Appendix A

The summary of the key libraries and their respective functionalities utilized in the implementation.

Table 1 The list of common Python libraries used in the implementation [1] [2] [3] [4] [5] [6]

Library	Ver.	Functions	Main Purpose
`scipy.stats`	1.15.3	`pearsonr`	To get the Pearson's Correlation Coefficient of frames
`skimage.metrics`	0.25.2	`structural_similarity` (ssim)	To measure frames' similarities with SSIM
`scipy.fft`	1.15.3	`fft`, `fftfreq`, `rfft`, `rfftfreq`	To apply Fast Fourier Transformation
`sklearn.preprocessing`	1.7.0	`MinMaxScaler`	To normalize the ranges of similarity signals
`scipy.signal`	1.15.3	`correlated2d`	To get the Pearson's Correlation of frames
`matplotlib.pyplot`	3.10.0	Plotting functions	To visualize data
`numpy`	2.3.0	Numerical functions	To efficiently store, process, and retrieve data
`cv2`	4.11.0	Image processing functions	To manipulate frames' images for processing
`os`	3.13.0	Operating system functions	To do common file processing such as storing and loading
`pandas`	2.3.0	Series	To flatten images
`time`	3.13.0	`perf_counter`	To accurately measure the execution time.

References

[1] Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., *et al.*: Scipy 1.0: fundamental algorithms for scientific computing in python. Nature methods **17**(3), 261–272 (2020) https://doi.org/10.1038/s41592-019-0686-2

[2] Harris, C.R., Millman, K.J., Van Der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., *et al.*: Array programming with numpy. Nature **585**(7825), 357–362 (2020) https://doi.org/10.1038/s41586-020-2649-2

[3] McKinney, W.: Data structures for statistical computing in Python. Proceedings of the Python in Science Conferences, 56–61 (2010) https://doi.org/10.25080/majora-92bf1922-00a

[4] Walt, S., Schönberger, J.L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T.: scikit-image: image processing in python. PeerJ **2**, 453 (2014)

[5] Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in science & engineering **9**(03), 90–95 (2007)

[6] Bradski, G.: The opencv library. Dr. Dobb's Journal: Software Tools for the Professional Programmer **25**(11), 120–123 (2000)

Appendix B

A summary of the primary functions as well as their implementation for `pizza.mp4`.

Table 1 Primary functions and their roles

Function / Input Parameters	**Main Role**
`normalize_list_sklearn` • `lst`	Normalizes the values of the input parameter `lst` to the range [0,1].
`extract_frames` • `video_path` • `output_folder`	Extracts the frames of the video stored at `video_path` and saves each as a `.jpg` file in `output_folder`. Frames are numbered starting at 0000.
`compare_images` • `func` • `image_path1` • `image_path2`	Depending on the method specified by `func`, it returns the similarity measure of the given images.
`similarity_signal` • `func`	Returns a list containing the similarity measures between the reference frame (frame 0001) and all frames.
`plot_similarity` • `sim_metric` • `y_values` • `y_range`	Plots the similarity signal in the specified range.
`second_max_index` • `arr`	Returns the index of the second maximum in the given array. Used to get the FFT's highest frequency.
`find_period` • `signal`	Using FFT, returns the period of the input signal.
`autoCorrelation` • `signal`	Using Autocorrelation, returns the period of the input signal.
`time_analysis` • `image_path1` • `image_path2`	Collects execution times for all similarity measurement functions used in the study.
`cepstrumAnalysis` • `signal`	Using Cepstrum Analysis, finds the period of the input signal.
`__main__`	Does the following steps: 1. Extracts frames from the video and stores them in a designated folder. 2. Calculates the similarity measures between the reference frame and all other frames using various methods and saves the results in a text file. 3. Reads the text file generated in the previous step and organizes the similarity values into a structured dataset. 4. Visualizes the similarity signals with respect to their original ranges using plots. 5. Normalizes the similarity signals to the range [0,1] and plots them for comparison. 6. Uses the selected methods to estimate the period of the selected similarity signal.

Listing 1 normalize_list_sklearn

```python
def normalize_list_sklearn(lst):
    scaler = MinMaxScaler()
    return scaler.fit_transform([[x] for x in lst]).flatten()
```

Listing 2 extract_frames

```python
def extract_frames(video_path, output_folder):
    # Ensure the output folder exists
    os.makedirs(output_folder, exist_ok=True)
```

```
    # Open the video file
    video = cv2.VideoCapture(video_path)
    if not video.isOpened():
        raise ValueError(f"Error opening video file: {video_path}")

    frame_count = 0
    while True:
        ret, frame = video.read()
        if not ret:
            break

        # Save the frame as an image file
        frame_filename = os.path.join(output_folder, f"frame_{frame_count:04d}.jpg")
        cv2.imwrite(frame_filename, frame)

        frame_count += 1

    # Release the video capture object
    video.release()

    print(f"Extracted {frame_count} frames to '{output_folder}'")
    return frame_count
```

Listing 3 compare_images

```
def compare_images(func, image_path1, image_path2):
    image1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    image2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

    # Check if images were loaded correctly
    if image1 is None or image2 is None:
        raise ValueError("One or both image paths are invalid or the images cannot be
            read.")

    # Resize images to the same dimensions, if necessary
    if image1.shape != image2.shape:
        image2 = cv2.resize(image2, (image1.shape[1], image1.shape[0]))

    # Compute SSIM between the images
    if func == 'ssim':
        similarity, _ = ssim(image1, image2, full=True)

    #very inefficient
    elif func == 'ncc':
        image1 = np.array(image1)
        image2 = np.array(image2)
        similarity = correlate2d(image1, image2, boundary='symm', mode='same').max()

    elif func == 'mse':
        similarity = np.mean((image1 - image2) ** 2)

    elif func == 'psnr':
        mse_value = np.mean((image1 - image2) ** 2)
        if mse_value == 0: # Images are identical
            return float('inf')
        max_pixel = 255.0
        similarity = 20 * np.log10(max_pixel / np.sqrt(mse_value))/30

    elif func == 'orb':
        orb = cv2.ORB_create()
        keypoints1, descriptors1 = orb.detectAndCompute(image1, None)
        keypoints2, descriptors2 = orb.detectAndCompute(image2, None)

        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(descriptors1, descriptors2)
        similarity = len(matches)
```

```
    elif func == 'histogram':
        hist1 = cv2.calcHist([image1], [0], None, [256], [0, 256])
        hist2 = cv2.calcHist([image2], [0], None, [256], [0, 256])
        similarity = cv2.compareHist(hist1, hist2, cv2.HISTCMP_CORREL)

    elif func == 'lpips':
        loss_fn = lpips.LPIPS(net='alex') # Pretrained on AlexNet
        image1_tensor = image1
        image2_tensor = image2
        similarity = loss_fn(image1_tensor, image2_tensor)

    elif func == 'cor1':
        # Calculate correlation
        image1 = image1.flatten()
        image2 = image2.flatten()
        similarity = pearsonr(image1, image2)[0]

    elif func == 'cor2':
        # Calculate correlation
        image1 = image1.flatten()
        image2 = image2.flatten()
        similarity = np.corrcoef(image1, image2)[0,1]

    elif func == 'diff':
        # Calculate correlation
        image1 = image1.flatten()
        image2 = image2.flatten()
        diff = np.subtract(image1, image2)
        similarity = diff

    return similarity
```

Listing 4 similarity_signal

```
def similarity_Signal(func):
    simSig = []

    #for pizza.mp4
    image_path1 = "output_frames/frame_0001.jpg"
    for i in range(305):
        image_path2 = "output_frames/frame_" + f"{i:04}" +".jpg"

        simSig.append(compare_images(func,image_path1, image_path2))

    return simSig
```

Listing 5 plot_similarity

```
def plot_similarity(sim_metric, y_values, y_range):
    # Generate x values in the range from 0 to 305
    x = np.linspace(0, 305, 305)

    y = y_values

    # Create the plot
    plt.plot(x, y, label=sim_metric)
    plt.ylim(y_range[0], y_range[1])

    # Add labels and title
    plt.xlabel("frame")
    plt.ylabel("similarity")
    plt.title("Frames Similarities to frame 0001")

    # Add a legend
```

```
    plt.legend()

    # Show the grid
    plt.grid()

    # Display the plot
    plt.show()
```

Listing 6 second_max_index

```
def second_max_index(arr):
    # Find the index of the maximum value
    max_index = np.argmax(arr)

    # Create a copy of the array and set the maximum value to negative infinity
    arr_copy = np.copy(arr)
    arr_copy[max_index] = -np.inf

    # Find the index of the second maximum value
    second_max_index = np.argmax(arr_copy)

    return second_max_index
```

Listing 7 find_period

```
def find_period(signal):
    # Sample data
    x = np.linspace(0, 305, 305)
    y = signal

    # Calculate FFT
    yf = fft(y)
    xf = fftfreq(len(y), x[1] - x[0])

    # Find the dominant frequency
    dominant_frequency = xf[second_max_index(np.abs(yf))]

    # Calculate the period
    period = 1 / dominant_frequency

    # Plot the results
    plt.figure(figsize=(12, 6))

    plt.subplot(2, 2, 2)
    plt.plot(x, y)
    plt.title('Original Signal')

    #xf = np.where(xf >= 0)

    plt.subplot(2, 2, 1)
    plt.plot(xf, np.abs(yf))
    plt.title('FFT')
    plt.xlabel('Frequency')
    plt.ylabel('Amplitude')

    plt.tight_layout()
    plt.show()

    print("\t\t\tDominant Frequency: %.3f , Period: %.3f" %(dominant_frequency, period))
```

Listing 8 autoCorrelation

```
def autoCorrelation(signal):
    fs = 305 # Sampling rate
    t = np.arange(0, 1, 1/fs) # Time vector
```

```
    autocorr = np.correlate(signal, signal, mode='full')
    autocorr = autocorr[len(signal)-1:] # Only take positive lags

    peaks = np.diff(np.sign(np.diff(autocorr))) < 0 # Find local maxima
    peak_indexes = np.where(peaks)[0] + 1

    if len(peak_indexes) <= 0:
        print("\t\t\tNo clear period found.")
        period = 0
    else:

        period = peak_indexes[0] / fs # Period in seconds
        print(f"\t\tPeriod: {(305/25)*period:.3f} seconds" ,end = "")
        print(f"\tPeriod: {305*period:.3f} frames") # 25 frames / second

    plt.figure(figsize=(12, 6))
    plt.subplot(2, 2, 2)
    plt.plot(t*(305/25), signal)
    plt.title("Signal")

    plt.subplot(2, 2, 1)
    plt.plot((305/25)*np.arange(len(autocorr))/fs, autocorr)
    plt.title("Autocorrelation")
    plt.xlabel("Lag (seconds)")
    plt.scatter((305/25)*period, autocorr[int(period*fs)], color='red', marker='o',
        label=f'Period: {(305/25)*period:.3f} s')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

Listing 9 time_analysis

```
def time_analysis(image_path1, image_path2):

    execution_times = []

    # Read the images
    image1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    image2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

    # Check if images were loaded correctly
    if image1 is None or image2 is None:
        raise ValueError("One or both image paths are invalid or the images cannot be
            read.")

    # Resize images to the same dimensions, if necessary
    if image1.shape != image2.shape:
        image2 = cv2.resize(image2, (image1.shape[1], image1.shape[0]))

    # SSIM
    start_time = time.perf_counter()

    ssim(image1, image2, full=True)

    end_time = time.perf_counter()
    execution_time = end_time - start_time
    execution_times.append( ("ssim", execution_time) )
    print(f"Execution time: {execution_time:.4f} seconds")


    # MSE
    start_time = time.perf_counter()

    np.mean((image1 - image2) ** 2)
```

```
end_time = time.perf_counter()
execution_time = end_time - start_time
execution_times.append( ("mse", execution_time) )
print(f"Execution time: {execution_time:.4f} seconds")

# PSNR
mse_value = np.mean((image1 - image2) ** 2)
if mse_value == 0: # Images are identical
    float('inf')
max_pixel = 255.0
start_time = time.perf_counter()

20 * np.log10(max_pixel / np.sqrt(mse_value))/30

end_time = time.perf_counter()
execution_time = end_time - start_time
execution_times.append( ("pnsr", execution_time) )
print(f"Execution time: {execution_time:.4f} seconds")

# ORB
start_time = time.perf_counter()

orb = cv2.ORB_create()
keypoints1, descriptors1 = orb.detectAndCompute(image1, None)
keypoints2, descriptors2 = orb.detectAndCompute(image2, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(descriptors1, descriptors2)
len(matches)

end_time = time.perf_counter()
execution_time = end_time - start_time
execution_times.append( ("orb", execution_time) )
print(f"Execution time: {execution_time:.4f} seconds")

# Histogram
start_time = time.perf_counter()

hist1 = cv2.calcHist([image1], [0], None, [256], [0, 256])
hist2 = cv2.calcHist([image2], [0], None, [256], [0, 256])
cv2.compareHist(hist1, hist2, cv2.HISTCMP_CORREL)

end_time = time.perf_counter()
execution_time = end_time - start_time
execution_times.append( ("histogram", execution_time) )
print(f"Execution time: {execution_time:.4f} seconds")

# Pearson's Correlation 1
start_time = time.perf_counter()

image1f1 = image1.flatten()
image2f2 = image2.flatten()
pearsonr(image1f1, image2f2)[0]

end_time = time.perf_counter()
execution_time = end_time - start_time
execution_times.append( ("cor1", execution_time) )
print(f"Execution time: {execution_time:.4f} seconds")

# Pearson's Correlation 2
start_time = time.perf_counter()

image1ff1 = image1.flatten()
image2ff2 = image2.flatten()

np.corrcoef(image1ff1, image2ff2)[0,1]
```

```python
    end_time = time.perf_counter()
    execution_time = end_time - start_time
    execution_times.append( ("cor2", execution_time) )
    print(f"Execution time: {execution_time:.4f} seconds")

    return execution_times
```

Listing 10 cepstrumAnalysis

```python
def cepstrumAnalysis(signal):
    # Define the signal
    fs = 305 # Sampling frequency

    t = np.arange(0, 1, 1/fs) # Time vector

    # Calculate the cepstrum
    spectrum = np.fft.fft(signal)
    log_spectrum = np.log(np.abs(spectrum))
    cepstrum = np.fft.ifft(log_spectrum)

    # Identify the peak
    cepstrum = np.abs(cepstrum) # Take the absolute value to get the real cepstrum
    peak_index = np.argmax(cepstrum[3:-3]) + 3 # Exclude first samples

    # Calculate the period
    period = peak_index / fs

    # Plot the signal and the cepstrum
    plt.figure(figsize=(12, 6))

    plt.subplot(2, 2, 2)
    plt.plot((305/25)*t, signal)
    plt.title('Signal')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')

    plt.subplot(2, 2, 1)
    plt.plot((305/25)*np.arange(len(cepstrum))/fs, cepstrum)
    plt.title('Cepstrum')
    plt.xlabel('Quefrency (s)')
    plt.ylabel('Amplitude')
    plt.axvline(x=(305/25)*period, color='r', linestyle='--', label=f'Period =
        {(305/25)*period:.3f} s')
    plt.legend()

    plt.tight_layout()
    plt.show()

    print(f"\t\tPeriod: {(305/25)*period:.3f} seconds\tPeriod: {305*period:.3f} frames")
```