Fourier Domain Watermarking of Videos

Divyansh Lalwani and Catherine Pollard

EN.553.493 Mathematical Image Analysis

Introduction

Watermarking is the action of modifying the content of a document in a manner that identifies its author or owner. Watermarking can be applied to a variety of different signals, including text, audio, images, and videos. In this report, we will discuss traditional applications and methods of watermarking, understand their shortcomings, and demonstrate four methods of watermarking signals that take advantage of Fourier analysis to leave an undetectable mark on various signals.

Background

Why Watermark?

Watermarking has many benefits, including

- Protection of author IP — watermarking can help to ensure that a user who paid for one copy of a document does not spread it or copy it in an unlicensed manner
- Marking AI-generated documents — this will prove necessary in order to protect against:
  - Academic dishonesty
  - Disruptive deepfakes
  - Training future models on AI-generated data, reducing quality of their output
- Detecting online document leaks — if each copy of a document sent out to users is watermarked with a different signature, leaks can be traced back to the original user.

Inherent Weaknesses of Watermarking

While the benefits listed above are attractive, there are several things that no watermarking scheme can achieve:

1. General AI detection [CITE PROPERLY ^1] — While some altruistic AI engineers may include watermarking to provide a means of distinguishing between their model's generations and non-AI works, there are already too many capable LLM's and image-generators that do not implement watermarking; this means that a user (for

example, a student) who wanted to generate a document with AI and pass it off as human already has plenty of ways to do so. This simply means that there is no watermarking scheme that can guarantee all AI-generated documents are recognizable as such.
2. Zero-distortion marking — since watermarking occurs in the actual content of the document (including text, pixel values and audio levels), there is no watermarking scheme that leaves the image exactly as it was originally found.

Traditional Watermarking Methods

Traditional watermarking consists of adding a signal to a document; this signal is easily detectable to the user, and may be easy or difficult to remove, depending on the author's level of trust in the user. The only way to access the original of a watermarked document should be to get it directly from the author.

Traditional watermarking tends to be relatively simple. Some (non-exhaustive) examples of its application include:

Audio: Inserting a statement of ownership in between audio context (ex. reading out the sponsor of a radio segment) makes it clear who the owner of the content is, but can be cropped out. Adding a noise signal to the audio signal disturbs the quality of the sound, but cannot be entirely removed without a drop in the audio quality.

Text: Adding the author's name is one form of text watermarking but is trivially easy to remove. Changing the words of an original text document in a known way (ex. making the words slightly more likely to start with "m" [1]) is a feasible method of text watermarking but changes the meaning of the text.

Images: Inconspicuous watermarks can be cropped or blurred out. Large, visible watermarks as shown in the figure below are historically effective but are becoming easier to remove with generative AI.

Figure: Watermark Sample

Videos: Traditional watermarking of videos simply consists of applying a visible watermark to each frame of the video separately. The watermark might also transform between frames, increasing the difficulty of removing it.

Traditional watermarking is limited because it must be applied to the document at its surface level - that is to say, the visible/audible level. In our day and age, *all* forms of traditional watermarking are either vulnerable to attack from a malicious user, or modify the document in a noticeable manner. Therefore, we need a method of marking images to show their authorship without alerting the user to the presence of a watermark, presenting them with no opportunities for malicious action.

The Fourier Transform

The Fourier transform is a powerful tool for signal processing, which takes in a signal and outputs its frequency spectrum, \textit{transforming} it from the time domain to the frequency domain. For continuous 1D real signals, the Fourier transform is defined as:

$$\mathscr{F} : \mathbb{R} \to \mathbb{R} \qquad S(f) = \mathscr{F}(s(t)) = \int_{-\infty}^{\infty} s(t) e^{-i2\pi ft}\, dt \tag{1}$$

Furthermore, the Fourier transform is invertible; to convert a signal from the frequency domain into the time domain, one uses the inverse Fourier transform:

$$\mathscr{F}^{-1} : \mathbb{R} \to \mathbb{R} \qquad s(t) = \mathscr{F}^{-1}(S(f)) = \int_{-\infty}^{\infty} S(f) e^{i2\pi ft}\, df \tag{2}$$

The Fourier transform is defined across both periodic and aperiodic signals, discrete and continuous, and for various quotient groups. For the purpose of studying Fourier watermarking of digital signals, we will be focusing on 1D and 2D real, discrete signals, which are operated on by the \textbf{Discrete Fourier Transform (DFT)} and its inverse. For all of our implementations, we use NumPy, a numerical computing library in Python, which implements the \textbf{Fast Fourier Transform} on signals. np.fft assumes that discrete signals are periodic with periodicity equal to their length and are defined on integer values. Therefore, the versions of the Fourier transforms used to act on these discrete signals are described below.

1D DFT: Transforms a 1D discrete periodic signal to a 1D discrete periodic signal.

$$\mathscr{F} : \frac{\mathbb{Z}(1)}{\mathbb{Z}(N)} \to \frac{\mathbb{Z}(1)}{\mathbb{Z}(N)} \qquad S[k] = \mathscr{F}(s[n]) = \sum_{n=0}^{N-1} s[n] e^{-\frac{i2\pi kn}{N}}$$

$$\mathscr{F} : \frac{\mathbb{Z}(1)}{\mathbb{Z}(N)} \to \frac{\mathbb{Z}(1)}{\mathbb{Z}(N)} \qquad s[n] = \mathscr{F}(S[k]) = \sum_{k=0}^{N-1} S[k] e^{\frac{i2\pi kn}{N}}$$

2D DFT: Transforms a 2D discrete periodic signal to a 2D discrete periodic signal.

$$\mathscr{F} : \frac{\mathbb{Z}(1) \times \mathbb{Z}(1)}{\mathbb{Z}(M) \times \mathbb{Z}(N)} \to \frac{\mathbb{Z}(1) \times \mathbb{Z}(1)}{\mathbb{Z}(M) \times \mathbb{Z}(N)}$$

$$S[k_1, k_2] = \mathscr{F}(s[n_1, n_2]) = \sum_{n_1=0}^{M-1} \sum_{n_2=0}^{N-1} s[n_1, n_2] e^{-\frac{i2\pi k_1 n_1}{M}} e^{-\frac{i2\pi k_2 n_2}{N}}$$

$$\mathscr{F}^{-1} : \frac{\mathbb{Z}(1) \times \mathbb{Z}(1)}{\mathbb{Z}(M) \times \mathbb{Z}(N)} \to \frac{\mathbb{Z}(1) \times \mathbb{Z}(1)}{\mathbb{Z}(M) \times \mathbb{Z}(N)}$$

$$s[n_1, n_2] = \mathscr{F}^{-1}(S[k_1, k_2]) = \sum_{k_1=0}^{M-1} \sum_{k_2=0}^{N-1} S[k_1, k_2] e^{\frac{i2\pi k_1 n_1}{M}} e^{\frac{i2\pi k_2 n_2}{N}}$$

Fourier Domain Watermarking Methods

The Fourier transform is useful for encoding watermarks since it allows the author to place information out of the sight/hearing of the user, while still embedding the information in the image itself. Various authors have proposed Fourier watermarking of signals before, for each of the signals under consideration.

Audio: Additive noise encoding binary information in inconspicuous regions of the human auditory spectrum [CITE PROPERLY Schalkwijk_2019] - Ultrasound spectrum (removed by most compression algorithms but undetectable based on physiology) - Low

frequency - it is difficult for humans to detect additive noise in the low frequency and compression algorithms tend to favor low frequency

Text Embedding a watermark in the Fourier domain is difficult because a small change in text is not only noticeable, but can change the meaning of the text entirely. There are two attempts at text Fourier watermarking that are noteworthy for this discussion.

CITE THE RIGHT FIGURES:

1. **Naive Text Watermarking:** If text is *tokenized* (by, for example, the ChatGPT tokenizing algorithm [^ChatGPT_tokenizer]), it can be treated as a 16-bit 1-dimensional signal (similar to an audio signal). If the DFT is performed on this signal, a watermark in the form of a signature filter is applied, and then the token signal is converted back to words, then information has been encoded in the Fourier domain of the text. However, this has disastrous results, as shown in Figure below; even a small change to the tokens in the Fourier domain causes a massive change to the text signal, since *tokens* of similar value do not correspond to *words* of a similar meaning.

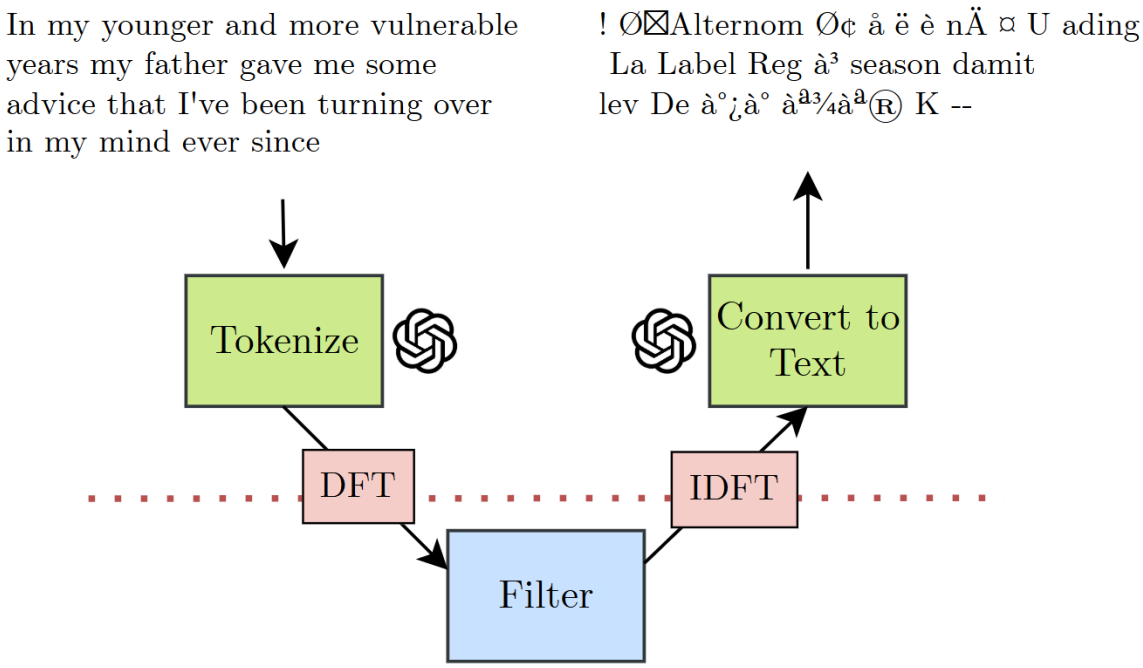


Figure: Native Text Watermarking

2. Using Token Ranks to Embed Watermark:

Fourier analysis can be also be used to *extract* a watermark from a signal. In a large language model (LLM), each subsequent generated word is chosen by calculating a probability for the next token based on the previous tokens. Traditionally, the word with

the highest probability, (and the highest rank) is chosen. However, by calculating an integer $f(k)$ corresponding to the k^{th} token in the sequence, and choosing the $f(k)^{th}$ ranked token, a sinusoidal signal can be embedded in the *ranks* of the tokens (see figure ATW) [5]. \[9pt]

By regenerating the text from an identical prompt, and calculating the difference in the rank for each token in the original and watermarked sequences, the watermark may be detected based on the signature frequency of the extracted difference (see figure ATW2). In this case, performing a DFT on the difference of ranks reduces the impact that noise/randomness has on the author's certainty that the document is or is not theirs.

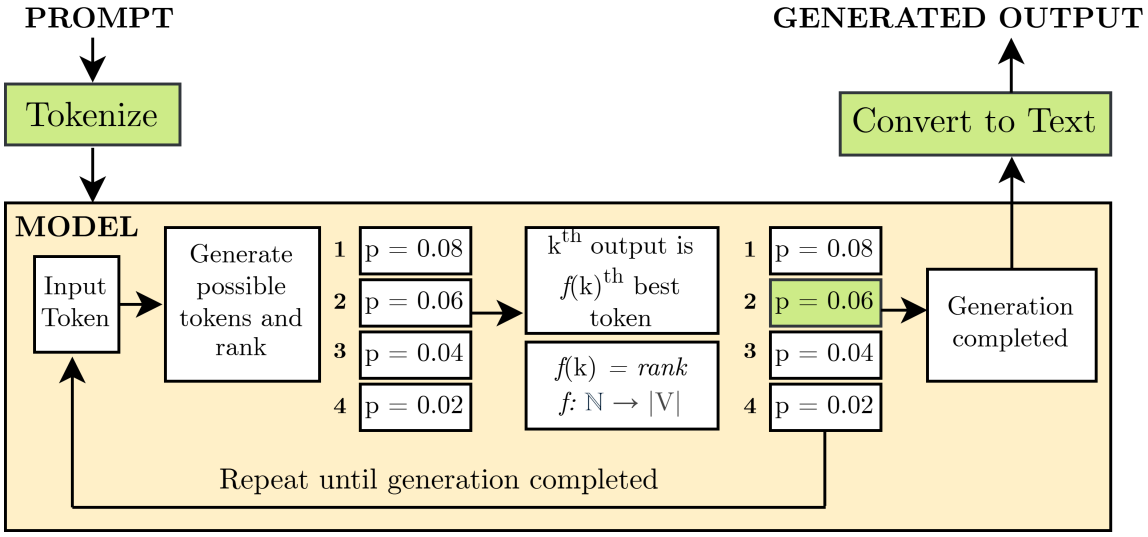


Figure ATW: Alternative Text Watermarking

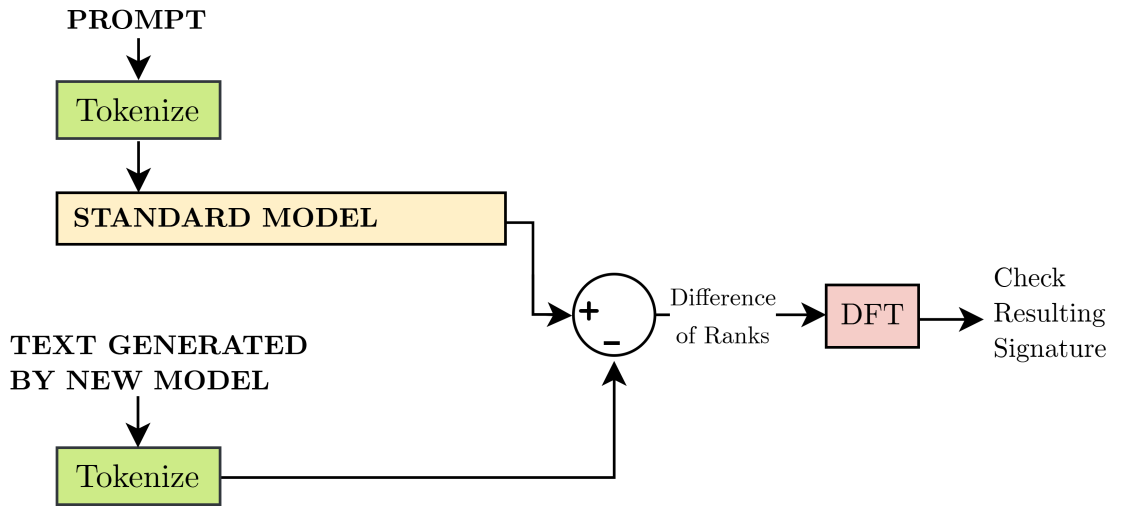


Figure ATW2: Alternative Text Watermarking with rank difference calculation

These two methods of text watermarking exemplify the two categories of "Fourier" watermarking; (1) the direct application of a watermark in the Fourier domain of a signal, and (2) the use of the Fourier transform to extract a watermark from a signal. These applications each see more use for watermarking images and videos.

Images: Fourier watermarking of images works in the same way as the naive attempt at text watermarking. For clarity, the algorithm for watermarking is reproduced below.

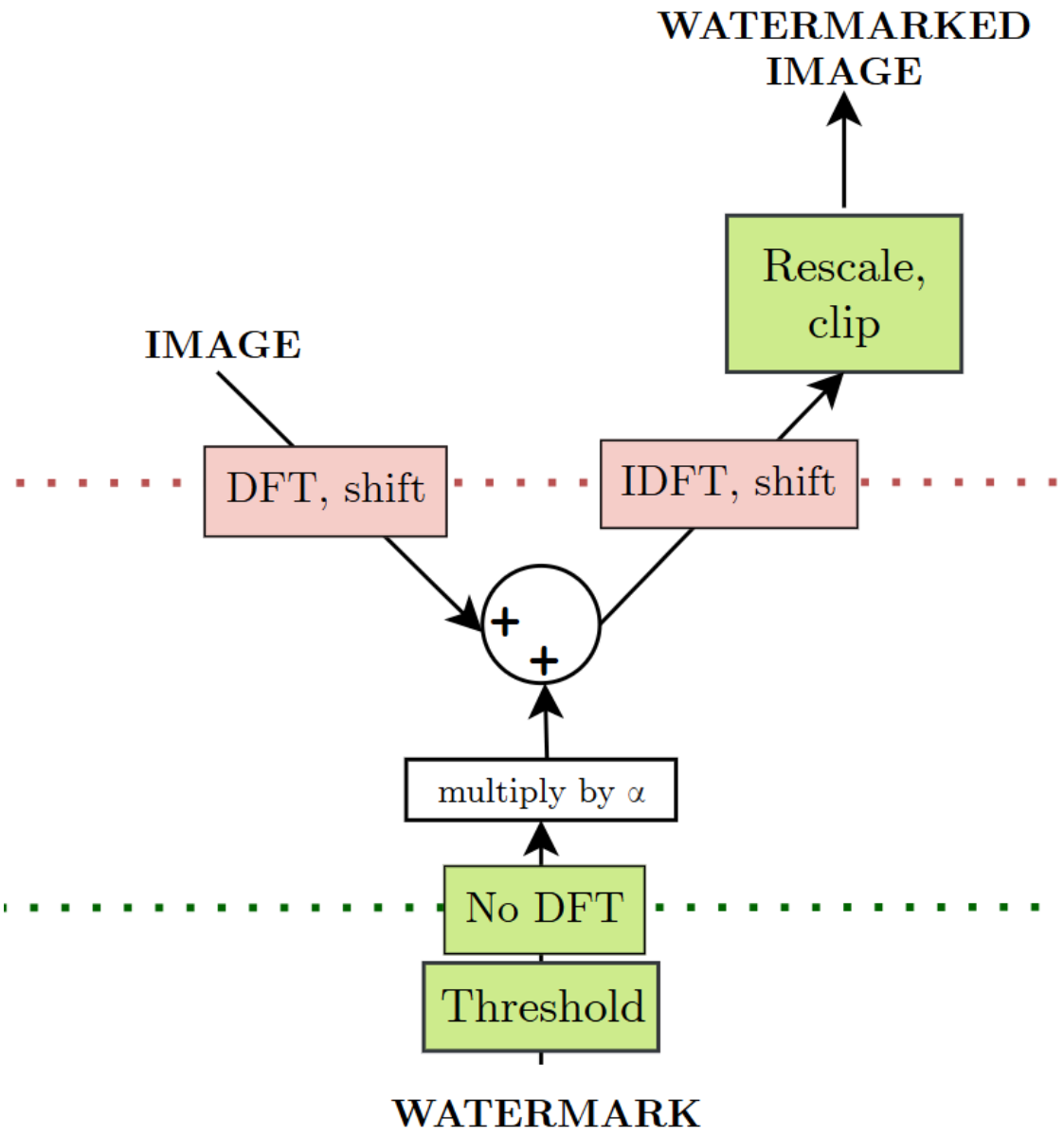


Figure: Image Watermarking

1. Load in Image
2. Perform DFT and DFT shift - high frequencies in the middle of the image
3. Load in Watermark
4. Threshold watermark image to convert to black and white
5. Multiply watermark image by parameter α to decrease intensity
6. Add watermark to image in upper left corner
7. Shift resulting image and apply IDFT
8. Rescale image to 0-255
9. Return watermarked image

Videos

We will present two methods of "Fourier" video watermarking: Simple Video Watermarking and Time-Domain Video Watermarking.

1. Simple Video Watermarking

The most intuitive method of video watermarking is to apply the image watermarking algorithm to each frame of the video separately. While this is simple and is effectively invisible to the eye (see Implementation), it requires FFT to be applied to every frame; although FFT runs in O(nlog(n))instead of the O(n^2) runtime of the FT itself, its speed is dependent on each of the dimensions of the video.

2. Time-Domain Video Watermarking

On the other hand, time-domain video watermarking only takes place in the time dimension of the video. By randomly selecting a pixel from the video, applying sinusoidal additive noise to it, and re-embedding it, a watermark can be added without needing to perform any Fourier transforms on the 2D frames of the video themselves (see figure below).

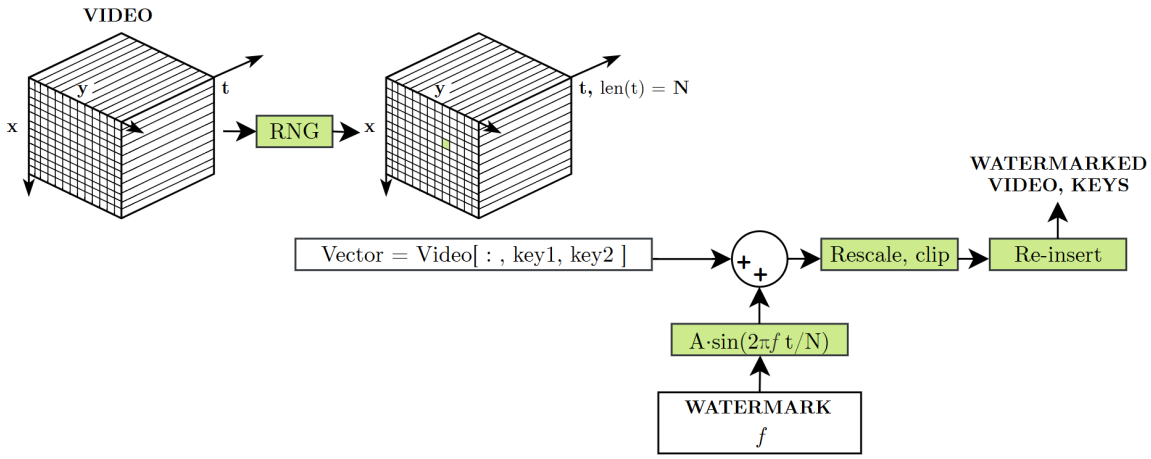


Figure: Video Watermarking

Implementation of Image and Video Watermarking

In [90]:
```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from matplotlib import cm
from matplotlib.animation import FuncAnimation, PillowWriter, ArtistAnimatic
import os
```

In [91]:
```
def load_image(image_path):
    """
    Load an image from the given path.
```

```
    Args:
        image_path (str): Path to the image file

    Returns:
        numpy.ndarray: Loaded image
    """
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if img is None:
        raise FileNotFoundError(f"Image not found at {image_path}")
    return img
```

In [92]:
```
def load_video(image_path):
    """
    Function to load in videos as 3D Numpy arrays
    """
    frames = []

    path = image_path
    cap = cv2.VideoCapture(path)
    ret = True

    while ret:
        ret, img = cap.read()  # read one frame
        if ret:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # convert to graysc
            frames.append(img)

    cap.release()

    video = np.stack(frames, axis=0)  # (T, H, W)

    return video
```

In [93]:
```
def embed_watermark(image, watermark, alpha=0.3):
    """
    Embed a watermark in a watermarked image using magnitude addition

    Args:
        image (numpy.ndarray): Image to be watermarked
        watermark (numpy.ndarray): logo/watermark to be embedded in the four
        optional: alpha: strength of watermark to be applied (stronger = mor
    """

    height, width = image.shape
    watermark = cv2.resize(watermark, (width//2, height//2))

    dft = np.fft.fft2(image)
    dft_shift = np.fft.fftshift(dft)
    magnitude = np.abs(dft_shift)
    phase = np.angle(dft_shift)

    construct_watermark = np.zeros_like(image)
    construct_watermark[0:watermark.shape[0], 0:watermark.shape[1]] = waterm
```

```
    # Add watermark to the central region (using slice assignment instead of
    magnitude += alpha * construct_watermark

    # Reconstruct the complex DFT
    dft_shift_watermarked = magnitude * np.exp(1j * phase)
    dft_ishift = np.fft.ifftshift(dft_shift_watermarked)
    img_back = np.fft.ifft2(dft_ishift)
    img_back = np.abs(img_back)
    img_back = np.clip(img_back, 0, 255).astype(np.uint8)
    return img_back
```

In [94]:
```
def embed_video_watermark_TD(video, watermark, alpha=0.3):
    """
    Embed a watermark in a video by marking one pixel on each frame

    Args:
        video (numpy.ndarray): video to be watermarked
        watermark (numpy.ndarray): a 1D signature/signal to be applied
        optional: alpha: strength of watermark to be applied
    """

    key1= int(np.random.uniform(low=0, high=video.shape[1]))
    key2 = int(np.random.uniform(low=0, high=video.shape[2]))

    print(key1, key2)

    video_back = video.copy()

    vector = video_back[:, key1, key2]

    vector += (watermark*alpha).astype(np.uint8)

    video_back[:, key1, key2] = np.clip(vector, 0, 255).astype(np.uint8)

    return video_back, key1, key2
```

In [95]:
```
def extract_watermark(original_image, watermarked_image, alpha=0.3):
    """
    Extract the watermark from a watermarked image using magnitude subtracti

    Args:
        original_image (numpy.ndarray): Original input image
        watermarked_image (numpy.ndarray): Watermarked image for watermark e
        optional: alpha: expected strength of watermark
    """
    dft_original = np.fft.fftshift(np.fft.fft2(original_image))
    dft_watermarked = np.fft.fftshift(np.fft.fft2(watermarked_image))
    mag_orig = np.abs(dft_original)
    mag_wm = np.abs(dft_watermarked)

    # Subtract the magnitude spectra to recover watermark
    raw_extracted = mag_wm - mag_orig

    raw_extracted = raw_extracted[0:original_image.shape[0]//2, 0: original_
```

```
    raw_extracted = cv2.resize(raw_extracted, (max(raw_extracted.shape), max

    # Recover the watermark intensities
    extracted = np.clip(raw_extracted / alpha, 0, 255).astype(np.uint8)
    return extracted
```

In [96]:
```
def extract_video_watermark_TD(video, watermarked_video, key1, key2, alpha):
    """
    Extract (Fourier-embedded) watermark from a video

    Args:
        video (numpy.ndarray): original video
        watermarked_video (numpy.ndarray): the video that has been watermark
        key1, key2: the pixel location used to embed the watermark
        alpha (float): embedding strength
    """
    # Get the pixel signal across frames
    original_vector = video[:, key1, key2].copy()
    watermarked_vector = watermarked_video[:, key1, key2].copy()

    diff = original_vector.astype(np.int16) - watermarked_vector.astype(np.i

    # Extract magnitude difference (watermark = diff / alpha)
    extracted = np.abs((np.fft.fft((diff)/alpha)))

    # Optional: interpolate
    extracted = np.interp((extracted), [np.min(extracted), np.max(extracted)
    return extracted
```

In [97]:
```
def visualize_spectrum(image, title="Magnitude Spectrum"):
    """
    Visualize the Fourier transform magnitude spectrum of an image.

    Args:
        image (numpy.ndarray): Input image
        title (str): Plot title
    """
    # Compute the 2D Fourier Transform
    f = np.fft.fft2(image)

    # Shift the zero frequency component to the center
    fshift = np.fft.fftshift(f)

    # Calculate the magnitude spectrum
    magnitude_spectrum = 20 * np.log(np.abs(fshift) + 1)

    # Display the results
    plt.figure(figsize=(10, 7))
    plt.subplot(121), plt.imshow(image, cmap='gray')
    plt.title('Input Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(magnitude_spectrum, cmap='gray')
    plt.title(title), plt.xticks([]), plt.yticks([])
    plt.show()
```

In [98]:

```python
def example_usage_image():
    """
    Example usage of the watermarking functions.
    """
    # Create output directory if it doesn't exist
    output_dir = "output"
    os.makedirs(output_dir, exist_ok=True)

    # Load images
    try:
        # Replace with your actual image paths
        original_image = load_image("input/ghibli_image.png")
        watermark = load_image("input/openai_watermark.png")

        # Embed watermark
        watermarked_image = embed_watermark(original_image, watermark, alpha

        # Save watermarked image
        cv2.imwrite(os.path.join(output_dir, "watermarked.png"), watermarked

        # Extract watermark
        extracted_watermark = extract_watermark(original_image, watermarked_

        # Save extracted watermark
        cv2.imwrite(os.path.join(output_dir, "extracted_image_watermark.png"

        # Visualize
        visualize_spectrum(original_image, "Original Image Spectrum")
        visualize_spectrum(watermarked_image, "Watermarked Image Spectrum")

        # Display the results
        plt.figure(figsize=(15, 10))

        plt.subplot(221), plt.imshow(original_image, cmap='gray')
        plt.title('Original Image'), plt.xticks([]), plt.yticks([])

        plt.subplot(222), plt.imshow(watermark, cmap='gray')
        plt.title('Watermark'), plt.xticks([]), plt.yticks([])

        plt.subplot(223), plt.imshow(watermarked_image, cmap='gray')
        plt.title('Watermarked Image'), plt.xticks([]), plt.yticks([])

        plt.subplot(224), plt.imshow(extracted_watermark, cmap='gray')
        plt.title('Extracted Watermark'), plt.xticks([]), plt.yticks([])

        plt.tight_layout()
        plt.savefig(os.path.join(output_dir, "image_watermarking_results.png
        plt.show()

        print("Watermarking process completed successfully!")
        print(f"All output files saved to {output_dir}/ directory.")

    except Exception as e:
        print(f"Error: {e}")
```

```
        print("Please ensure you have valid input images in the 'input' dire
        print("Required files: 'input/original.png' and 'input/watermark.png
```

In [ ]:

```
def example_usage_video():
    """
    Example usage of the watermarking function on each frame of a video
    """

    video = load_video("input/meatthezoo.mp4")
    watermark = cv2.resize(load_image("input/youtube_watermark.jpg"), (video
    _, watermark = cv2.threshold(watermark, 100, 255, cv2.THRESH_BINARY)
    plt.imshow(watermark, cmap=cm.gray, vmin=0, vmax=255)
    plt.title("Watermark")
    watermarked_video=np.stack([embed_watermark(video[i],watermark,alpha=0.3

    visualize_spectrum(video[0], "Original Image Spectrum")
    visualize_spectrum(watermarked_video[0], "Watermarked Image Spectrum")

    fig = plt.figure()
    video_frames =[]

    for i in range(video.shape[0]):
        video_frames.append([plt.imshow(watermarked_video[i], cmap=cm.gray,a

    video_ani = ArtistAnimation(fig, video_frames, interval=50, blit=True,
                                repeat_delay=1000)
    output_dir = "output"
    os.makedirs(output_dir, exist_ok=True)
    video_ani.save(os.path.join(output_dir, "watermarked_video.gif"), writer
    print("Saved watermarked video to output/watermarked_video.gif")

    watermark_frames =[]
    for i in range(video.shape[0]):
        watermark_frames.append([plt.imshow(extract_watermark(video[i], wate

    watermark_ani = ArtistAnimation(fig, watermark_frames, interval=50, blit
                                    repeat_delay=1000)

    output_dir = "output"
    os.makedirs(output_dir, exist_ok=True)
    watermark_ani.save(os.path.join(output_dir, "extracted_video_watermark.g
    print("Saved extracted watermark to output/extracted_video_watermark.gif
```

In [100…

```
def example_usage_video_TD():
    video = load_video("input/meatthezoo.mp4")
    time = np.linspace(0, video.shape[0], video.shape[0])
    freq = 10
    N = len(time)
    watermark = np.sin(2 * np.pi * time * freq / N)

    plt.title("watermark frequency spectrum")
    plt.plot(time, (np.fft.fft(watermark)))
    plt.show()
```

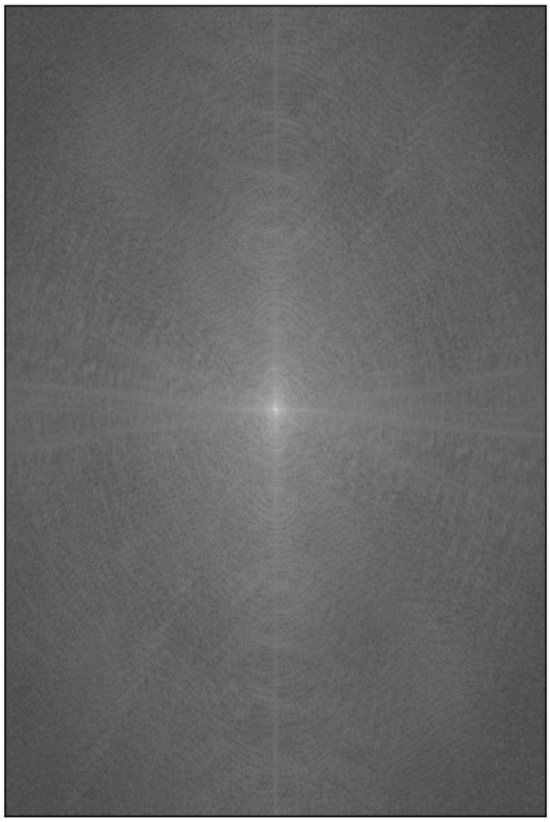
```
    watermarked_video, key1, key2 = embed_video_watermark_TD(video, watermar

    plt.title("watermarked video vector")
    plt.plot(time, watermarked_video[:, key1, key2])
    plt.show()
    plt.title("video vector")
    plt.plot(time, video[:, key1, key2])
    plt.show()
    plt.title("difference of watermarked and original videos")
    diff = watermarked_video[:, key1, key2].astype(np.int16) - video[:, key1
    plt.plot(time, diff)
    plt.show()

    fig = plt.figure()
    video_frames = []
    for i in range(video.shape[0]):
        video_frames.append([plt.imshow(watermarked_video[i], cmap=cm.gray,

    video_ani = ArtistAnimation(fig, video_frames, interval=50, blit=True, r

    output_dir = "output"
    os.makedirs(output_dir, exist_ok=True)
    video_ani.save(os.path.join(output_dir, "watermarked_video_TD.gif"), wri
    print("Saved watermarked video to output/watermarked_video_TD.gif")


    extracted_watermark = extract_video_watermark_TD(video, watermarked_vide

    # Create figure with two subplots
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

    # First subplot for the extracted watermark
    ax1.set_title("Frequency spikes of extracted watermark")
    ax1.plot(time, extracted_watermark)

    # Find the index of the max spike in the specified range
    search_range = extracted_watermark[0:len(time)//2]
    peak_index = np.argmax(search_range)
    peak_time = time[peak_index]

    # Add a dashed vertical line at the peak
    ax1.axvline(x=peak_time, color='r', linestyle='--', label=f'Peak at {pea
    ax1.legend()

    # Second subplot for the incorrect key extraction
    ax2.set_title("Extraction with incorrect keys")
    ax2.plot(time, extract_video_watermark_TD(video, watermarked_video, key1

    plt.tight_layout()
    plt.show()
```

In [101…

```
example_usage_image()
```

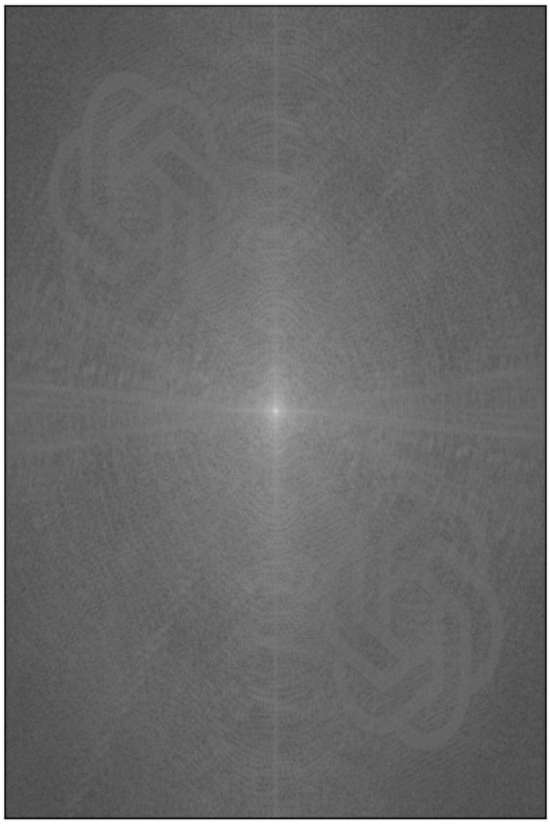
Input Image

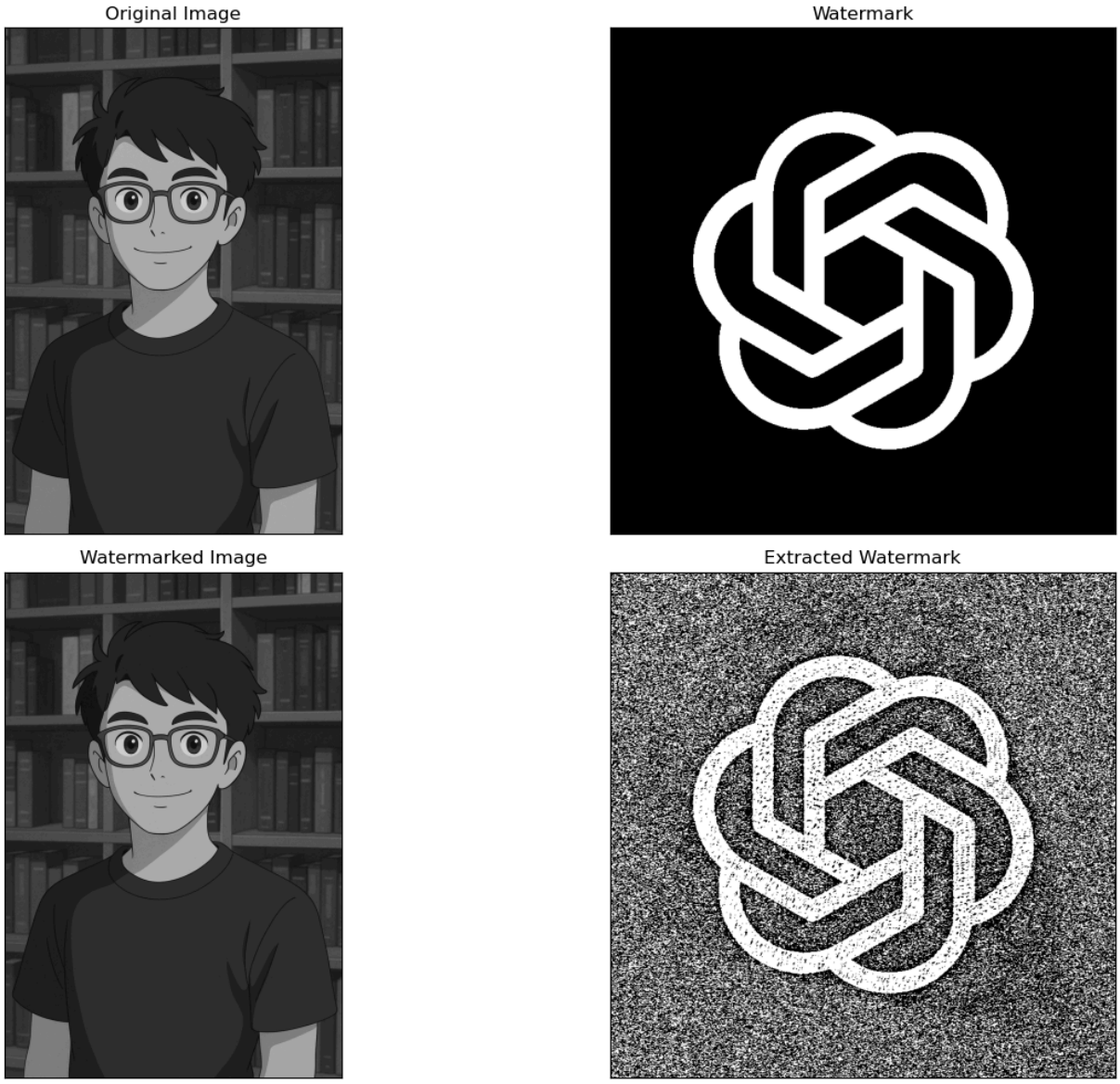
Original Image Spectrum



Input Image

Watermarked Image Spectrum





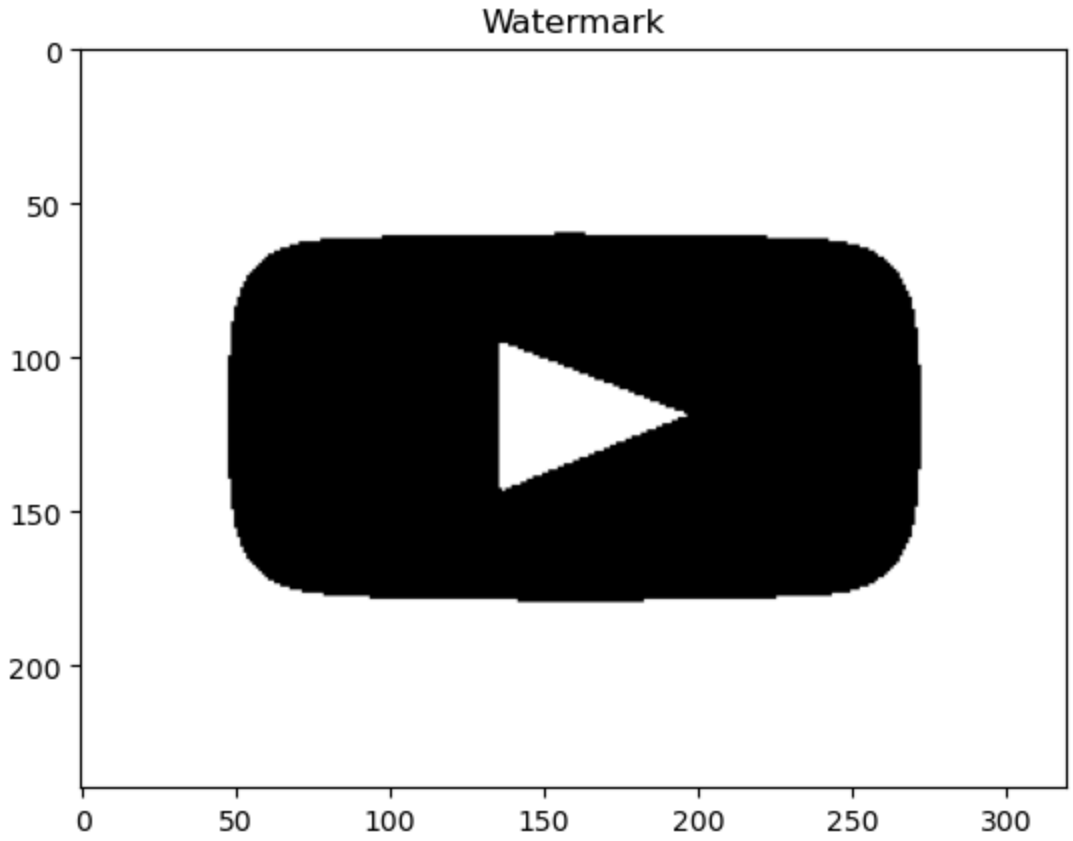

```
Watermarking process completed successfully!
All output files saved to output/ directory.
```

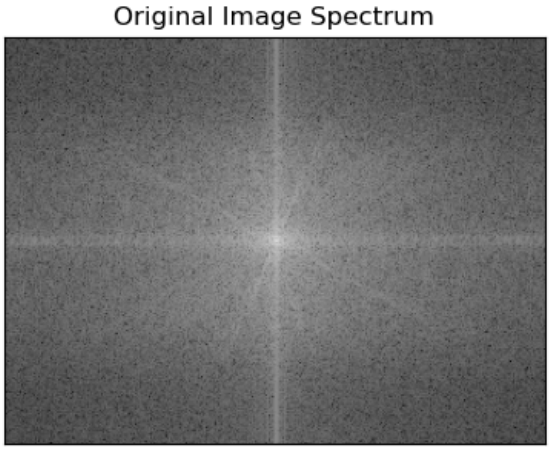
Analysis

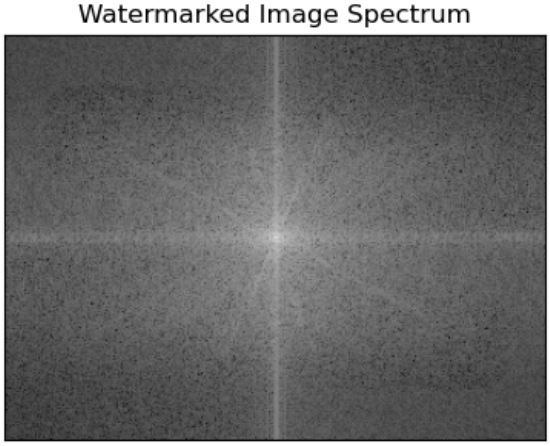
The results demonstrate the effectiveness of the DFT-based watermarking technique. The watermarked image spectrum shows the OpenAI logo watermark embedded in the image. The "Original Image" panel shows the unaltered input, while the "Watermarked Image" retains high visual fidelity, indicating that the watermark embedding process is imperceptible. The "Watermark" panel displays the binary pattern used for embedding. Notably, the "Extracted Watermark" panel reveals a clear and recognizable watermark, closely matching the original. This successful extraction underscores the robustness of the method, confirming that the watermark can be reliably retrieved without degrading the host image. Overall, the results validate the method's applicability for secure and imperceptible digital watermarking.

In [102…

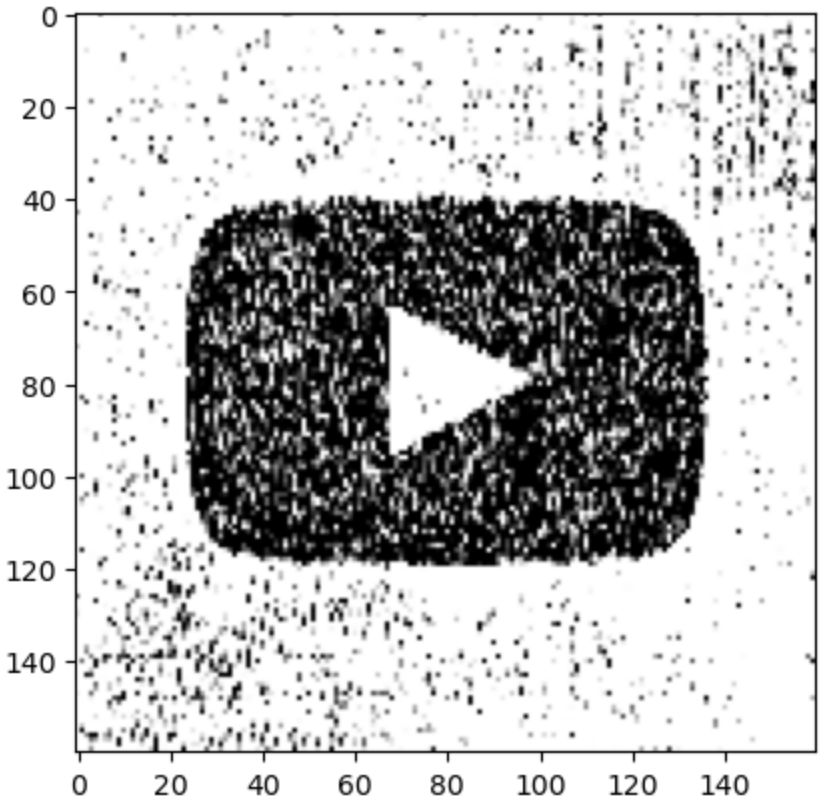
```
example_usage_video()
```








```
Saved watermarked video to output/watermarked_video.gif
Saved extracted watermark to output/extracted_watermark.gif
```



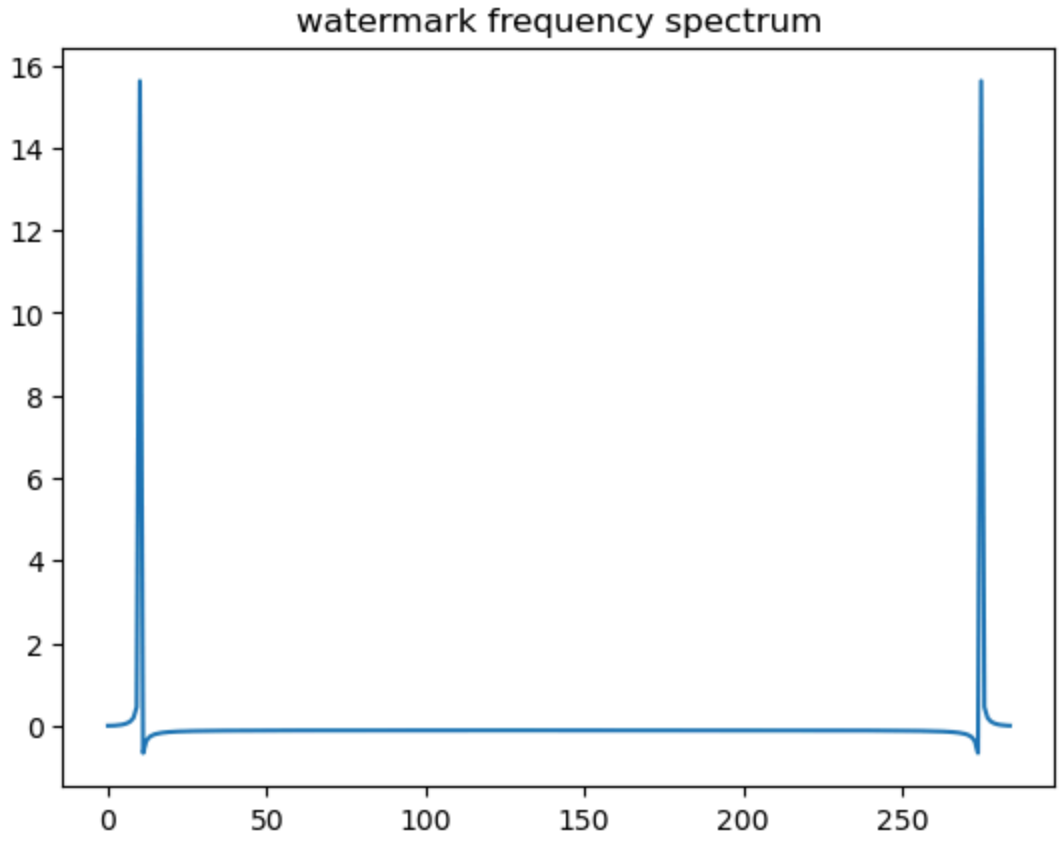
Analysis

The video watermarking technique effectively embeds and extracts a watermark using the DFT. An analysis of the provided results reveals several key observations. The spectrum of the watermarked image exhibits noticeable changes compared to the original image spectrum. In particular, darker pixels appear in specific regions, especially in the top-left and bottom-right corners. These changes indicate the presence of the watermark, which is embedded in the frequency domain. The alterations in these areas suggest that the watermarking process targets specific frequency components, allowing the watermark to remain imperceptible in the spatial domain while remaining detectable in the frequency domain. Furthermore, the extracted watermark image highlights the technique's robustness. The watermark is clearly visible and retains the recognizable features of the original design. This successful extraction confirms the method’s reliability in recovering the watermark from watermarked video frames, even after undergoing compression and other video processing operations.

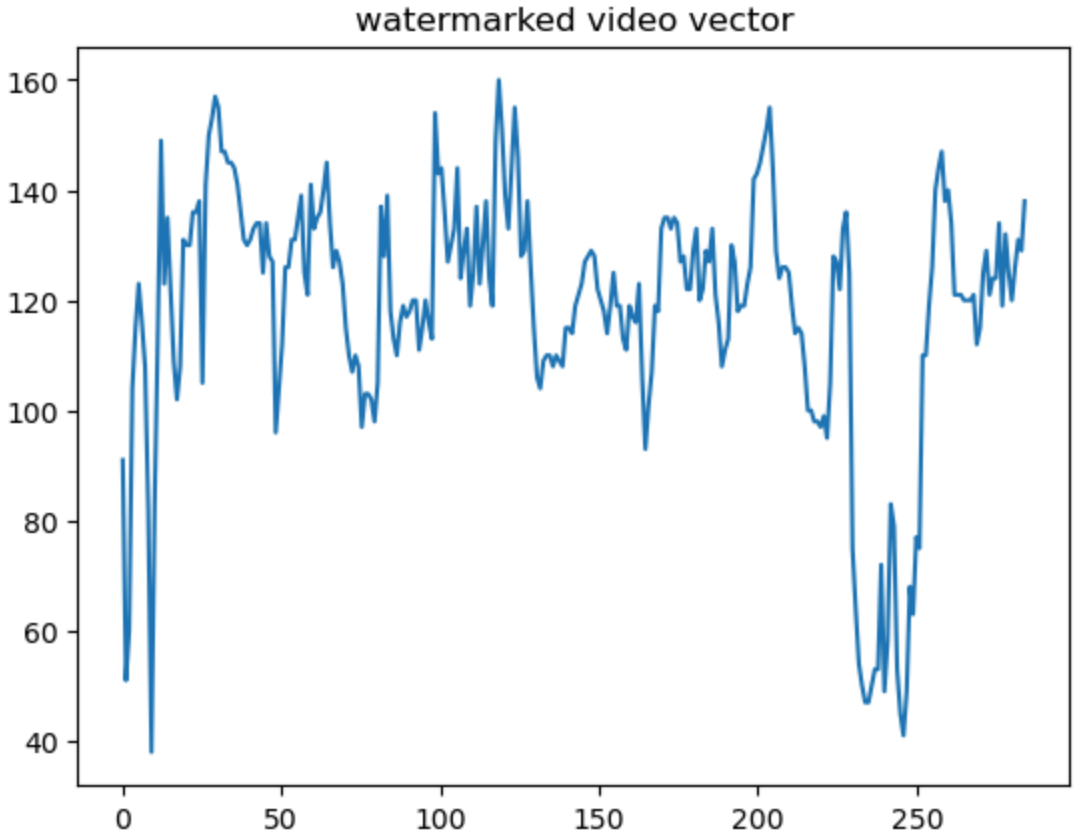
In [103…

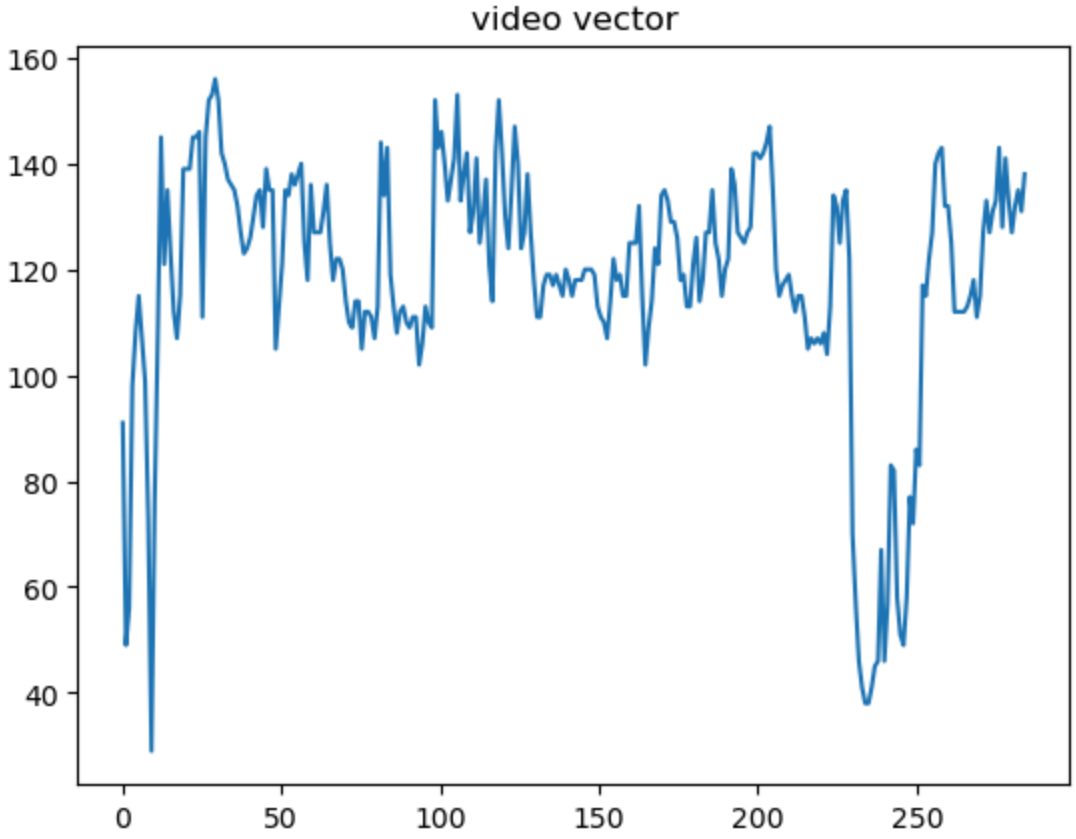
```
example_usage_video_TD()
```

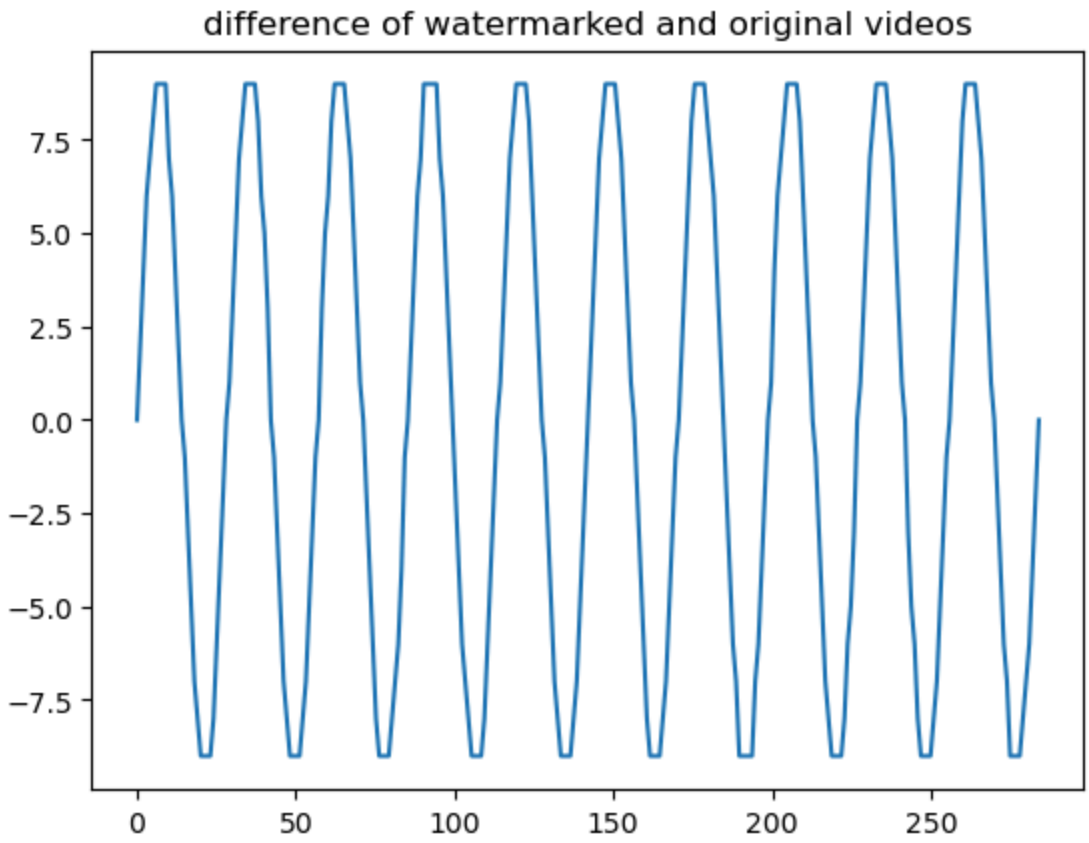
```
/Users/divyanshlalwani/miniconda3/envs/IntroDS/lib/python3.13/site-packages/
matplotlib/cbook.py:1709: ComplexWarning: Casting complex values to real dis
cards the imaginary part
  return math.isfinite(val)
/Users/divyanshlalwani/miniconda3/envs/IntroDS/lib/python3.13/site-packages/
matplotlib/cbook.py:1345: ComplexWarning: Casting complex values to real dis
cards the imaginary part
  return np.asarray(x, float)
```



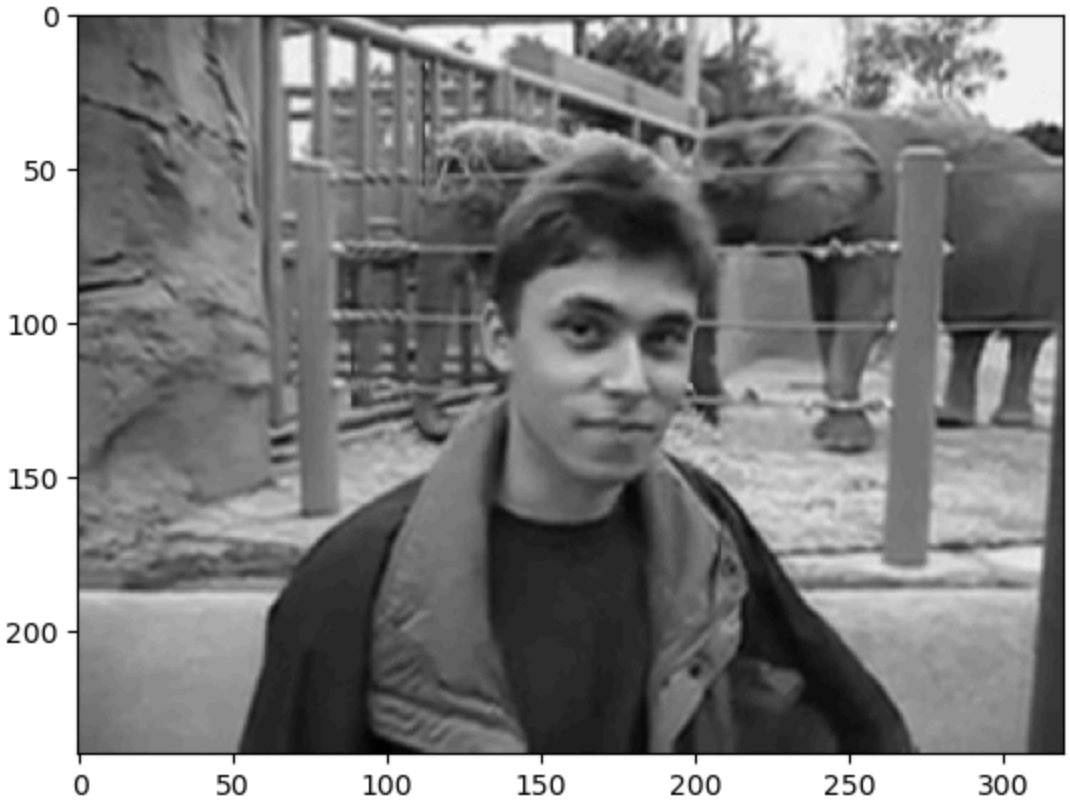

```
101 237
```

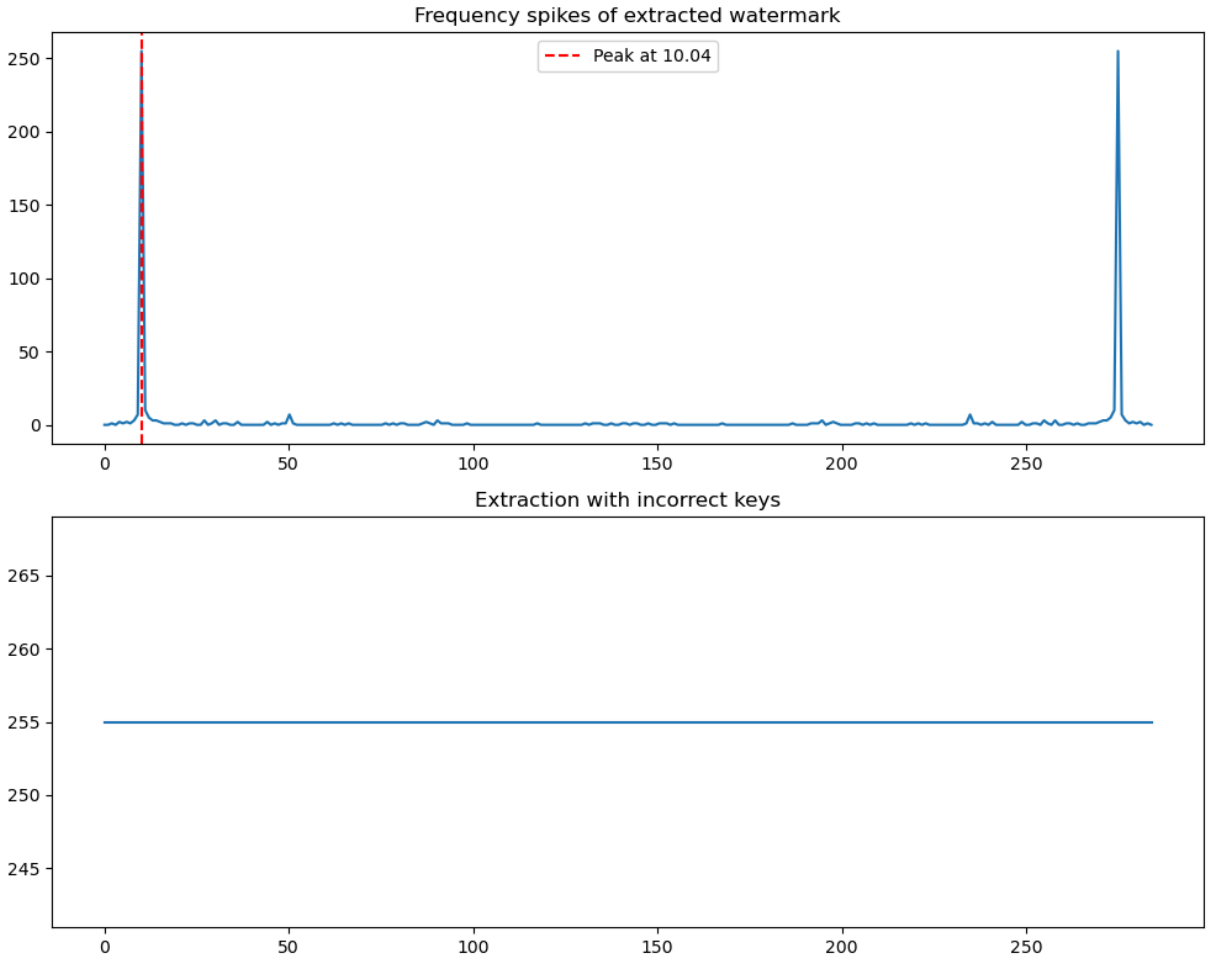

watermarked video vector
160
140
120
100
80
60
40
0
50
100
150
200
250


video vector
160
140
120
100
80
60
40
0
50
100
150
200
250




```
Saved watermarked video to output/watermarked_video_TD.gif
```





Analysis

watermark frequency spectrum figure:

The figure illustrates the frequency spectrum of a sinusoidal watermark signal embedded within a video. This watermark is created using a sine wave with a frequency of 10 Hz, which is clearly reflected by the presence of two prominent spikes in the spectrum. These peaks represent the positive and negative frequency components of the sine wave, confirming the successful embedding of the watermark in the frequency domain. The spectrum shows minimal energy outside these peaks, indicating that the watermark is a pure sinusoidal signal. Such a signal is particularly suitable for watermarking due to its simplicity, predictability, and ease of detection. The embedding process strategically targets specific frequency components, enhancing the watermark's robustness against typical video processing operations while maintaining invisibility in the spatial domain. Overall, this analysis demonstrates the effectiveness of using a sinusoidal watermark for video content, as it produces distinct frequency markers that can be reliably identified for purposes such as verification and authentication.

watermarked video vector, video vector, and difference of watermarked and original videos figures:

These figures demonstrate the impact of watermark embedding on a video signal by comparing a vector from the original video, the corresponding vector from the watermarked version, and their difference. The plot of the watermarked video vector reveals changes in pixel intensity over time, indicating the presence of the embedded watermark. These variations, while subtle, are integrated seamlessly into the video content. In contrast, the plot of the original video vector provides a baseline for comparison, enabling a clearer understanding of the modifications introduced by the watermark. The difference plot, which shows the subtraction of the original vector from the watermarked one, reveals a distinct periodic waveform. This waveform corresponds to the sinusoidal watermark signal, confirming that the watermark has been successfully embedded. The consistent sinusoidal pattern in the difference plot suggests that the watermark is applied uniformly across the video, preserving the integrity of the original content while ensuring the watermark can be reliably detected and extracted.

frequency spikes of extracted watermark and extraction with incorrect keys figure:

These plots demonstrate the effectiveness of the watermark extraction process and highlight the critical role of using correct keys. In the top plot, the frequency spectrum of the extracted watermark displays two prominent spikes, which correspond to the original watermark's frequency components. A dashed red line marks a peak at approximately 10.04 Hz, verifying that the watermark was successfully extracted at its intended frequency. The presence of these distinct peaks confirms the robustness of the watermarking technique and its ability to accurately recover the embedded signal. In contrast, the bottom plot shows the result of attempting extraction with incorrect keys. Here, the spectrum is flat, indicating that no meaningful signal was retrieved. This stark difference underscores the security strength of the watermarking method: only with the correct keys can the watermark be successfully extracted, ensuring that unauthorized attempts yield no useful information.

Conclusion

Watermarking, like cryptography, remains effective only as long as it can ensure protection against malicious actors. With the advent of generative AI, conventional watermarking techniques have become less effective; visible watermarks can easily be removed. The Fourier watermarking methods explored in this project provide a compromise between the easily removable watermarks of today and the concept of an ideal, indelible watermark.

Future work could focus on:

- Developing techniques to automatically detect and extract watermarks from videos
- Assessing the robustness of these watermarking methods against common modifications such as cropping, rotation, and noise addition

- Enhancing the invisibility and resilience of the watermark while maintaining minimal distortion to the original content

References

1. David Gilbertson. "Why llm watermarking will never work." https://ai.gopubby.com/why-llm-watermarking-will-never-work-1b76bdeebbd1, 2024. Accessed: 2025-05-05.

2. OpenAI Tokenizer. https://platform.openai.com/tokenizer, 2024. Accessed: 2025-05-05.

3. Jerome Schalkwijk. "Hiding data in sound," Oct 2019. Accessed: 2025-05-05. https://medium.com/intrasonics/hiding-data-in-sound-c8db3de5d6e0.

4. Wikimedia Commons. "Watermark sample.jpg," 2012. https://commons.wikimedia.org/wiki/File:Watermark_sample.jpg, 2025.

5. Zhenyu Xu and Victor S. Sheng. "Signal watermark on large language models," 2024. https://arxiv.org/pdf/2410.06545, 2025.