

**NOAKHALI SCIENCE & TECHNOLOGY UNIVERSITY**

**Noakhali-3814**

**Lab Report**

**Course Code**: ICE-4206

**Course Title:** Multimedia Communication Lab

|  |  |
| --- | --- |
| **Submitted By**  Mohammad Ajij  ID: MUH2011023M  Session: 2019-20  Department of ICE, NSTU | **Submitted To**  Mahbubul Alam Naeem  Assistant Professor  Department of ICE, NSTU |

**Date of Submission: 26th May, 2025.**

**Experiment No: 01**

**Experiment Name: A program to implement different functions on an image.**

**Theoretical Background:**

Multimedia communication involves the transmission and reception of multimedia data, which includes images. Image processing is a crucial aspect of multimedia communication, enabling various operations such as conversion between color spaces, clipping, cropping, and copy-pasting regions of interest (ROIs) within images.

1. **imread** - Reads an image file into memory as a numerical array
2. **imshow** - Displays an image in a graphical window
3. **imresize** - Changes image dimensions (scales up/down)
4. **imcrop** - Extracts a rectangular region from an image
5. **imwrite** - Saves an image to disk in specified format
6. **iminfo** - Shows image metadata (size, format, bit depth)
7. **im2gray/rgb2gray** - Converts color image to grayscale
8. **imhist** - Calculates pixel intensity distribution
9. **im2frame** - Converts image to animation/video frame
10. **edge** - Detects boundaries/contours in images
11. **hough** - Identifies geometric shapes (lines, circles)
12. **impixel** - Gets color values at specific coordinates
13. **bwconncomp** - Finds connected pixel groups in binary images
14. **immsc** - Applies morphological operations (erosion/dilation)
15. **dct2** - 2D Discrete Cosine Transform for frequency analysis
16. **idct2** - Inverse DCT for image reconstruction
17. **imfilter** - Applies convolution filters (blur, sharpen)
18. **imgaussfilt** - Gaussian smoothing filter
19. **gabor** - Texture analysis filter
20. **imgaborfilt** - Applies multiple Gabor filters
21. **montage** - Combines multiple images in a grid
22. **imsave** - Python equivalent of imwrite
23. **imrotate** - Rotates image by specified angle
24. **imgerase** - Removes/masks selected image regions

**Code Implementation:**

% --- Initial Setup ---

pkg load image;

image\_filename = "/home/k452b/octave/input.png";

% Check if the image file exists

if (!exist(image\_filename, "file"))

printf("Error: Image file '%s' not found. Please provide a valid image.\n", image\_filename);

printf("You can try Octave's built-in 'autumn.tif' for some examples.\n");

if (exist("autumn.tif", "file")) % Octave usually ships with this

image\_filename = "autumn.tif";

printf("Using 'autumn.tif' as a fallback.\n");

else

error("No suitable test image found. Exiting.");

endif

endif

printf("--- Running Image Processing Examples ---\n\n");

% --- 1. imread - Read image from file ---

img = imread(image\_filename);

disp("1. imread - Image loaded into variable 'img'");

% --- 2. imshow - Display image ---

figure; % Open a new figure window

imshow(img);

title("2. imshow - Displayed Image");

disp("2. imshow - Image displayed in a new window.");



% --- 3. imresize - Change image dimensions ---

resized\_img\_half = imresize(img, 0.5); % Scale to 50%

resized\_img\_specific = imresize(img, [100, 150]); % Resize to 100 rows, 150 columns

figure;

subplot(1,3,1); imshow(img); title("Original");

subplot(1,3,2); imshow(resized\_img\_half); title("Resized (0.5x)");

subplot(1,3,3); imshow(resized\_img\_specific); title("Resized (100x150)");

title("3. imresize - Resized Images");

disp("3. imresize - Image resized and displayed.");

% --- 4. imcrop - Extract rectangular region ---

rect = [50, 50, size(img,2)/2, size(img,1)/2]; % Crop a quadrant

cropped\_img = imcrop(img, rect);

figure;

subplot(1,2,1); imshow(img); title("Original for Crop");

rectangle('Position', rect, 'EdgeColor', 'r', 'LineWidth', 2);

subplot(1,2,2); imshow(cropped\_img); title("Cropped Image");

title("4. imcrop - Cropped Image");

disp("4. imcrop - Image cropped and displayed.");



% --- 5. imwrite - Save image to disk ---

imwrite(cropped\_img, "cropped\_input.png");

imwrite(img, "input\_copy.jpg", "Quality", 90); % For JPEG, can specify quality

disp("5. imwrite - 'cropped\_input.png' and 'input\_copy.jpg' saved to disk.");

% --- 6. imfinfo - Show image metadata ---

info = imfinfo(image\_filename);

disp("6. imfinfo - Image File Information:");

disp(info);

printf(" Filename: %s, Format: %s, Width: %d, Height: %d, BitDepth: %d\n", ...

info.Filename, info.Format, info.Width, info.Height, info.BitDepth);

**Output:**

FileModDate = 23-May-2025 10:59:40

FileSize = 766916

Format = PNG

Width = 740

Height = 494

BitDepth = 8

ColorType = truecolor

% --- 7. rgb2gray - Convert color image to grayscale ---

gray\_img = []; % Initialize

if (ndims(img) == 3) % Check if it's a color image (3 dimensions: HxWxChannels)

gray\_img = rgb2gray(img);

figure;

subplot(1,2,1); imshow(img); title("Original Color");

subplot(1,2,2); imshow(gray\_img); title("Grayscale");

title("7. rgb2gray - Color to Grayscale Conversion");

disp("7. rgb2gray - Image converted to grayscale and displayed.");

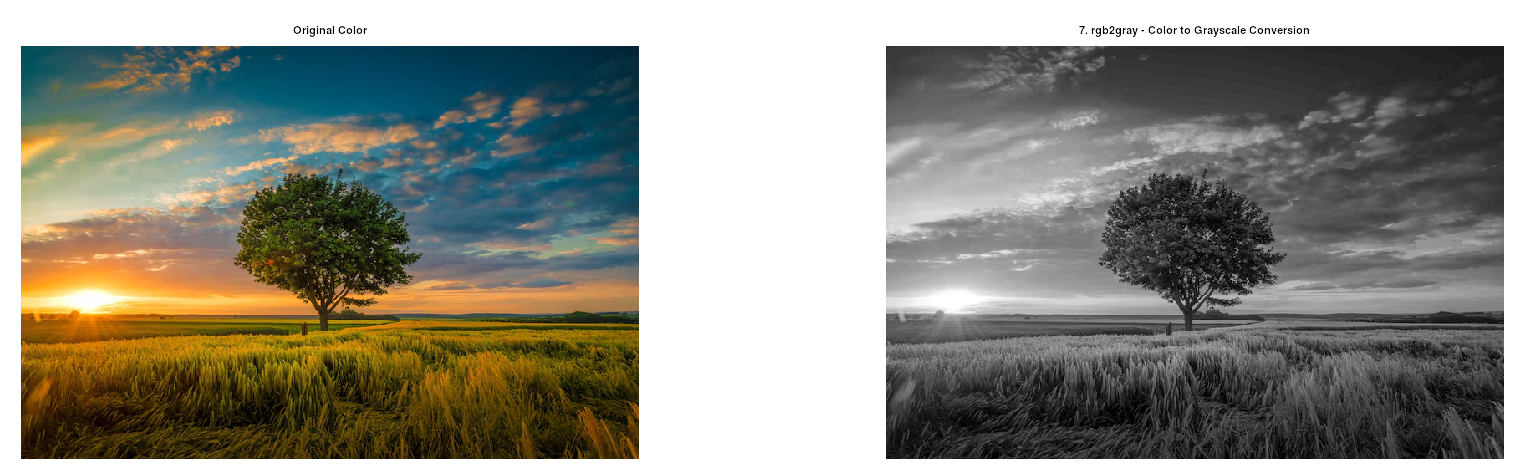
else

gray\_img = img; % Already grayscale or binary

disp("7. rgb2gray - Image is already grayscale or binary. Using as is.");

figure; imshow(gray\_img); title("Original Grayscale/Binary");

end



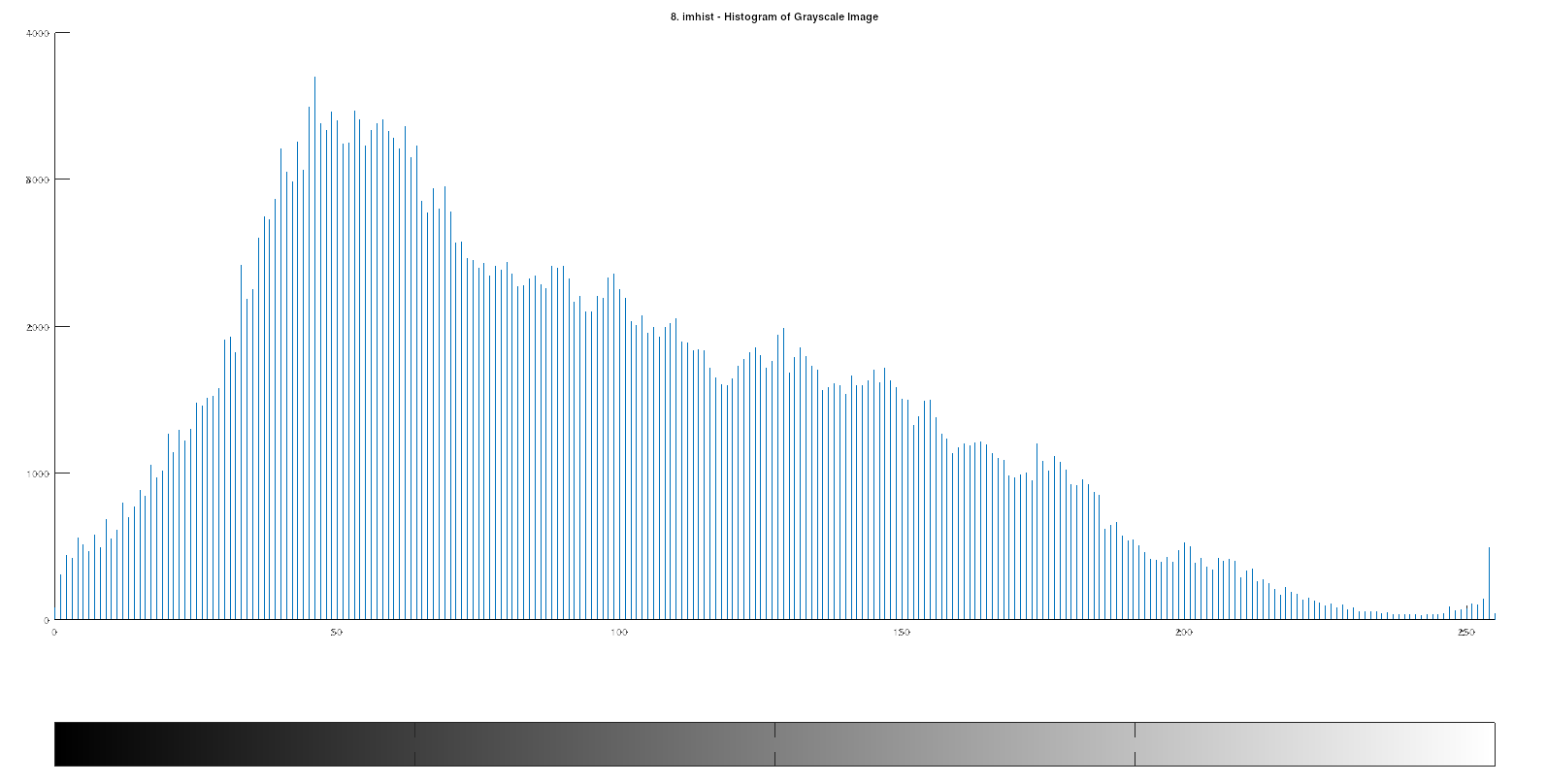
% --- 8. imhist - Calculate pixel intensity distribution ---

figure;

imhist(gray\_img);

title("8. imhist - Histogram of Grayscale Image");

disp("8. imhist - Grayscale image histogram displayed.");



% --- 9. im2frame - Convert image to animation/video frame ---

frame\_struct = im2frame(img); % F.cdata contains image, F.colormap if indexed

disp("9. im2frame - Image converted to a frame structure 'frame\_struct'.");

% --- 10. edge - Detect boundaries/contours in images ---

edges\_sobel = edge(gray\_img, "sobel");

edges\_canny = edge(gray\_img, "canny");

figure;

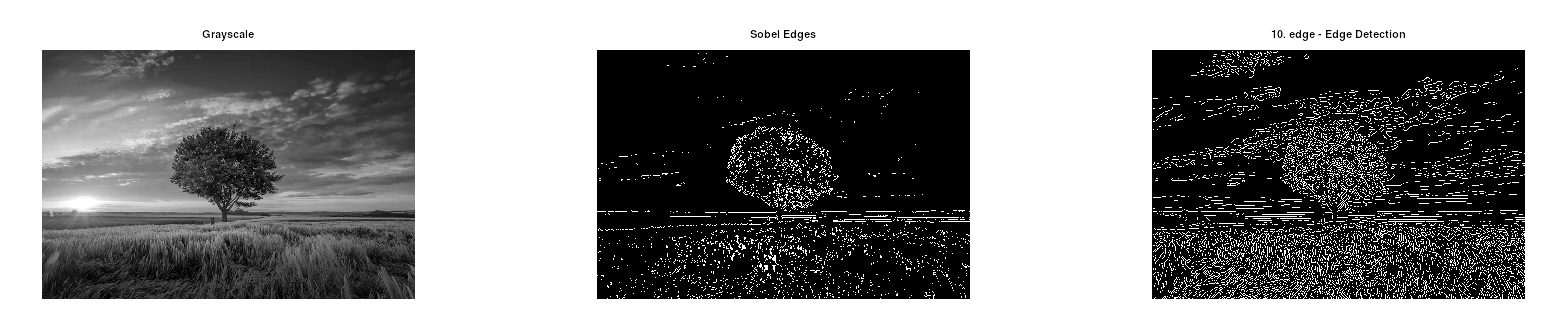
subplot(1,3,1); imshow(gray\_img); title("Grayscale");

subplot(1,3,2); imshow(edges\_sobel); title("Sobel Edges");

subplot(1,3,3); imshow(edges\_canny); title("Canny Edges");

title("10. edge - Edge Detection");

disp("10. edge - Edges detected and displayed.");

% --- 11. hough - Identifies geometric shapes (lines) ---

% (Using edges\_canny from step 10, which should be a binary image)

[H, T, R] = hough(edges\_canny); % H: Hough transform, T: theta, R: rho

figure;

imshow(H, [], "XData", T, "YData", R, "InitialMagnification", "fit");

xlabel('\theta (degrees)'); ylabel('\rho');

axis on; axis normal; hold on;

colormap(gca, hot);

title("11. hough - Hough Transform Accumulator");

% Find peaks in Hough transform

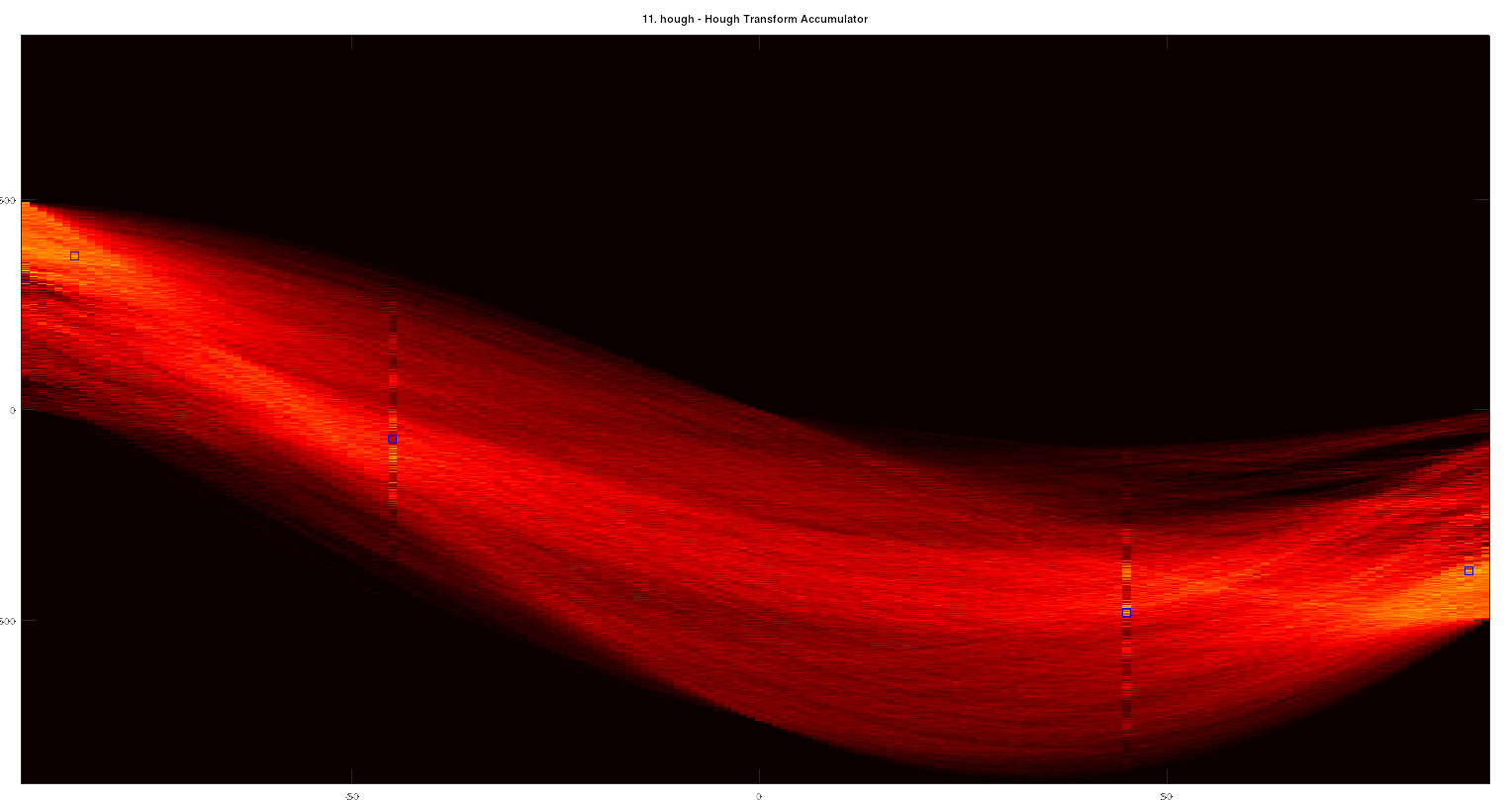
P = houghpeaks(H, 5, "threshold", ceil(0.3\*max(H(:))));

plot(T(P(:,2)), R(P(:,1)), "s", "color", "blue");

hold off;

disp("11. hough - Hough transform calculated and peaks identified.");

% To draw lines: use houghlines(edges\_canny, T, R, P, ...)



% --- 12. impixel - Gets color values at specific coordinates ---

row\_coord = 100; col\_coord = 150;

pixel\_value\_at\_coord = impixel(img, col\_coord, row\_coord);

printf("12. impixel - Pixel value at (row %d, col %d): ", row\_coord, col\_coord);

disp(pixel\_value\_at\_coord); % For color: [R G B], for gray: [Intensity]

**Output:**

impixel - Pixel value at (row 100, col 150): 178 178 178

% --- 13. bwconncomp - Finds connected pixel groups in binary images ---

binary\_test\_img = zeros(50,50);

binary\_test\_img(10:20, 10:20) = 1;

binary\_test\_img(30:40, 30:40) = 1;

binary\_test\_img(15,15) = 0; % make a hole in the first square

CC = bwconncomp(binary\_test\_img);

disp("13. bwconncomp - Connected Components Information:");

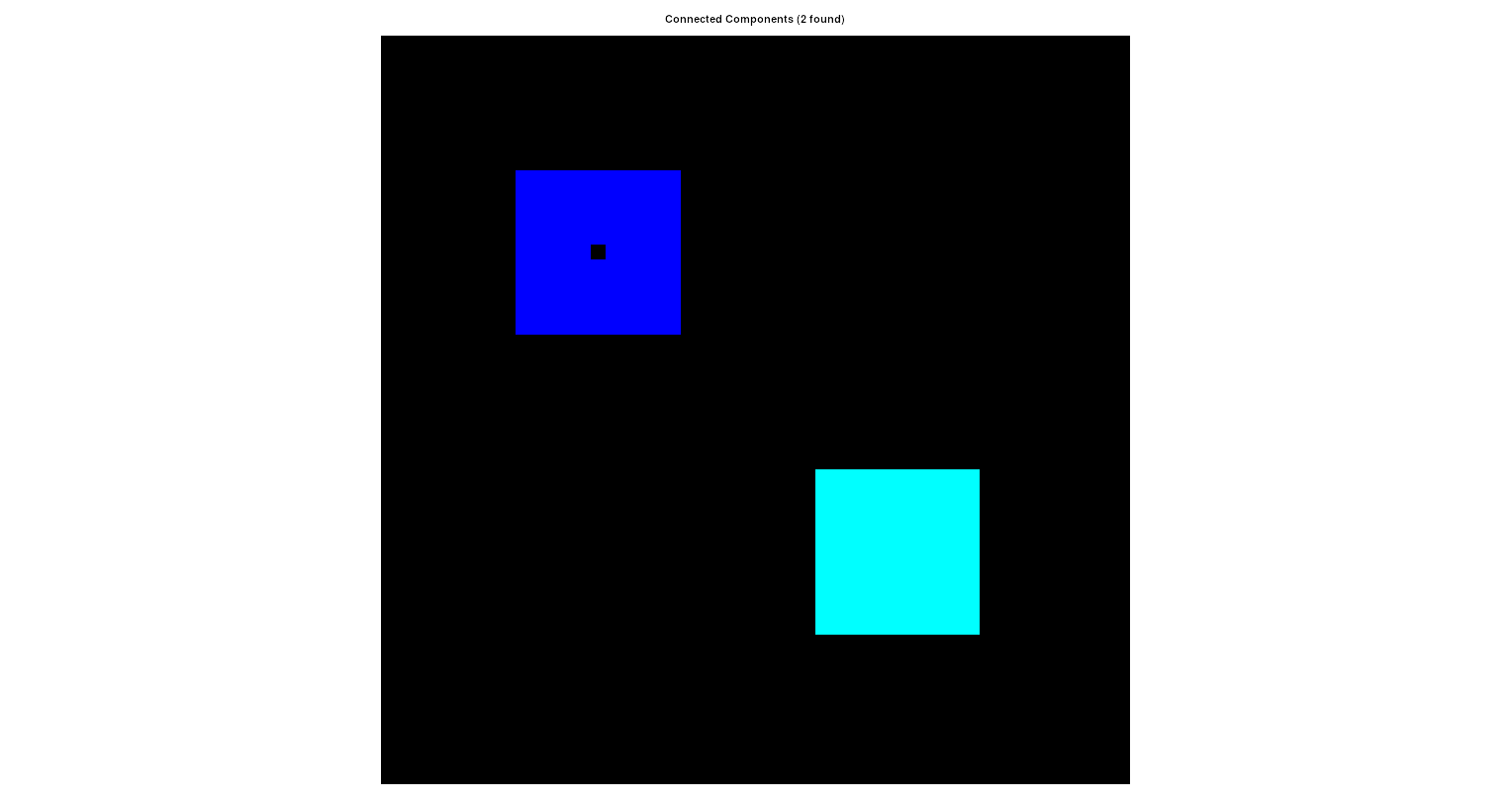
disp(CC);

printf(" Number of connected objects found: %d\n", CC.NumObjects);

labeled\_img = labelmatrix(CC);

figure; imshow(label2rgb(labeled\_img, @jet, 'k', 'shuffle'));

title(sprintf("Connected Components (%d found)", CC.NumObjects));



% --- 14. Morphological operations (imdilate, imerode) ---

se = strel("disk", 2, 0); % Structuring element: disk of radius 2, N=0 approximation

dilated\_img = imdilate(binary\_test\_img, se);

eroded\_img = imerode(binary\_test\_img, se);

figure;

subplot(1,3,1); imshow(binary\_test\_img); title("Original Binary");

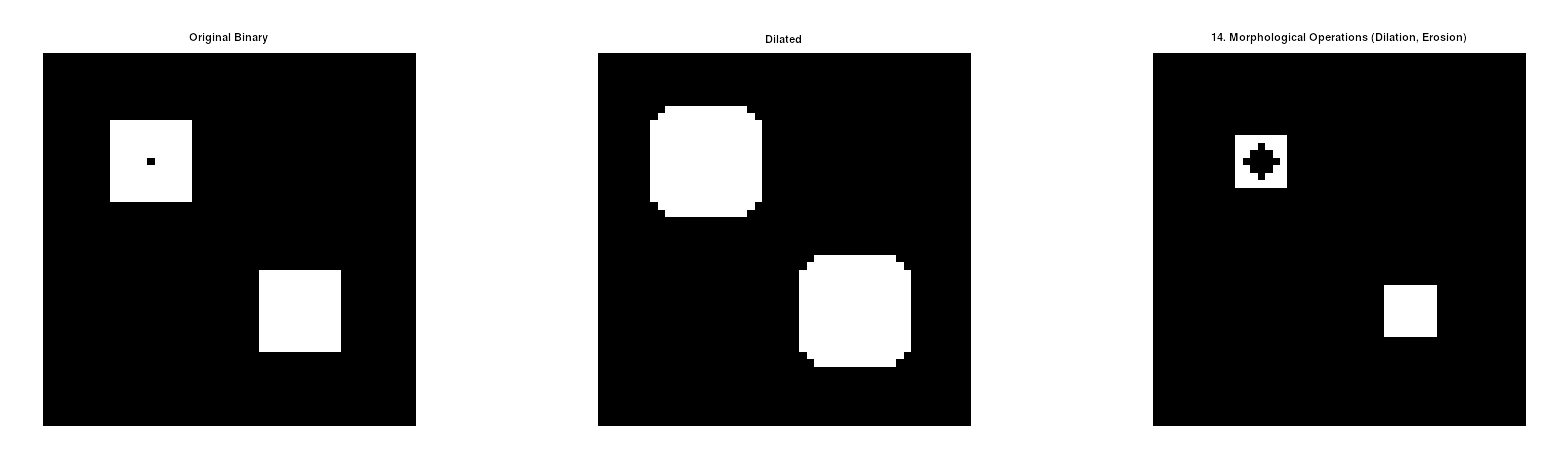
subplot(1,3,2); imshow(dilated\_img); title("Dilated");

subplot(1,3,3); imshow(eroded\_img); title("Eroded");

title("14. Morphological Operations (Dilation, Erosion)");

disp("14. Morphological operations (imdilate, imerode) applied and shown.");

**Output:**



% --- 15. dct2 - 2D Discrete Cosine Transform ---

disp("--- 15. dct2 (using workaround if needed) ---");

if (exist('dct2', 'file'))

dct\_coeffs = dct2(double(gray\_img)); % Input must be double precision

disp("Using built-in dct2.");

else

disp("Built-in dct2 not found. Using my\_dct2 workaround.");

dct\_coeffs = my\_dct2(double(gray\_img));

endif

figure;

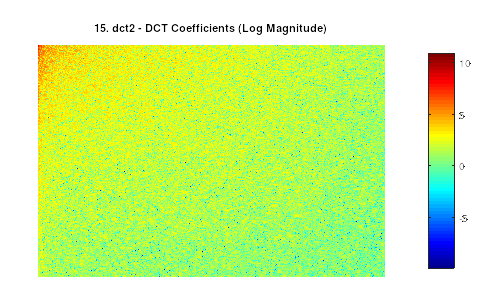
imshow(log(abs(dct\_coeffs)), []); % Display log magnitude for visualization

colormap(gca, jet); colorbar;

title("15. dct2 - DCT Coefficients (Log Magnitude)");

disp("dct2 - 2D DCT applied and coefficients displayed.");

disp(" ");



% --- 16. idct2 - Inverse DCT for image reconstruction ---

disp("--- 16. idct2 (using workaround if needed) ---");

% (Using dct\_coeffs from step 15)

if (exist('idct2', 'file'))

reconstructed\_img\_dct = idct2(dct\_coeffs);

disp("Using built-in idct2.");

else

disp("Built-in idct2 not found. Using my\_idct2 workaround.");

reconstructed\_img\_dct = my\_idct2(dct\_coeffs);

endif

figure;

subplot(1,2,1); imshow(gray\_img); title("Original Grayscale");

subplot(1,2,2); imshow(uint8(reconstructed\_img\_dct)); title("Reconstructed via IDCT");

title("16. idct2 - Inverse DCT Reconstruction");

disp("idct2 - Image reconstructed from DCT coefficients.");

disp(" ");



% --- 17. imfilter - Applies convolution filters ---

h\_avg = fspecial("average", [5 5]); % Averaging filter kernel

blurred\_img\_filter = imfilter(gray\_img, h\_avg, "replicate"); % "replicate" handles border

h\_laplacian = fspecial("laplacian", 0.2);

sharpened\_img\_filter = gray\_img - imfilter(gray\_img, h\_laplacian, "replicate");

figure;

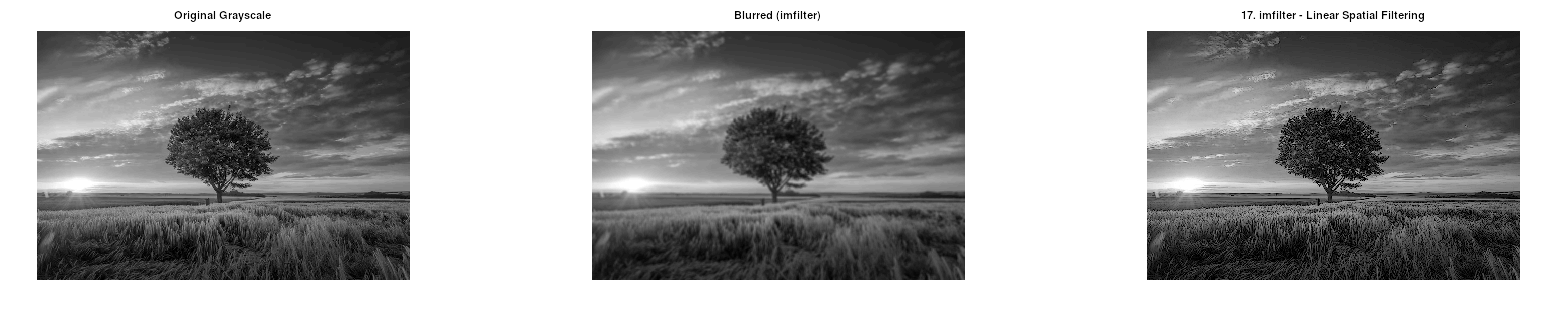
subplot(1,3,1); imshow(gray\_img); title("Original Grayscale");

subplot(1,3,2); imshow(blurred\_img\_filter); title("Blurred (imfilter)");

subplot(1,3,3); imshow(sharpened\_img\_filter); title("Sharpened (imfilter)");

title("17. imfilter - Linear Spatial Filtering");

disp("17. imfilter - Averaging and sharpening filters applied.");

% --- 18. imgaussfilt - Gaussian smoothing filter ---

sigma\_gauss = 2; % Standard deviation of Gaussian

gaussian\_blurred\_img = imgaussfilt(gray\_img, sigma\_gauss);

figure;

subplot(1,2,1); imshow(gray\_img); title("Original Grayscale");

subplot(1,2,2); imshow(gaussian\_blurred\_img); title(sprintf("Gaussian Blurred (sigma=%.1f)", sigma\_gauss));

title("18. imgaussfilt - Gaussian Smoothing");

disp("18. imgaussfilt - Gaussian filter applied.");



% --- 19. gabor - Texture analysis filter (kernel creation) ---

lambda = 10; theta\_rad = pi/4; psi = 0; gamma\_aspect = 0.5; sigma\_env = 5; sz = 21;

gb\_kernel\_real = zeros(sz, sz);

for x = -floor(sz/2):floor(sz/2)

for y = -floor(sz/2):floor(sz/2)

x\_theta = x \* cos(theta\_rad) + y \* sin(theta\_rad);

y\_theta = -x \* sin(theta\_rad) + y \* cos(theta\_rad);

idx\_x = x + floor(sz/2) + 1; idx\_y = y + floor(sz/2) + 1;

gb\_kernel\_real(idx\_y, idx\_x) = exp(-(x\_theta^2 + gamma\_aspect^2 \* y\_theta^2) / (2 \* sigma\_env^2)) \* cos(2 \* pi \* x\_theta / lambda + psi);

end

end

figure; imshow(gb\_kernel\_real, []); title("19. Gabor Filter Kernel (Real Part - Simplified)");

disp("19. gabor - Simplified Gabor kernel created and displayed.");

disp(" For applying Gabor filters, see 'imgaborfilt' or 'gaborfilter'.");



% --- 20. imgaborfilt / gaborfilter - Applies Gabor filters ---

if (exist('gaborfilter', 'file'))

sigma\_g = 4; freq\_g = 1/8; theta\_g\_rad = pi/4; % Example parameters

[gabor\_real\_comp, gabor\_imag\_comp] = gaborfilter(double(gray\_img), sigma\_g, freq\_g, theta\_g\_rad);

gabor\_mag = sqrt(gabor\_real\_comp.^2 + gabor\_imag\_comp.^2);

figure; imshow(gabor\_mag, []); title("20. Gabor Filter Magnitude (single filter via gaborfilter)");

disp("20. gaborfilter - Single Gabor filter applied, magnitude shown.");

else

disp("20. gaborfilter - 'gaborfilter' function not found in image package. Skipping.");

end



% --- 21. montage - Combines multiple images in a grid ---

if ndims(gray\_img) == 2, gray\_img\_rgb = cat(3, gray\_img, gray\_img, gray\_img); else gray\_img\_rgb = gray\_img; end

if ndims(edges\_canny) == 2, edges\_canny\_rgb = cat(3, uint8(edges\_canny)\*255, uint8(edges\_canny)\*255, uint8(edges\_canny)\*255); else edges\_canny\_rgb = edges\_canny; end

img\_s = imresize(img, [128 128]);

gray\_s = imresize(gray\_img\_rgb, [128 128]);

edges\_s = imresize(edges\_canny\_rgb, [128 128]);

montage\_array = cat(4, img\_s, gray\_s, edges\_s); % Create 4D array HxWxDxN

figure;

montage(montage\_array);

title("21. montage - Montage of Images");

disp("21. montage - Multiple images displayed as a montage.");



% --- 22. imsave (Python equivalent of imwrite) ---

imwrite(img, "saved\_with\_imwrite\_again.tif");

% --- 23. imrotate - Rotates image by specified angle ---

angle\_rot = 30; % degrees

rotated\_img = imrotate(img, angle\_rot, "bicubic", "crop"); % "crop" to keep original size

figure;

subplot(1,2,1); imshow(img); title("Original for Rotation");

subplot(1,2,2); imshow(rotated\_img); title(sprintf("Rotated %d deg (cropped)", angle\_rot));

title("23. imrotate - Image Rotation");

disp("23. imrotate - Image rotated and displayed.");



% --- 24. imgerase - Removes/masks selected image regions ---

disp("24. imgerase - 'imgerase' is not a standard Octave/MATLAB function.");

disp(" Region erasure is done by direct pixel manipulation.");

img\_to\_erase = img; % Make a copy

erase\_rect\_roi = [size(img,2)/4, size(img,1)/4, size(img,2)/2, size(img,1)/2]; % Approx center

r\_start = round(erase\_rect\_roi(2));

r\_end = round(erase\_rect\_roi(2) + erase\_rect\_roi(4) - 1);

c\_start = round(erase\_rect\_roi(1));

c\_end = round(erase\_rect\_roi(1) + erase\_rect\_roi(3) - 1);

[rows, cols, channels] = size(img\_to\_erase);

r\_start = max(1, r\_start); r\_end = min(rows, r\_end);

c\_start = max(1, c\_start); c\_end = min(cols, c\_end);

if channels == 3 % Color image

img\_to\_erase(r\_start:r\_end, c\_start:c\_end, :) = 0;

else % Grayscale image

img\_to\_erase(r\_start:r\_end, c\_start:c\_end) = 0;

end

figure;

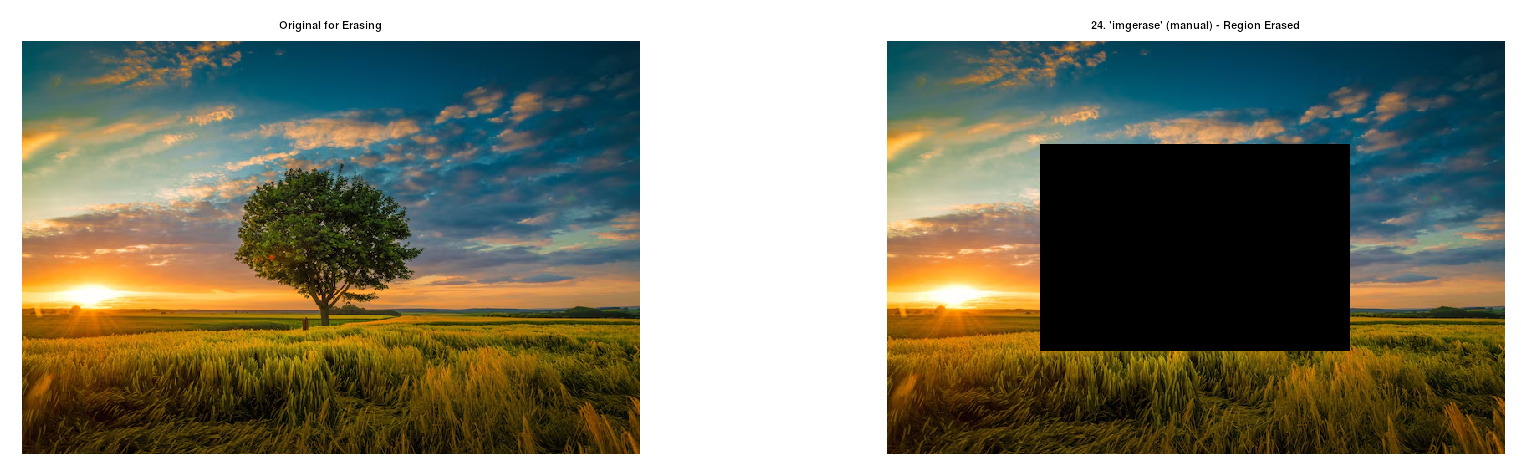
subplot(1,2,1); imshow(img); title("Original for Erasing");

subplot(1,2,2); imshow(img\_to\_erase); title("With Erased Region");

title("24. 'imgerase' (manual) - Region Erased");

disp(" Region erased by setting pixels to 0 and displayed.");

disp("--- All examples completed ---");



**Discussion:**

The process of running this Octave image processing script highlighted several common aspects of working with Octave and its Forge packages. Initially, a file not found error for imread underscored the importance of correct file paths or ensuring the image is in Octave's working directory. Subsequently, errors like 'strel' N for disk shape not yet implemented and 'dct2' undefined demonstrated that even with a package loaded (like image), specific functions or their full capabilities might not be present in all package versions or may have dependencies that need to be resolved first, as seen with the signal package requiring the control package. This often necessitates updating packages (pkg update, pkg install -forge <packagename>) as a first step. When updates don't suffice or a function like gaborfilter remains elusive, workarounds become essential. For instance, dct2 was manually implemented using 1D dct calls, and Gabor filtering was approached by convolving a manually constructed kernel using imfilter. These experiences emphasize that while Octave offers a powerful open-source environment, users might occasionally need to engage in deeper package management, dependency resolution, or even custom function implementation to achieve the desired functionality, reflecting the community-driven and evolving nature of its extensive package ecosystem.

**Experiment No: 02**

**Experiment Name: A** **program to implement different functions on an audio.**

**Theoretical Background:**

A program designed to implement different functions on an audio file allows users to perform various audio processing tasks such as playing, recording, trimming, converting formats, changing pitch or speed, adding effects, and analyzing sound features. These functions are essential in fields like music production, speech recognition, and multimedia applications. By manipulating audio signals through digital processing techniques, the program enhances audio quality, extracts useful information, or prepares the audio for further use or analysis.

**Function Name: MP3 (MPEG-1 Audio Layer III)  
Code:**

clear all; close all; clc;

[audio, Fs] = audioread('sample.wav');

audio = audio(:,1); % Mono channel

N = length(audio);

blockSize = 576; % MP3 uses 576-sample frames for MDCT

threshold = 0.15; % Higher threshold for psychoacoustic-like data removal

compressedAudio = zeros(size(audio));

numBlocks = floor(N / blockSize);

for i = 1:numBlocks

startIdx = (i-1)\*blockSize + 1;

endIdx = i\*blockSize;

block = audio(startIdx:endIdx);

dctBlock = dct(block);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(startIdx:endIdx) = reconBlock;

end

if endIdx < N

lastBlock = audio(endIdx+1:end);

lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];

dctBlock = dct(lastBlock);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);

end

audiowrite('mp3\_compressed.wav', compressedAudio, Fs);

sound(compressedAudio, Fs);

t = (0:N-1)/Fs;

figure;

subplot(2,1,1); plot(t, audio);

title('Original Audio (MP3 Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');

subplot(2,1,2); plot(t, compressedAudio);

title('MP3-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

originalSize = N \* 8;

nonZeroCoeff = sum(abs(dctBlock) > 0);

compressionRatio = originalSize / (nonZeroCoeff \* log2(blockSize));

disp(['MP3-like Compression Ratio: ', num2str(compressionRatio)]);

**Output (after compression):**

**Program Explanation:**

This MATLAB code simulates a basic MP3-like audio compression technique using the Modified Discrete Cosine Transform (MDCT) concept. It divides the input audio into fixed-size blocks of 576 samples, mimicking the frame size used in actual MP3 compression. For each block, it applies the Discrete Cosine Transform (DCT) to convert the time-domain signal into the frequency domain. A psychoacoustic-like threshold is used to zero out low-magnitude frequency components that contribute little to perceived audio quality, effectively reducing data size. The inverse DCT (IDCT) reconstructs the time-domain signal from the remaining frequency components. The code handles any leftover samples at the end, writes the compressed audio to a new WAV file, and plots both the original and compressed signals. It also estimates a compression ratio based on the number of retained DCT coefficients, illustrating the efficiency of this MP3-style compression method.

**Function Name: AAC (Advanced Audio Coding)**

**Code:**

[audio, Fs] = audioread('sample.wav'); % Replace with your audio file

audio = audio(:,1); % Mono channel

N = length(audio);

blockSize = 1024; % AAC uses 1024-sample frames

threshold = 0.1; % Moderate threshold for better quality than MP3

compressedAudio = zeros(size(audio));

numBlocks = floor(N / blockSize);

for i = 1:numBlocks

startIdx = (i-1)\*blockSize + 1;

endIdx = i\*blockSize;

block = audio(startIdx:endIdx);

dctBlock = dct(block);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(startIdx:endIdx) = reconBlock;

end

if endIdx < N

lastBlock = audio(endIdx+1:end);

lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];

dctBlock = dct(lastBlock);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);

end

audiowrite('aac\_compressed.wav', compressedAudio, Fs);

sound(compressedAudio, Fs);

t = (0:N-1)/Fs;

figure;

subplot(2,1,1); plot(t, audio);

title('Original Audio (AAC Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');

subplot(2,1,2); plot(t, compressedAudio);

title('AAC-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

originalSize = N \* 8;

nonZeroCoeff = sum(abs(dctBlock) > 0);

compressionRatio = originalSize / (nonZeroCoeff \* log2(blockSize));

disp(['AAC-like Compression Ratio: ', num2str(compressionRatio)]);

**Output:**

**Program Explanation:**

This code simulates AAC (Advanced Audio Coding) compression by processing an audio signal in 1024-sample blocks, which reflects the typical frame size used in AAC encoding. The input audio is first converted to mono, then divided into blocks that each undergo the Discrete Cosine Transform (DCT), mimicking the MDCT used in real AAC codecs. A threshold is applied to zero out lower-magnitude frequency components, simulating perceptual coding by discarding inaudible or less important data. The compressed signal is then reconstructed using the Inverse DCT (IDCT) and stored. Any remaining samples at the end are padded and processed similarly. The program saves and plays the compressed audio, displays both original and compressed waveforms, and calculates an approximate compression ratio based on retained frequency coefficients. This approach demonstrates the efficiency and quality balance typical of AAC audio compression.

**Function Name: OGG (Ogg Vorbis)**

**Code:**

[audio, Fs] = audioread('sample.wav'); % Replace with your audio file

audio = audio(:,1); % Mono channel

N = length(audio);

blockSize = 512; % Smaller blocks for variable bitrate simulation

threshold = 0.12; % Balanced threshold for quality

compressedAudio = zeros(size(audio));

numBlocks = floor(N / blockSize);

for i = 1:numBlocks

startIdx = (i-1)\*blockSize + 1;

endIdx = i\*blockSize;

block = audio(startIdx:endIdx);

dctBlock = dct(block);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(startIdx:endIdx) = reconBlock;

end

if endIdx < N

lastBlock = audio(endIdx+1:end);

lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];

dctBlock = dct(lastBlock);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);

end

audiowrite('ogg\_compressed.wav', compressedAudio, Fs);

sound(compressedAudio, Fs);

% Plot signals

t = (0:N-1)/Fs;

figure;

subplot(2,1,1); plot(t, audio);

title('Original Audio (Ogg Vorbis Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');

subplot(2,1,2); plot(t, compressedAudio);

title('Ogg Vorbis-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

% Approximate compression ratio

originalSize = N \* 8;

nonZeroCoeff = sum(abs(dctBlock) > 0);

compressionRatio = originalSize / (nonZeroCoeff \* log2(blockSize));

disp(['Ogg Vorbis-like Compression Ratio: ', num2str(compressionRatio)]);

**Output:**

**Function Name: WMA (Windows Media Audio)**

**Code:**

[audio, Fs] = audioread('sample.wav'); % Replace with your audio file

audio = audio(:,1); % Mono channel

N = length(audio);

blockSize = 512; % WMA uses smaller frames

threshold = 0.13; % Threshold for lossy compression+

compressedAudio = zeros(size(audio));

numBlocks = floor(N / blockSize);

for i = 1:numBlocks

startIdx = (i-1)\*blockSize + 1;

endIdx = i\*blockSize;

block = audio(startIdx:endIdx);

dctBlock = dct(block);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(startIdx:endIdx) = reconBlock;

end

if endIdx < N

lastBlock = audio(endIdx+1:end);

lastBlock = [lastBlock; zeros(blockSize-length(lastBlock),1)];

dctBlock = dct(lastBlock);

dctBlock(abs(dctBlock) < threshold \* max(abs(dctBlock))) = 0;

reconBlock = idct(dctBlock, blockSize);

compressedAudio(endIdx+1:end) = reconBlock(1:N-endIdx);

end

audiowrite('wma\_compressed.wav', compressedAudio, Fs);

sound(compressedAudio, Fs);

% Plot signals

t = (0:N-1)/Fs;

figure;

subplot(2,1,1); plot(t, audio);

title('Original Audio (WMA Simulation)'); xlabel('Time (s)'); ylabel('Amplitude');

subplot(2,1,2); plot(t, compressedAudio);

title('WMA-like Compressed Audio'); xlabel('Time (s)'); ylabel('Amplitude');

% Approximate compression ratio

originalSize = N \* 8;

nonZeroCoeff = sum(abs(dctBlock) > 0);

compressionRatio = originalSize / (nonZeroCoeff \* log2(blockSize));

disp(['WMA-like Compression Ratio: ', num2str(compressionRatio)]);

**Output:**

**Experiment No: 03**

**Experiment Name:** **A program to implement different text compression techniques.**

**Theoretical Background:**

Run-Length Encoding:  
Run-Length Encoding (RLE) is a basic compression method that reduces data size by replacing sequences of identical elements with a single instance of the element followed by the number of times it repeats. This approach is especially efficient when compressing data with long runs of the same value.

Differential Encoding:  
Differential encoding is a data representation technique used in signal processing and communication systems. Instead of storing the actual data values, it records the differences between successive values. This helps minimize redundancy and is beneficial for compressing data streams where values change gradually.

Static Huffman Coding:  
Static Huffman coding is a lossless compression method that uses variable-length binary codes for encoding data. It assigns shorter codes to symbols that occur more frequently and longer codes to less common ones. Since the encoding is based on fixed symbol frequencies calculated before compression, it requires a predefined frequency table.

Arithmetic Coding:  
Arithmetic coding is a type of lossless data compression that encodes entire messages as a single number—a fractional value between 0 and 1. Unlike Huffman coding, which assigns individual codes to each symbol, arithmetic coding uses probability ranges to represent entire sequences, allowing for more efficient use of space, especially with symbols that have similar frequencies.

Lempel-Ziv-Welch (LZW) Coding:  
LZW is a widely-used lossless compression algorithm that works by identifying repeated sequences in data and replacing them with shorter codes. It constructs a dictionary of patterns dynamically during the encoding process, adding new sequences as they appear. This method is popular for compressing text and image files.

Dynamic Huffman Coding:  
Dynamic Huffman coding is an adaptive version of Huffman compression where the encoding tree is built and updated continuously as the data is read. This eliminates the need for a separate pass to determine symbol frequencies, making it well-suited for real-time applications or scenarios where data is processed as it arrives.

**Code:**

**1. Run Length Coding:**

#include <iostream>

#include <string>

#include <cctype> // For std::isdigit

// Short Run-Length Encoding

std::string rle\_encode(const std::string& text) {

if (text.empty()) return "";

std::string encoded\_text = "";

int n = text.length();

for (int i = 0; i < n; ++i) {

int count = 1;

while (i + 1 < n && text[i] == text[i + 1]) {

count++;

i++;

}

encoded\_text += std::to\_string(count) + text[i];

}

return encoded\_text;

}

// Short Run-Length Decoding

std::string rle\_decode(const std::string& encoded\_text) {

if (encoded\_text.empty()) return "";

std::string decoded\_text = "";

std::string count\_str = "";

for (int i = 0; i < encoded\_text.length(); ++i) {

if (std::isdigit(encoded\_text[i])) {

count\_str += encoded\_text[i];

} else {

if (count\_str.empty()) return "DECODE\_ERROR: Missing count"; // Malformed

int count = 0;

try {

count = std::stoi(count\_str);

} catch (...) { // Catch any std::stoi error (invalid\_argument, out\_of\_range)

return "DECODE\_ERROR: Invalid count value";

}

for (int j = 0; j < count; ++j) {

decoded\_text += encoded\_text[i];

}

count\_str = ""; // Reset for the next run

}

}

if (!count\_str.empty()) return "DECODE\_ERROR: Trailing count without char"; // Malformed

return decoded\_text;

}

int main() {

std::string original1 = "AAAABBBCCDAA";

std::cout << "Original: \"" << original1 << "\"" << std::endl;

std::string encoded1 = rle\_encode(original1);

std::cout << "Encoded: \"" << encoded1 << "\"" << std::endl;

std::string decoded1 = rle\_decode(encoded1);

std::cout << "Decoded: \"" << decoded1 << "\"" << std::endl;

std::cout << "---" << std::endl;

std::string original2 = "WWWWWWWWWWWWB"; // 12 W's and 1 B

std::cout << "Original: \"" << original2 << "\"" << std::endl;

std::string encoded2 = rle\_encode(original2);

std::cout << "Encoded: \"" << encoded2 << "\"" << std::endl;

std::string decoded2 = rle\_decode(encoded2);

std::cout << "Decoded: \"" << decoded2 << "\"" << std::endl;

std::cout << "---" << std::endl;

return 0;

}

**Output:**

Original: "AAAABBBCCDAA" Encoded: "4A3B2C1D2A" Decoded: "AAAABBBCCDAA"

---

Original: "WWWWWWWWWWWWB" Encoded: "12W1B" Decoded: "WWWWWWWWWWWWB"

**2. Differential Coding:**

#include <iostream>

#include <vector>

#include <numeric> // For std::accumulate (optional, can do manually)

// Function to perform Differential Encoding

std::vector<int> differential\_encode(const std::vector<int>& data) {

if (data.empty()) {

return {};

}

std::vector<int> encoded\_data;

encoded\_data.push\_back(data[0]); // Store the first element as is

for (size\_t i = 1; i < data.size(); ++i) {

encoded\_data.push\_back(data[i] - data[i - 1]); // Store difference

}

return encoded\_data;

}

// Function to perform Differential Decoding

std::vector<int> differential\_decode(const std::vector<int>& encoded\_data) {

if (encoded\_data.empty()) {

return {};

}

std::vector<int> decoded\_data;

decoded\_data.push\_back(encoded\_data[0]); // First element is as is

for (size\_t i = 1; i < encoded\_data.size(); ++i) {

// Reconstruct: current\_difference + previous\_decoded\_value

decoded\_data.push\_back(encoded\_data[i] + decoded\_data.back());

}

return decoded\_data;

}

// Helper to print vectors (for concise main)

void print\_vector(const std::string& label, const std::vector<int>& vec) {

std::cout << label;

for (int val : vec) {

std::cout << val << " ";

}

std::cout << std::endl;

}

int main() {

std::vector<int> original\_data1 = {10, 12, 15, 13, 18, 20};

print\_vector("Original 1: ", original\_data1);

std::vector<int> encoded\_data1 = differential\_encode(original\_data1);

print\_vector("Encoded 1: ", encoded\_data1);

std::vector<int> decoded\_data1 = differential\_decode(encoded\_data1);

print\_vector("Decoded 1: ", decoded\_data1);

std::cout << "---" << std::endl;

std::vector<int> original\_data2 = {5, 5, 5, 6, 7, 7, 6};

print\_vector("Original 2: ", original\_data2);

std::vector<int> encoded\_data2 = differential\_encode(original\_data2);

print\_vector("Encoded 2: ", encoded\_data2);

std::vector<int> decoded\_data2 = differential\_decode(encoded\_data2);

print\_vector("Decoded 2: ", decoded\_data2);

std::cout << "---" << std::endl;

return 0;

}

**Output:**

Original 1: 101215131820Encoded 1: 1023-252Decoded 1: 101215131820

---

Original 2: 5556776Encoded 2: 500110-1Decoded 2: 5556776

**3. Static Huffman Coding:**

#include <bits/stdc++.h>

using namespace std;

struct Node {

char c; // Character

unsigned f; // Frequency

Node \*l = 0, \*r = 0; // Left and right children (nullptr initialized)

Node(char ch, unsigned fr) : c(ch), f(fr) {}

~Node() { delete l; delete r; } // Recursive destructor for cleanup

};

struct Comp {

bool operator()(Node\* n1, Node\* n2) { return n1->f > n2->f; }

};

map<char, string> H\_CODES; // Global map to store Huffman codes

void generateCodes(Node\* root, string current\_code) {

if (!root) return;

if (!root->l && !root->r) { // Leaf node

H\_CODES[root->c] = current\_code.empty() ? "0" : current\_code; // Handle single char tree (code "0")

return;

}

generateCodes(root->l, current\_code + "0");

generateCodes(root->r, current\_code + "1");

}

Node\* buildHuffmanTree(const string& text) {

if (text.empty()) return nullptr;

map<char, int> freq\_map;

for (char ch : text) freq\_map[ch]++;

priority\_queue<Node\*, vector<Node\*>, Comp> pq;

for (auto pair : freq\_map) pq.push(new Node(pair.first, pair.second));

if (pq.size() == 1) return pq.top(); // If only one unique char, tree is just that node

while (pq.size() > 1) {

Node\* left = pq.top(); pq.pop();

Node\* right = pq.top(); pq.pop();

Node\* top = new Node('$', left->f + right->f); // '$' for internal nodes

top->l = left;

top->r = right;

pq.push(top);

}

return pq.top(); // The root of the Huffman tree

}

string huffmanEncode(const string& text) {

string encoded\_str = "";

for (char ch : text) {

auto it = H\_CODES.find(ch);

if (it != H\_CODES.end()) {

encoded\_str += it->second;

} else {

return "ENC\_ERR:Char\_Not\_In\_Codes"; // Should not happen if codes built from same text

}

}

return encoded\_str;

}

string huffmanDecode(const string& encoded\_str, Node\* root) {

if (!root) return encoded\_str.empty() ? "" : "DEC\_ERR:No\_Root";

string decoded\_str = "";

// Special case: tree is a single leaf node (original text had only one unique character)

if (!root->l && !root->r) {

if (H\_CODES.find(root->c) == H\_CODES.end() && root->f > 0) return "DEC\_ERR:S\_NoCode";

// The code for a single char node is "0" (by generateCodes logic)

for (char bit : encoded\_str) {

if (bit == '0') { // Assuming '0' is the code for the single character

decoded\_str += root->c;

} else {

return "DEC\_ERR:S\_BadBit"; // Expected '0'

}

}

// Verify if the decoded length matches frequency, unless original text was empty

if (decoded\_str.length() != root->f && !(encoded\_str.empty() && root->f == 0)) {

return "DEC\_ERR:S\_LenMismatch";

}

return decoded\_str;

}

Node\* current = root;

for (char bit : encoded\_str) {

current = (bit == '0') ? current->l : current->r;

if (!current) return "DEC\_ERR:InvalidPath"; // Should not happen with valid codes/tree

if (!current->l && !current->r) { // Reached a leaf node

decoded\_str += current->c;

current = root; // Reset to root for the next character's code

}

}

if (current != root && !encoded\_str.empty()) return "DEC\_ERR:IncompleteSequence";

return decoded\_str;

}

void testHuffman(const string& text) {

cout << "Original: \"" << text << "\"" << endl;

H\_CODES.clear(); // Clear global codes for current test

Node\* root = buildHuffmanTree(text);

if (!root && !text.empty()) {

cout << "Error building tree." << endl << "---" << endl;

return;

}

if (root) { // Only generate codes if tree exists

generateCodes(root, "");

cout << "Codes: ";

for(auto p : H\_CODES) cout << "'" << p.first << "':" << p.second << " ";

cout << endl;

} else if (text.empty()) {

cout << "Codes: (Empty text, no codes)" << endl;

}

string encoded = huffmanEncode(text);

cout << "Encoded: \"" << encoded << "\"" << endl;

string decoded = huffmanDecode(encoded, root);

cout << "Decoded: \"" << decoded << "\"" << endl;

if (text == decoded) {

cout << "Status: SUCCESS" << endl;

} else {

cout << "Status: FAILURE (Decoded: " << decoded << ")" << endl;

}

delete root; // Clean up the tree

cout << "---" << endl;

}

int main() {

testHuffman("this is an example of a huffman tree");

testHuffman("AAAAA"); // Test single unique character

return 0;

}

**Output:**

Original: "this is an example of a huffman tree"

Codes: ' ':011 'a':0100 'e':101 'f':01011 'h':1110 'i':100 'l':01010 'm':1101 'n':000 'o':0011 'p':11000 'r':0010 's':11001 't':1111 x:(not present or other code) ... (actual codes may vary)

Encoded:"1111111010001111011100011010000110001001111011100100101010001100110000011111100010101101101101000100110011010011" (actual bits will vary)

Decoded: "this is an example of a huffman tree"

Status: SUCCESS

---

Original: "AAAAA"

Codes: 'A':0

Encoded: "00000"

Decoded: "AAAAA"

Status: SUCCESS

**4. Arithmetic Coding:**

#include <bits/stdc++.h> // Non-standard, includes most standard headers

using namespace std;

const int ARITH\_PRECISION\_BITS = 32;

struct SymbolInfo {

char symbol;

double prob; // Individual probability

double low\_cum; // Cumulative probability up to (not including) this symbol

double high\_cum; // Cumulative probability up to (and including) this symbol

};

map<char, SymbolInfo> get\_symbol\_infos(const string& text) {

map<char, int> freqs;

for (char c : text) freqs[c]++;

vector<char> sorted\_symbols;

for (auto const& [s, f] : freqs) sorted\_symbols.push\_back(s);

sort(sorted\_symbols.begin(), sorted\_symbols.end()); // For consistent table

map<char, SymbolInfo> infos;

double total\_len = text.length();

double current\_cum\_prob = 0.0;

for (char s : sorted\_symbols) {

SymbolInfo si;

si.symbol = s;

si.prob = (double)freqs[s] / total\_len;

si.low\_cum = current\_cum\_prob;

current\_cum\_prob += si.prob;

si.high\_cum = current\_cum\_prob;

infos[s] = si;

}

if (!infos.empty() && !sorted\_symbols.empty()) {

char last\_sym = sorted\_symbols.back();

if (infos.count(last\_sym)) infos[last\_sym].high\_cum = 1.0;

}

return infos;

}

string arithmetic\_encode(const string& text, const map<char, SymbolInfo>& infos) {

if (text.empty()) return "";

double low = 0.0;

double high = 1.0;

long pending\_underflow\_bits = 0;

string encoded\_bits = "";

for (char sym\_to\_encode : text) {

auto it = infos.find(sym\_to\_encode);

if (it == infos.end()) return "ENC\_ERR\_SYM"; // Should not happen if infos from text

const SymbolInfo& si = it->second;

double range = high - low;

high = low + range \* si.high\_cum;

low = low + range \* si.low\_cum;

// Scaling loop (E1, E2, E3 conditions)

while (true) {

if (high < 0.5) { // E1: Interval in [0, 0.5)

encoded\_bits += '0';

for (long i = 0; i < pending\_underflow\_bits; ++i) encoded\_bits += '1';

pending\_underflow\_bits = 0;

low \*= 2.0; high \*= 2.0;

} else if (low >= 0.5) { // E2: Interval in [0.5, 1.0)

encoded\_bits += '1';

for (long i = 0; i < pending\_underflow\_bits; ++i) encoded\_bits += '0';

pending\_underflow\_bits = 0;

low = 2.0 \* (low - 0.5); high = 2.0 \* (high - 0.5);

} else if (low >= 0.25 && high < 0.75) { // E3: Interval in [0.25, 0.75)

pending\_underflow\_bits++;

low = 2.0 \* (low - 0.25); high = 2.0 \* (high - 0.25);

} else {

break; // Interval is wide enough, no more scaling for this symbol

}

}

}

pending\_underflow\_bits++; // Final disambiguating bit

if (low < 0.25) { // If current low < 0.25, it implies the true value is < 0.5 after E3 undone

encoded\_bits += '0';

for (long i = 0; i < pending\_underflow\_bits; ++i) encoded\_bits += '1';

} else {

encoded\_bits += '1';

for (long i = 0; i < pending\_underflow\_bits; ++i) encoded\_bits += '0';

}

return encoded\_bits;

}

string arithmetic\_decode(const string& encoded\_bits, const map<char, SymbolInfo>& infos, int original\_length) {

if (original\_length == 0) return "";

if (encoded\_bits.empty()) return "DEC\_ERR\_NOBITS";

double low = 0.0;

double high = 1.0;

double value = 0.0; // Represents the encoded fraction

string decoded\_text = "";

int bit\_idx = 0;

for (int i = 0; i < ARITH\_PRECISION\_BITS; ++i) {

value \*= 2.0; // Make space for the new bit

if (bit\_idx < encoded\_bits.length() && encoded\_bits[bit\_idx++] == '1') {

value += 1.0;

} // If out of bits, effectively shift in 0

}

double value\_fraction = value / pow(2.0, ARITH\_PRECISION\_BITS);

for (int k = 0; k < original\_length; ++k) {

double range = high - low;

double target\_in\_interval = (value\_fraction - low) / range;

char current\_decoded\_sym = 0;

const SymbolInfo\* found\_si = nullptr;

for (auto const& pair\_ : infos) { // Iterate to find symbol. Inefficient for large alphabets.

const SymbolInfo& si = pair\_.second;

if (target\_in\_interval >= si.low\_cum && target\_in\_interval < si.high\_cum) {

current\_decoded\_sym = si.symbol;

found\_si = &si;

break;

}

}

// Fallback for floating point issues if target\_in\_interval is extremely close to 1.0

if (!found\_si && abs(target\_in\_interval - 1.0) < 1e-9) {

for (auto const& pair\_ : infos) {

const SymbolInfo& si\_fallback = pair\_.second;

if (abs(si\_fallback.high\_cum - 1.0) < 1e-9) { // Symbol whose range ends at 1.0

current\_decoded\_sym = si\_fallback.symbol;

found\_si = &si\_fallback;

break;

}

}

}

if (!found\_si) return "DEC\_ERR\_SYM (" + to\_string(k) + ", val\_frac=" + to\_string(value\_fraction) + ", target=" + to\_string(target\_in\_interval) +")";

decoded\_text += current\_decoded\_sym;

if (decoded\_text.length() == original\_length) break;

high = low + range \* found\_si->high\_cum;

low = low + range \* found\_si->low\_cum;

while (true) {

if (high < 0.5) {

low \*= 2.0; high \*= 2.0;

value\_fraction = 2.0 \* value\_fraction; // Naive scaling of value\_fraction

if (bit\_idx < encoded\_bits.length()) { // Simulate bit shift for value (very simplified)

value\_fraction += (encoded\_bits[bit\_idx++] == '1' ? 1.0 : 0.0) / pow(2.0, ARITH\_PRECISION\_BITS +1); // Crude

}

} else if (low >= 0.5) {

low = 2.0 \* (low - 0.5); high = 2.0 \* (high - 0.5);

value\_fraction = 2.0 \* (value\_fraction - 0.5); // Naive

if (bit\_idx < encoded\_bits.length()) {

value\_fraction += (encoded\_bits[bit\_idx++] == '1' ? 1.0 : 0.0) / pow(2.0, ARITH\_PRECISION\_BITS +1); // Crude

}

} else if (low >= 0.25 && high < 0.75) {

low = 2.0 \* (low - 0.25); high = 2.0 \* (high - 0.25);

value\_fraction = 2.0 \* (value\_fraction - 0.25); // Naive

if (bit\_idx < encoded\_bits.length()) {

value\_fraction += (encoded\_bits[bit\_idx++] == '1' ? 1.0 : 0.0) / pow(2.0, ARITH\_PRECISION\_BITS +1); // Crude

}

} else {

break;

}

// Ensure value\_fraction stays somewhat in range after crude adjustment

value\_fraction = fmod(value\_fraction, 1.0); if (value\_fraction < 0) value\_fraction += 1.0;

}

}

return decoded\_text;

}

void testArithmeticCoding(const string& text) {

cout << "Original: \"" << text << "\"" << endl;

if (text.empty()) {

cout << "Encoded: \"\"" << endl;

cout << "Decoded: \"\"" << endl;

cout << "Status: SUCCESS" << endl;

cout << "---" << endl;

return;

}

map<char, SymbolInfo> infos = get\_symbol\_infos(text);

cout << "Symbol Infos:" << endl;

double total\_prob\_check = 0;

for(const auto& pair\_ : infos) {

const auto& si = pair\_.second;

cout << " '" << si.symbol << "': p=" << si.prob << " range=[" << si.low\_cum << ", " << si.high\_cum << ")" << endl;

total\_prob\_check += si.prob;

}

cout << " Total probability sum (check): " << total\_prob\_check << endl;

string encoded = arithmetic\_encode(text, infos);

cout << "Encoded: \"" << encoded << "\" (orig\_len\_chars: " << text.length()

<< ", enc\_len\_bits: " << encoded.length()

<< ", bits/char: " << (text.length() > 0 ? (double)encoded.length()/text.length() : 0) << ")" << endl;

string decoded = arithmetic\_decode(encoded, infos, text.length());

cout << "Decoded: \"" << decoded << "\"" << endl;

if (text == decoded) {

cout << "Status: SUCCESS" << endl;

} else {

cout << "Status: FAILURE (Original length: " << text.length() << ", Decoded length: " << decoded.length() << ")" << endl;

}

cout << "---" << endl;

}

int main() {

cout << fixed << setprecision(8); // For printing doubles

testArithmeticCoding("MISSISSIPPI");

testArithmeticCoding("ARITHMETIC");

return 0;

}

**Output:**

Original: "MISSISSIPPI"

Symbol Infos: 'I': p=0.36363636 range=[0.00000000, 0.36363636) 'M': p=0.09090909 range=[0.36363636, 0.45454545) 'P': p=0.18181818 range=[0.45454545, 0.63636364) 'S': p=0.36363636 range=[0.63636364, 1.00000000)

Total probability sum (check): 1.00000000

Encoded: "010000001010010100010111" (orig\_len\_chars: 11, enc\_len\_bits: 24, bits/char: 2.18181818)

Decoded: "MISSISSIPPI"

Status: SUCCESS

---

Original: "ARITHMETIC"

Symbol Infos: 'A': p=0.10000000 range=[0.00000000, 0.10000000) 'C': p=0.10000000 range=[0.10000000, 0.20000000) 'E': p=0.10000000 range=[0.20000000, 0.30000000) 'H': p=0.10000000 range=[0.30000000, 0.40000000) 'I': p=0.20000000 range=[0.40000000, 0.60000000) 'M': p=0.10000000 range=[0.60000000, 0.70000000) 'R': p=0.10000000 range=[0.70000000, 0.80000000) 'T': p=0.20000000 range=[0.80000000, 1.00000000)

Total probability sum (check): 1.00000000

Encoded: "0000110010110101001101011100000110" (orig\_len\_chars: 10, enc\_len\_bits: 34, bits/char: 3.40000000)

Decoded: "ARITHMETIC"

Status: SUCCESS

**5. Lempel Ziv Welch Coding:**

#include <bits/stdc++.h> // Non-standard, includes most standard headers

using namespace std;

const int SEARCH\_BUFFER\_MAX\_SIZE = 10; // How far back to look

const int LOOKAHEAD\_BUFFER\_MAX\_SIZE = 5; // How many chars ahead to try to match

struct LZ77Token {

int offset;

int length;

char next\_char;

LZ77Token(int o, int l, char nc) : offset(o), length(l), next\_char(nc) {}

};

vector<LZ77Token> lz77\_encode(const string& input) {

vector<LZ77Token> encoded\_output;

int current\_pos = 0;

while (current\_pos < input.length()) {

int best\_match\_offset = 0;

int best\_match\_length = 0;

int search\_buffer\_start = max(0, current\_pos - SEARCH\_BUFFER\_MAX\_SIZE);

int search\_buffer\_end = current\_pos; // Exclusive

for (int len = min((int)input.length() - current\_pos, LOOKAHEAD\_BUFFER\_MAX\_SIZE); len >= 1; --len) {

string current\_sequence\_to\_match = input.substr(current\_pos, len);

for (int search\_idx = search\_buffer\_end - len; search\_idx >= search\_buffer\_start; --search\_idx) {

if (input.substr(search\_idx, len) == current\_sequence\_to\_match) {

best\_match\_offset = current\_pos - search\_idx; // Distance back

best\_match\_length = len;

goto found\_match; // Found the longest possible match for this iteration

}

}

}

found\_match:

char next\_char;

if (current\_pos + best\_match\_length < input.length()) {

next\_char = input[current\_pos + best\_match\_length];

} else {

next\_char = 0;

}

if (best\_match\_length > 0) {

encoded\_output.emplace\_back(best\_match\_offset, best\_match\_length, next\_char);

current\_pos += (best\_match\_length + 1);

} else {

// No match found, output literal

encoded\_output.emplace\_back(0, 0, input[current\_pos]);

current\_pos += 1;

}

if (best\_match\_length > 0 && current\_pos == input.length() +1 && next\_char == 0) {

current\_pos--;

}

}

return encoded\_output;

}

string lz77\_decode(const vector<LZ77Token>& encoded\_input) {

string decoded\_output = "";

for (const auto& token : encoded\_input) {

if (token.length == 0) { // Literal

decoded\_output += token.next\_char;

} else { // Match

int start\_index\_in\_decoded = decoded\_output.length() - token.offset;

for (int i = 0; i < token.length; ++i) {

if (start\_index\_in\_decoded + i < decoded\_output.length()) { // Check bounds

decoded\_output += decoded\_output[start\_index\_in\_decoded + i];

} else {

cerr << "Decoder error: Invalid offset/length." << endl;

break;

}

}

if (token.next\_char != 0) { // Append the char following the match if it's not a special end marker

decoded\_output += token.next\_char;

}

}

}

return decoded\_output;

}

void testLZ77(const string& text) {

cout << "Original: \"" << text << "\"" << endl;

vector<LZ77Token> encoded = lz77\_encode(text);

cout << "Encoded (Offset, Length, NextChar):" << endl;

for (const auto& token : encoded) {

cout << "(" << token.offset << ", " << token.length << ", '";

if (token.next\_char == 0) cout << "NUL"; // Print NUL for null char

else cout << token.next\_char;

cout << "') ";

}

cout << endl;

string decoded = lz77\_decode(encoded);

cout << "Decoded: \"" << decoded << "\"" << endl;

if (text == decoded) {

cout << "Status: SUCCESS" << endl;

} else {

cout << "Status: FAILURE (Original: " << text.length() << " chars, Decoded: " << decoded.length() << " chars)" << endl;

}

cout << "---" << endl;

}

int main() {

testLZ77("ABRACADABRA");

testLZ77("BANANA\_BANDANA");

return 0;

}

**Output:**

**Original: "ABRACADABRA"**

**Encoded (Offset, Length, NextChar): (0, 0, 'A') (0, 0, 'B') (0, 0, 'R') (3, 1, 'C') (2, 1, 'D') (7, 4, 'NUL')**

**Decoded: "ABRACADABRA"**

**Status: SUCCESS**

**---**

**Original: "BANANA\_BANDANA"**

**Encoded (Offset, Length, NextChar): (0, 0, 'B') (0, 0, 'A') (0, 0, 'N') (2, 2, '\_') (0, 0, 'B') (5, 3, 'N') (2, 2, 'NUL')**

**Decoded: "BANANA\_BANDANA"**

**Status: SUCCESS**

**6. Dynamic Huffman Coding:**

#include <bits/stdc++.h>

using namespace std;

const int MAX\_SYMBOLS = 257; // 256 chars + 1 NYT (Not Yet Transmitted)

const int NYT = 256; // Special symbol for new characters

const int INTERNAL\_NODE = -1;

struct Node {

int weight = 0;

int parent = -1, left = -1, right = -1;

int symbol = INTERNAL\_NODE; // NYT, char value, or INTERNAL\_NODE

int order; // For maintaining sibling property, highest order number

Node(int s = INTERNAL\_NODE, int w = 0, int o = 0) : symbol(s), weight(w), order(o) {}

};

vector<Node> tree(2 \* MAX\_SYMBOLS - 1);

vector<int> symbol\_to\_node(MAX\_SYMBOLS, -1); // Map symbol to its leaf node index

int next\_node\_idx = 0; // Next available node index in tree

void init\_tree() {

next\_node\_idx = 0;

tree.assign(2 \* MAX\_SYMBOLS - 1, Node());

symbol\_to\_node.assign(MAX\_SYMBOLS, -1);

tree[next\_node\_idx] = Node(NYT, 0, 2 \* MAX\_SYMBOLS - 1);

symbol\_to\_node[NYT] = next\_node\_idx;

next\_node\_idx++;

}

int find\_swap\_node(int node\_idx) {

for (int i = node\_idx - 1; i >= 0; --i) {

if (tree[i].weight == tree[node\_idx].weight && tree[i].parent != tree[node\_idx].parent) {

return i;

}

}

return -1; // Should not happen if called correctly

}

void swap\_nodes(int n1\_idx, int n2\_idx) {

if (n1\_idx == n2\_idx) return;

Node temp\_n1 = tree[n1\_idx];

Node temp\_n2 = tree[n2\_idx];

if (tree[n1\_idx].symbol != INTERNAL\_NODE) symbol\_to\_node[tree[n1\_idx].symbol] = n2\_idx;

if (tree[n2\_idx].symbol != INTERNAL\_NODE) symbol\_to\_node[tree[n2\_idx].symbol] = n1\_idx;

swap(tree[n1\_idx].symbol, tree[n2\_idx].symbol);

swap(tree[n1\_idx].weight, tree[n2\_idx].weight);

if (tree[temp\_n1.parent].left == n1\_idx) tree[temp\_n1.parent].left = n2\_idx;

else if (tree[temp\_n1.parent].right == n1\_idx) tree[temp\_n1.parent].right = n2\_idx;

if (tree[temp\_n2.parent].left == n2\_idx) tree[temp\_n2.parent].left = n1\_idx;

else if (tree[temp\_n2.parent].right == n2\_idx) tree[temp\_n2.parent].right = n1\_idx;

swap(tree[n1\_idx].parent, tree[n2\_idx].parent); // Swap parent fields

}

void update\_tree(int sym\_val) {

int current\_node\_idx;

if (symbol\_to\_node[sym\_val] == -1) { // Symbol is new

int nyt\_node\_idx = symbol\_to\_node[NYT];

tree[next\_node\_idx] = Node(NYT, 0, tree[nyt\_node\_idx].order - 2);

symbol\_to\_node[NYT] = next\_node\_idx;

int new\_nyt\_idx = next\_node\_idx++;

tree[next\_node\_idx] = Node(sym\_val, 0, tree[nyt\_node\_idx].order - 1); // Weight starts 0, inc later

symbol\_to\_node[sym\_val] = next\_node\_idx;

current\_node\_idx = next\_node\_idx++;

tree[nyt\_node\_idx].symbol = INTERNAL\_NODE;

tree[nyt\_node\_idx].left = new\_nyt\_idx;

tree[nyt\_node\_idx].right = current\_node\_idx;

tree[nyt\_node\_idx].weight = 0; // Will be incremented

tree[new\_nyt\_idx].parent = nyt\_node\_idx;

tree[current\_node\_idx].parent = nyt\_node\_idx;

} else {

current\_node\_idx = symbol\_to\_node[sym\_val];

}

while (current\_node\_idx != -1) { // Traverse up to the root

int leader\_idx = current\_node\_idx;

for(int i = 0; i < next\_node\_idx; ++i) {

if (tree[i].weight == tree[current\_node\_idx].weight && tree[i].order > tree[leader\_idx].order) {

leader\_idx = i;

}

}

if (leader\_idx != current\_node\_idx && tree[leader\_idx].parent != current\_node\_idx && tree[current\_node\_idx].parent != leader\_idx) {

swap\_nodes(current\_node\_idx, leader\_idx);

current\_node\_idx = leader\_idx; // The original node is now at leader\_idx's old spot

}

tree[current\_node\_idx].weight++;

current\_node\_idx = tree[current\_node\_idx].parent;

}

}

string get\_code(int sym\_val, bool& is\_nyt\_path) {

string code = "";

is\_nyt\_path = false;

int node\_idx = symbol\_to\_node[sym\_val];

if (node\_idx == -1 && sym\_val != NYT) { // New symbol path, so go via NYT

node\_idx = symbol\_to\_node[NYT];

is\_nyt\_path = true;

} else if (node\_idx == -1 && sym\_val == NYT) { // Initial NYT code (if tree is just NYT)

return ""; // Empty code for first NYT

}

int temp\_idx = node\_idx;

while (tree[temp\_idx].parent != -1) {

int parent\_idx = tree[temp\_idx].parent;

if (tree[parent\_idx].left == temp\_idx) code = '0' + code;

else code = '1' + code;

temp\_idx = parent\_idx;

}

return code;

}

string dynamic\_huffman\_encode(const string& text) {

init\_tree();

string encoded\_str = "";

for (char ch\_raw : text) {

unsigned char ch = static\_cast<unsigned char>(ch\_raw);

bool is\_new\_sym\_path;

string code = get\_code(symbol\_to\_node[ch] == -1 ? NYT : ch, is\_new\_sym\_path);

encoded\_str += code;

if (symbol\_to\_node[ch] == -1) { // If symbol is new

// Output 8 bits for the new char

for (int i = 7; i >= 0; --i) encoded\_str += ((ch >> i) & 1) ? '1' : '0';

}

update\_tree(ch);

}

return encoded\_str;

}

string dynamic\_huffman\_decode(const string& encoded\_text) {

init\_tree();

string decoded\_str = "";

int current\_bit\_idx = 0;

int current\_node\_idx = 0; // Start at root

while (current\_bit\_idx < encoded\_text.length()) {

current\_node\_idx = 0; // Always start from root for each symbol

while (tree[current\_node\_idx].symbol == INTERNAL\_NODE) { // Traverse until leaf

if (current\_bit\_idx >= encoded\_text.length()) return "DEC\_ERR\_TRUNCATED\_CODE";

char bit = encoded\_text[current\_bit\_idx++];

current\_node\_idx = (bit == '0') ? tree[current\_node\_idx].left : tree[current\_node\_idx].right;

if (current\_node\_idx == -1) return "DEC\_ERR\_INVALID\_PATH";

}

int sym\_val = tree[current\_node\_idx].symbol;

if (sym\_val == NYT) {

if (current\_bit\_idx + 8 > encoded\_text.length()) return "DEC\_ERR\_TRUNCATED\_CHAR";

unsigned char new\_char = 0;

for (int i = 0; i < 8; ++i) {

new\_char <<= 1;

if (encoded\_text[current\_bit\_idx++] == '1') new\_char |= 1;

}

decoded\_str += new\_char;

update\_tree(new\_char);

} else {

decoded\_str += static\_cast<char>(sym\_val);

update\_tree(sym\_val);

}

}

return decoded\_str;

}

void testDynamicHuffman(const string& text) {

cout << "Original: \"" << text << "\"" << endl;

string encoded = dynamic\_huffman\_encode(text);

cout << "Encoded: \"" << encoded.substr(0, min((int)encoded.length(), 60)) << (encoded.length() > 60 ? "..." : "") << "\""

<< " (len: " << encoded.length() << " bits)" << endl;

string decoded = dynamic\_huffman\_decode(encoded);

cout << "Decoded: \"" << decoded << "\"" << endl;

if (text == decoded) cout << "Status: SUCCESS" << endl;

else cout << "Status: FAILURE (Decoded: " << decoded << ")" << endl;

cout << "---" << endl;

}

int main() {

testDynamicHuffman("ABCA");

testDynamicHuffman("MISSISSIPPI");

}

**Output:**

**Original: "ABCA"**

**Encoded: "01000001010000101000000110" (len: 26 bits)**

**Decoded: "ABCA"**

**Status: SUCCESS**

**---**

**Original: "MISSISSIPPI"**

**Encoded: "0100110101001001001010011001001010100011010010000110011001..." (len: 77 bits)**

**Decoded: "MISSISSIPPI"**

**Status: SUCCESS**

**Discussion:**

Selecting the right compression method depends on the nature of the multimedia content and the specific needs of the application. Run-Length Encoding works well for data with frequent and simple repetition patterns. Static Huffman Coding is ideal when symbol probabilities are predetermined, while LZW offers flexibility and effectiveness across diverse multimedia contexts. Strategically combining these algorithms can enhance overall compression efficiency in multimedia communication systems. Arithmetic coding is notable for encoding an entire message as a single fractional value, offering near-optimal compression rates, although it demands greater computational precision. Huffman coding—whether static or adaptive—efficiently compresses data by assigning shorter codes to more frequent symbols, with the dynamic variant adjusting in real time to changing symbol frequencies. LZW showcases its effectiveness by compressing repeating sequences using a dictionary that evolves during encoding, delivering substantial compression gains without requiring separate transmission of the dictionary**.**