Mini projet:

Compressed sensing for ECG signals acquisition

I- Nyquist-Shannon sampling theorem vs Compressed Sensing

The Nyquist-Shannon Theorem

The Nyquist-Shannon sampling theorem states, "To reconstruct a signal, the sampling rate must be at least twice the highest frequency present in the signal”. This ensures that no information is lost during sampling. If the sampling rate is not sufficiently high, fast oscillations or high-frequency signal components will not be accurately represented in the digital reconstruction. This is because the sampling frequency is too low to capture the rapid changes in the signal.

The Nyquist-Shannon Theorem assumption vs CS assumption

In The Shannon-Nyquist theorem, we are limited by the signal bandwidth. In this case, this theorem assumes that the signal being sampled is band-limited, meaning that all signal frequencies are contained within a certain range ([-Fsn, +Fsn]). Yet, when the signal is compressible, only a few frequencies matter. In the Compressed Sensing problem, we’re not trying to reconstruct a band-limited signal, but a signal that has a sparse representation. In other words, the solution we’re trying to solve is sparse.

There are two main assumptions for the CS: sparsity & the compressibility of the natural signal.

1-Sparsity :

Sparsity is the property of a signal where only a small subset of its coefficients significantly contribute to its overall behavior. A signal is called sparse if most of its components are zero, except for a few significant ones.

Definition 1 :

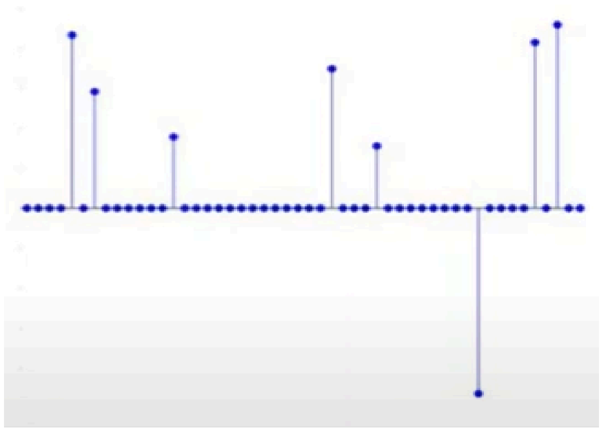
The support of a vector x E IR^n, x = (x1, x2,..., xn) is the index set of its non-zero entries, i.e.,

$$supp(x) = \{j \text{ E } [|1,n|], xj \text{ != } 0\}$$

Definition 2 :

A signal x E IR^n is said to be k-sparse if at most k of its components are non-zero, i.e.,

$$||x||0 = card\ (supp(x)) =< k$$



$\|x\|_0$ This notation isn't a norm. It only counts the non-zero elements of the vector x.

The problem of sparsity often involves knowing which elements in the vector are non-zero and where they are located within the structure (a combinatorial problem).

2-The compressibility of natural signal :

This compressibility lies in the rarity of natural signals within the vast space of randomness.

To illustrate, consider a space of images with dimensions of 20x20 pixels, where each pixel is binary, either black or white.

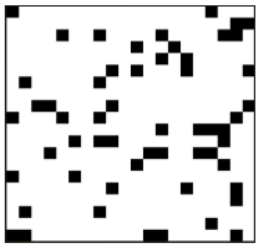


Figure: Image 20x20 (black and white)

How many possible images can exist in this space?

Calculation gives us a 2^(20x20) image. This number is bigger than the total number of nucleons in the universe which is 10^(80). This comparison shows how large this space is.

Figure: Illustration of pixel space

Moreover, expanding the pixel choices to include more than just black and white => The pixel space is a vast space of randomness.

If we take a random image, it will be a noise. This is because every pixel can be randomly chosen and independent of the other pixels.

Yet, in natural images, pixels cannot be randomly chosen for the picture to have a natural pattern.

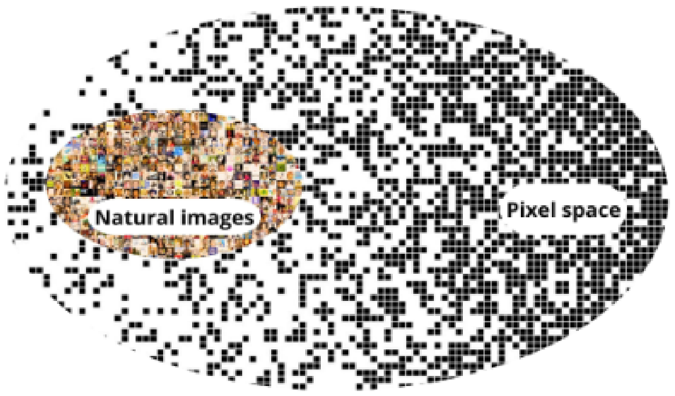


Figure: Illustrative comparison between image space and natural images

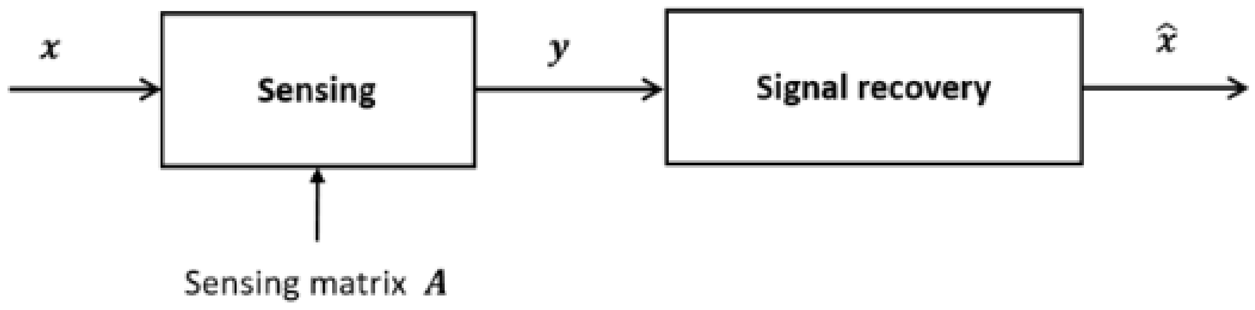
This implies that natural images only occupy a small fraction of the pixel space. This rarity of natural signals in the pixel space is the essence of their compressibility.

II- An Overview of Compression

A signal is a sequence of coefficients (discrete or continuous). Compression aims to reduce the amount of data (coefficients) needed to represent the signal while preserving its essential features. In classical compression, we collect high-dimensional data to compress and discard most of the information. For instance, to compress an audio file, we transform it into the transform domain, retain just 1% of the information, and discard the remaining 99% of it. The intuition of CS is “What if we just collect the right information to begin with”. In Compressed sensing, we collect a few compressed or random measurements and then infer the sparse representation in the transformed basis (TD). This offers numerous advantages. In terms of storage, storing only the significant coefficients reduces memory requirements. Moreover, during transmission, transmitting fewer coefficients saves bandwidth. Additionally, sparse representations allow faster computations (e.g., denoising, reconstruction) in the realm of processing.

III- Analytical Problem

The CS process comprises two main steps, acquisition or sensing, and signal recovery.



1- Sensing :

Let's consider the following system: **Y = A.X**, where :

- Y E IR^m
- X E IR^n is a sparse signal
- A E IR^{mxn} is the sensing matrix
- m << n where (m,n) E INxIN.

Notice that “m" is the number of measurements taken Y, and "n" represents the size of the original signal X.

So we have the vector of measurements Y = (y1,.., ym) and for each i E [|1, m|] : yi = <Ai, x>

a) Properties of the measurements y1, y2..., ym:

a.1) Non-adaptive :

The measurements are taken without any prior knowledge of the signal. And, the measurement process does not change or adapt based on previous measurements.

a. 2) No causality :

The measurement process does not depend on the order or any sequential information. Each measurement is independent of the others.

a. 3) Not supposed to match the signal structure at all:

The measurements should not be specifically tailored to the characteristics or structure of the signal.

a. 4) Should look like a random noise (the noisier they look like the better):

This randomness helps ensure that they capture diverse and independent pieces of information about the signal, which is crucial for effective reconstruction.

a. 5) The yi are equally important => Democratic:

This "democratic" property means that no single measurement is more critical than the others, ensuring a balanced and fair representation of the signal.

a. 6) Losing a few does not affect the reconstruction => Robust:

The measurement system can tolerate the loss of some measurements without significantly degrading the quality of the signal reconstruction.

a. 7) The number of measurements increases => Reconstruction error decreases:

As the number of measurements increases, the error in the reconstructed signal should decrease.

b) The Sensing matrix design: When the CS is working?

The sensing matrix should obey the following conditions:

- The measurement matrix A satisfies the RIP
- Coherence: Lower coherence between the columns of A improves the performance of sparse recovery.

Given a sensing matrix A $\in$ IR^(m×n) with columns $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$,.

b.1) Definition of RIP :

The matrix A said to satisfy the RIP of order k if there exists a constant δk ∈ (0,1) (called the restricted isometry constant) such that for all k-sparse vectors X :

$$(1-\delta_k)\|\mathbf{x}\|_2^2 \leq \|\mathbf{Ax}\|_2^2 \leq (1+\delta_k)\|\mathbf{x}\|_2^2$$

b.2) Interpretation of RIP :

The RIP ensures that the measurement matrix A approximately preserves the Euclidean (or $\ell 2$) norm of any k-sparse vector X. This implies that A does not significantly distort the length of sparse vectors when they are projected into the lower-dimensional measurement space. Here's why this property is important:

- **Lower Bound (1- δk) :** The (1- δk)(// $\mathbf{x}$ // 2)² means that the measurements process does not compress the signal too much

- **Upper Bound (1+ δk)** : The (1+ δk)(// $\mathbf{x}$ // 2)² means that the measurements process does not expand the signal too much

In essence, RIP ensures that the structure of the sparse signal is retained after transformation by A.

b.1) Definition of Coherence :

The coherence μ(A) of the matrix A is defined as:

$$\mu(A) = \max_{1 \leq i < j \leq n} \frac{|\langle \mathbf{a}_i, \mathbf{a}_j \rangle|}{\|\mathbf{a}_i\|_2 \|\mathbf{a}_j\|_2}$$

Here:

- $\langle ai, aj \rangle$ denotes the inner product between columns ai and aj.
- // ai‖2 is the Euclidean norm (or $\ell 2$-norm) of column ai.

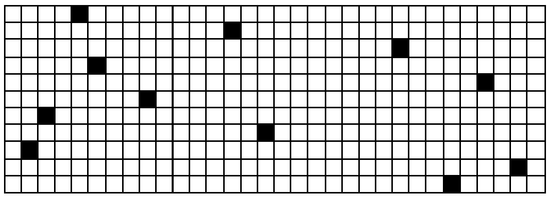
b.2) Interpretation of Coherence :

The coherence μ(A) ranges from 0 to 1. When the coherence is nearly zero, the columns of A are nearly orthogonal. This helps in distinguishing different sparse representations of the signal. On the opposite, when the coherence is nearly one, the columns of A are highly similar. This can lead to ambiguity in recovering the sparse signal, as different sparse combinations might explain the measurements equally well.

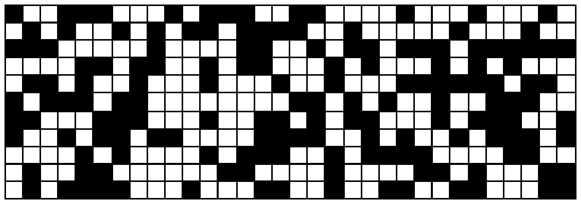
=> To conclude, lower coherence is desirable because it improves the performance of sparse recovery algorithms.

Here are some examples of good random measurement matrices :

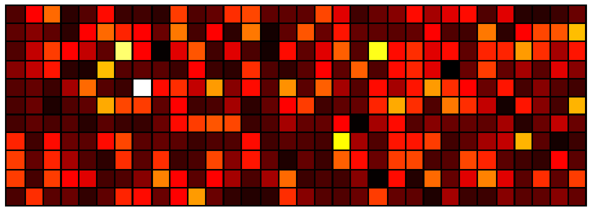
- Random Single Pixel: Each row of the matrix contains only one non-zero element, randomly positioned.



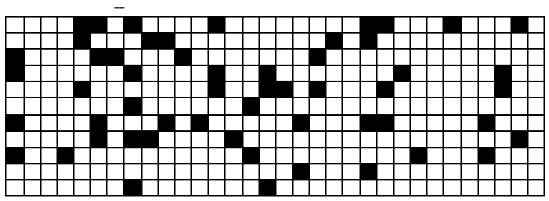
- Bernoulli Matrix: It is created by randomly assigning each element to either -1 or 1 with equal probability.



- Gaussian Random Matrix: It is created by sampling each element from a Gaussian distribution.

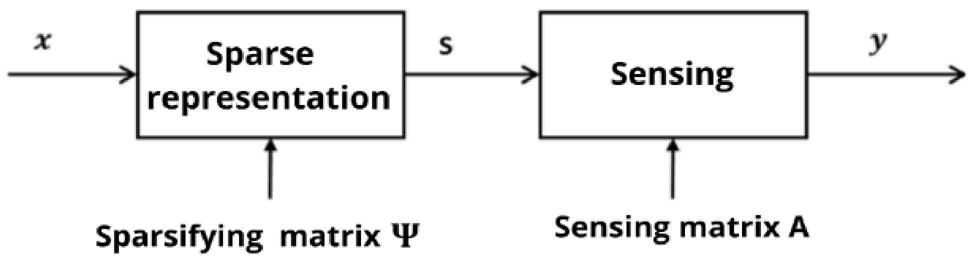


- Sparse Random Matrix: A sparse random matrix is similar to a Gaussian random matrix but with a limited number of non-zero elements per column.



2- Sparsification :

Natural signals may often not be sparse in their representation in the original domain. One common approach is to transform these signals into a domain where they exhibit sparsity.



Sparsification refers to transforming a non-sparse signal into a sparse signal representation on a suitable basis.

The sparsification equation can be written as: **X = Ψ.S**, where :

- S is a sparse signal with the same dimension as X.
- Ψ E IR^(nxn) represents the basis of the transform domain.

In consequence, the sensing system is reformulated as **Y = A.Ψ.S**.

Let's consider **Φ = A.Ψ**, where Φ E IR^(mxn). Accordingly, the resulting system is **Y = Φ.S**.

a) Transform domain (TD):

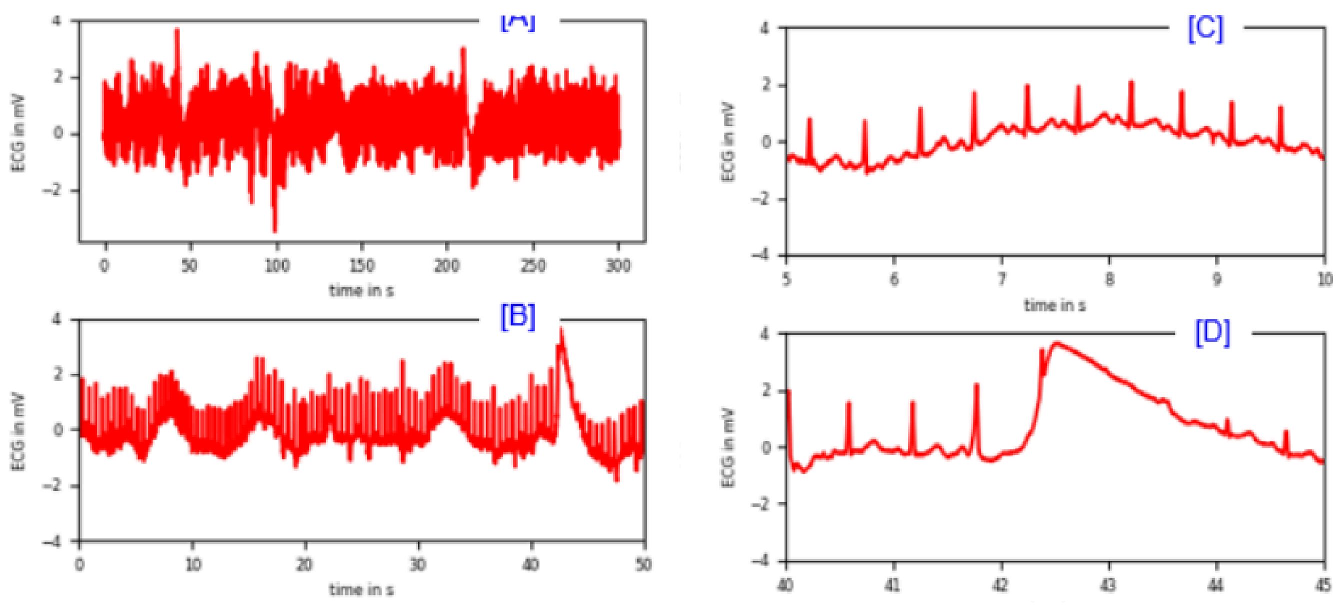
The transform domain plays a crucial role in CS because it provides the basis functions that can efficiently represent the original signal with a minimal number of non-zero coefficients.

The TD refers to the domain in which the signal exhibits sparsity. Some common transform domains include:

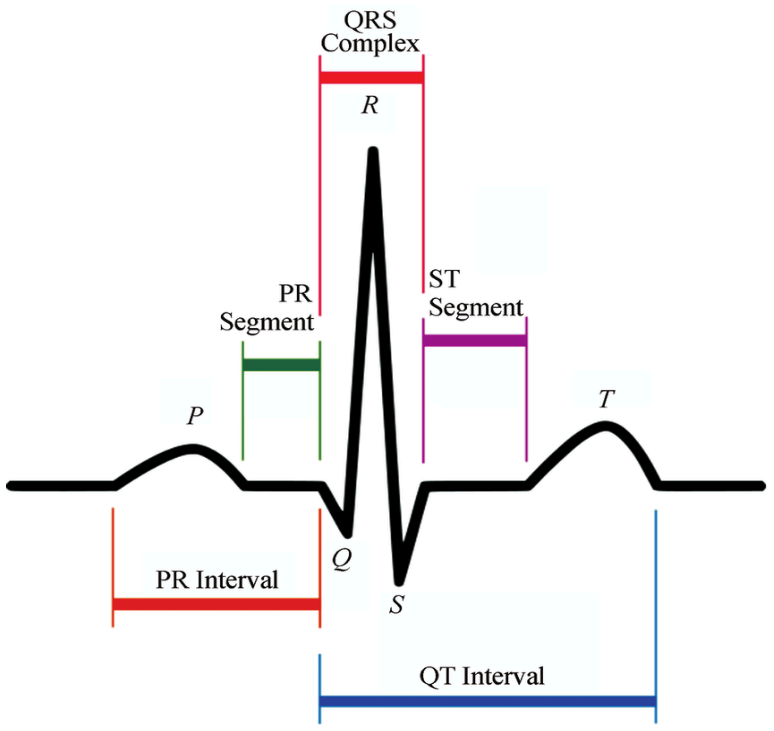
- Fourier Transform, Wavelet Transform, Discrete Cosine Transform (DCT), etc…

Question: Which transform is suitable for sparsifying ECG signals?

Let’s see some examples of noisy ECG signals :



ECG signals capture the electrical activity of the heart over time, characterized by distinct waveforms : P wave, QRS complex, and T wave.



These waves can vary due to factors like physical activity, emotional state, and health conditions.

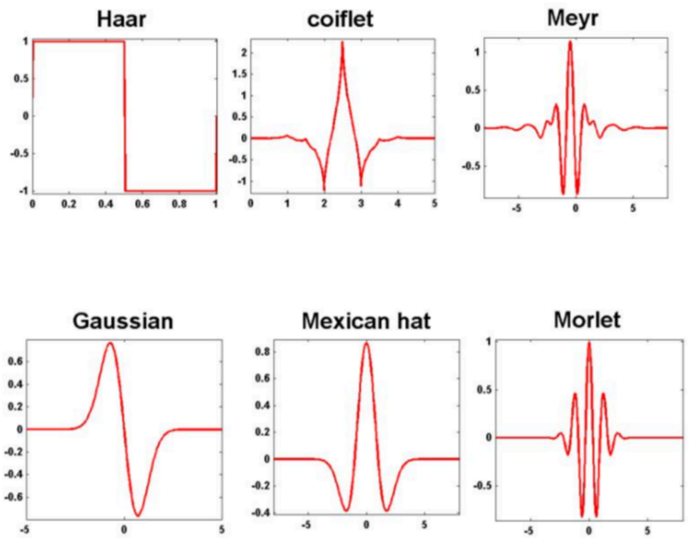
As a consequence, these features reflect various cardiac activities essential for diagnosing heart conditions. This highlights the non-stationary nature of ECG signals.

In this case, the wavelet transform is particularly preferred due to its ability to capture both the time and frequency characteristics of the signal, which aligns well with the transient nature of ECG signal features.

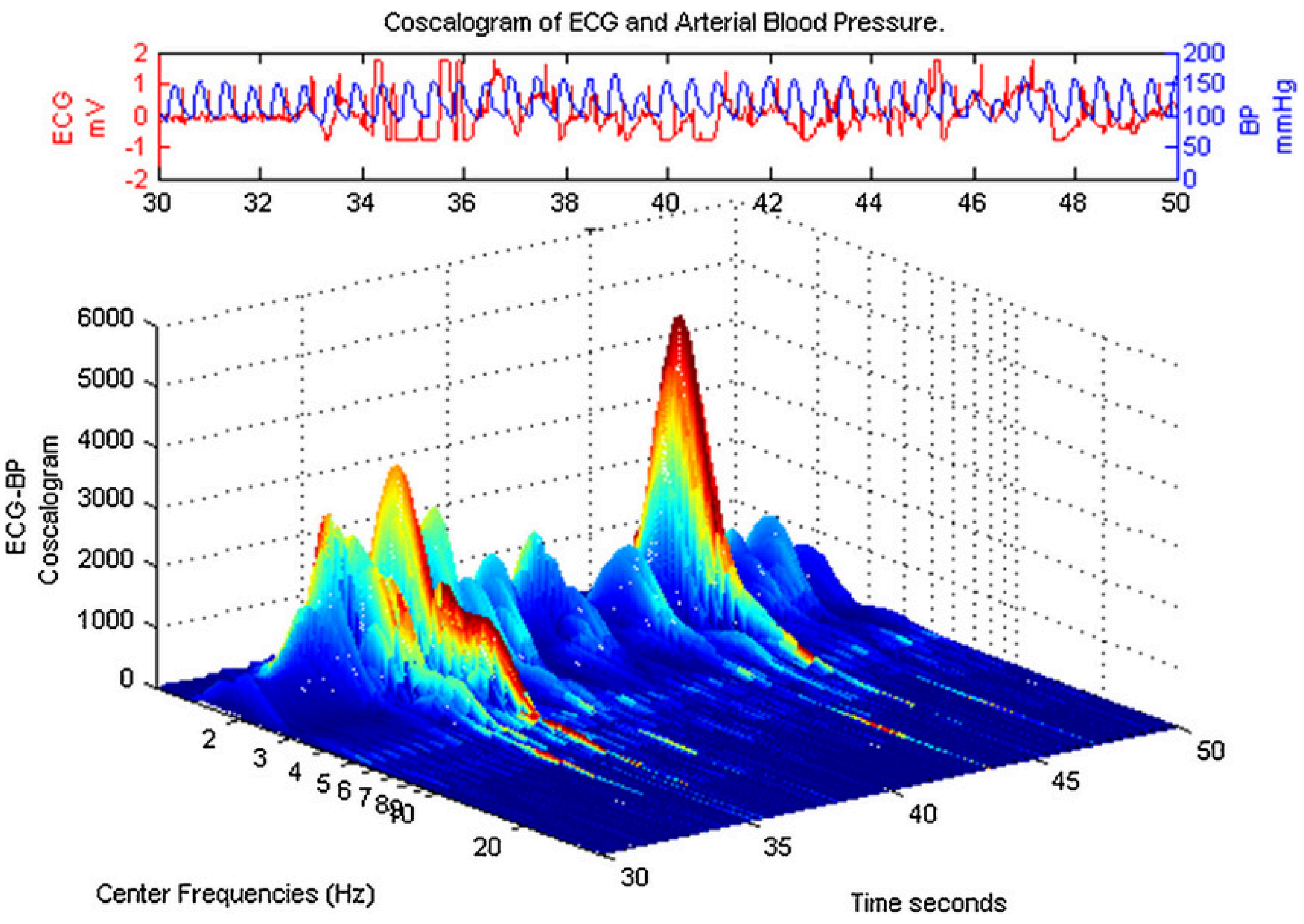
Here are the primary reasons:

- Time-Frequency Localization: Wavelet Transform provides good localization in both time and frequency domains. It can analyze the signal at different scales (resolutions) and adapt to changes in the signal, capturing both short-duration high-frequency components and long-duration low-frequency components.

$$F(\tau, s) = \frac{1}{\sqrt{|s|}} \int_{-\infty}^{+\infty} f(t)\, \psi^*\left(\frac{t-\tau}{s}\right) dt$$



- Non-Stationary Signal Analysis: Wavelet Transform is particularly well-suited for non-stationary signals whose frequency content changes over time.



3- Implementation of the Compressed Sensing (CS) Sensing Step with Python

a) Generating a Sparse Signal (signal with a majority of its components being zero)

```python
import numpy as np

def generating_sparse_signal(n, k):
    """
    Generating a sparse signal of length n with a given sparsity level k.
    """
    x = np.zeros(n)
    x[np.random.choice(n, k, replace=False)] = np.random.randn(k)
    return x

# Defining funcion's parameters
Length_signal = 1000  # Length of the signal
k_sparsity = 50  # Number of non-zero elements

# Generating the sparse signal
sparse_signal = generating_sparse_signal(Length_signal, k_sparsity)
print(sparse_signal)
```

b) Creating a Sensing Matrix (we’re using Gaussian random matrix).

```python
def generating_sensing_matrix(m, n):
    """
    Generating a sensing matrix of size m x n.
    """
    ## Sparse random matrix
    # A = np.zeros((m, n))
    # num_nonzero_elements = int(sparsity * m)
    # for i in range(n):
    #     idx = np.random.choice(m, num_nonzero_elements, replace=False)
    #     A[idx, i] = np.random.randn(num_nonzero_elements)
    # return A
    ## Random single pixel matrix
    # A = np.zeros((m, n))
    # for i in range(m):
    #     j = np.random.randint(0, n)
    #     A[i, j] = np.random.randn()
    # return A

    ## Bernoulli matrix
    # return np.random.choice([-1, 1], size=(rows, cols))

    ## Gaussian random matrix
    return np.random.randn(m, n)

# Parameters
nb_measurements = 300  # Number of measurements (M < N)

# Generate the sensing matrix
sensing_matrix = generating_sensing_matrix(nb_measurements, Length_signal)
print(sensing_matrix)
```

c) Acquire Measurements (Multiplication of the sparse signal by the sensing matrix to obtain the measurements)

```
measurements = sensing_matrix.dot(sparse_signal)

# Print the shapes to verify
print("Sensing matrix shape:", sensing_matrix.shape)
print("Sparse signal shape:", sparse_signal.shape)
print("Measurements shape:", measurements.shape)
```

```
Sensing matrix shape: (300, 1000)
Sparse signal shape: (1000,)
Measurements shape: (300,)
```

Full Implementation (A is a Gaussian random matrix) :

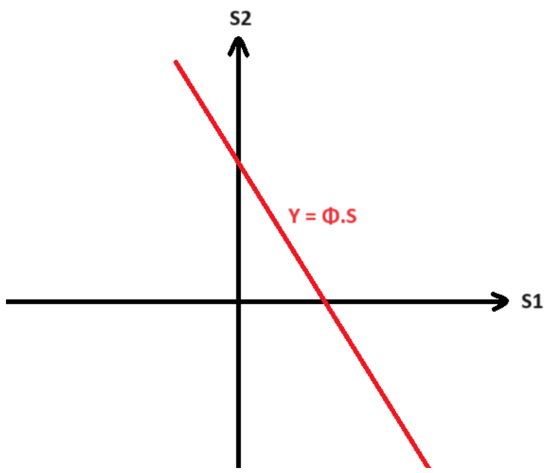
```python
import numpy as np

def generating_sparse_signal(n, k):
    x = np.zeros(n)
    x[np.random.choice(n, k, replace=False)] = np.random.randn(k)
    return x

def generating_sensing_matrix(m, n):
    return np.random.randn(m, n)

Length_signal, k_sparsity, nb_measurements = 1000, 50, 300
sparse_signal = generating_sparse_signal(Length_signal, k_sparsity)
sensing_matrix = generating_sensing_matrix(nb_measurements, Length_signal)

measurements = sensing_matrix.dot(sparse_signal)
```

4- Signal recovery

The idea of CS is to be able to recover the signal X E IR^n from a few samples or measurements

Y E IR^m when m << n if X admits a k-sparse representation is some orthogonal basis Ψ.

The phrase "m is too small compared with n" means that the system has more unknowns than equations. As a consequence, the previous system of equations **Y = Φ.S (*)** is called an **"Underdetermined system of equations"**. In this case, we can have infinite solutions that satisfy (*).



However, compressed sensing aims to identify the sparsest vector **S** that aligns with the measurement **Y**. To achieve this unique solution, we must apply a constraint, which leads us to the concept of norm minimization.

a) Norm minimization :

To find the true sparse solution, we rely on norm minimization techniques that involve finding a solution X that minimizes a certain norm, under the constraint that the solution fits the observed measurements Y.

The previous two conditions mentioned “Low Coherence” and “RIP” lead to using the $\ell 1$ norm minimization.

a.1) ℓ1 Norm Definition :

The ℓ1 norm of a vector x is defined as the sum of the absolute values of its elements:

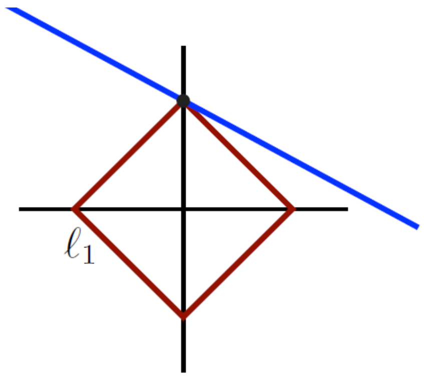
$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$ where $x=[x1,x2,\ldots,xn]T$ is an n-dimensional vector.

a.2) ℓ1 Norm Minimization Formulation:

Here’s the formulation: **min // x // 1 subject to Ax=y.**

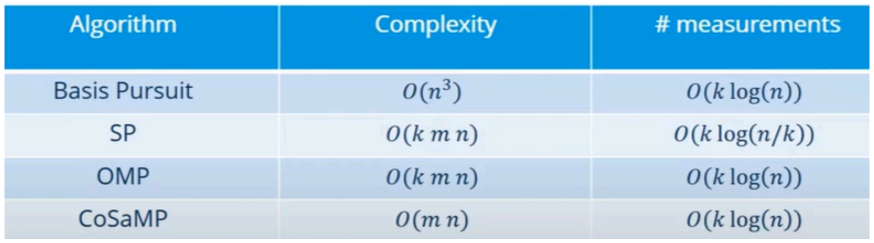
This norm promotes sparsity and this problem can be solved efficiently using linear programming techniques. This is because:

- Convexity: The ℓ1 norm is a convex function, which makes the minimization problem a convex optimization problem. Convex optimization problems are well-studied and can be solved efficiently using established algorithms.
- Sparsity Encouragement: The ℓ1 norm tends to shrink coefficients towards zero. This behavior is due to the geometry of the ℓ1 norm, which prefers solutions with fewer non-zero entries.



b) Practical Algorithms:

Our sparse recovery problem is an optimization problem. Several algorithms have been developed to solve the $\ell 1$ norm minimization problem. Yet, the algorithm choice depends on the System's specification e.g. tolerance to errors, recovery speed, number of available measurements, etc.



Algorithm	Complexity	# measurements
Basis Pursuit	$O(n^3)$	$O(k\log(n))$
SP	$O(k\,m\,n)$	$O(k\log(n/k))$
OMP	$O(k\,m\,n)$	$O(k\log(n))$
CoSaMP	$O(m\,n)$	$O(k\log(n))$

a.1) Basis Pursuit (BP):

Solves the exact $\ell 1$ norm minimization problem: **$\min // x //_1$ subject to $Ax=y$.**

```
import numpy as np
import cvxpy as cp
'''
cvxpy is a Python library designed for convex optimization.
It provides a simple and intuitive way to formulate and solve
convex optimization problems.


In the context of convex optimization and the cvxpy library,
the Variable class is a fundamental component used to represent
decision variables. Decision variables are the variables that need
to be optimized in an optimization problem. They can take on
different values subject to constraints to minimize or maximize
an objective function."
'''

def basis_pursuit(Sensing_matrix, y):
    n = Sensing_matrix.shape[1]
    solution = cp.Variable(n)
    objective = cp.Minimize(cp.norm(solution, 1))
    constraints = [Sensing_matrix @ solution == y]
    problem = cp.Problem(objective, constraints)
    problem.solve()
    return solution.value
```

```python
# Example usage
A = np.random.randn(50, 100)  # Sensing matrix
X = np.zeros(100)
X[np.random.choice(100, 7, replace=False)] = np.random.randn(7)  # Sparse signal
Y = A @ X  # Measurements

print("Recovered signal:", basis_pursuit(A, Y))
```

a.2) Basis Pursuit Denoising (BPDN):

Basis Pursuit Denoising (BPDN) is a technique for recovering clean signals from noisy observations.

Problem Formulation: **minimize** $\| x\|_1$ **subject to** $\| \mathbf{Ax}-\mathbf{y} \|_2 \leq \epsilon$, here:

- $\| Ax-y\|_2$ represents the $\ell 2$-norm (Euclidean norm) of the residual between the reconstructed signal Ax and the noisy observation y.
- ϵ is a noise threshold parameter that controls the level of denoising.

```python
def BPDN(Sensing_matrix, y, erreur):
    n = Sensing_matrix.shape[1]

    # Define variables
    solution = cp.Variable(n)

    # Define objective function
    objective = cp.Minimize(cp.norm(solution, 1))

    # Define constraints
    constraints = [cp.norm(Sensing_matrix @ solution - y, 2) <= erreur]

    # Define problem
    problem = cp.Problem(objective, constraints)

    # Solve problem
    problem.solve()

    return solution.value
```

```
A = np.random.randn(50, 100)  # Sensing matrix
X = np.zeros(100)
X[np.random.choice(100, 7, replace=False)] = np.random.randn(7)  # Sparse signal
Y = A @ X  # Measurements

# Set epsilon value based on noise standard deviation
epsilon = 0.1 * np.sqrt(len(y))

print("Recovered signal:", BPDN(A, Y, epsilon))
```

a.3) Greedy Algorithms (MP, OMP, CoSaMP) :

a.3.1) Matching Pursuit (MP)

Matching Pursuit is the simplest of the three algorithms. It iteratively selects the dictionary element (column of the sensing matrix) that has the highest correlation with the current residual (initially, the signal itself). It then updates the residual by subtracting the projection of the signal onto this dictionary element.

Algorithm:

1. Initialize the residual $\mathbf{r}_0 = \mathbf{y}$.
2. For each iteration k:
    a. Find the column of A that is most correlated with $\mathbf{r}_k$.
    b. Update the signal estimate.
    c. Update the residual.

Implementation of the signal recovery using MP in Python:

```python
import numpy as np

def matching_pursuit(A, y, max_iterations):
    residual = y
    index_set = []
    x_estimated = np.zeros(A.shape[1])

    for _ in range(max_iterations):
        # Find the column with the highest correlation with the residual
        correlations = A.T @ residual
        best_index = np.argmax(np.abs(correlations))
        index_set.append(best_index)

        # Update the estimated signal
        x_estimated[best_index] += correlations[best_index]

        # Update the residual
        residual = y - A @ x_estimated

    return x_estimated

A = np.random.randn(50, 100)  # Sensing matrix
X = np.zeros(100)
X[np.random.choice(100, 7, replace=False)] = np.random.randn(7)  # Sparse signal
Y = A @ X  # Measurements

# Set iteration value
iterations = 20

print("Recovered signal:", matching_pursuit(A, Y, iterations))
```

a.3.1) Orthogonal Matching Pursuit (OMP)

OMP improves upon MP by ensuring that the selected atoms (columns) remain orthogonal to the residual. This is achieved by orthogonalizing the selected atoms at each iteration, which generally leads to better convergence properties.

Algorithm:

1. Initialize the residual $\mathbf{r}_0 = \mathbf{y}$, the index set $\Lambda=\varnothing$, and the signal estimate $\mathbf{x}=\mathbf{0}$.
2. For each iteration k:
    a. Find the column of A that is most correlated with $\mathbf{r}_k$.
    b. Add this column's index to the set Λ.
    c. Solve the least-squares problem on the submatrix A_Λ.
    d. Update the residual.

Implementation of the signal recovery using OMP in Python:

```python
def orthogonal_matching_pursuit(A, y, max_iterations):
    residual = y
    index_set = []
    x_estimated = np.zeros(A.shape[1])

    for _ in range(max_iterations):
        # Find the column with the highest correlation with the residual
        correlations = A.T @ residual
        best_index = np.argmax(np.abs(correlations))
        index_set.append(best_index)

        # Solve the least-squares problem on the submatrix
        A_selected = A[:, index_set]
        x_ls, _, _, _ = np.linalg.lstsq(A_selected, y, rcond=None)

        # Update the estimated signal
        x_estimated[index_set] = x_ls

        # Update the residual
        residual = y - A @ x_estimated

    return x_estimated
```

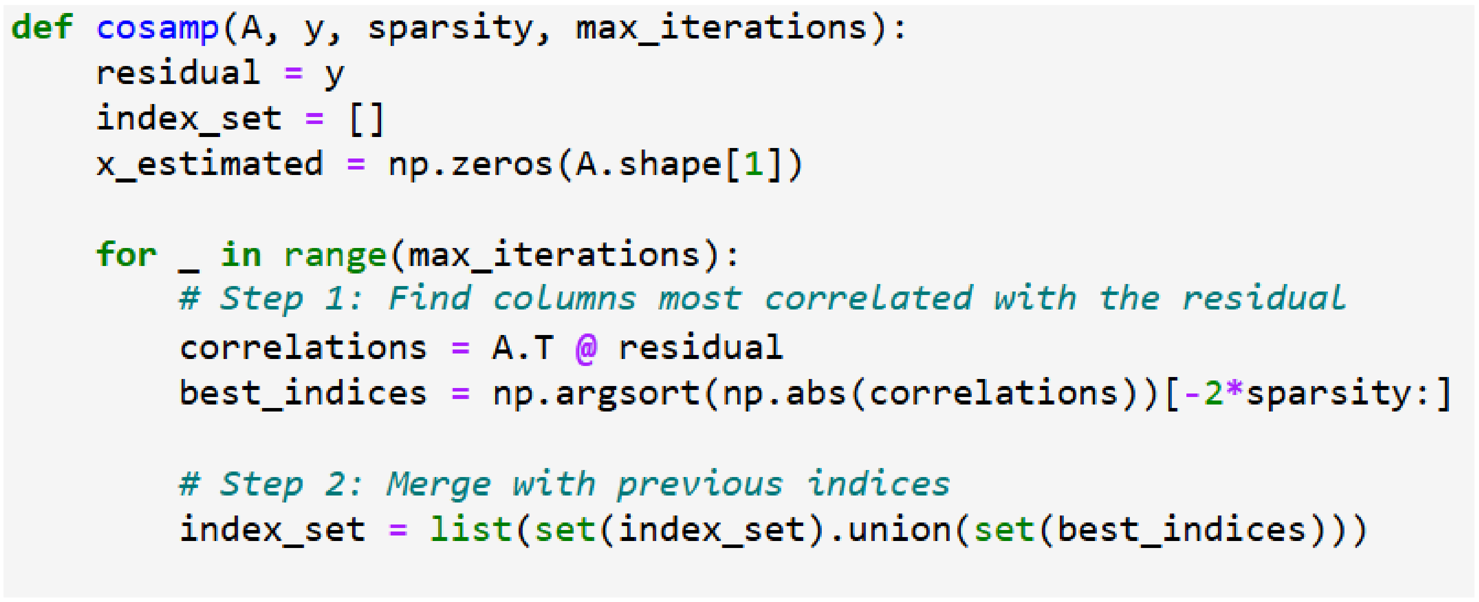
a.3.1) Compressive Sampling Matching Pursuit (CoSaMP)

CoSaMP is more sophisticated than OMP and includes a process of selecting multiple columns per iteration and then pruning to maintain sparsity. It combines elements of both greedy algorithms and iterative thresholding.

Algorithm:

1. Initialize the residual $\mathbf{r}_0 = \mathbf{y}$, the index set $\Lambda=\varnothing$, and the signal estimate $\mathbf{x}=\mathbf{0}$.
2. For each iteration k:
    a. Identify the columns most correlated with the residual.
    b. Merge these columns with the previously selected columns.
    c. Solve the least-squares problem on the merged set of columns.
    d. Prune the least significant coefficients to maintain sparsity.
    e. Update the residual.

Implementation of the signal recovery using OMP in Python:



```python
def cosamp(A, y, sparsity, max_iterations):
    residual = y
    index_set = []
    x_estimated = np.zeros(A.shape[1])

    for _ in range(max_iterations):
        # Step 1: Find columns most correlated with the residual
        correlations = A.T @ residual
        best_indices = np.argsort(np.abs(correlations))[-2*sparsity:]

        # Step 2: Merge with previous indices
        index_set = list(set(index_set).union(set(best_indices)))
        # Step 3: Solve the least-squares problem on the merged set
        A_selected = A[:, index_set]
        x_ls, _, _, _ = np.linalg.lstsq(A_selected, y, rcond=None)

        # Step 4: Prune to maintain sparsity
        largest_indices = np.argsort(np.abs(x_ls))[-sparsity:]
        new_index_set = [index_set[i] for i in largest_indices]

        x_estimated = np.zeros(A.shape[1])
        x_estimated[new_index_set] = x_ls[largest_indices]

        # Step 5: Update the residual
        residual = y - A @ x_estimated

        # Update the index set
        index_set = new_index_set

    return x_estimated
```

Application of compressed sensing to electrocardiogram (ECG) signals:

```
Entrée [75]: import numpy as np
             import pandas as pd
             import pywt
             from sklearn.linear_model import OrthogonalMatchingPursuit
             import matplotlib.pyplot as plt
```

```
Entrée [76]: # Load MIT-BIH Arrhythmia Database from CSV
             data = pd.read_csv('mitbih_test.csv')
             ecg_signals = np.array(data)
             ecg_signal = ecg_signals[0]
```

```
Entrée [77]: # Wavelet transformation
             coeffs = pywt.wavedec(ecg_signal, 'db4', level=3)
```

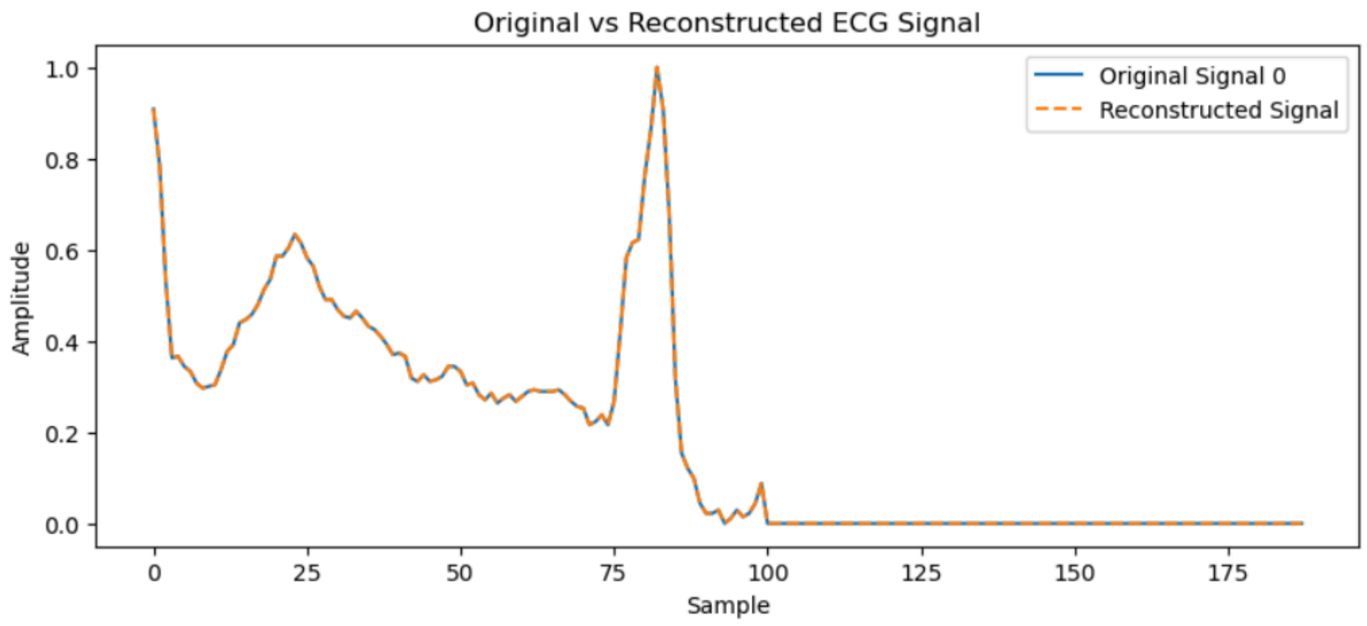
```
Entrée [78]: # Concatenate coefficients
             X = np.concatenate(coeffs)

             # Compressive sensing parameters
             M = 200  # Number of measurements
             N = len(X)  # Signal length

             # Random sensing matrix
             sensing_matrix = np.random.randn(M, N)

             # Compressed measurements
             measurements = sensing_matrix.dot(X)
```

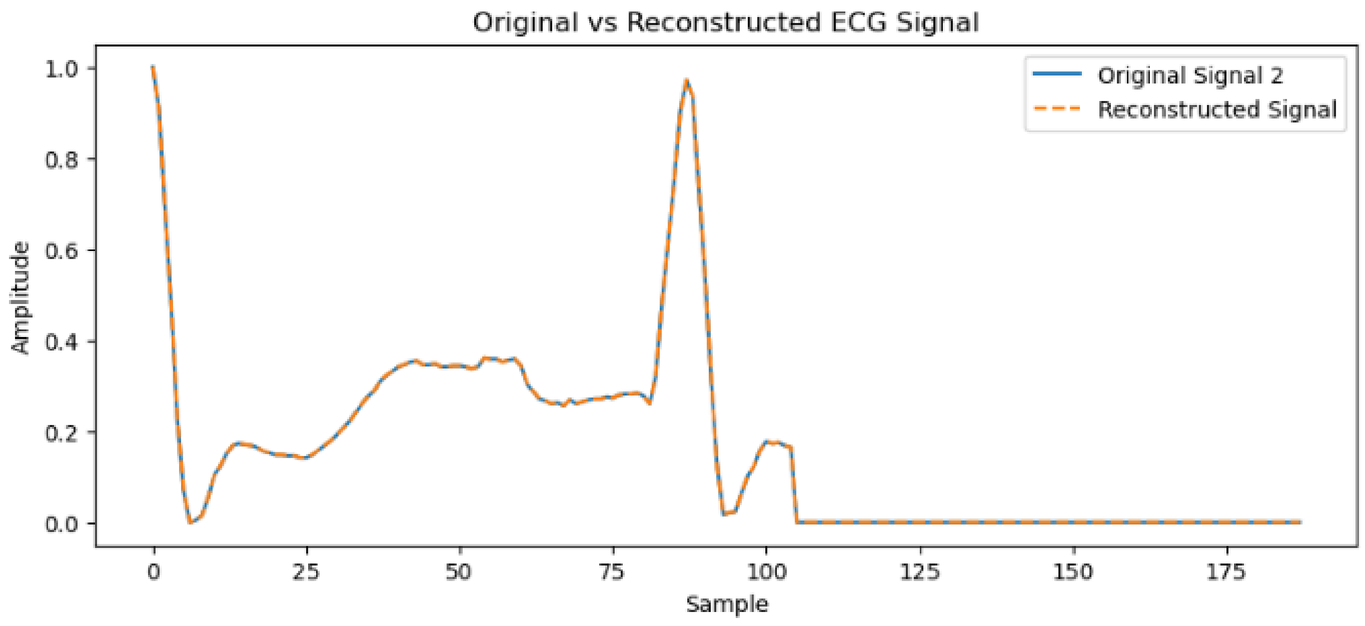
```
Entrée [79]: # Orthogonal Matching Pursuit (OMP) algorithm for signal reconstruction
             omp = OrthogonalMatchingPursuit(n_nonzero_coefs=N)
             omp.fit(sensing_matrix, measurements)
             reconstructed_coeffs = omp.coef_
```

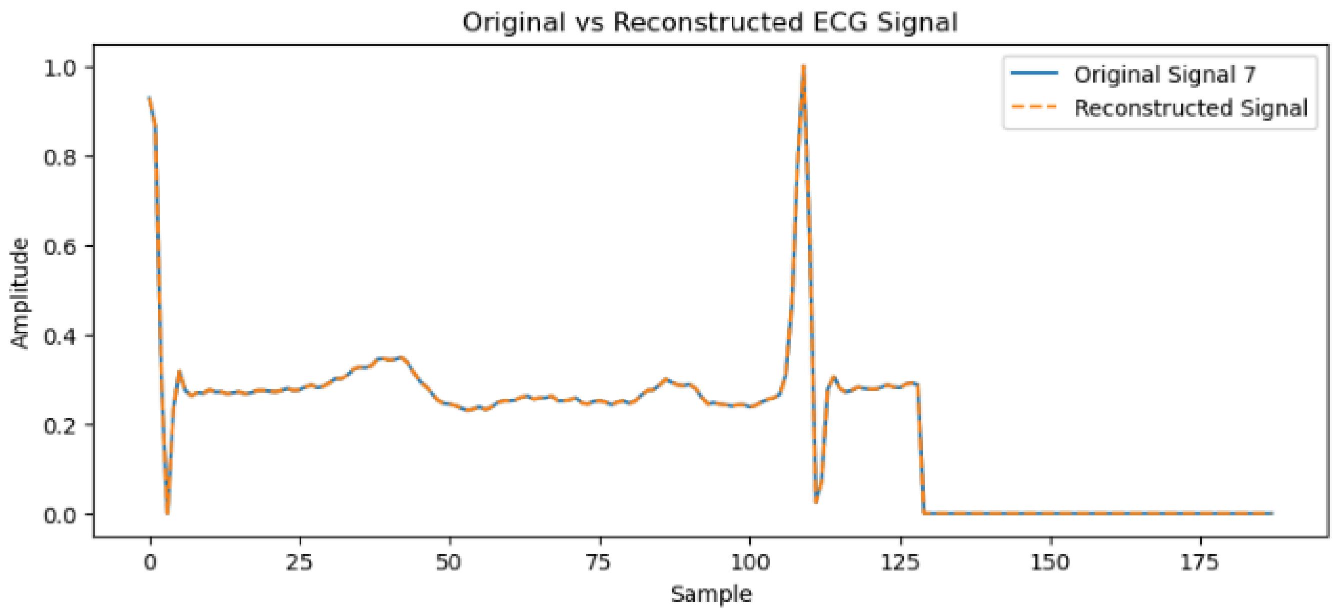
```
Entrée [80]: # Inverse wavelet transform
             # We need to split the reconstructed_coeffs back to the original wavelet coefficient shapes
             coeff_shapes = [coeff.shape[0] for coeff in coeffs]
             split_indices = np.cumsum(coeff_shapes)[:-1]
             reconstructed_coeffs_list = np.split(reconstructed_coeffs, split_indices)

             # Wavelet reconstruction
             reconstructed_signal = pywt.waverec(reconstructed_coeffs_list, 'db4')
```

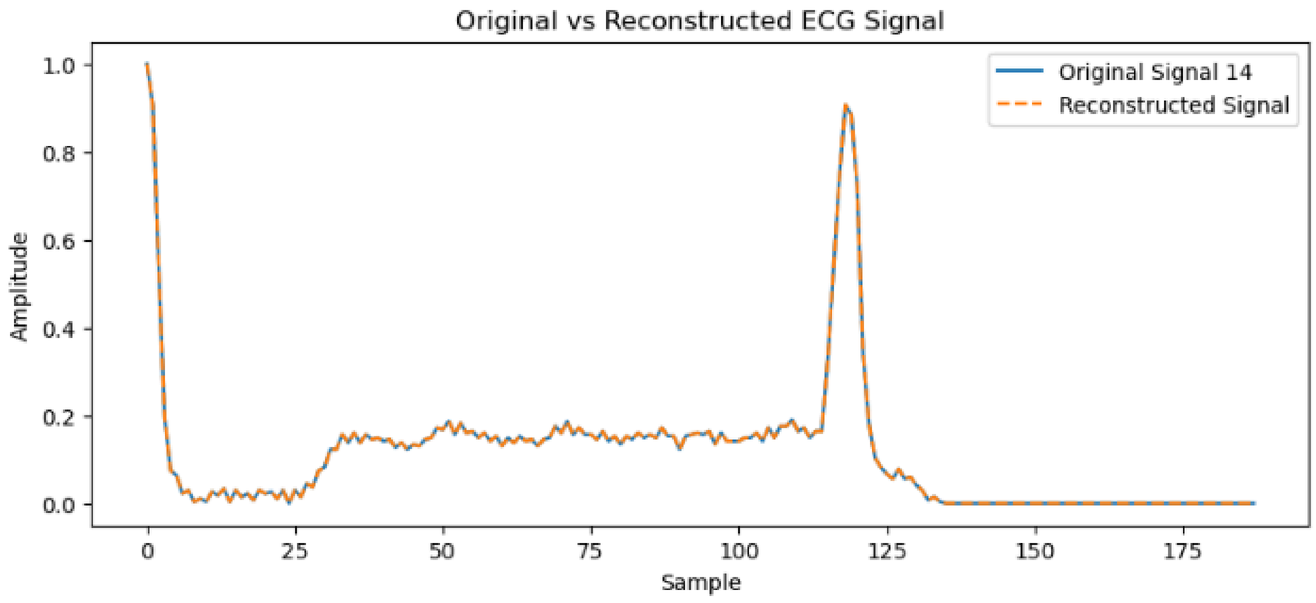
```
Entrée [81]: # Plot original and reconstructed signals for comparison
             plt.figure(figsize=(10, 4))
             plt.plot(ecg_signal, label='Original Signal 0')
             plt.plot(reconstructed_signal, label='Reconstructed Signal', linestyle='--')
             plt.legend()
             plt.xlabel('Sample')
             plt.ylabel('Amplitude')
             plt.title('Original vs Reconstructed ECG Signal')
             plt.show()
```

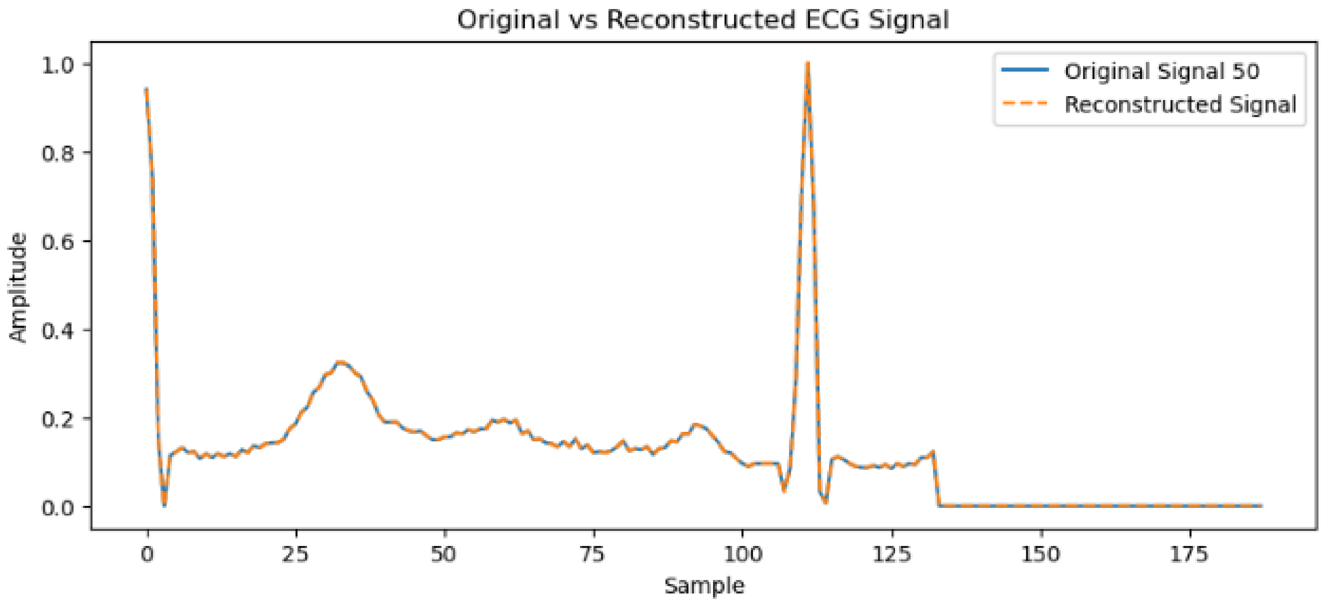
Here's the plot for different reconstructed ECG signals vs the original ones :

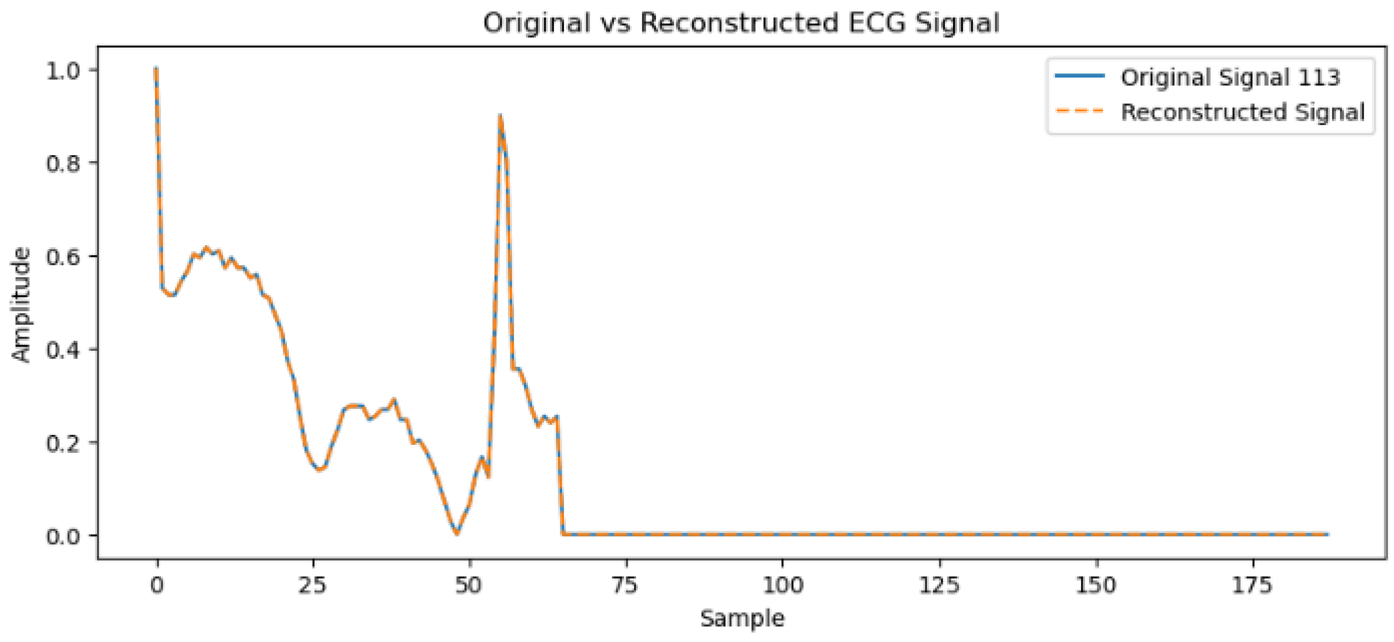





Original vs Reconstructed ECG Signal
Original Signal 7
Reconstructed Signal
Amplitude
Sample
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175


Original vs Reconstructed ECG Signal
Original Signal 14
Reconstructed Signal
Amplitude
Sample
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175


Original vs Reconstructed ECG Signal
Original Signal 50
Reconstructed Signal
Amplitude
Sample
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175



note :Il existe 4 classe de probleme majeure d’optimisation : Classe P, NP, NP-Complet, NP-Difficile