Matching Pursuit Alghorithm for Sparse Signal Decomposition

Ghazaleh Hadian Ghahfarokhi,7216547
Dortmund University of Applied Sciences and Arts
ghazaleh.hadianghahfarokhi001@stud.fh-dortmund.de

***Abstract*— This paper presents the implementation of the Matching Pursuit algorithm, replicating the animation of signal decomposition as depicted on the Matching Pursuit Wikipedia page. The algorithm iteratively decomposes a signal into a sparse representation using a redundant dictionary of basis functions, selecting the best matching atom at each step. The implementation, developed in MATLAB, demonstrates this decomposition process through dynamic animation, illustrating how the signal is progressively approximated over successive iterations. This approach highlights the flexibility and adaptability of Matching Pursuit in sparse signal representation, particularly in non-stationary signal analysis, and showcases its potential applications in modern signal processing.**

Keywords— ***Matching Pursuit Algorithm, Signal Decomposition, Redundant Dictionary, Sparse Representation, Signal Processing***

I. INTRODUCTION

In recent years, the demand for efficient signal processing techniques has increased due to the explosive growth of data in various fields such as communications, image processing, and machine learning. A key approach to processing high-dimensional signals is sparse signal representation, where a signal is expressed as a linear combination of a small number of basis functions or atoms from an overcomplete dictionary. Among the methods for sparse signal decomposition, the Matching Pursuit (MP) algorithm has garnered significant attention for its simplicity and adaptability.

Originally introduced by Mallat and Zhang in 1993, the Matching Pursuit algorithm iteratively decomposes a signal into a linear combination of dictionary elements that best approximate the signal's structure. Unlike traditional transforms, such as the Fourier or wavelet transforms that rely on fixed bases, MP adaptively selects atoms that match the signal's features, making it highly effective for sparse signal decomposition. MP has found extensive application in areas such as image compression, denoising, and compressed sensing[3].

A key feature of the MP algorithm is its ability to iteratively refine the signal approximation by subtracting the contribution of the selected atom from the signal at each step. This process continues until a stopping criterion is met, typically based on the residual error or the number of iterations. The result is a sparse representation of the signal, where only a few nonzero coefficients are required to accurately approximate the original signal.

This paper focuses on the implementation and visualization of the Matching Pursuit algorithm for sparse signal decomposition. In particular, it replicates the dynamic animation of the MP algorithm shown on the Wikipedia page for Matching Pursuit[2]. The animation illustrates the iterative decomposition of a signal into its sparse components, highlighting the algorithm's step-by-step selection of atoms and the resulting sparse representation.

Using MATLAB, a script is developed to simulate the animation, visualizing the signal approximation process and the evolution of the sparse coefficients over time. The implementation demonstrates the effectiveness of MP in decomposing signals and provides insights into the behavior of the algorithm, particularly in terms of the impact of dictionary selection and convergence properties. Through this project, I aim to bridge the theoretical aspects of the MP algorithm with its practical application and visualization.

This paper is organized as follows; first, a brief overview of the Matching Pursuit algorithm is provided, including its mathematical formulation, applications and extensions. Next, the implementation in MATLAB is detailed, focusing on the construction of the dynamic animation. Following this, the results of the experiments are presented, comparing the animation with theoretical expectations and highlighting key observations. Finally, the paper concludes with reflections on the MP algorithm's performance.

II. MATCHING PURSUIT THEORY

The Matching Pursuit (MP) algorithm is an iterative greedy algorithm used for sparse signal decomposition. The goal of the algorithm is to approximate a given signal as a linear combination of a few elements (or "atoms") chosen from a larger dictionary. This approach is particularly useful when the signal can be represented sparsely, i.e., when only a small number of atoms are needed to describe the signal adequately[3].

A. Mathematical Formulation

Given a signal $f \in \mathbb{R}^N$ and a dictionary $D = \{g_\gamma\}_{\gamma \in \Gamma}$ consisting of normalized atoms $g_\gamma \in \mathbb{R}^N$, the matching pursuit algorithm seeks to express f as a sum of a few dictionary elements. Mathematically, f is approximated as:

$$f \approx \sum_{n=0}^{M-1} c_n g_{\gamma_n}$$

where c_n are the coefficients, g_{γ_n} are the selected atoms from the dictionary, and M is the number of iterations or the desired level of sparsity [3].

The MP algorithm proceeds as follows:

1. **Initialization**: Start with the original signal f and define the initial residual $\mathcal{R}_0 f = f$. Set the iteration counter $n = 0$.

2. **Atom Selection**: At each iteration n, select the atom g_{γ_n} from the dictionary that maximizes the absolute inner product with the current residual:

$$g_{\gamma_n} = arg \max_{g_\gamma \in \mathcal{D}} |\langle \mathcal{R}_n f, g_\gamma \rangle|$$

This step identifies the atom that has the greatest correlation with the residual signal, which means it captures the largest remaining portion of the signal's structure.

3. **Coefficient Computation**: Compute the coefficient c_n corresponding to the selected atom:

$$c_n = \langle \mathcal{R}_n f, g_\gamma \rangle$$

This coefficient quantifies how much of the selected atom g_{γ_n} contributes to the signal.

4. **Residual Update**: Update the residual by subtracting the contribution of the selected atom from the current residual:

$$\mathcal{R}_{n+1} f = \mathcal{R}_n f - c_n g_{\gamma_n}$$

This step removes the part of the signal already represented by the selected atom, leaving behind a new residual for the next iteration.

5. **Iteration**: The process is repeated by incrementing n, selecting new atoms, and updating the residual until a stopping criterion is met. Common stopping criteria include reaching a predefined number of iterations M or achieving a sufficiently small residual norm $\|\mathcal{R}_n f\|$.

B. Convergence and Properties

The Matching Pursuit algorithm guarantees that the residual norm decreases with each iteration. However, since it is a greedy algorithm, MP does not always yield the sparsest possible representation of the signal. Its performance depends on the structure of the dictionary and the signal being analyzed. In practice, MP is often used when computational efficiency and simplicity are more critical than obtaining the absolute sparsest solution.

A variation of the algorithm, Orthogonal Matching Pursuit (OMP), enforces orthogonality between the residual and the chosen atoms, leading to potentially faster convergence and sparser solutions. However, OMP is computationally more demanding.

Matching Pursuit is widely applied in signal processing tasks such as compression, denoising, and compressed sensing, were signals exhibit sparsity in specific domains[3].

C. Redundant Dictionaries and Sparsity

Representing a signal in a particular basis involves finding the unique set of expansion coefficients in that basis. While there are many advantages to signal representation in a basis, particularly an orthogonal basis, there are also disadvantages. The ability of a basis to provide a sparse representation depends on how well the signal characteristics match the characteristics of the basis vectors. For example, smooth continuous signals are sparsely represented in a Fourier basis, while impulses are not. A smooth signal with isolated discontinuities is sparsely represented in a wavelet basis. However, a wavelet basis is not efficient at representing a signal whose Fourier transform has narrow high frequency support.

Real-world signals often contain features that prohibit sparse representation in any single basis. For these signals, you want the ability to choose vectors from a set not limited to a single basis. Because you want to ensure that you can represent every vector in the space, the *dictionary* of vectors you choose from must span the space. However, because the set is not limited to a single basis, the dictionary is not linearly independent.

Because the vectors in the dictionary are not a linearly independent set, the signal representation in the dictionary is not unique. However, by creating a redundant dictionary, you can expand your signal in a set of vectors that adapt to the time-frequency or time-scale characteristics of your signal. You are free to create a dictionary consisting of the union of several bases. For example, you can form a basis for the space of square-integrable functions consisting of a wavelet packet basis and a local cosine basis. A wavelet packet basis is well adapted to signals with different behavior in different frequency intervals. A local cosine basis is well adapted to signals with different behavior in different time intervals. The ability to choose vectors from each of these bases greatly increases your ability to sparsely represent signals with varying characteristics[5].

D. Algorithm Summary

To summarize, the key steps of the Matching Pursuit algorithm are:

- Initialize the residual as the original signal.
- Iteratively select atoms from the dictionary that best match the residual.
- Compute the corresponding coefficients and update the residual.
- Repeat until the signal is sufficiently decomposed into a sparse set of atoms.

The ability to adaptively select atoms that match the signal's structure makes MP a powerful tool for sparse signal decomposition and approximation.

III. MATCHING PURSUIT APPLICATIONS

In the Matching pursuit has been applied to signal, image and video coding, shape representation and recognition, 3D objects coding, and in interdisciplinary applications like structural health monitoring. It has been shown that it performs better than DCT based coding for low bit rates in both efficiency of coding and quality of image. The main problem with matching pursuit is the computational complexity of the encoder. In the basic version of an algorithm, the large dictionary needs to be searched at each iteration. Improvements include the use of approximate dictionary representations and suboptimal ways of choosing the best match at each iteration (atom extraction). The matching pursuit algorithm is used in MP/SOFT, a method of simulating quantum dynamics[1].

MP is also used in dictionary learning. In this algorithm, atoms are learned from a database (in general, natural scenes such as usual images) and not chosen from generic dictionaries.

A very recent application of MP is its use in linear computation coding to speed-up the computation of matrix-vector products[1].

IV. Matching Pursuit Extensions

Matching Pursuit (MP) has several important variants that address different needs in terms of accuracy, computational efficiency, and performance in signal decomposition. Below are the three primary forms of the algorithm:

1) Standard Matching Pursuit (MP):

MP is a greedy algorithm that iteratively selects the atom (a basis function from a predefined dictionary) which has the highest correlation with the current residual of the signal. At each iteration, the residual signal is updated by subtracting the contribution of the selected atom. It is simple and computationally efficient, making it effective for quick approximations and real-time applications. On the other hand, since it doesn't ensure orthogonality between selected atoms, some redundancy can occur, resulting in less optimal reconstructions in comparison to more refined methods[3].

2) Orthogonal Matching Pursuit (OMP):

OMP extends MP by ensuring that each selected atom is orthogonal to the previously chosen atoms. This is achieved by using orthogonal projection methods during the selection process, ensuring that atoms do not interfere with each other. A prototype of the OMP algorithm first appeared in the statistics community at some point in the 1950s, where it was called stagewise regression. The algorithm later developed a life of its own in the signal processing and approximation theory, literatures. The orthogonality constraint improves accuracy, reducing redundancy and providing a more stable and efficient signal decomposition. It often converges faster in terms of reaching an accurate approximation. In contract, the cost of ensuring orthogonality increases computational complexity, making it less suitable for real-time or large-scale applications where speed is critical[4].

3) Weak Matching Pursuit (WMP):

WMP relaxes the selection criteria of MP. Instead of choosing the atom with the highest correlation, it selects any atom that meets a weaker correlation condition. This allows for faster computation by reducing the strictness of atom selection. WMP offers a balance between speed and accuracy. It is faster than OMP and can be more efficient than standard MP in some cases where finding the exact maximum correlation is less important. The downside of WMP is the relaxation in selection criteria can lead to a less accurate representation of the signal. The weaker selection condition may result in a slower convergence to the optimal solution compared to OMP[5].

V. Implementation

This section describes the MATLAB implementation of the Matching Pursuit algorithm and the process used to generate the dynamic animation that visually demonstrates the algorithm's iterative decomposition of a signal.

The goal of the implementation was to replicate the dynamic animation, Figure 1, found on the Wikipedia page for Matching Pursuit, where the algorithm approximates a signal by selecting atoms from a predefined dictionary. The animation illustrates the evolution of the signal approximation over time as atoms are chosen and coefficients are computed[3].

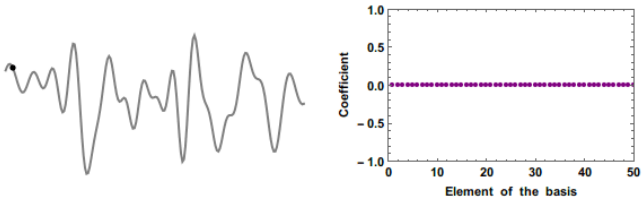

Figure 1: The original intended animation [2]

Retrieving an unknown signal (black line) from a small number of measurements (black dots) is in general impossible. The signal can be represented with only a few nonzero elements of a basis, then "compressed sensing" can be used. By making the measurement in a basis that is incoherent with the basis the signal is sparse in, it is ensured that each measurement sample as many coefficients as possible. In this project, an "orthogonal-matching-pursuit" algorithm is implemented. The starting guess is that all coefficient is zero. At each iteration, one nonzero coefficient is added, choosing the one that will affect our error metric the most. The values of those few nonzero coefficients are then estimated by a least-square fitting, and the process is iterated until the error is smaller than a given threshold. Let's delve more into coding structureVIII.

A. Initial signal generation

The initialization of signals presented here replicates the known values and overall formulation implemented by the Mathematica 11.0 code found in the animation of Orthogonal Matching Pursuit on Wikimedia Commons [2].

In this MATLAB implementation, several key elements for the matching pursuit algorithm are defined. The signal's dimensionality and the number of non-zero coefficients are set, with random positions and values assigned to these coefficients within a sparse vector. The original signal is constructed using the sum of a weighted combination of Hermite polynomials functions, applied over a set of values ranging from -7 to 7. A measurement matrix is created to capture 30 sampled positions from the original signal, simulating the signal acquisition process. Additionally, a transformation matrix is constructed using Hermite polynomials, forming the basis for projecting the signal into a new space. A measurement matrix is then formed by multiplying the measurement matrix with the transposed Hermite basis, and the observed signal $\hat{f}$ is calculated using the sampled positions of the original signal. A brief introduction of Hermite polynomials functions is as follows:

1) Hermite polynomials

The Hermite polynomials $H_n(x)$ are a sequence of orthogonal polynomials that can be defined using the following general recursion formula[6]:

$$H_{n+1}(x) = 2x.H_n(x) - 2n.H_{n-1}(x)$$

with the initial conditions:

$$H_0(x) = 1, \quad H_1(x) = 2x$$

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n}\left(e^{-x^2}\right)$$

With this said, the original $f(z)$ is computed using a function that sums the contributions of Hermite polynomials[2]:

$$f(z) = \sum_{j=1}^{n} c_j . hermitFunction\left(coefficient_j, z\right)$$

Given,

$$hermitFunction(n,z) = \frac{\exp\left(-\frac{x^2}{2}\right) . hermitH(n,x)}{\sqrt{\sqrt{\pi}\, n!\, 2^n}}$$

B. Error Calculation and Update Algorithm

Initialization: The algorithm begins with an initial residual, which represents the difference between the actual measurements ($\hat{f}$) and the predicted values.

Error Calculation: While the norm of the residual vector is greater than a small threshold (ϵ_0), the algorithm performs the following steps:

Compute Error Vector: For each candidate index in the complement of the support set, the error vector ϵ is calculated. For each candidate index j, the error is computed using the formula:

$$\epsilon_j = \left\| \frac{A(:,j)'.residual}{\|A(:,j)\|^2} . A(:,j) - residual \right\|^2$$

where:

- $A(:,j)$ is the j-th column of the measurement projection matrix A.
- residual is the current residual vector.
- $\|A(:,j)\|$ denotes the norm of the column vector $A(:,j)$.

Select Best Fit: Identify the index with the minimum error value. This index corresponds to the column that best reduces the residual.

Update Error: Set the current error (ϵ) to the minimum error value found.

Update Sets: Remove the selected index from the complement set and update the support set to include this index. This index is added to the support set and removed from the complement set.

Update Solution: Using the new support set, waiting factors are computed by solving the least-squares problem that minimizes the error between the observed measurements and the projections of the signal, given the current set of basis vectors

$$waiting\_factors = \left(A_{support}^T A_{support}\right)^{-1} \left(A_{support}^T \hat{f}\right)$$

Where, $A_{support}$ is the matrix A restricted to the current support set columns.

Update Residual: Calculate the new residual vector:

$$residual = \hat{f} - A.waiting\_factors$$

Store Results: Append the current solution to a list of results.

Termination: The loop continues until the norm of the residual is less than or equal to ϵ_0, indicating that the approximation is sufficiently close to the original data[2].

C. Visualization and Plot Customization

To ensure clarity the visualization in Figure 2, the original signal is plotted in black. The evolving signal approximation is plotted in orange to highlight its progression. The selected measurements are marked as filled black dots to show which parts of the signal are being used at each iteration. The number of measurements is 30 out of 200 points of the original signal. The coefficients of the selected atoms are plotted in magenta using stem plots to depict their evolution. The visualizations are adjusted to have consistent scaling, ensuring that the full range of the signal and coefficients are visible throughout the animation. Figure 3 shows the last plot of the simulation.

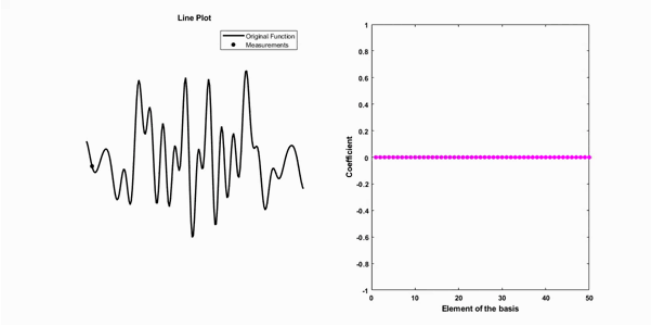

Figure 2: The replicated animation results of OMP algorithm[7]

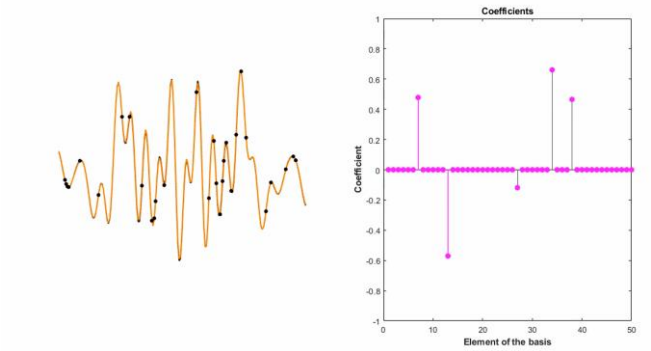

Figure 3: Final overview of approximated signal

The implementation of OMP algorithm successfully decomposed the original signal into a sparse representation using a dictionary of Hermite basis functions. The iterative process illustrated in the animation shows how the algorithm selects the best-fitting atoms at each step to reconstruct the signal. The accuracy of the reconstructed signal, $\hat{f}$, closely matched the original signal $f_{original}$, validating the effectiveness of OMP in sparse signal decomposition.

The results demonstrate the potential of MP in various applications, particularly in signal processing, where sparse representations are critical for compression and denoising tasks. The decomposition of signals into meaningful components highlights MP's utility in fields like machine learning, where sparse coding can lead to better interpretability, and in communications, where efficient signal representation is essential. Future work could focus on enhancing the computational efficiency of the algorithm or exploring alternative dictionaries to improve performance in diverse signal conditions.

The whole project, including the document, source code and the replicated animation can be found in the GitHub repository here.

VI. CONCLUSION

This paper implemented the Orthogonal Matching Pursuit (OMP) algorithm to replicate the signal decomposition process shown in the animation. Through iterative selection of the best-matching basis functions from a redundant dictionary, the algorithm successfully decomposed the original signal into a sparse representation. This demonstrated the flexibility and effectiveness of MP in processing non-stationary signals and its utility in applications like compression and denoising. While the implementation revealed challenges such as sensitivity to dictionary selection, noise, and computational cost, MP remains a valuable tool with potential for future applications in machine learning, communications, and real-time signal processing.

VII. REFERENCES

[1] Angoletta, Maria Elena. "Digital signal processor fundamentals and system design." (2008).

[2] Jacopo Bertolotti, "Orthogonal Matching Pursuit," *Wikimedia Commons*. [Online]. Available: https://commons.wikimedia.org/wiki/File:Orthogonal_Matching_Pursuit.gif . [Accessed: Aug. 30, 2024].

[3] "Matching Pursuit Algorithm," *Wikipedia: The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Matching_pursuit_algorithm. [Accessed: August 30, 2024]

[4] J. A. Tropp and A. C. Gilbert, "Signal Recovery From Random Measurements Via Orthogonal Matching Pursuit," in IEEE Transactions on Information Theory, vol. 53, no. 12, pp. 4655-4666, Dec. 2007, doi: 10.1109/TIT.2007.909108.

[5] “Matching Pursuit Algorithms”, MathWorks, [Online]. Available:, https://de.mathworks.com/help/wavelet/ug/matching-pursuit-algorithms.html, [Accessed: August 30, 2024]

[6] " Hermite polynomials," *Wikipedia: The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Hermite_polynomials [Accessed: August 30, 2024]

[7] Gh. Hadian, "Matching Pursuit Algorithm," GitHub, 2024. [Online]. Available: https://github.com/Hadian1989/matching_pursuit/blob/main/matlab%20implementation/matching_pursuit_animation.gif

VIII. MATLAB SCRIPT

```
clc;
close all;
clear all;

%% Step 1: Initialization
dim = 200;
numOfMeasurmentOfOrigSignal = 30;
numOfCoefficient = 6;
posisionOfCoffs=  sort(randsample(40, numOfCoefficient));
valueOfCoeffs = -0.8 + (1.6) .* rand(1, numOfCoefficient);

% Create a sparse vector
idealX_CoeffSparseArray = sparse(posisionOfCoffs, 1, valueOfCoeffs, dim, 1);

z_values = linspace(-7, 7, dim); %(1x200)
f_original = zeros(1, dim);
for i = 1:dim
    % Compute g(z) for each point in z_values
    f_original(i) = compute_g(z_values(i),numOfCoefficient,valueOfCoeffs,posisionOfCoffs);
end

measuringPositionsOfOrigSignal = sort(randperm(dim, numOfMeasurmentOfOrigSignal));
%  30x200
AtomSparseMatrix = sparse(1:numOfMeasurmentOfOrigSignal, measuringPositionsOfOrigSignal, 1,
numOfMeasurmentOfOrigSignal, dim);

% Matrix of change of basis from Hermite to canonical
Phi_HermitePolyMatrix = zeros(dim, dim);
for i = 1:dim
    % Basis matrix (Hermite polynomials)
    Phi_HermitePolyMatrix(i, :) = arrayfun(@(x) hermitFunc(i-1, x), z_values);
end
% Measurement matrix (30x200)
A = AtomSparseMatrix * transpose(Phi_HermitePolyMatrix);
f_hat = f_original(measuringPositionsOfOrigSignal)';

%% Step 2: Orthogonal Matching Pursuit Algorithm

% Initialization
% Error threshold
epsilon0 = 1e-10;
% Initial guess (all coefficients zero) 200x1
gaussed_waiting_factors = zeros(dim, 1);
% Initial residual is the measurement vector
residual = f_hat;
% Complement of the support (all indices)
indexOfCompOfSupport = 1:dim;
% Calculate the error for all columns of A that are not already in the support
epsilon = arrayfun(@(j) norm((A(:, j)' * residual / norm(A(:, j))^2)* A(:, j) - residual)^2,
indexOfCompOfSupport);
% Select the one with the smallest error
[~, min_idx] = min(epsilon);
j_MinValueErrorIndex = indexOfCompOfSupport(min_idx);
% Update the error and complement of the support
epsilon = epsilon(j_MinValueErrorIndex);
% Update the complement of the support
indexOfCompOfSupport(indexOfCompOfSupport == j_MinValueErrorIndex) = [];
% Update the support
support = setdiff(1:dim, indexOfCompOfSupport);
% Extract the columns of A corresponding to the support set
A_OfSupport = A(:, support);
% Find the best fit for the new estimate of x (least squares solution)
gaussed_waiting_factors(support) = (A_OfSupport' * A_OfSupport) \ (A_OfSupport' * f_hat);
% Update the residuals
residual = f_hat - A * gaussed_waiting_factors;
```

```
tmp = gaussed_waiting_factors;
% To store results
result_waiting_factors = {};


% Orthogonal Matching Pursuit
while norm(residual) > epsilon0
    % Calculate error vector epsilon
    epsilon = arrayfun(@(j) norm((A(:, j)' * residual / norm(A(:, j))^2) * A(:, j) - residual)^2,
indexOfCompOfSupport);
    % Select the index with the minimum error (best fit)
    [~, min_idx] = min(epsilon);
    % Update epsilon
    epsilon = epsilon(min_idx);
    % Remove the selected index from the complement set
    indexOfCompOfSupport(min_idx) = [];
    % Update the support set
    support = setdiff(1:dim, indexOfCompOfSupport);
    % Update the solution for the new support set
    A_OfSupport = A(:, support);
    % Least squares solution
    gaussed_waiting_factors(support) = (A_OfSupport' * A_OfSupport) \ (A_OfSupport' * f_hat);
    % Update the residuals
    residual = f_hat - A * gaussed_waiting_factors;
    result_waiting_factors{end+1} = gaussed_waiting_factors;
end
result_waiting_factors=[tmp,result_waiting_factors];

%% Step 3: Generate the animation
% % Define video dimensions

% Initialize video writer
videoFileName = 'matching_pursuit_animation.mp4';
frameRate = 6;  % Frame rate
% Create a video writer object
v = VideoWriter(videoFileName, 'MPEG-4');
v.FrameRate = frameRate;  % Set frame rate
open(v);  % Open the video file for writing

p0 = cell(1, numOfMeasurmentOfOrigSignal);

for k = 1:numOfMeasurmentOfOrigSignal
    fig = figure('Visible', 'off');
     % Adjust figure size: [left, bottom, width, height]
    set(fig, 'Position', [100, 100, 1200, 600]);
    % Create the first subplot for the line plot
    subplot(1, 2, 1);
    plot(f_original, 'k', 'LineWidth', 2);
    hold on;
    plot(measuringPositionsOfOrigSignal(1:k), f_hat(1:k), 'ko', 'MarkerSize', 5, 'MarkerFaceColor',
'k');
    ylim([-1, 1]);
    legend('Original Function', 'Measurements');
    title('Line Plot');
    axis off;
     % Second plot (horizontal line with shaded area)
    subplot(1, 2, 2);
    stem(1:50, zeros(1, 50), 'm', 'MarkerSize', 5, 'MarkerFaceColor', 'm');
    ylim([-1, 1]);
    xlim([0,50]);
    xlabel('Element of the basis', 'FontWeight', 'bold');
    ylabel('Coefficient', 'FontWeight', 'bold');
    set(gca, 'XColor', 'k', 'YColor', 'k', 'FontWeight', 'bold');
    p0{k} = gcf;  % Save the current figure handle into the cell array
    % Capture the current frame and write it to the video
    frame = getframe(gcf);
    writeVideo(v, frame);
```

```
    % Close the figure to free up memory
    close(gcf);
end
tau_values = 0:0.1:1;  % Tau values for the second loop

% Second Loop: Create the second set of figures and add them to the video
for tau = tau_values
    % Create a new figure for each frame
    fig = figure('Visible', 'off');
    set(fig, 'Position', [100, 100, 1200, 600]);
    subplot(1, 2, 1);
    plot(f_original, 'k', 'LineWidth', 1.5);
    hold on;
    evolving_signal = (1 - tau)*0 + tau*transpose(Phi_HermitePolyMatrix)*result_waiting_factors{1};
    plot(evolving_signal, 'Color', [1, 0.5, 0] ,'LineWidth', 1.5);  % Plot evolving signal
    plot(measuringPositionsOfOrigSignal, f_hat, 'ko', 'MarkerSize', 4, 'MarkerFaceColor', 'k');
    ylim([-1, 1]);  % Set y-axis limits
    axis off;  % Hide axes

    % Second subplot (Right): Plot the evolution of coefficients
    subplot(1, 2, 2);  % Create second subplot
    coeffs = tau * result_waiting_factors{1};  % Compute the evolving coefficients
    h = stem(1:200, coeffs, 'm', 'MarkerSize', 6, 'MarkerFaceColor', 'm');  % Plot coefficients
    % Ensure the color is consistent
    set(h, 'Color', 'm');  % Set color for the stem lines
    set(h, 'MarkerEdgeColor', 'm');  % Set color for marker edges
    set(h, 'MarkerFaceColor', 'm');  % Set color for marker faces
    ylim([-1, 1]);  % Set y-axis limits
    xlim([0,50]);
    xlabel('Element of the basis', 'FontWeight', 'bold');
    ylabel('Coefficient', 'FontWeight', 'bold');
    title('Coefficients');  % Add title

    % Capture the current frame and write it to the video
    frame = getframe(gcf);
    writeVideo(v, frame);

    % Close the figure to free up memory
    close(gcf);
end

for k = 2:length(support) - 1
    for tau = tau_values
        % Create a new figure for each frame
        fig = figure('Visible', 'off');
        set(fig, 'Position', [100, 100, 1200, 600]);  % Adjust figure size: [left, bottom, width,
height]
        % First subplot: Plot df and evolving signal
        subplot(1, 2, 1);  % Create the first subplot
        plot(f_original, 'k' ,'LineWidth', 1.5);  % Plot df as gray line
        hold on;
        evolving_signal_p2 = (1 - tau) * Phi_HermitePolyMatrix' * result_waiting_factors{k - 1} ...
                           + tau * Phi_HermitePolyMatrix' * result_waiting_factors{k};
        plot(evolving_signal_p2 ,'Color', [1, 0.5, 0], 'LineWidth', 1.5);  % Plot evolving signal
        plot(measuringPositionsOfOrigSignal, f_hat, 'ko', 'MarkerSize', 4, 'MarkerFaceColor', 'k');
        ylim([-1, 1]);  % Set y-axis limits
        axis off;  % Hide axes
        % Second subplot: Plot coefficients
        subplot(1, 2, 2);  % Create the second subplot
        coeffs_p2 = (1 - tau) * result_waiting_factors{k - 1} + tau * result_waiting_factors{k};
        h2 = stem(1:200, coeffs_p2, 'm', 'MarkerSize', 6, 'MarkerFaceColor', 'm');
        % Ensure the color is consistent
        set(h2, 'Color', 'm');  % Set color for the stem lines
        set(h2, 'MarkerEdgeColor', 'm');  % Set color for marker edges
        set(h2, 'MarkerFaceColor', 'm');  % Set color for marker faces
        ylim([-1, 1]);  % Set y-axis limits
```

```
        xlim([0,50]);
        xlabel('Element of the basis', 'FontWeight', 'bold');
        ylabel('Coefficient', 'FontWeight', 'bold');
        title('Coefficients');  % Add title

        % Capture the current frame and write it to the video
        frame = getframe(gcf);
        writeVideo(v, frame);

        % Close the figure to free up memory
        close(gcf);
    end
end

% Finalize and close the video file
close(v);

function g_val = compute_g(z,nb,c,nzc)
g_val = 0;
for j = 1:nb
    g_val = g_val + c(j) * hermitFunc(nzc(j), z);  % Sum of terms
end
end

function y = hermitFunc(n, x)
    % Compute the normalization factor
    norm_factor = sqrt(factorial(n) * 2^n * sqrt(pi));

    % Define the normalized Hermite function
    y = exp(-x.^2 / 2) .* hermiteH(n, x) / norm_factor;
end
```