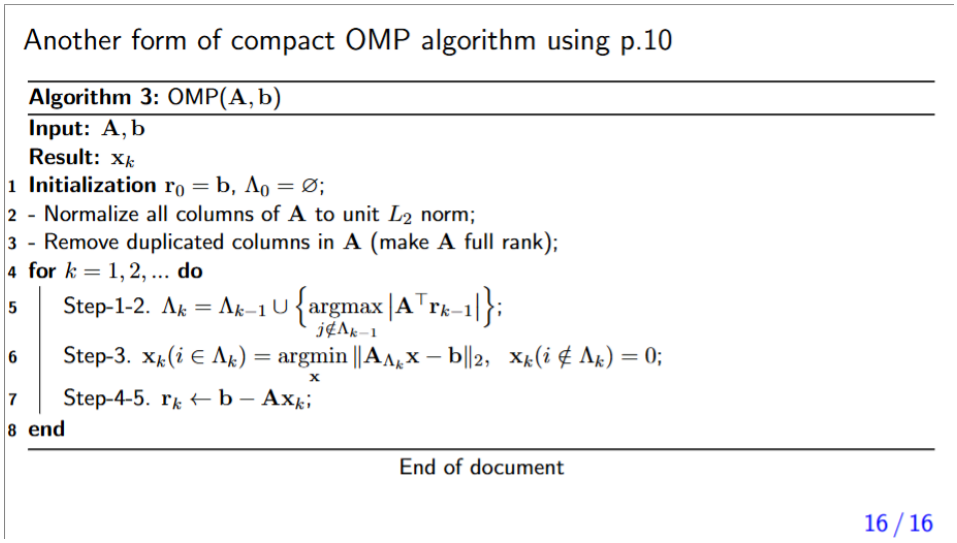
The goal of compressed sensing is to recover $\mathbf{x}$ (or $\mathbf{s}$) from the measurements $\mathbf{y}$, given knowledge of ϕ (and Ψ if applicable), by exploiting the sparsity of the signal.

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 in (Anderson 2022)


Another form of compact OMP algorithm using p.10

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

End of document

16 / 16

Figure 1: OMP Algorithm

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

3.1.1 Algorithm Implementation

- **Python**

The algorithm takes a sum of three sinusoidal signals as input and it undergoes Discrete Cosine Transform (DCT), to convert the input to its sparse domain. The code implementing the same is given as shown,

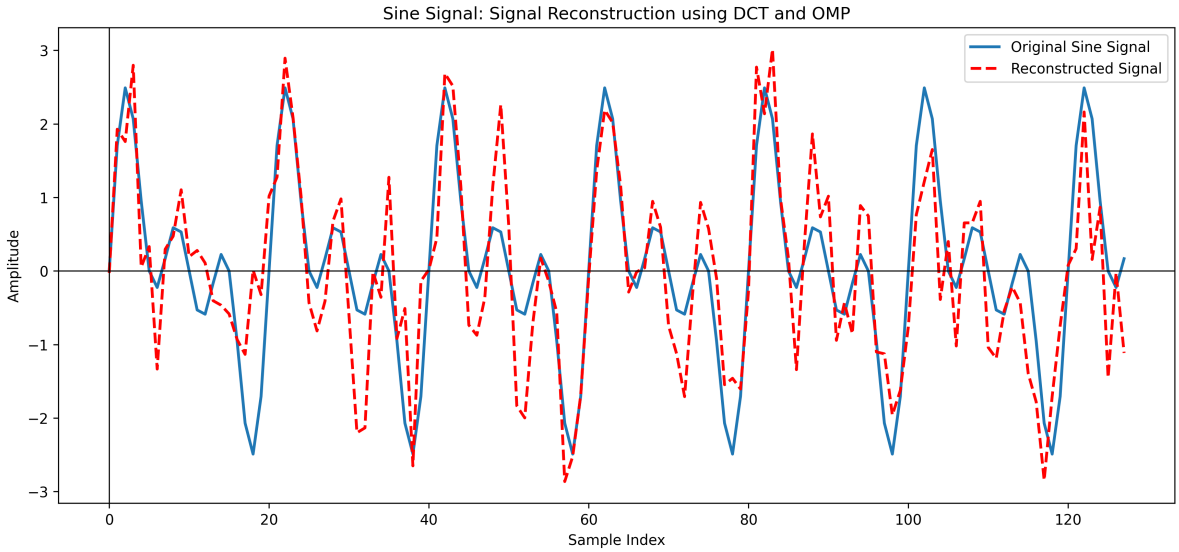
CODE:-

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

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

def measurement(m, n):
    Psi = idct(np.eye(n), norm='ortho')
    Phi = np.random.randn(m, n)
    return Phi @ Psi

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

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

n = 128
m = 64
k = 3
snr_db = 10

x_time = generate_sine_signal(n, k, 5)
x_sparse = dct(x_time, norm='ortho')
A = measurement(m, n)
y = A @ x_sparse
y_noisy = add_noise(y, snr_db)
x_sparse_rec = omp(y, A)
x_time_rec = idct(x_sparse_rec, norm='ortho')

print("\nReconstruction error (L2 norm):", norm(x_time - x_time_rec))

plt.figure(figsize=(14, 6))
plt.plot(x_time, label="Original Sine Signal", linewidth=2)
plt.plot(x_time_rec, '--r', label="Reconstructed Signal", linewidth=2)
plt.title("Sine Signal: Signal Reconstruction using DCT and OMP")
plt.xlabel("Sample Index")
plt.ylabel("Amplitude")
plt.legend()
plt.axhline(0, color='black', linewidth=0.8)
```

```
plt.axvline(0, color='black', linewidth=0.8)
plt.show()
```

```
Reconstruction error (L2 norm): 8.858774504014669
```



- **MATLAB**

Here, the sum of two sinusoids is taken as input and it is made sparse using the built-in dft matrix taken as the basis.

CODE:-

```
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

n = 256;
m = 50;
k = 2;

freqs = [randi([1, 10]),randi([1, 10])];
x_freq = zeros(n, 1);
x_freq(freqs) = [1; 1];
x_time = real(ifft(x_freq))*n;

psi = randn(m,n);
b = psi * x_time;

phi = dftmtx(n);
Theta = psi * phi';

x_freq_rec = omp(Theta, b, k);
x_rec = real(ifft(x_freq_rec))*n;

figure;
```

```
t = 0:n-1;

% Plot the original and recovered signals smoothly
plot(t, x_time, 'b-'); hold on;
plot(t, x_rec, 'r--');
legend;
xlabel('Index');
ylabel('Amplitude');
title('OMP (sinusoidal)');
```

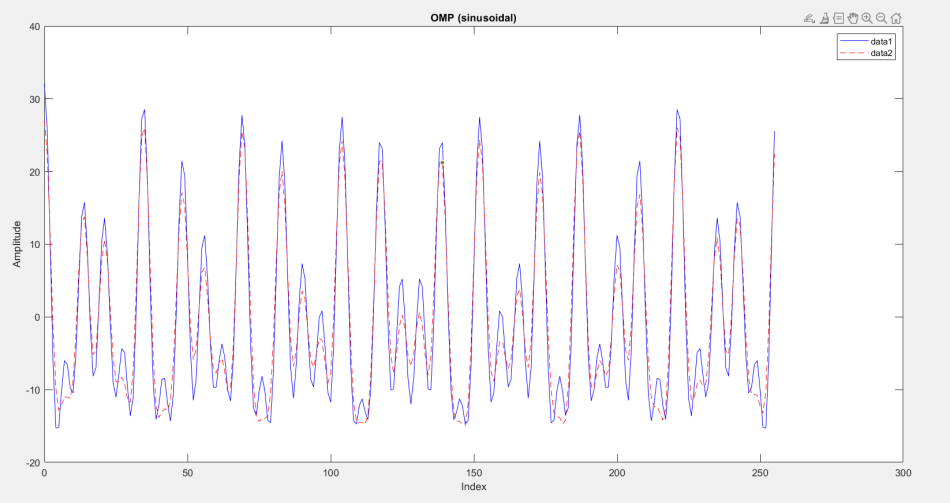


Figure 2: OMP Signal Reconstruction: Original vs Reconstructed Signal

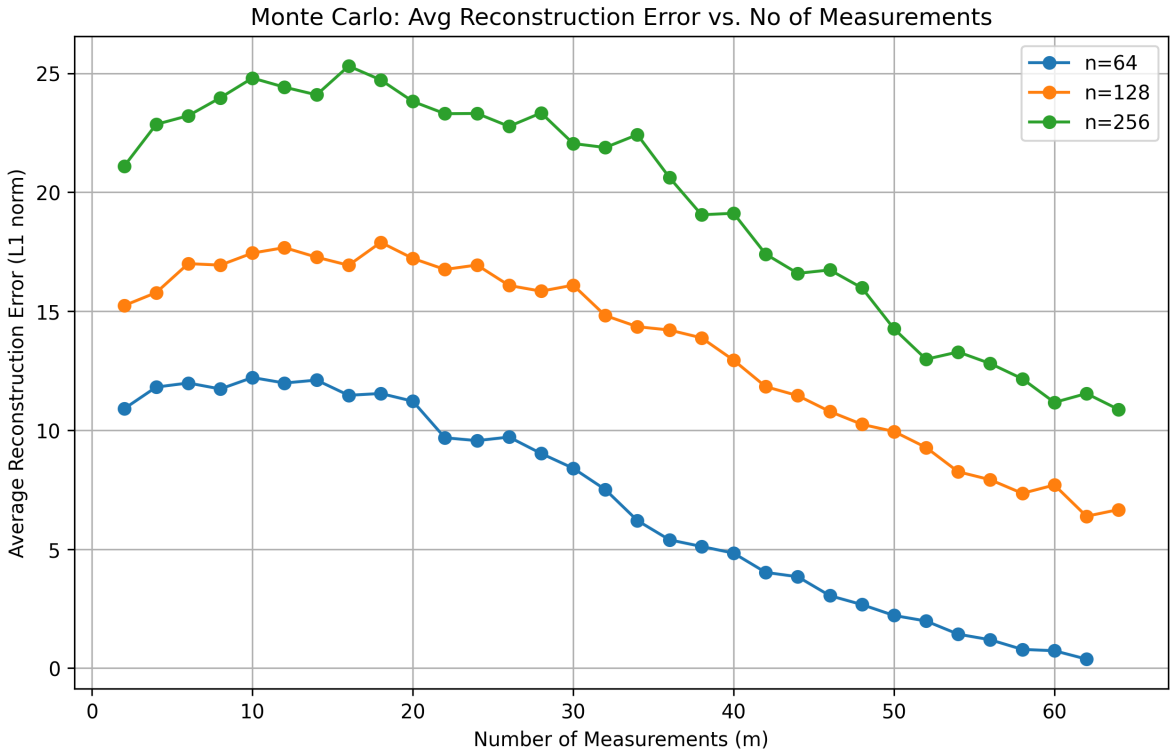
3.1.2 Monte-Carlo Simulation

Monte Carlo methods 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. The code implementing these simulations are given below,

- **Measurements**

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

def generate_sine_signal(n, fs=100):
    t = np.arange(n) / fs
    return np.sin(2*np.pi*5*t)+np.sin(2*np.pi*10*t)+np.sin(2*np.pi*20*t)

def measurement(m, n):
    Psi = idct(np.eye(n), norm='ortho')
    Phi = np.random.randn(m, n)
    return Phi @ Psi

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))
        if idx not in idx_set:
            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
    x_hat[idx_set] = x_ls
    return x_hat

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
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}')

plt.title("Monte Carlo: Avg Reconstruction Error vs. No of Measurements")
plt.xlabel("Number of Measurements (m)")
plt.ylabel("Average Reconstruction Error (L1 norm)")
plt.legend()
plt.grid(True)
plt.show()
```

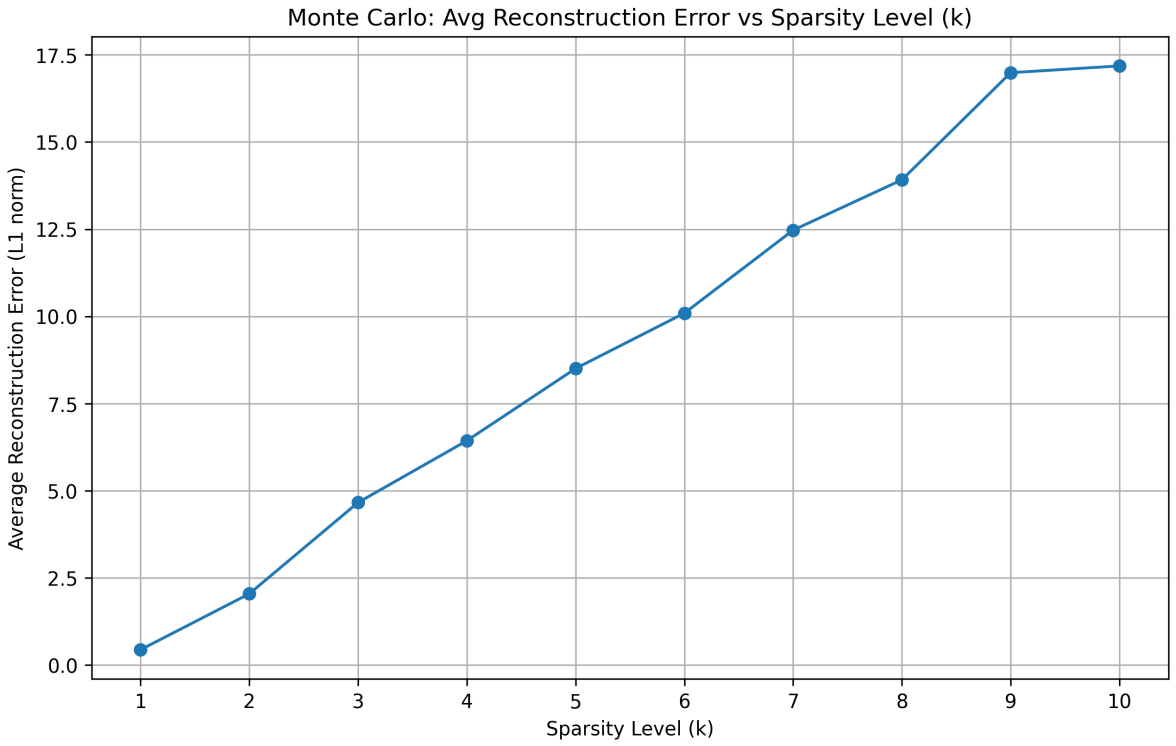


- **Sparsity**

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

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

def measurement(m, n):
    Psi = idct(np.eye(n), norm='ortho')
```

```
    Phi = np.random.randn(m, n)
    return Phi @ Psi

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))
        if idx not in idx_set:
            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
    x_hat[idx_set] = x_ls
    return x_hat

def monte_carlo_trial(sampling_rate, k, n=128, m=80):
    x_time = generate_sine_signal(n, k, 5, 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
```

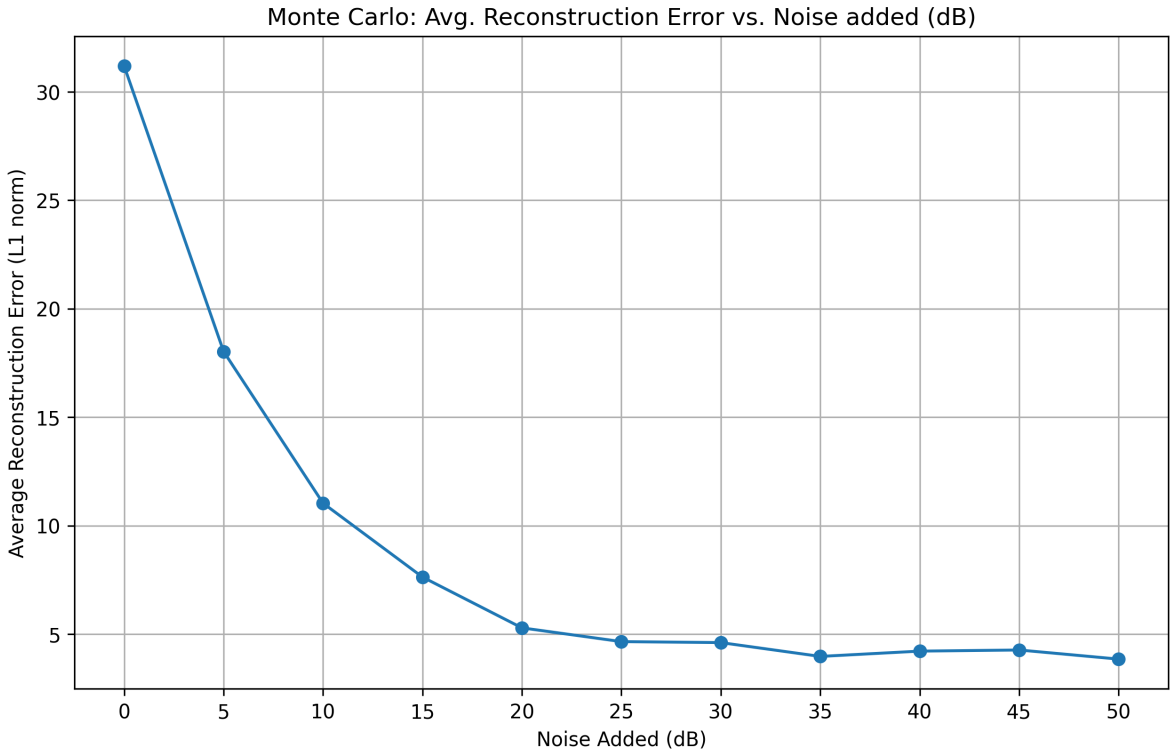
```
plt.figure(figsize=(10, 6))
avg_errors = []
for k in k_values:
    errors = []
    for _ in range(num_trials):
        errors.append(monte_carlo_trial(sampling_rate, k))
    avg_errors.append(np.mean(errors))

plt.plot(k_values, avg_errors, marker='o')
plt.title("Monte Carlo: Avg Reconstruction Error vs Sparsity Level (k)")
plt.xlabel("Sparsity Level (k)")
plt.xticks(k_values)
plt.ylabel("Average Reconstruction Error (L1 norm)")
plt.grid(True)
plt.show()
```



- **Noise**

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

def generate_sine_signal(n, freq=5, fs=100):
    t = np.arange(n) / fs
    return np.sin(2*np.pi*freq*t)+np.sin(2*np.pi*10*t)+np.sin(2*np.pi*20*t)

def measurement(m, n):
    Psi = idct(np.eye(n), norm='ortho')
    Phi = np.random.randn(m, n)
    return Phi @ Psi

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))
        if idx not in idx_set:
            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
    x_hat[idx_set] = x_ls
    return x_hat

def monte_carlo_trial(sampling_rate, snr_db, n=128, m=80):
    x_time = generate_sine_signal(n, 5, sampling_rate)
    x_sparse = dct(x_time, norm='ortho')
    A = measurement(m, n)
    y = A @ x_sparse
    y_noisy = add_noise(y, snr_db)
```

```
    x_sparse_rec = omp(y_noisy, A)
    x_time_rec = idct(x_sparse_rec, norm='ortho')
    error = norm(x_time - x_time_rec, ord=2)
    return error

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

# ---- 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
sampling_rate = 100

plt.figure(figsize=(10, 6))
avg_errors = []
for noise in noise_val:
    errors = []
    for _ in range(num_trials):
        errors.append(monte_carlo_trial(sampling_rate, noise))
    avg_errors.append(np.mean(errors))

plt.plot(noise_val, avg_errors, marker='o')
plt.title("Monte Carlo: Avg. Reconstruction Error vs. Noise added (dB)")
plt.xlabel("Noise Added (dB)")
plt.xticks(noise_val)
plt.ylabel("Average Reconstruction Error (L1 norm)")
plt.grid(True)
plt.show()
```



3.2 ITERATIVE SHRINKAGE THRESHOLDING ALGORITHM (ISTA)

To be continued……

CHAPTER 4: CONCLUSION

Summary and conclusion go here…

REFERENCES

References

Anderson, Ang. 2022. *Orthogonal Matching Pursuit Algorithm: A Brief Introduction*.