DFT and FFT in Image Steganography (and PV fault detection)

# Introduction

* Introduction to Fourier Transform and its Applications

The Fourier Transform (FT) is one of the most significant mathematical tools in modern science and engineering. It provides the foundation for important algorithms like the Discrete Fourier Transform (DFT) and its computationally efficient implementation, the Fast Fourier Transform (FFT). These techniques are pivotal in a wide range of applications, from signal processing to image analysis and even in the construction of the Gaussian function, which is a cornerstone in probability and data analysis.

* Motivation for the Project

This project explores the applications of DFT and FFT in two domains:

* Image Steganography:

Inspired by the increasing demand for data security on the internet, image steganography leverages FFT to encode hidden messages within images. The focus is to understand and apply these Fourier-based techniques to securely embed and extract hidden messages, contributing to the field of secure communication.

* PV Fault Detection Analysis (Side Project):

As an additional objective, this project aims to apply Fourier-based methods to analyze fault signals in photovoltaic (PV) systems. By leveraging DFT/FFT for signal adjustment and fault analysis, this project explores their potential in improving the reliability of PV systems.

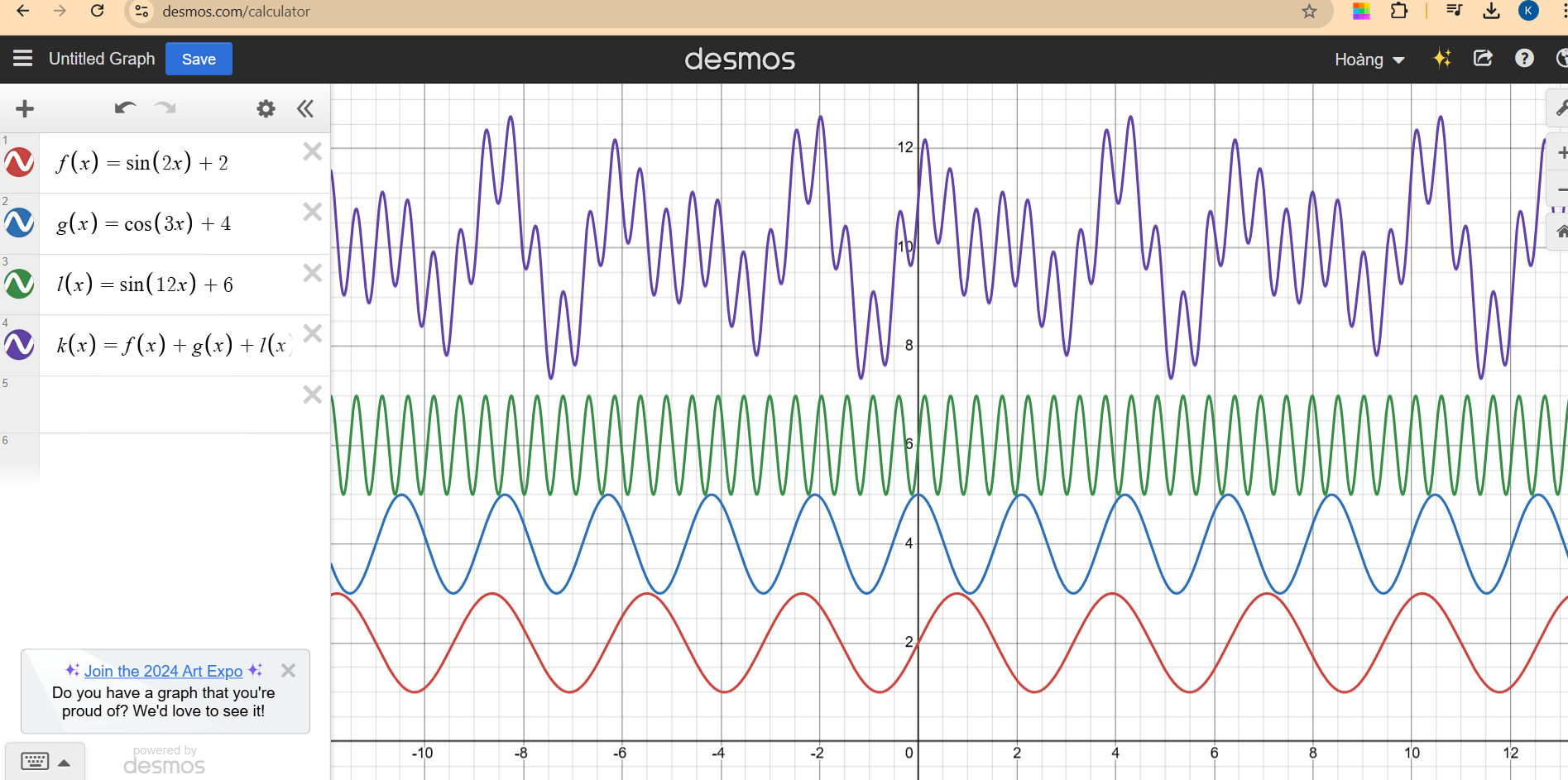
* Why Image Steganography?

The inspiration for choosing image steganography stems from the growing need for secure methods of data transmission in today's digital world. Among the many approaches researched, using FFT for hiding images offers a robust and efficient solution for encoding messages. This project demonstrates how FFT can be employed to meet these demands by securely embedding and extracting data from digital images.

# Theory background

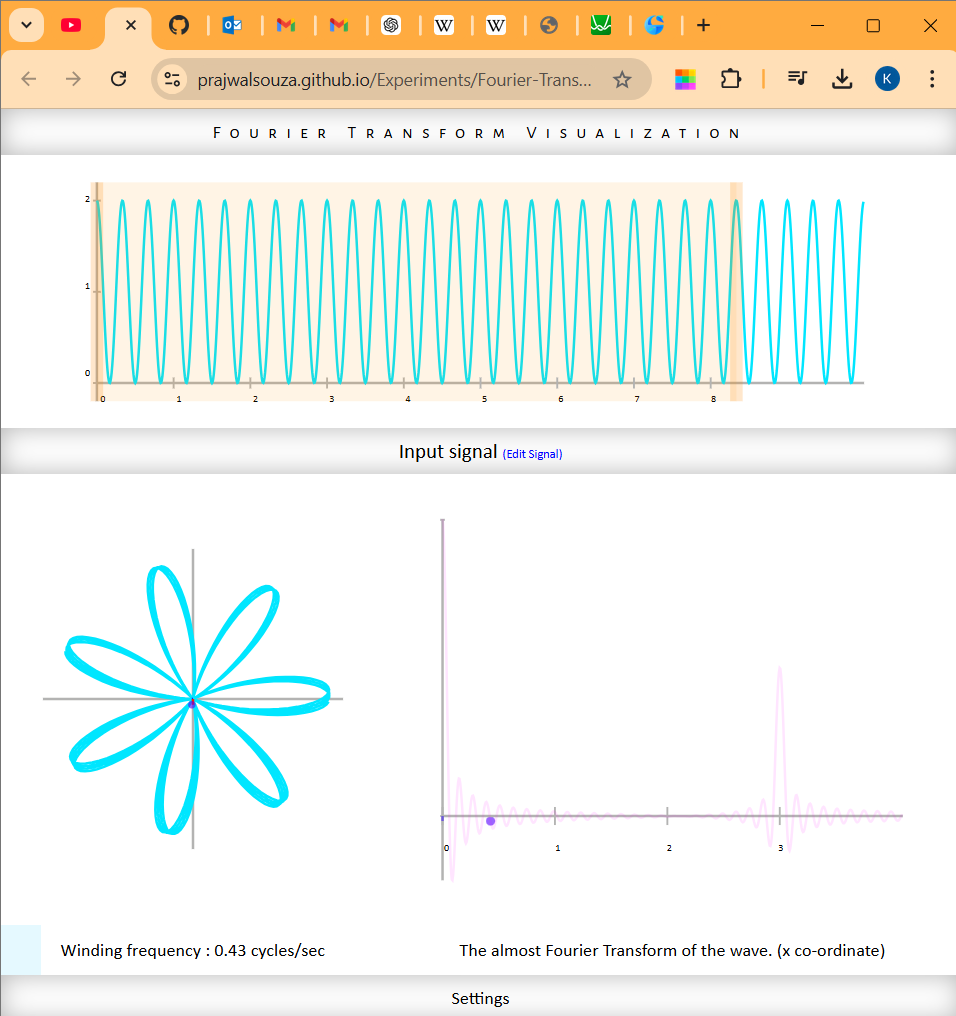
## FT

Fourier transform is an interesting function. And it is the answer for the question how can we unmix a complicated signal we record from the real life. Why? Take an example the microphone can measure the air pressure and then give us a signal like this:



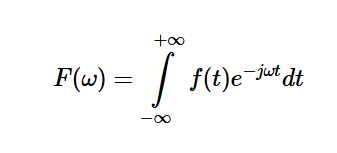
<https://prajwalsouza.github.io/Experiments/Fourier-Transform-Visualization.html>

Imagine that all the signal wrapping around a circle. Now we have a picture like this.



So, there are now 2 frequency we need to deal with the first one is the frequency of the signal which is 3 Hz. Another one is the frequency of a rotation around the circle. However, we can control the frequency wrapping around the circle, maybe we want it faster or maybe we want it slower. (show in the website). Now we come to the end, we assume this graph has a weight, and for the weight, we can easily find the center of mass using the differential of the weight function. Then we get the point on the graph.

At a certain rotational frequency, we can find out that the center of mass is not at the origin anymore, that how our function FT work.



Here you can see that exp part of the function is the vector which is pointed around the circle, the f(t) serve as the magnitude of the vector and the differential is the function for the center of mass.

## DFT

## FFT

The Fast Fourier Transform (FFT) is an algorithm used to compute the Discrete Fourier Transform (DFT) efficiently. The DFT is a mathematical technique that transforms a discrete signal from the time domain into the frequency domain, where the signal is represented as a combination of sinusoids of different frequencies, amplitudes, and phases.

The FFT reduces the computational complexity of the DFT, making it practical for large datasets and real-time applications.

First, we assume the DFT has N equal to an even number:

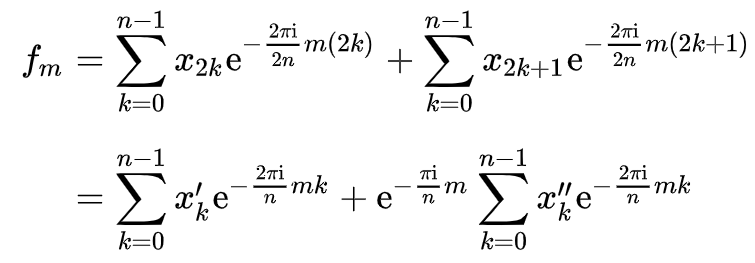


So now we have even indices and odd indices:

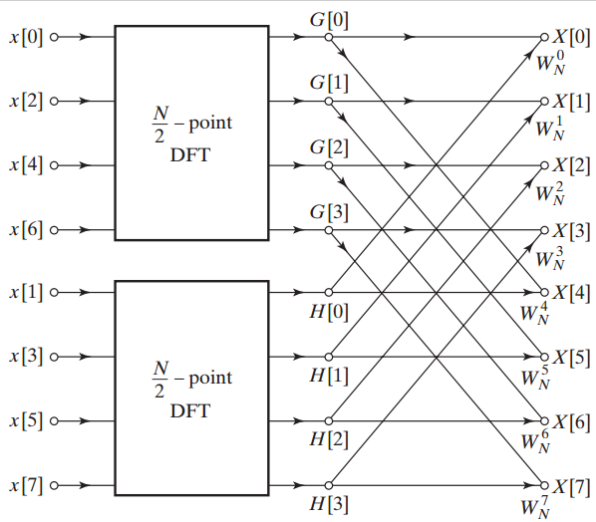




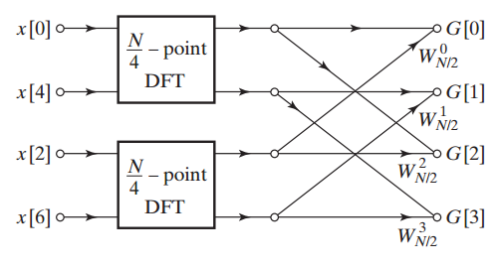
Now we will separate the fm into 2 part one is even and one is odd:



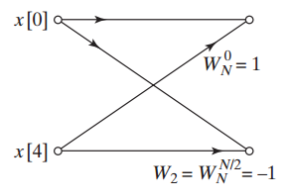
We can summarize it as follow:



And then we continue apply the FFT for the even and odd terms, which we will also get the split

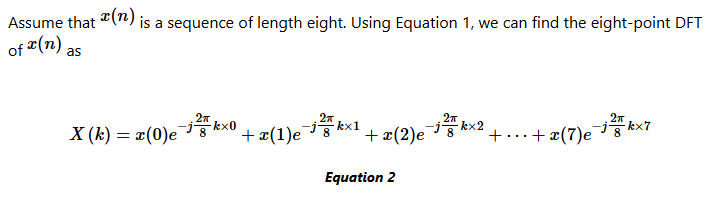


And again, one more time:

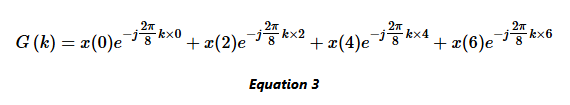


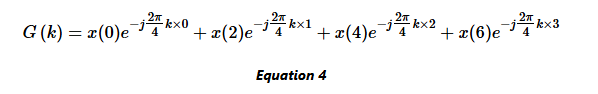
This will help us saving a lot of time cause for DFT we will have to compute computation but for the FFT we only have to compute times.

For example:

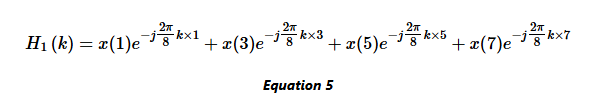


we examine choosing all the terms with an even sample index, i.e., x(0)x(0), x(2)x(2), x(4)x(4), and x(6)x(6). Hence, we have

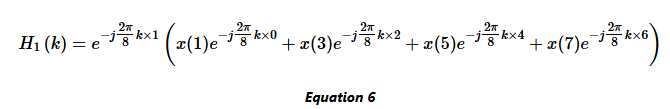


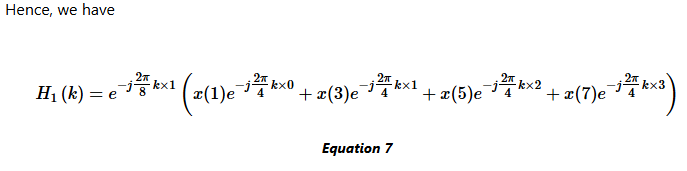


The remaining terms, which correspond to odd-index samples, are given by



To simplify the fractions , we can simply factor and obtain





## Image processing

## Moving average filter (optional)

# Image steganography

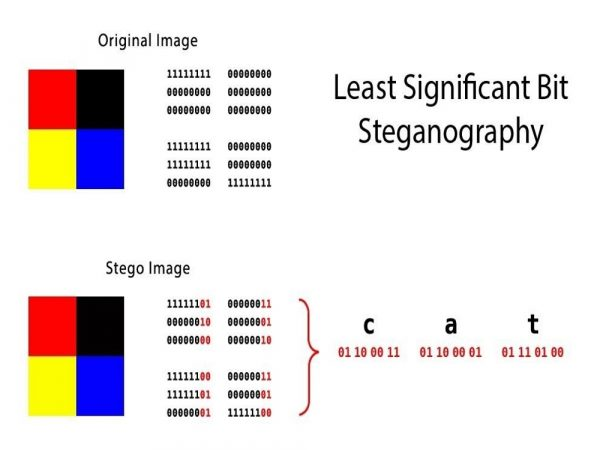
## LSB – Least significant bit

<https://www.geeksforgeeks.org/lsb-based-image-steganography-using-matlab/>

Steganography is the method of hiding secret data inside any form of digital media. The main idea behind steganography is to hide the existence of data in any medium like audio, video, image, etc. When we talk about image steganography, the idea is quite simple. Images are made up of pixels which usually refer to the color of that particular pixel. In a grayscale (black and white) image, these pixel values range from **0-255**, 0 being black and 255 being white.

**Concept of the LSB**

LSB stands for Least Significant Bit. The idea behind LSB embedding is that if we change the last bit value of a pixel, there won’t be much visible change in the color. For example, 0 is black. Changing the value to 1 won’t make much of a difference since it is still black, just a lighter shade.



1. Convert the image to grayscale
2. Resize the image if needed
3. Convert the message to its binary format
4. Initialize output image same as input image
5. Traverse through each pixel of the image and do the following:

* Convert the pixel value to binary
* Get the next bit of the message to be embedded
* Create a variable temp
* If the message bit and the LSB of the pixel are same, set temp = 0
* If the message bit and the LSB of the pixel are different, set temp = 1
* This setting of temp can be done by taking XOR of message bit and the LSB of the pixel
* Update the pixel of output image to input image pixel value + temp

1. Keep updating the output image till all the bits in the message are embedded
2. Finally, write the input as well as the output image to local system.

**Advantage:**

* This method is very fast and easy to implement in comparison to other methods of image Steganography.
* The output image has very slight difference to the input image.
* Instead of embedding the message in only the LSB, we can embed the message in last two LSBs, thus embedding even large messages.
* This method forms the basics of many other complex algorithms
* Instead of embedding the message in only the LSB, we can embed the message in last two LSBs, thus embedding even large messages.

**Disadvantages of this method:**

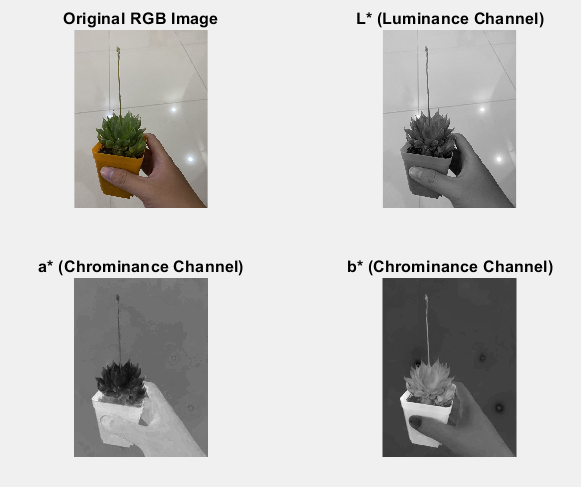
* This type of encoding the data is weak since it can be easily decoded by taking the LSBs of the image and getting the message in binary format.
* This is method is too old because it was used long ago when other encoding methods were not yet developed.
* When embedding the message in more than one LSB, the image quality may reduce depending on how many pixels are changed.

## FFT method

## Result \_ Methodology

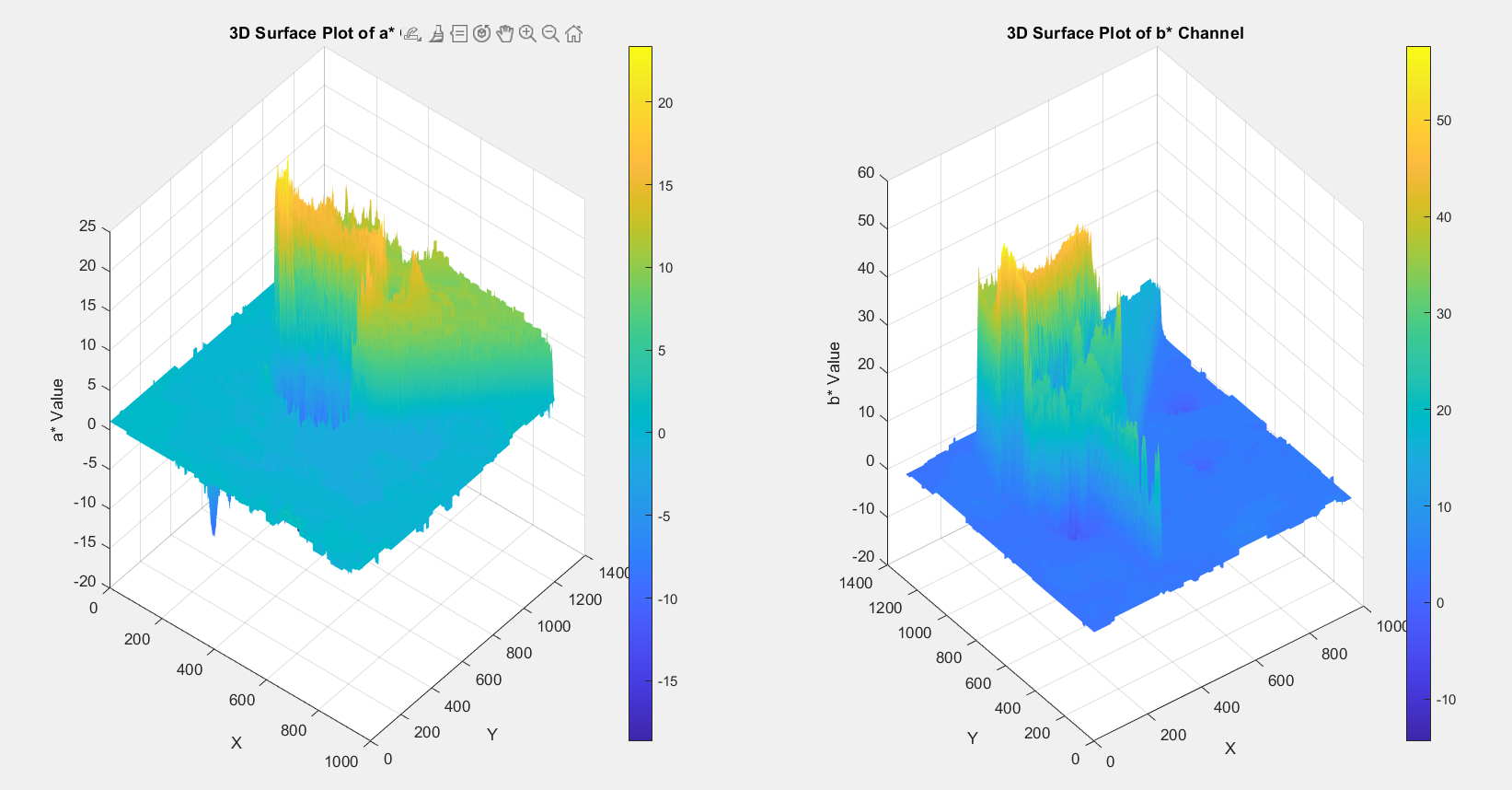
### FFT method

1. From RGB to Lab

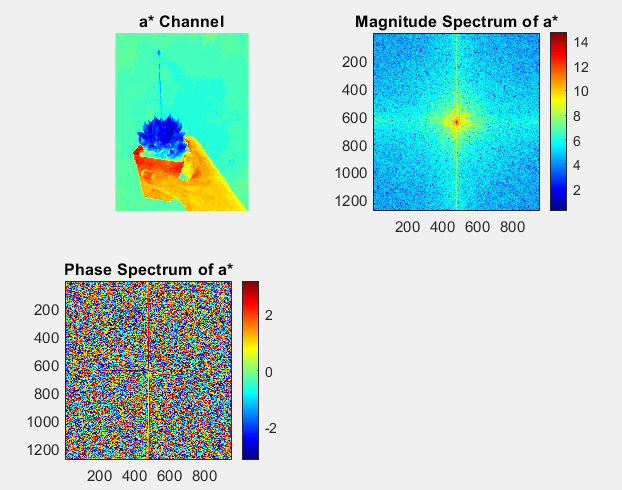




It is change from the RGB space to XYZ than to Lab



1. Then we start to apply FFT 2D to the channel a\*



% Apply 2D FFT to a\* channel

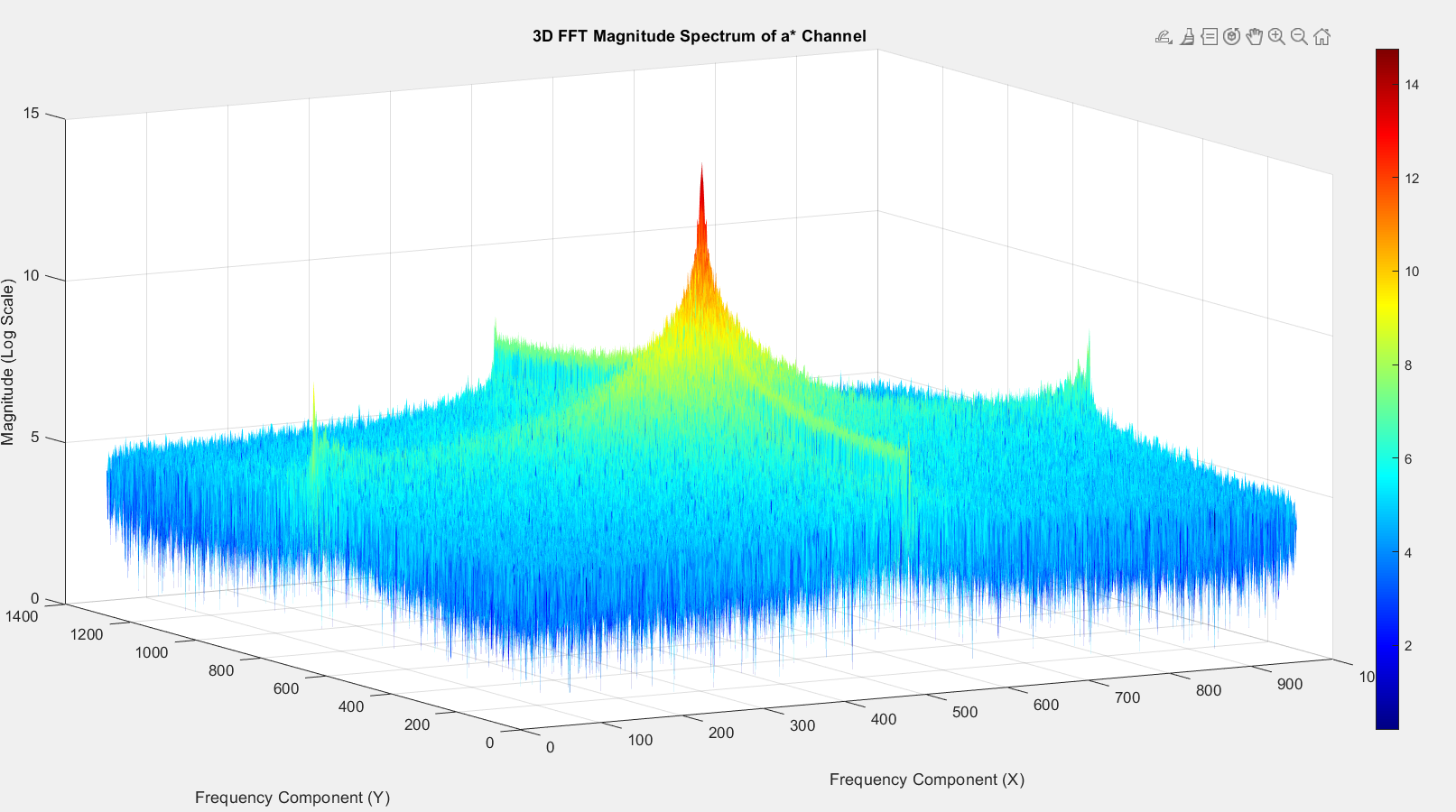
fftA = fft2(a); % Compute FFT

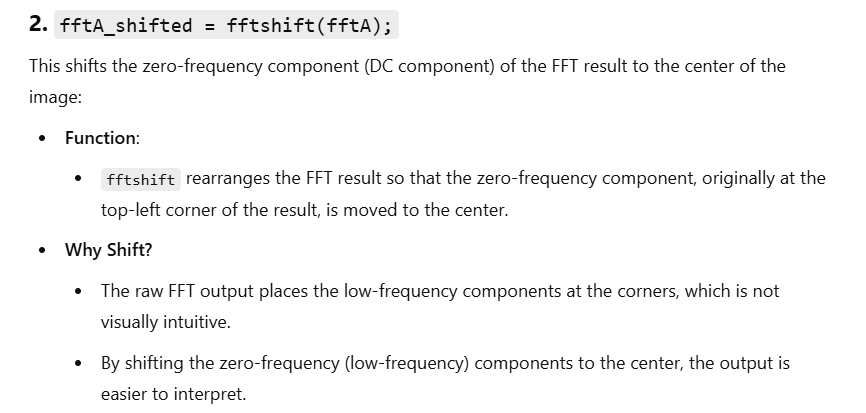
fftA\_shifted = fftshift(fftA); % Shift zero-frequency component to center

magnitudeA = abs(fftA\_shifted); % Magnitude spectrum

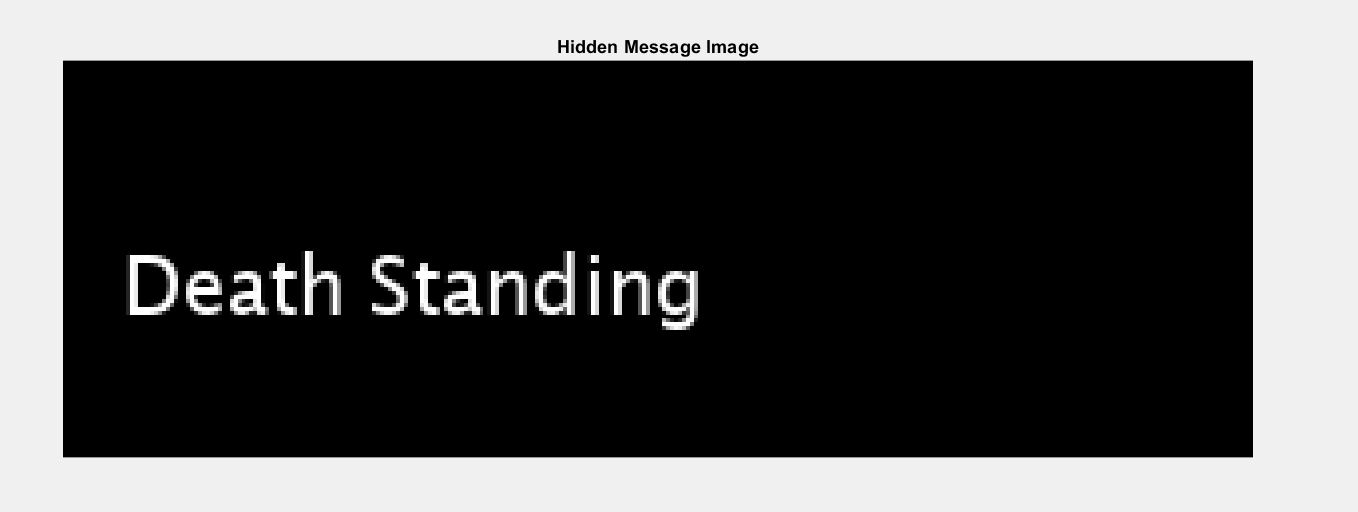
phaseA = angle(fftA\_shifted); % Phase spectrum

Here is the 3D view of the magnitude of the channel a\*, notice that the reason why I focus on the magnitude of the channel is due to the less important of its during the whole process.





1. Here is the hidden message



% Create a grayscale image of the message

message = 'Death Stranding';

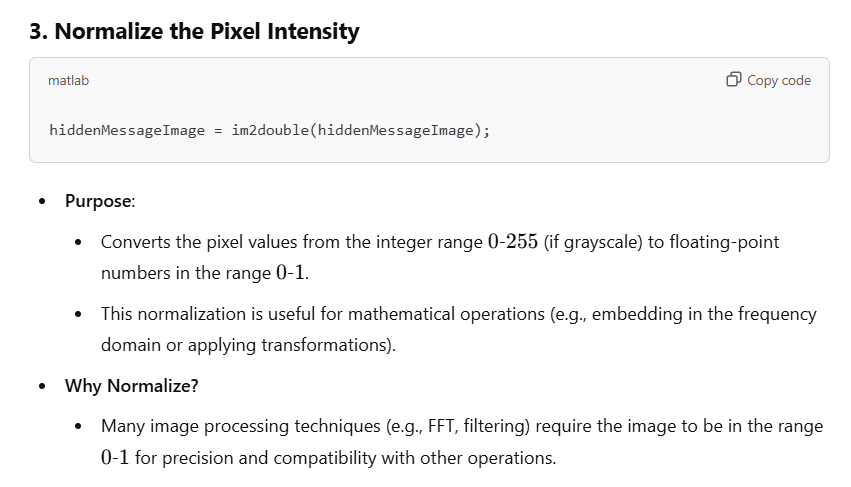
fontSize = 20; % Adjust font size

hiddenMessageImage = insertText(zeros(100, 300), [10, 40], message, ...

'FontSize', fontSize, 'BoxOpacity', 0, 'TextColor', 'white');

hiddenMessageImage = rgb2gray(hiddenMessageImage); % Convert to grayscale

hiddenMessageImage = im2double(hiddenMessageImage); % Normalize to [0,1]



1. Implement the hidden message into the chosen region of the magnitude matrix

% Resize the hidden message to match the embedding region

[hiddenRows, hiddenCols] = size(hiddenMessageImage);

embedSizeRows = round(size(magnitudeA, 1) / 8); % Size of the rectangular region

embedSizeCols = round(size(magnitudeA, 2) / 8);

hiddenMessageResized = imresize(hiddenMessageImage, [embedSizeRows, embedSizeCols]);

% Amplification factor

amplification = 8; % Increase embedding strength (adjust as needed)

% Normalize and scale the hidden message to the FFT range

maxMagnitude = max(magnitudeA(:));

hiddenMessageResized = hiddenMessageResized \* maxMagnitude / amplification;

% Define the rectangular embedding region next to the center

centerRow = floor(size(magnitudeA, 1) / 2); % Middle row

centerCol = floor(size(magnitudeA, 2) / 2); % Middle column

startRow = centerRow - floor(embedSizeRows / 2); % Vertically aligned

startCol = centerCol + embedSizeCols; % Start next to the center horizontally

% Embed the hidden message into the next-to-center region

for i = 1:embedSizeRows

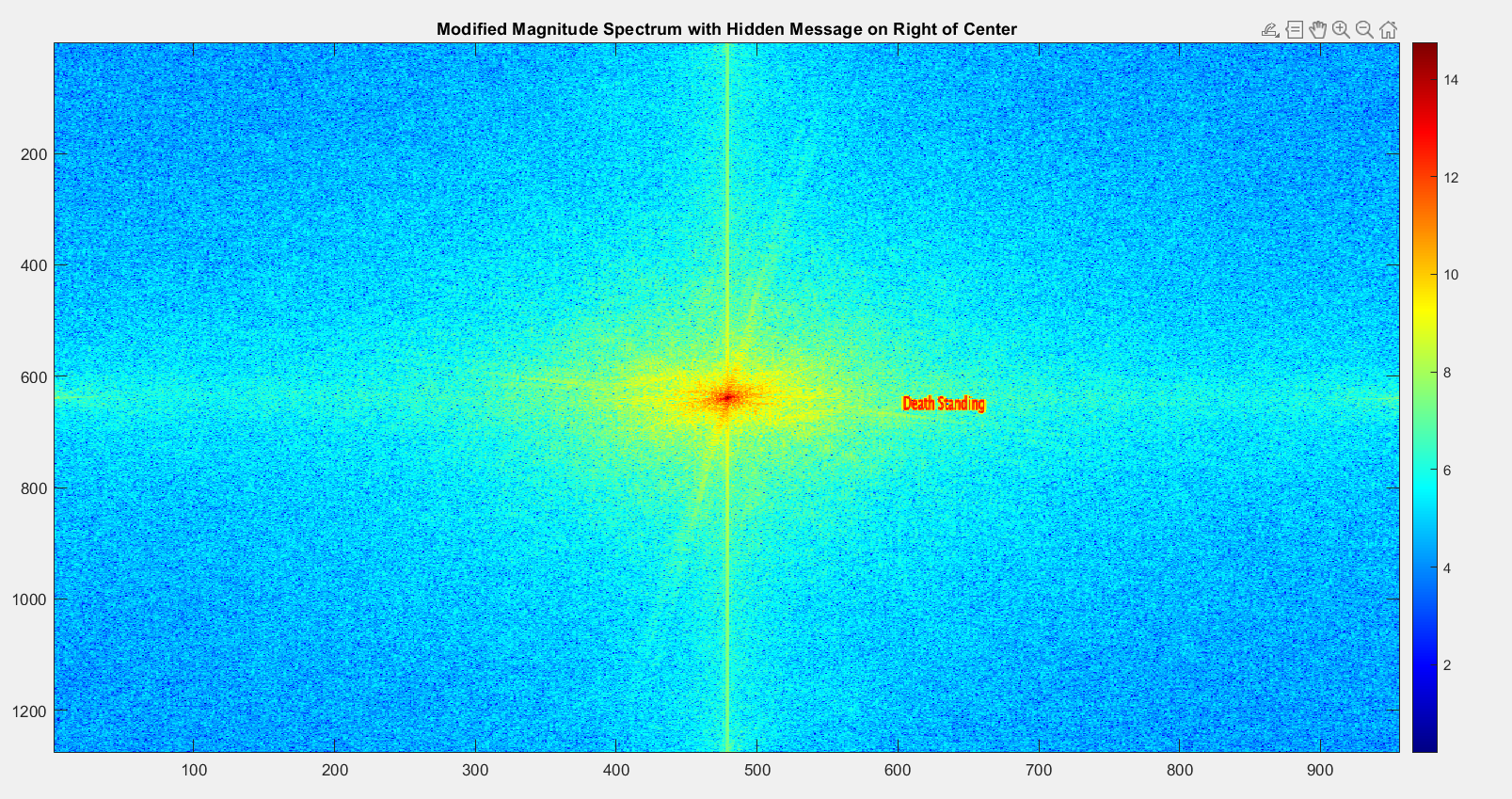
for j = 1:embedSizeCols

magnitudeA(startRow + i - 1, startCol + j - 1) = ...

magnitudeA(startRow + i - 1, startCol + j - 1) + hiddenMessageResized(i, j);

end

end





After implemented the message into the picture, we than computed new channel a\*

% Combine the modified magnitude spectrum with the original phase spectrum

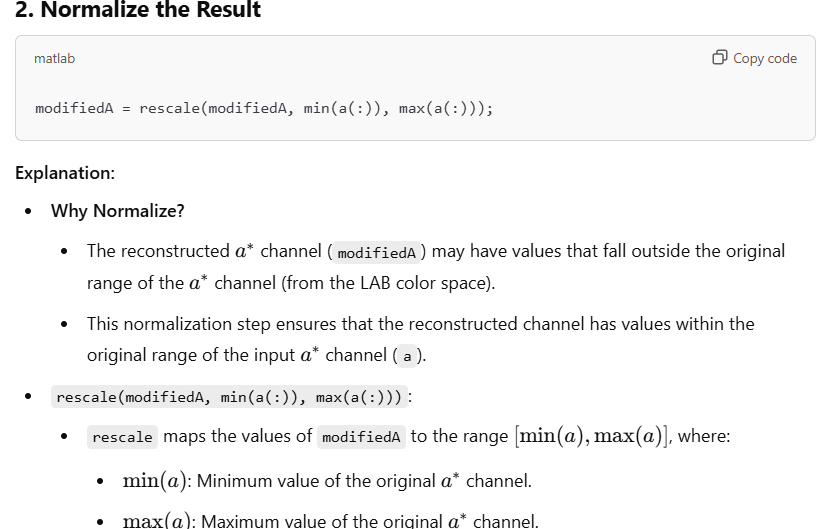
modifiedFFT\_A = magnitudeA .\* exp(1i \* phaseA);

then We do IFFT 2D for the modified channel a\*

% Perform IFFT to reconstruct the modified a\* channel

modifiedA = real(ifft2(ifftshift(modifiedFFT\_A))); % Apply inverse FFT

modifiedA = rescale(modifiedA, min(a(:)), max(a(:))); % Normalize back to original a\* range



And than change from Lab space to RGB and we got the modified picture with hiding image

The process for decoding also the same

We first change the image from RGB to Lab space then we FFT 2D the channel a\*. This is the most important part we need to choose the correct rectangular window where the message is hold.

% Required as key factors to open the message

% START

% Amplification factor (must match encoding process)

amplification = 8; % Same as used during encoding

% Define the embedding region size (must match encoding process)

embedSizeRows = round(size(magnitudeDecodedA, 1) / 8); % Embedding region height

embedSizeCols = round(size(magnitudeDecodedA, 2) / 8); % Embedding region width

% Define the dimensions of the original hidden message

originalRows = 100; % Height of the hidden message

originalCols = 300; % Width of the hidden message

% END

% Define the rectangular region for decoding (next to the center in the frequency domain)

centerRow = floor(size(magnitudeDecodedA, 1) / 2); % Middle row

centerCol = floor(size(magnitudeDecodedA, 2) / 2); % Middle column

startRow = centerRow - floor(embedSizeRows / 2); % Vertically aligned

startCol = centerCol + embedSizeCols; % Start next to the center horizontally

% Extract the same next-to-center region from the FFT spectrum

extractedRegion = magnitudeDecodedA(startRow:startRow + embedSizeRows - 1, ...

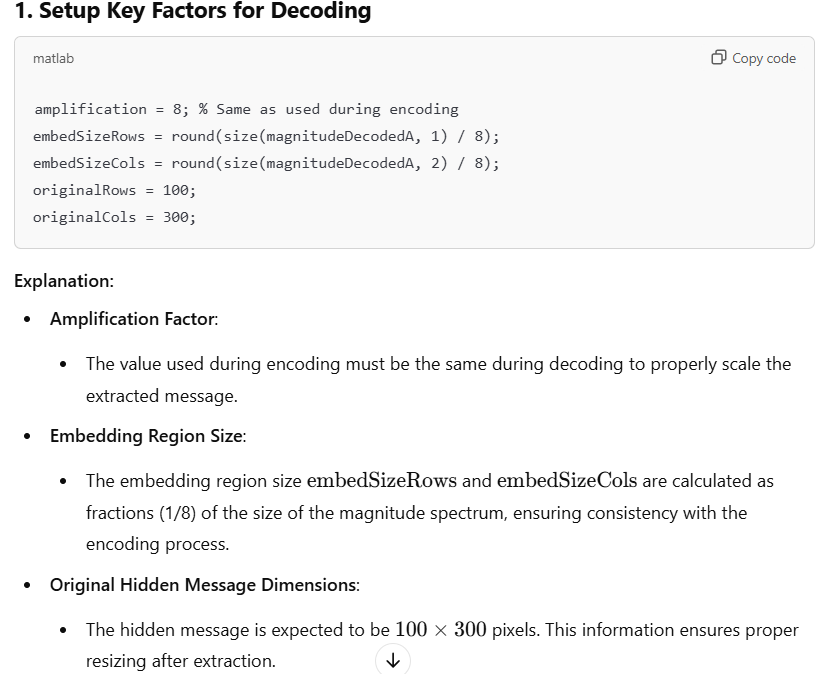
startCol:startCol + embedSizeCols - 1);

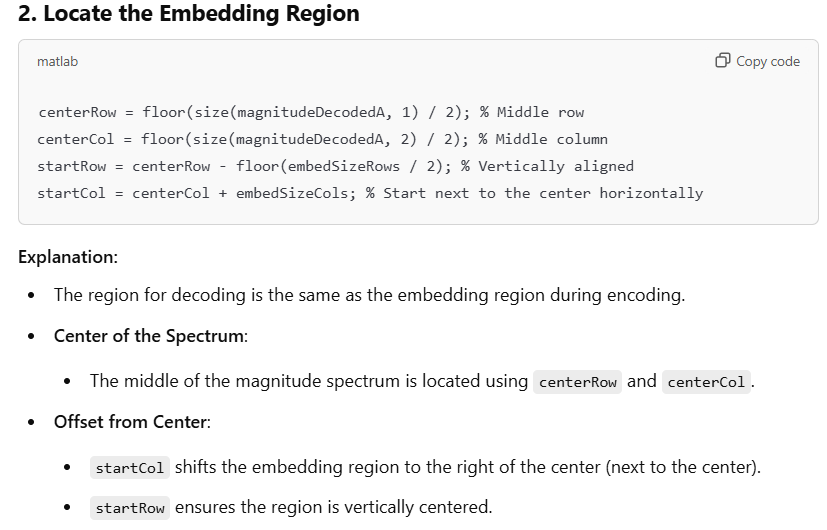
% Amplify the extracted region to restore visibility

amplifiedRegion = extractedRegion \* amplification;

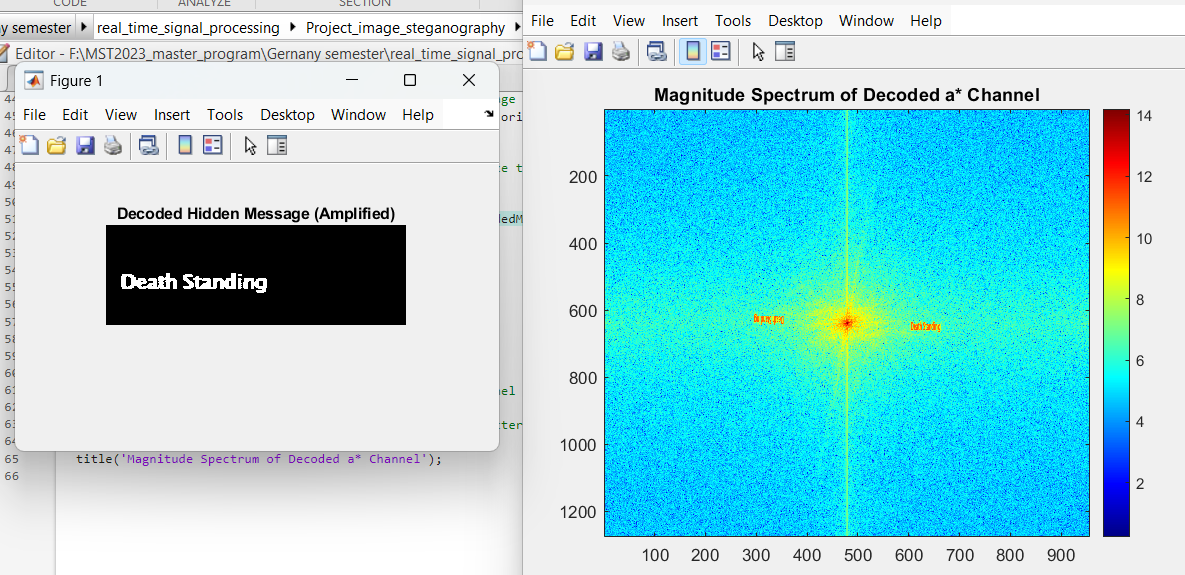
% Resize the extracted region to the original hidden message dimensions

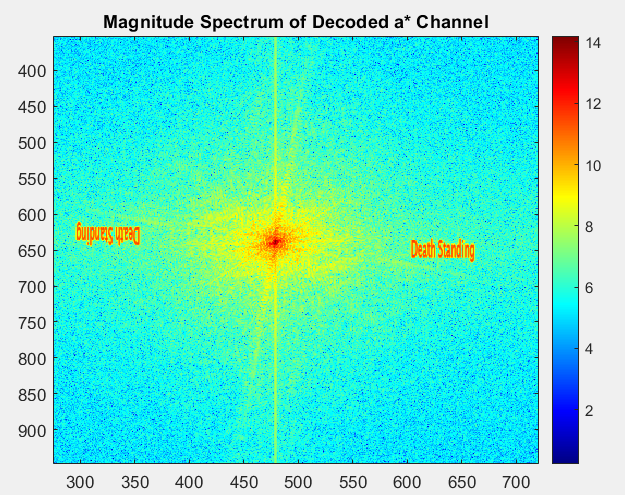
decodedMessage = imresize(amplifiedRegion, [originalRows, originalCols]);

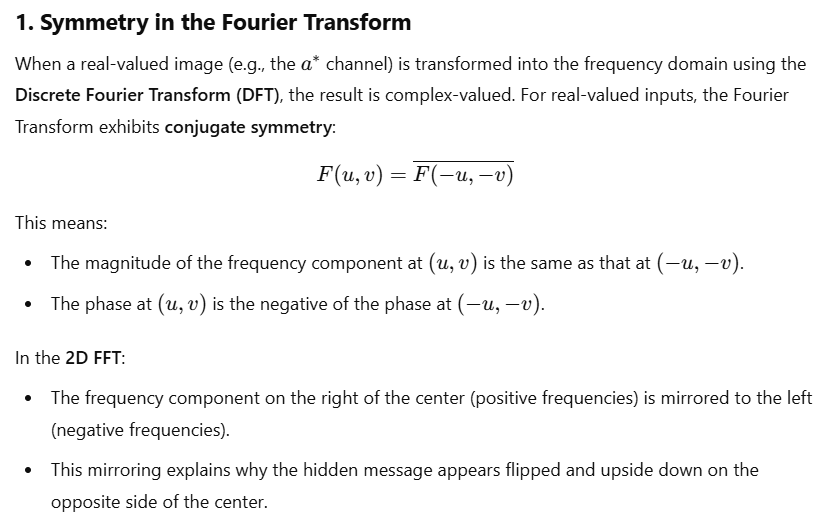




Here is the result which is the properties of FFT, we get not only 1 but 2 hidden message that is place opposite and upside down in the magnitude of the channel a\*.







### LSB method

% Traverse through the image

for i = 1 : height

for j = 1 : width

% If more bits are remaining to embed

if(embed\_counter <= len)

% Finding the Least Significant Bit of the current pixel

LSB = mod(double(input(i, j)), 2);

% Find whether the bit is same or needs to change

temp = double(xor(LSB, bin\_num\_message(embed\_counter)));

% Updating the output to input + temp

output(i, j) = input(i, j)+temp;

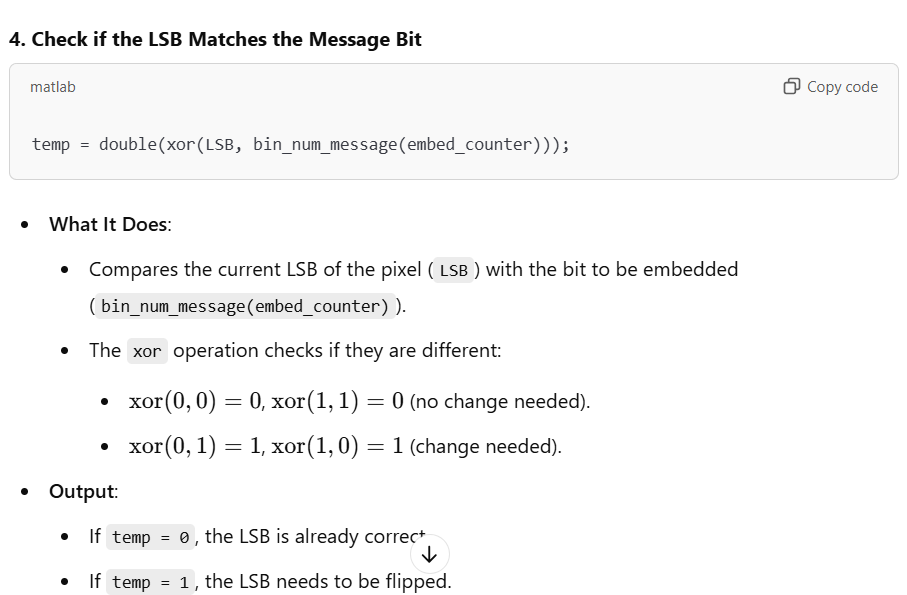
% Increment the embed counter

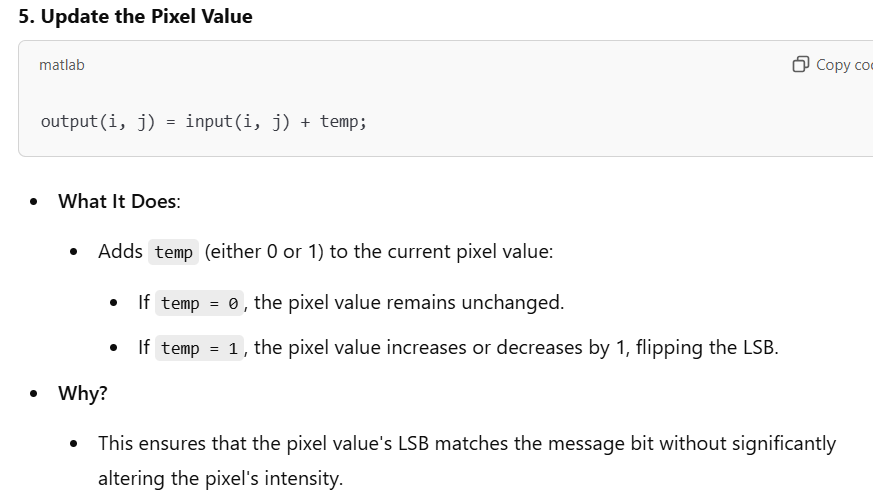
embed\_counter = embed\_counter+1;

end

end

end





## Conclusion

# PV fault detection

## Threshold based method

## Mean and variation method

# Conclusion

# Reference