**Appendix A: Experimental MATLAB Scripts**

This appendix includes the MATLAB scripts developed and used for implementing, evaluating, and profiling all compression techniques discussed in this thesis. Each compression method — whether existing or unexplored — was tested using three core script modules:

* Image Quality Evaluation Script: Implements the method and computes FWHM, SSIM, CNR, and CR.
* Transmission Performance Evaluation Script: Simulates image transmission over 3G and 4G networks using the simulate\_transfer function.
* Timing and Memory Profiling Script: Uses MATLAB's tic, toc, and whos functions to assess execution time and memory consumption.

**Appendix A.1: Existing Compression Techniques**

**A.1.1 Discrete Cosine Transform (DCT)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

dct\_image = dct2(img);

threshold = 5;

dct\_image(abs(dct\_image) < threshold) = 0;

compressed\_img = idct2(dct\_image);

profile = double(compressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = NaN;

ssim\_val = NaN;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_val = ssim(uint8(compressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_val];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(dct\_image ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_val, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

Transmission Performance Evaluation

Imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

bandwidth\_3g = 384 \* 1024;

bandwidth\_4g = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3g = zeros(1, num\_images);

transmission\_times\_4g = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

dct\_image = dct2(img);

threshold = 5;

dct\_image(abs(dct\_image) < threshold) = 0;

compressed\_img = idct2(dct\_image);

compressed\_size\_bits = numel(find(dct\_image ~= 0)) \* 8;

transmission\_times\_3g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3g, chunk\_size);

transmission\_times\_4g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4g, chunk\_size);

fprintf('image %d: transfer time (3g): %.2f sec, transfer time (4g): %.2f sec\n', ...

Idx, transmission\_times\_3g(idx), transmission\_times\_4g(idx));

end

fprintf('average transfer time (3g): %.2f sec\n', mean(transmission\_times\_3g));

fprintf('average transfer time (4g): %.2f sec\n', mean(transmission\_times\_4g));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

try

tic;

run\_DCT\_compression(img);

exec\_times(idx) = toc;

catch ME

warning(['Error on image ', num2str(idx), ': ', ME.message]);

exec\_times(idx) = NaN;

end

end

avg\_time = nanmean(exec\_times);

mem\_info = memory;

mem\_usage = mem\_info.MemUsedMATLAB / 1e6;

fprintf('\nDCT Compression Benchmarking Results:\n');

fprintf('Average Execution Time (s): %.4f\n', avg\_time);

fprintf('Memory Usage (MB): %.2f\n', mem\_usage);

function run\_DCT\_compression(img)

I = double(img);

D = dct2(I);

threshold = 5;

D(abs(D) < threshold) = 0;

C = idct2(D);

end

**A.1.2 Discrete Wavelet Transform (DWT)**

Image Quality Evaluation

imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

wavelettype = 'db1';

decompositionlevel = 2;

pixel\_size = 0.3528;

For idx = 1:length(imagefiles)

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[c, s] = wavedec2(double(img), decompositionlevel, wavelettype);

approx\_length = prod(s(1, :));

compressed\_c = c;

compressed\_c(approx\_length + 1:end) = 0;

compressed\_img = waverec2(compressed\_c, s, wavelettype);

profile = double(compressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = nan;

ssim\_value = nan;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

Std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(compressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(compressed\_c ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('image %d: fwhm: %.2f mm, cnr: %.2f, ssim: %.2f, cr: %.2f\n', ...

Idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('average fwhm: %.2f mm\n', mean(fwhm\_values));

fprintf('average cnr: %.2f\n', mean(cnr\_values));

fprintf('average ssim: %.2f\n', mean(ssim\_values));

fprintf('average cr: %.2f\n', mean(cr\_values));

Transmission Performance Evaluation

Imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

bandwidth\_3g = 384 \* 1024;

bandwidth\_4g = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3g = zeros(1, num\_images);

transmission\_times\_4g = zeros(1, num\_images);

wavelettype = 'db1';

decompositionlevel = 2;

for idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[c, s] = wavedec2(double(img), decompositionlevel, wavelettype);

approx\_length = prod(s(1, :));

compressed\_c = c;

compressed\_c(approx\_length + 1:end) = 0;

compressed\_img\_cropped = waverec2(compressed\_c, s, wavelettype);

original\_size\_bits = numel(img) \* 8;

compressed\_size\_bits = numel(find(compressed\_c ~= 0)) \* 8;

transmission\_times\_3g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3g, chunk\_size);

transmission\_times\_4g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4g, chunk\_size);

fprintf('image %d: transfer time (3g): %.2f sec, transfer time (4g): %.2f sec\n', ...

Idx, transmission\_times\_3g(idx), transmission\_times\_4g(idx));

end

fprintf('average transfer time (3g): %.2f sec\n', mean(transmission\_times\_3g));

fprintf('average transfer time (4g): %.2f sec\n', mean(transmission\_times\_4g));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

Clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images';

imagefiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imagefiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

wavelettype = 'db1';

decompositionlevel = 2;

for idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

I = double(img);

[c, s] = wavedec2(i, decompositionlevel, wavelettype);

approx\_length = prod(s(1, :));

compressed\_c = c;

compressed\_c(approx\_length + 1:end) = 0;

irec = waverec2(compressed\_c, s, wavelettype);

exec\_times(idx) = toc;

catch

exec\_times(idx) = nan;

end

mem\_info = memory;

mem\_usages(idx) = mem\_info.memusedmatlab / 1e6;

end

fprintf('dwt compression timing and memory benchmark:\n');

fprintf('avg time per image: %.4f seconds\n', nanmean(exec\_times));

fprintf('avg memory used: %.2f mb\n', nanmean(mem\_usages));

**A.1.3 Hybrid DCT-DWT**

Image Quality Evaluation

imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

wavelettype = 'db1';

decompositionlevel = 2;

For idx = 1:length(imagefiles)

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[c, s] = wavedec2(double(img), 1, wavelettype);

approx\_ll1 = appcoef2(c, s, wavelettype, 1);

[c2, s2] = wavedec2(double(approx\_ll1), 1, wavelettype);

approx\_ll2 = appcoef2(c2, s2, wavelettype, 1);

dct\_ll2 = dct2(approx\_ll2);

threshold = 20;

dct\_ll2(abs(dct\_ll2) < threshold) = 0;

compressed\_ll2 = idct2(dct\_ll2);

compressed\_c2 = c2;

compressed\_c2(1:numel(approx\_ll2)) = compressed\_ll2(:);

compressed\_img\_ll1 = waverec2(compressed\_c2, s2, wavelettype);

compressed\_c = c;

compressed\_c(1:numel(approx\_ll1)) = compressed\_img\_ll1(:);

compressed\_img = waverec2(compressed\_c, s, wavelettype);

profile = double(compressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* 0.3528;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = nan;

ssim\_value = nan;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

Std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(compressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(dct\_ll2 ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('image %d: fwhm: %.2f mm, cnr: %.2f, ssim: %.2f, cr: %.2f\n', ...

Idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('average fwhm: %.2f mm\n', mean(fwhm\_values));

fprintf('average cnr: %.2f\n', mean(cnr\_values));

fprintf('average ssim: %.2f\n', mean(ssim\_values));

fprintf('average cr: %.2f\n', mean(cr\_values));

Transmission Performance Evaluation

imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

bandwidth\_3g = 384 \* 1024;

bandwidth\_4g = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3g = zeros(1, num\_images);

transmission\_times\_4g = zeros(1, num\_images);

wavelettype = 'db1';

decompositionlevel = 2;

For idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[c, s] = wavedec2(double(img), 1, wavelettype);

approx\_ll1 = appcoef2(c, s, wavelettype, 1);

[c2, s2] = wavedec2(double(approx\_ll1), 1, wavelettype);

approx\_ll2 = appcoef2(c2, s2, wavelettype, 1);

dct\_ll2 = dct2(approx\_ll2);

threshold = 20;

dct\_ll2(abs(dct\_ll2) < threshold) = 0;

compressed\_ll2 = idct2(dct\_ll2);

compressed\_c2 = c2;

compressed\_c2(1:numel(approx\_ll2)) = compressed\_ll2(:);

compressed\_img\_ll1 = waverec2(compressed\_c2, s2, wavelettype);

compressed\_c = c;

compressed\_c(1:numel(approx\_ll1)) = compressed\_img\_ll1(:);

compressed\_img = waverec2(compressed\_c, s, wavelettype);

original\_size\_bits = numel(img) \* 8;

compressed\_size\_bits = numel(find(dct\_ll2 ~= 0)) \* 8;

transmission\_times\_3g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3g, chunk\_size);

transmission\_times\_4g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4g, chunk\_size);

fprintf('image %d: transfer time (3g): %.2f sec, transfer time (4g): %.2f sec\n', ...

Idx, transmission\_times\_3g(idx), transmission\_times\_4g(idx));

end

fprintf('average transfer time (3g): %.2f sec\n', mean(transmission\_times\_3g));

fprintf('average transfer time (4g): %.2f sec\n', mean(transmission\_times\_4g));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images';

imagefiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imagefiles));

exec\_times = zeros(num\_images, 1);

For idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

run\_hybrid\_compression(img);

exec\_times(idx) = toc;

catch me

warning(['error in hybrid compression on image ', num2str(idx), ': ', me.message]);

exec\_times(idx) = nan;

end

end

Avg\_time = nanmean(exec\_times);

mem\_info = memory;

avg\_memory = mem\_info.memusedmatlab / 1e6;

Fprintf('\nhybrid compression timing and memory benchmark:\n');

fprintf('avg time per image: %.4f seconds\n', avg\_time);

fprintf('avg memory used: %.2f mb\n', avg\_memory);

Function run\_hybrid\_compression(img)

img = imresize(img, [256, 256]);

img = double(img);

[c1, s1] = wavedec2(img, 1, 'db1');

ll1 = appcoef2(c1, s1, 'db1', 1);

[c2, s2] = wavedec2(ll1, 1, 'db1');

ll2 = appcoef2(c2, s2, 'db1', 1);

d = dct2(ll2);

d(abs(d) < 20) = 0;

ll2\_c = idct2(d);

c2(1:numel(ll2)) = ll2\_c(:);

ll1\_c = waverec2(c2, s2, 'db1');

c1(1:numel(ll1)) = ll1\_c(:);

waverec2(c1, s1, 'db1');

end

**A.1.4 Binary-cLuster (BL) Code**

Image Quality Evaluation

imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

blocksize = 64;

s = 1;

pixel\_size = 0.3528;

For idx = 1:length(imagefiles)

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

padrows = blocksize \* ceil(rows / blocksize) - rows;

padcols = blocksize \* ceil(cols / blocksize) - cols;

imgpadded = padarray(img, [padrows, padcols], 'replicate', 'post');

compressed\_img = zeros(size(imgpadded));

for I = 1:blocksize:size(imgpadded, 1)

for j = 1:blocksize:size(imgpadded, 2)

block = imgpadded(i:i+blocksize-1, j:j+blocksize-1);

encoded\_block = arrayfun(@(val) bl\_encode(val, s), double(block));

compressed\_block = arrayfun(@(val) bl\_decode(val, s), encoded\_block);

compressed\_img(i:i+blocksize-1, j:j+blocksize-1) = compressed\_block;

end

end

compressed\_img\_cropped = compressed\_img(1:rows, 1:cols);

profile = double(compressed\_img\_cropped(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img\_cropped));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(compressed\_img\_cropped), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

Std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(compressed\_img\_cropped), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(compressed\_img ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('image %d: fwhm: %.2f mm, cnr: %.2f, ssim: %.2f, cr: %.2f\n', ...

Idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('average fwhm: %.2f mm\n', mean(fwhm\_values));

fprintf('average cnr: %.2f\n', mean(cnr\_values));

fprintf('average ssim: %.2f\n', mean(ssim\_values));

fprintf('average cr: %.2f\n', mean(cr\_values));

function encoded = bl\_encode(z, s)

m = ceil(log2((z + 2 \* s) / (2 \* s)));

k = floor((1 + sqrt(1 + 8 \* m)) / 2);

x = m - (k \* (k - 1) / 2);

prefix\_value = (2^(k - (x - 1)) - 1) \* 10 + (2^(x - 1) - 1);

suffix\_value = z - 2 \* s \* (2^(m - 1) - 1) - 1;

encoded = prefix\_value \* 1000 + suffix\_value;

end

Function z = bl\_decode(encoded, s)

prefix\_value = floor(encoded / 1000);

suffix\_value = mod(encoded, 1000);

k = floor(log2(prefix\_value)) + 1;

t = k - 1; % continuous '1's in the binary representation of the prefix

m = k \* (k - 1) / 2 + t + 1;

z = suffix\_value + 2 \* s \* (2^(m - 1) - 1) + 1;

end

Transmission Performance Evaluation

imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

bandwidth\_3g = 384 \* 1024;

bandwidth\_4g = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3g = zeros(1, num\_images);

transmission\_times\_4g = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

blocksize = 64;

s = 1;

For idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

padrows = blocksize \* ceil(rows / blocksize) - rows;

padcols = blocksize \* ceil(cols / blocksize) - cols;

imgpadded = padarray(img, [padrows, padcols], 'replicate', 'post');

original\_size\_bits = numel(imgpadded) \* 8;

compressed\_img = zeros(size(imgpadded));

for I = 1:blocksize:size(imgpadded, 1)

for j = 1:blocksize:size(imgpadded, 2)

block = imgpadded(i:i+blocksize-1, j:j+blocksize-1);

encoded\_block = arrayfun(@(val) bl\_encode(val, s), double(block));

compressed\_block = arrayfun(@(val) bl\_decode(val, s), encoded\_block);

compressed\_img(i:i+blocksize-1, j:j+blocksize-1) = compressed\_block;

end

end

compressed\_img\_cropped = compressed\_img(1:rows, 1:cols);

compressed\_size\_bits = numel(find(compressed\_img\_cropped ~= 0)) \* 8;

cr\_values(idx) = original\_size\_bits / compressed\_size\_bits;

transmission\_times\_3g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3g, chunk\_size);

transmission\_times\_4g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4g, chunk\_size);

fprintf('image %d: transfer time (3g): %.2f sec, transfer time (4g): %.2f sec\n', ...

Idx, transmission\_times\_3g(idx), transmission\_times\_4g(idx));

end

fprintf('average transfer time (3g): %.2f sec\n', mean(transmission\_times\_3g));

fprintf('average transfer time (4g): %.2f sec\n', mean(transmission\_times\_4g));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function encoded = bl\_encode(z, s)

m = ceil(log2((z + 2 \* s) / (2 \* s)));

k = floor((1 + sqrt(1 + 8 \* m)) / 2);

x = m - (k \* (k - 1) / 2);

prefix\_value = (2^(k - (x - 1)) - 1) \* 10 + (2^(x - 1) - 1);

suffix\_value = z - 2 \* s \* (2^(m - 1) - 1) - 1;

encoded = prefix\_value \* 1000 + suffix\_value;

end

function z = bl\_decode(encoded, s)

prefix\_value = floor(encoded / 1000);

suffix\_value = mod(encoded, 1000);

k = floor(log2(prefix\_value)) + 1;

t = k - 1; % continuous '1's in the binary representation of the prefix

m = k \* (k - 1) / 2 + t + 1;

z = suffix\_value + 2 \* s \* (2^(m - 1) - 1) + 1;

end

Timing and Memory Profiling

Clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images';

imagefiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imagefiles));

times = zeros(num\_images, 1);

mem\_usage = zeros(num\_images, 1);

For idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

try

mem\_start = memory;

tic;

run\_blcode\_compression(img);

times(idx) = toc;

mem\_stop = memory;

mem\_usage(idx) = mem\_stop.memusedmatlab - mem\_start.memusedmatlab;

catch

times(idx) = nan;

mem\_usage(idx) = nan;

end

end

Fprintf('blcode compression timing and memory benchmark:\n');

fprintf('avg time per image: %.4f seconds\n', nanmean(times));

fprintf('avg memory used: %.2f mb\n', nanmean(mem\_usage) / 1e6);

Function run\_blcode\_compression(img)

blocksize = 64; s = 1;

[rows, cols] = size(img);

padrows = blocksize \* ceil(rows / blocksize) - rows;

padcols = blocksize \* ceil(cols / blocksize) - cols;

imgpadded = padarray(img, [padrows, padcols], 'replicate', 'post');

compressed\_img = zeros(size(imgpadded));

for I = 1:blocksize:size(imgpadded,1)

for j = 1:blocksize:size(imgpadded,2)

block = double(imgpadded(i:i+blocksize-1, j:j+blocksize-1));

encoded\_block = arrayfun(@(val) bl\_encode(val, s), block);

decoded\_block = arrayfun(@(val) bl\_decode(val, s), encoded\_block);

compressed\_img(i:i+blocksize-1, j:j+blocksize-1) = decoded\_block;

end

end

compressed\_img(1:rows, 1:cols);

end

Function encoded = bl\_encode(z, s)

m = ceil(log2((z + 2 \* s) / (2 \* s)));

k = floor((1 + sqrt(1 + 8 \* m)) / 2);

x = m - (k \* (k - 1) / 2);

prefix\_value = (2^(k - (x - 1)) - 1) \* 10 + (2^(x - 1) - 1);

suffix\_value = z - 2 \* s \* (2^(m - 1) - 1) - 1;

encoded = prefix\_value \* 1000 + suffix\_value;

end

Function z = bl\_decode(encoded, s)

prefix\_value = floor(encoded / 1000);

suffix\_value = mod(encoded, 1000);

k = floor(log2(prefix\_value)) + 1;

t = k - 1;

m = k \* (k - 1) / 2 + t + 1;

z = suffix\_value + 2 \* s \* (2^(m - 1) - 1) + 1;

end

**A.1.5 Walsh Matrix Transformation**

Image Quality Evaluation

Imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

m = 1024;

blocksize = 32;

for idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

next\_pow2\_rows = 2^nextpow2(rows);

next\_pow2\_cols = 2^nextpow2(cols);

padded\_img = padarray(img, [next\_pow2\_rows - rows, next\_pow2\_cols - cols], 'post');

compressed\_img = fwt(padded\_img);

quantized\_data = compressed\_img(2:end);

compressed\_data = golomb\_rice\_compress(quantized\_data, m);

decompressed\_data = golomb\_rice\_decompress(compressed\_data, m);

decompressed\_img = ifwt(compressed\_img);

[decompressed\_rows, decompressed\_cols] = size(decompressed\_img);

if decompressed\_rows ~= rows || decompressed\_cols ~= cols

decompressed\_img = imresize(decompressed\_img, [rows, cols]);

end

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(compressed\_img ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('image %d: fwhm: %.2f mm, cnr: %.2f, ssim: %.2f, cr: %.2f\n', ...

Idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('average fwhm: %.2f mm\n', mean(fwhm\_values));

fprintf('average cnr: %.2f\n', mean(cnr\_values));

fprintf('average ssim: %.2f\n', mean(ssim\_values));

fprintf('average cr: %.2f\n', mean(cr\_values));

function transformed\_data = fwt(data)

[rows, cols] = size(data);

if mod(rows, 2) ~= 0 || mod(cols, 2) ~= 0

error('both rows and columns of the image must be powers of 2.');

End

for I = 1:rows

data(i, :) = walsh1d(data(i, :));

end

for j = 1:cols

data(:, j) = walsh1d(data(:, j));

end

transformed\_data = data;

end

function output = walsh1d(input)

n = length(input);

if n == 1

output = input;

return;

end

even\_part = walsh1d(input(1:2:end));

odd\_part = walsh1d(input(2:2:end));

output = [even\_part + odd\_part, even\_part - odd\_part];

end

function inverse\_transformed\_data = ifwt(data)

[rows, cols] = size(data);

if mod(rows, 2) ~= 0 || mod(cols, 2) ~= 0

error('both rows and columns of the image must be powers of 2.');

End

for I = 1:rows

data(i, :) = inv\_walsh1d(data(i, :));

end

for j = 1:cols

data(:, j) = inv\_walsh1d(data(:, j));

end

inverse\_transformed\_data = data;

end

function output = inv\_walsh1d(input)

n = length(input);

if n == 1

output = input;

return;

end

even\_part = inv\_walsh1d(input(1:2:end));

odd\_part = inv\_walsh1d(input(2:2:end));

output = [even\_part + odd\_part, even\_part - odd\_part];

end

function compressed\_data = golomb\_rice\_compress(data, m)

compressed\_data = [];

for I = 1:length(data)

symbol = data(i);

quotient = floor(symbol / m);

remainder = mod(symbol, m);

quotient\_code = dec2bin(quotient, log2(m));

remainder\_code = dec2bin(remainder, log2(m));

compressed\_data = [compressed\_data, quotient\_code, remainder\_code];

end

end

function decoded\_data = golomb\_rice\_decompress(encoded\_data, m)

decoded\_data = [];

data\_length = length(encoded\_data);

I = 1;

while I <= data\_length

quotient = bin2dec(encoded\_data(i:i + log2(m) - 1));

remainder = bin2dec(encoded\_data(I + log2(m):i + 2\*log2(m) - 1));

symbol = quotient \* m + remainder;

decoded\_data = [decoded\_data, symbol];

I = I + 2 \* log2(m);

end

end

Transmission Performance Evaluation

Imagefiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.png')); ...

Dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\my documents\matlab\10 images\images', '\*.jpg'))];

num\_images = min(10, length(imagefiles));

bandwidth\_3g = 384 \* 1024;

bandwidth\_4g = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3g = zeros(1, num\_images);

transmission\_times\_4g = zeros(1, num\_images);

m = 1024;

for idx = 1:num\_images

img = imread(fullfile(imagefiles(idx).folder, imagefiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

next\_pow2\_rows = 2^nextpow2(rows);

next\_pow2\_cols = 2^nextpow2(cols);

padded\_img = padarray(img, [next\_pow2\_rows - rows, next\_pow2\_cols - cols], 'post');

compressed\_img = fwt(padded\_img);

quantized\_data = compressed\_img(2:end);

compressed\_data = golomb\_rice\_compress(quantized\_data, m);

compressed\_size\_bits = numel(find(compressed\_img ~= 0)) \* 8;

transmission\_times\_3g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3g, chunk\_size);

transmission\_times\_4g(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4g, chunk\_size);

decompressed\_data = golomb\_rice\_decompress(compressed\_data, m);

decompressed\_img = ifwt(compressed\_img);

[decompressed\_rows, decompressed\_cols] = size(decompressed\_img);

if decompressed\_rows ~= rows || decompressed\_cols ~= cols

decompressed\_img = imresize(decompressed\_img, [rows, cols]);

end

fprintf('image %d: transfer time (3g) = %.2f s, transfer time (4g) = %.2f s\n', ...

Idx, transmission\_times\_3g(idx), transmission\_times\_4g(idx));

end

fprintf('average transfer time (3g): %.2f s\n', mean(transmission\_times\_3g));

fprintf('average transfer time (4g): %.2f s\n', mean(transmission\_times\_4g));

function transfer\_time = simulate\_transfer(data\_size\_bits, bandwidth, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth;

transfer\_time = num\_chunks \* time\_per\_chunk;

end

function transformed\_data = fwt(data)

[rows, cols] = size(data);

For I = 1:rows

data(i, :) = walsh1d(data(i, :));

end

For j = 1:cols

data(:, j) = walsh1d(data(:, j));

end

Transformed\_data = data;

end

function output = walsh1d(input)

n = length(input);

If n == 1

output = input;

return;

end

Even\_part = walsh1d(input(1:2:end));

odd\_part = walsh1d(input(2:2:end));

Output = [even\_part + odd\_part, even\_part - odd\_part];

end

function inverse\_transformed\_data = ifwt(data)

[rows, cols] = size(data);

For I = 1:rows

data(i, :) = inv\_walsh1d(data(i, :));

end

For j = 1:cols

data(:, j) = inv\_walsh1d(data(:, j));

end

Inverse\_transformed\_data = data;

end

function output = inv\_walsh1d(input)

n = length(input);

If n == 1

output = input;

return;

end

Even\_part = inv\_walsh1d(input(1:2:end));

odd\_part = inv\_walsh1d(input(2:2:end));

Output = [even\_part + odd\_part, even\_part - odd\_part];

end

function compressed\_data = golomb\_rice\_compress(data, m)

compressed\_data = [];

for I = 1:length(data)

symbol = data(i);

quotient = floor(symbol / m);

remainder = mod(symbol, m);

Quotient\_code = dec2bin(quotient, log2(m));

remainder\_code = dec2bin(remainder, log2(m));

Compressed\_data = [compressed\_data, quotient\_code, remainder\_code];

end

end

function decoded\_data = golomb\_rice\_decompress(encoded\_data, m)

decoded\_data = [];

data\_length = length(encoded\_data);

I = 1;

while I <= data\_length

quotient = bin2dec(encoded\_data(i:i + log2(m) - 1));

remainder = bin2dec(encoded\_data(I + log2(m):i + 2\*log2(m) - 1));

Symbol = quotient \* m + remainder;

decoded\_data = [decoded\_data, symbol];

I = I + 2 \* log2(m);

end

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

walshmatrix\_compression(img);

times(idx) = toc;

catch

times(idx) = NaN;

end

end

mem\_info = memory;

fprintf('\nWalshMatrix Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(times));

fprintf('Avg Memory Used: %.2f MB\n', mem\_info.MemUsedMATLAB / 1e6);

function walshmatrix\_compression(img)

[rows, cols] = size(img);

next\_pow2\_rows = 2^nextpow2(rows);

next\_pow2\_cols = 2^nextpow2(cols);

padded\_img = padarray(img, [next\_pow2\_rows - rows, next\_pow2\_cols - cols], 'post');

M = 1024;

compressed\_img = fwt(padded\_img);

quantized\_data = compressed\_img(2:end);

compressed\_data = golomb\_rice\_compress(quantized\_data, M);

decompressed\_data = golomb\_rice\_decompress(compressed\_data, M);

decompressed\_img = ifwt(compressed\_img);

if size(decompressed\_img, 1) ~= rows || size(decompressed\_img, 2) ~= cols

decompressed\_img = imresize(decompressed\_img, [rows, cols]);

end

end

function transformed\_data = fwt(data)

[rows, cols] = size(data);

for i = 1:rows

data(i, :) = walsh1d(data(i, :));

end

for j = 1:cols

data(:, j) = walsh1d(data(:, j));

end

transformed\_data = data;

end

function output = walsh1d(input)

N = length(input);

if N == 1

output = input;

return;

end

even\_part = walsh1d(input(1:2:end));

odd\_part = walsh1d(input(2:2:end));

output = [even\_part + odd\_part, even\_part - odd\_part];

end

function inverse\_transformed\_data = ifwt(data)

[rows, cols] = size(data);

for i = 1:rows

data(i, :) = inv\_walsh1d(data(i, :));

end

for j = 1:cols

data(:, j) = inv\_walsh1d(data(:, j));

end

inverse\_transformed\_data = data;

end

function output = inv\_walsh1d(input)

N = length(input);

if N == 1

output = input;

return;

end

even\_part = inv\_walsh1d(input(1:2:end));

odd\_part = inv\_walsh1d(input(2:2:end));

output = [even\_part + odd\_part, even\_part - odd\_part];

end

function compressed\_data = golomb\_rice\_compress(data, M)

compressed\_data = '';

for i = 1:length(data)

symbol = data(i);

quotient = floor(symbol / M);

remainder = mod(symbol, M);

qcode = dec2bin(quotient, log2(M));

rcode = dec2bin(remainder, log2(M));

compressed\_data = [compressed\_data, qcode, rcode];

end

end

function decoded\_data = golomb\_rice\_decompress(encoded\_data, M)

decoded\_data = [];

i = 1;

L = length(encoded\_data);

while i <= L

q = bin2dec(encoded\_data(i : i + log2(M) - 1));

r = bin2dec(encoded\_data(i + log2(M) : i + 2\*log2(M) - 1));

decoded\_data = [decoded\_data, q \* M + r];

i = i + 2 \* log2(M);

end

end

**A.1.6 Adjusted Quick Shift Phase Preserving Dynamic Range Compression (AQS-APPDRC)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

kernel\_size = 10;

color\_ratio = 0.8;

max\_distance = 5;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

preprocessed\_img = preprocess\_image(img);

superpixels = adjusted\_quick\_shift(preprocessed\_img, kernel\_size, color\_ratio, max\_distance);

segmented\_region = binary\_thresholding(superpixels);

profile = double(preprocessed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

dynamic\_threshold = mean(preprocessed\_img(:)) + 2 \* std(preprocessed\_img(:));

signal\_region = preprocessed\_img(preprocessed\_img > dynamic\_threshold);

background\_region = preprocessed\_img(preprocessed\_img <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

end

cnr\_values = [cnr\_values, cnr];

ssim\_value = ssim(uint8(preprocessed\_img), uint8(img));

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(superpixels > 0)) \* 8;

cr = original\_size / compressed\_size;

cr\_values = [cr\_values, cr];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', idx, fwhm\_mm, cnr, ssim\_value, cr);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function preprocessed\_img = preprocess\_image(img)

cutoff\_freq = 220;

high\_pass = 1/10000;

appdrc\_img = apply\_appdrc(img, cutoff\_freq, high\_pass);

frosted\_img = frost\_filter(appdrc\_img);

preprocessed\_img = frosted\_img;

end

function appdrc\_img = apply\_appdrc(img, cutoff\_freq, high\_pass)

[amplitude, phase] = extract\_phase\_amplitude(img);

compressed\_amp = log(amplitude + 1);

appdrc\_img = compressed\_amp .\* sin(phase);

end

function frosted\_img = frost\_filter(img)

frosted\_img = adapthisteq(imfilter(img, fspecial('gaussian', 5, 2)));

end

function superpixels = adjusted\_quick\_shift(img, kernel\_size, color\_ratio, max\_distance)

img = double(img);

[rows, cols] = size(img);

density = zeros(rows, cols);

labels = zeros(rows, cols);

spatial\_kernel = fspecial('gaussian', [2\*kernel\_size+1, 2\*kernel\_size+1], kernel\_size / 3);

for r = 1:rows

for c = 1:cols

row\_start = max(1, r-kernel\_size);

row\_end = min(rows, r+kernel\_size);

col\_start = max(1, c-kernel\_size);

col\_end = min(cols, c+kernel\_size);

local\_region = img(row\_start:row\_end, col\_start:col\_end);

local\_spatial\_kernel = spatial\_kernel(1:(row\_end-row\_start+1), 1:(col\_end-col\_start+1));

local\_intensity\_weights = exp(-abs(local\_region - img(r, c)) / max\_distance);

density(r, c) = sum(local\_spatial\_kernel(:) .\* local\_intensity\_weights(:));

end

end

for r = 1:rows

for c = 1:cols

labels(r, c) = sub2ind([rows, cols], r, c);

row\_start = max(1, r-kernel\_size);

row\_end = min(rows, r+kernel\_size);

col\_start = max(1, c-kernel\_size);

col\_end = min(cols, c+kernel\_size);

local\_density = density(row\_start:row\_end, col\_start:col\_end);

local\_indices = find(local\_density > density(r, c));

if ~isempty(local\_indices)

[~, max\_idx] = max(local\_density(local\_indices));

[max\_r, max\_c] = ind2sub(size(local\_density), local\_indices(max\_idx));

labels(r, c) = sub2ind([rows, cols], row\_start+max\_r-1, col\_start+max\_c-1);

end

end

end

superpixels = labels;

end

function segmented\_region = binary\_thresholding(superpixels)

labeled\_img = bwlabel(superpixels > 0);

region\_props = regionprops(labeled\_img, 'Area', 'PixelIdxList');

[~, largest\_idx] = max([region\_props.Area]);

segmented\_region = ismember(labeled\_img, largest\_idx);

end

function [amplitude, phase] = extract\_phase\_amplitude(img)

fft\_img = fft2(double(img));

amplitude = abs(fft\_img);

phase = angle(fft\_img);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

preprocessed\_img = preprocess\_image(img);

superpixels = adjusted\_quick\_shift(preprocessed\_img, 10, 0.8, 5);

segmented\_region = binary\_thresholding(superpixels);

unique\_values = unique(segmented\_region);

compressed\_size\_bits = numel(find(segmented\_region > 0)) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function preprocessed\_img = preprocess\_image(img)

cutoff\_freq = 220;

high\_pass = 1/10000;

appdrc\_img = apply\_appdrc(img, cutoff\_freq, high\_pass);

frosted\_img = frost\_filter(appdrc\_img);

preprocessed\_img = frosted\_img;

end

function appdrc\_img = apply\_appdrc(img, cutoff\_freq, high\_pass)

[amplitude, phase] = extract\_phase\_amplitude(img);

compressed\_amp = log(amplitude + 1);

appdrc\_img = compressed\_amp .\* sin(phase);

end

function frosted\_img = frost\_filter(img)

frosted\_img = adapthisteq(imfilter(img, fspecial('gaussian', 5, 2)));

end

function superpixels = adjusted\_quick\_shift(img, kernel\_size, color\_ratio, max\_distance)

img = double(img);

[rows, cols] = size(img);

density = zeros(rows, cols);

labels = zeros(rows, cols);

spatial\_kernel = fspecial('gaussian', [2\*kernel\_size+1, 2\*kernel\_size+1], kernel\_size / 3);

for r = 1:rows

for c = 1:cols

row\_start = max(1, r-kernel\_size);

row\_end = min(rows, r+kernel\_size);

col\_start = max(1, c-kernel\_size);

col\_end = min(cols, c+kernel\_size);

local\_region = img(row\_start:row\_end, col\_start:col\_end);

local\_spatial\_kernel = spatial\_kernel(1:(row\_end-row\_start+1), 1:(col\_end-col\_start+1));

local\_intensity\_weights = exp(-abs(local\_region - img(r, c)) / max\_distance);

density(r, c) = sum(local\_spatial\_kernel(:) .\* local\_intensity\_weights(:));

end

end

for r = 1:rows

for c = 1:cols

labels(r, c) = sub2ind([rows, cols], r, c);

row\_start = max(1, r-kernel\_size);

row\_end = min(rows, r+kernel\_size);

col\_start = max(1, c-kernel\_size);

col\_end = min(cols, c+kernel\_size);

local\_density = density(row\_start:row\_end, col\_start:col\_end);

local\_indices = find(local\_density > density(r, c));

if ~isempty(local\_indices)

[~, max\_idx] = max(local\_density(local\_indices));

[max\_r, max\_c] = ind2sub(size(local\_density), local\_indices(max\_idx));

labels(r, c) = sub2ind([rows, cols], row\_start+max\_r-1, col\_start+max\_c-1);

end

end

end

superpixels = labels;

end

function segmented\_region = binary\_thresholding(superpixels)

threshold = graythresh(superpixels);

segmented\_region = imbinarize(superpixels, threshold);

if all(segmented\_region(:) == 1)

segmented\_region = superpixels > mean(superpixels(:));

end

end

function [amplitude, phase] = extract\_phase\_amplitude(img)

fft\_img = fft2(double(img));

amplitude = abs(fft\_img);

phase = angle(fft\_img);

end

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

try

tic;

AQS\_APPDRC\_compress(img);

exec\_times(idx) = toc;

catch ME

warning(['Error on image ', num2str(idx), ': ', ME.message]);

exec\_times(idx) = NaN;

end

end

avg\_time = nanmean(exec\_times);

mem\_info = memory;

mem\_used\_mb = mem\_info.MemUsedMATLAB / 1e6;

fprintf('\nAQS-APPDRC Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', avg\_time);

fprintf('Avg Memory Used: %.2f MB\n', mem\_used\_mb);

function output = AQS\_APPDRC\_compress(I)

I = double(I);

kernel\_size = 3;

color\_ratio = 0.5;

max\_distance = 10;

I\_filtered = imgaussfilt(I, 2);

superpixels = adjusted\_quick\_shift(I\_filtered, kernel\_size, color\_ratio, max\_distance);

segmented = binary\_thresholding(superpixels);

output = imgaussfilt(double(segmented), 1);

end

function superpixels = adjusted\_quick\_shift(I, kernel\_size, color\_ratio, max\_distance)

[h, w] = size(I);

superpixels = zeros(h, w);

step = kernel\_size;

for i = 1:step:h

for j = 1:step:w

row\_end = min(i+step-1, h);

col\_end = min(j+step-1, w);

region = I(i:row\_end, j:col\_end);

mean\_val = mean(region(:));

superpixels(i:row\_end, j:col\_end) = mean\_val;

end

end

end

function binary = binary\_thresholding(I)

T = graythresh(I);

binary = imbinarize(I, T);

end

**A.1.7 Contextual Vector Quantization (CVQ)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

codebook\_size\_croi = 16;

codebook\_size\_bg = 8;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = double(img);

img\_preprocessed = preprocess(img);

[CROI, BG] = segment\_region(img\_preprocessed);

CROI\_encoded = CVQ\_encode(CROI, codebook\_size\_croi);

BG\_encoded = CVQ\_encode(BG, codebook\_size\_bg);

CROI\_decoded = CVQ\_decode(CROI\_encoded, size(CROI));

BG\_decoded = CVQ\_decode(BG\_encoded, size(BG));

compressed\_img = CROI\_decoded + BG\_decoded;

profile = double(compressed\_img(round(end / 2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

end

cnr\_values = [cnr\_values, cnr];

ssim\_value = ssim(uint8(compressed\_img), uint8(img));

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(CROI\_encoded.indices) + numel(BG\_encoded.indices);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function img\_out = preprocess(img\_in)

img\_out = wiener2(img\_in, [5 5]);

se = strel('disk', 2);

img\_out = imopen(img\_out, se);

end

function [CROI, BG] = segment\_region(img)

[M, N] = size(img);

CROI = zeros(M, N);

BG = zeros(M, N);

seed = [round(M/2), round(N/2)];

CROI(seed(1), seed(2)) = img(seed(1), seed(2));

threshold = 10;

for x = 2:M-1

for y = 2:N-1

neighbors = CROI(x-1:x+1, y-1:y+1);

mean\_intensity = mean(neighbors(neighbors > 0));

if abs(img(x, y) - mean\_intensity) <= threshold

CROI(x, y) = img(x, y);

else

BG(x, y) = img(x, y);

end

end

end

end

function encoded = CVQ\_encode(region, codebook\_size)

[M, N] = size(region);

if mod(M, 2) ~= 0

region = padarray(region, [1, 0], 'replicate', 'post');

end

if mod(N, 2) ~= 0

region = padarray(region, [0, 1], 'replicate', 'post');

end

blocks = mat2cell(region, repmat(2, floor(size(region, 1) / 2), 1), ...

repmat(2, floor(size(region, 2) / 2), 1));

codebook = zeros(codebook\_size, 4);

indices = zeros(size(blocks));

for i = 1:codebook\_size

idx = randi(numel(blocks));

codebook(i, :) = blocks{idx}(:)';

end

for i = 1:numel(blocks)

block = blocks{i}(:)';

[~, idx] = min(vecnorm(codebook - block, 2, 2));

indices(i) = idx;

end

encoded = struct('codebook', codebook, 'indices', indices);

end

function decoded = CVQ\_decode(encoded, region\_size)

codebook = encoded.codebook;

indices = encoded.indices;

[rows, cols] = size(indices);

decoded = zeros(region\_size);

for i = 1:rows

for j = 1:cols

idx = indices(i, j);

start\_row = (i - 1) \* 2 + 1;

start\_col = (j - 1) \* 2 + 1;

decoded(start\_row:start\_row+1, start\_col:start\_col+1) = reshape(codebook(idx, :), [2, 2]);

end

end

decoded = decoded(1:region\_size(1), 1:region\_size(2));

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

codebook\_size\_croi = 16;

codebook\_size\_bg = 8;

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = double(img);

img\_preprocessed = preprocess(img);

[CROI, BG] = segment\_region(img\_preprocessed);

CROI\_encoded = CVQ\_encode(CROI, codebook\_size\_croi);

BG\_encoded = CVQ\_encode(BG, codebook\_size\_bg);

CROI\_decoded = CVQ\_decode(CROI\_encoded, size(CROI));

BG\_decoded = CVQ\_decode(BG\_encoded, size(BG));

compressed\_img = CROI\_decoded + BG\_decoded;

compressed\_size\_bits = numel(CROI\_encoded.indices) + numel(BG\_encoded.indices);

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f s, Transfer Time (4G): %.2f s\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f s\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f s\n', mean(transmission\_times\_4G));

function img\_out = preprocess(img\_in)

img\_out = wiener2(img\_in, [5 5]);

se = strel('disk', 2);

img\_out = imopen(img\_out, se);

end

function [CROI, BG] = segment\_region(img)

[M, N] = size(img);

CROI = zeros(M, N);

BG = zeros(M, N);

seed = [round(M/2), round(N/2)];

CROI(seed(1), seed(2)) = img(seed(1), seed(2));

threshold = 10;

for x = 2:M-1

for y = 2:N-1

neighbors = CROI(x-1:x+1, y-1:y+1);

mean\_intensity = mean(neighbors(neighbors > 0));

if abs(img(x, y) - mean\_intensity) <= threshold

CROI(x, y) = img(x, y);

else

BG(x, y) = img(x, y);

end

end

end

end

function encoded = CVQ\_encode(region, codebook\_size)

[M, N] = size(region);

if mod(M, 2) ~= 0

region = padarray(region, [1, 0], 'replicate', 'post');

end

if mod(N, 2) ~= 0

region = padarray(region, [0, 1], 'replicate', 'post');

end

blocks = mat2cell(region, repmat(2, floor(size(region, 1) / 2), 1), ...

repmat(2, floor(size(region, 2) / 2), 1));

codebook = zeros(codebook\_size, 4);

indices = zeros(size(blocks));

for i = 1:codebook\_size

idx = randi(numel(blocks));

codebook(i, :) = blocks{idx}(:)';

end

for i = 1:numel(blocks)

block = blocks{i}(:)';

[~, idx] = min(vecnorm(codebook - block, 2, 2));

indices(i) = idx;

end

encoded = struct('codebook', codebook, 'indices', indices);

end

function decoded = CVQ\_decode(encoded, region\_size)

codebook = encoded.codebook;

indices = encoded.indices;

[rows, cols] = size(indices);

decoded = zeros(region\_size);

for i = 1:rows

for j = 1:cols

idx = indices(i, j);

start\_row = (i - 1) \* 2 + 1;

start\_col = (j - 1) \* 2 + 1;

decoded(start\_row:start\_row+1, start\_col:start\_col+1) = reshape(codebook(idx, :), [2, 2]);

end

end

decoded = decoded(1:region\_size(1), 1:region\_size(2));

end

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

codebook\_size\_croi = 16;

codebook\_size\_bg = 8;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

img = double(img);

try

mem\_start = memory;

tic;

img\_preprocessed = preprocess(img);

[CROI, BG] = segment\_region(img\_preprocessed);

CROI\_encoded = CVQ\_encode(CROI, codebook\_size\_croi);

BG\_encoded = CVQ\_encode(BG, codebook\_size\_bg);

CROI\_decoded = CVQ\_decode(CROI\_encoded, size(CROI));

BG\_decoded = CVQ\_decode(BG\_encoded, size(BG));

compressed\_img = CROI\_decoded + BG\_decoded;

exec\_times(idx) = toc;

mem\_end = memory;

mem\_usages(idx) = (mem\_end.MemUsedMATLAB - mem\_start.MemUsedMATLAB) / 1e6;

catch

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('CVQ Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function img\_out = preprocess(img\_in)

img\_out = wiener2(img\_in, [5 5]);

se = strel('disk', 2);

img\_out = imopen(img\_out, se);

end

function [CROI, BG] = segment\_region(img)

[M, N] = size(img);

CROI = zeros(M, N);

BG = zeros(M, N);

seed = [round(M/2), round(N/2)];

CROI(seed(1), seed(2)) = img(seed(1), seed(2));

threshold = 10;

for x = 2:M-1

for y = 2:N-1

neighbors = CROI(x-1:x+1, y-1:y+1);

mean\_intensity = mean(neighbors(neighbors > 0));

if abs(img(x, y) - mean\_intensity) <= threshold

CROI(x, y) = img(x, y);

else

BG(x, y) = img(x, y);

end

end

end

end

function encoded = CVQ\_encode(region, codebook\_size)

[M, N] = size(region);

if mod(M, 2) ~= 0, region = padarray(region, [1, 0], 'replicate', 'post'); end

if mod(N, 2) ~= 0, region = padarray(region, [0, 1], 'replicate', 'post'); end

blocks = mat2cell(region, repmat(2, floor(size(region, 1)/2), 1), repmat(2, floor(size(region, 2)/2), 1));

codebook = zeros(codebook\_size, 4);

indices = zeros(size(blocks));

for i = 1:codebook\_size

idx = randi(numel(blocks));

codebook(i, :) = blocks{idx}(:)';

end

for i = 1:numel(blocks)

block = blocks{i}(:)';

[~, idx] = min(vecnorm(codebook - block, 2, 2));

indices(i) = idx;

end

encoded = struct('codebook', codebook, 'indices', indices);

end

function decoded = CVQ\_decode(encoded, region\_size)

codebook = encoded.codebook;

indices = encoded.indices;

[rows, cols] = size(indices);

decoded = zeros(region\_size);

for i = 1:rows

for j = 1:cols

idx = indices(i, j);

start\_row = (i - 1) \* 2 + 1;

start\_col = (j - 1) \* 2 + 1;

decoded(start\_row:start\_row+1, start\_col:start\_col+1) = reshape(codebook(idx, :), [2, 2]);

end

end

decoded = decoded(1:region\_size(1), 1:region\_size(2));

end

**A.1.8 Pixel-Position-Based**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

fprintf('Original Image %d Size: %dx%d\n', idx, size(img, 1), size(img, 2));

[compressedData, uniformRegions] = pixelPositionBasedCompressionWithDecompression(img);

decompressed\_img = decompress(compressedData, size(img, 1), size(img, 2));

fprintf('Decompressed Image %d Size: %dx%d\n', idx, size(decompressed\_img, 1), size(decompressed\_img, 2));

[rows, cols] = size(img);

[dec\_rows, dec\_cols] = size(decompressed\_img);

if rows ~= dec\_rows || cols ~= dec\_cols

fprintf('Size mismatch detected: Original (%dx%d), Decompressed (%dx%d)\n', ...

rows, cols, dec\_rows, dec\_cols);

decompressed\_img = imresize(decompressed\_img, [rows, cols], 'nearest');

end

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

profile = double(decompressed\_img(round(end / 2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

if ~isequal(size(decompressed\_img), size(img))

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

fprintf('Final Decompressed Image %d Size (after resize): %dx%d\n', idx, size(decompressed\_img, 1), size(decompressed\_img, 2));

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

bits\_per\_error = 8;

bits\_per\_region = 64;

compressed\_size = numel(predictionErrors) \* bits\_per\_error + ...

numel(compressedData.uniformRegions) \* bits\_per\_region;

original\_size = numel(img) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [compressedData, uniformRegions] = pixelPositionBasedCompressionWithDecompression(inputImage)

img = double(inputImage);

[rows, cols] = size(img);

predictionErrors = zeros(rows, cols);

isType1 = mod((1:rows)' + (1:cols), 2);

uniformRegions = [];

[uniformRegions, uniformMask] = detectUniformRegions(img);

img(uniformMask) = NaN;

for i = 1:rows

for j = 1:cols

if isnan(img(i, j))

continue;

end

if i == 1 || j == 1 || i == rows || j == cols

predictionErrors(i, j) = handleBorders(img, i, j, rows, cols);

else

if isType1(i, j) == 1

predictedValue = floor((img(i-1, j-1) + img(i-1, j+1)) / 2);

else

predictedValue = floor((img(i-1, j) + img(i, j-1) + img(i, j+1) + img(i+1, j)) / 4);

end

predictionErrors(i, j) = img(i, j) - predictedValue;

end

end

end

compressedData = arithmeticCoding(predictionErrors, uniformRegions);

end

function errorValue = handleBorders(img, i, j, rows, cols)

if i == 1

if j == 1

errorValue = img(i, j);

else

errorValue = img(i, j) - img(i, j-1);

end

elseif j == 1

errorValue = img(i, j) - img(i-1, j);

elseif i == rows

errorValue = img(i, j) - floor((img(i-1, j) + img(i, j-1)) / 2);

elseif j == cols

errorValue = img(i, j) - floor((img(i-1, j) + img(i, j-1)) / 2);

else

errorValue = img(i, j);

end

end

function [uniformRegions, uniformMask] = detectUniformRegions(img)

uniformMask = false(size(img));

uniformRegions = [];

regionCount = 0;

for i = 1:size(img, 1)

uniformValue = img(i, 1);

if all(img(i, :) == uniformValue)

uniformMask(i, :) = true;

regionCount = regionCount + 1;

uniformRegions(regionCount, :) = [uniformValue, i, 1, size(img, 2)];

end

end

for j = 1:size(img, 2)

uniformValue = img(1, j);

if all(img(:, j) == uniformValue)

uniformMask(:, j) = true;

regionCount = regionCount + 1;

uniformRegions(regionCount, :) = [uniformValue, 1, j, size(img, 1)];

end

end

end

function compressedData = arithmeticCoding(predictionErrors, uniformRegions)

predictionErrors = predictionErrors(:);

minVal = floor(min(predictionErrors(:)));

maxVal = ceil(max(predictionErrors(:)));

range = minVal:maxVal;

freq = histcounts(predictionErrors, [range, maxVal + 1]);

if sum(freq) == 0

error('Frequency array is empty; check input data.');

end

cumFreq = [0; cumsum(freq(:))];

if length(cumFreq) < 2

error('Invalid cumulative frequency array. Check input data.');

end

low = 0;

high = 1;

for k = 1:length(predictionErrors)

symbol = predictionErrors(k) - minVal + 1;

symbol = max(1, min(symbol, length(cumFreq)-1));

rangeWidth = high - low;

high = low + rangeWidth \* cumFreq(symbol + 1) / cumFreq(end);

low = low + rangeWidth \* cumFreq(symbol) / cumFreq(end);

end

compressedData.encodedValue = (low + high) / 2;

compressedData.uniformRegions = uniformRegions;

end

function decompressedImage = decompress(compressedData, rows, cols)

decompressedImage = zeros(rows, cols);

uniformRegions = compressedData.uniformRegions;

for k = 1:size(uniformRegions, 1)

value = uniformRegions(k, 1);

startRow = uniformRegions(k, 2);

startCol = uniformRegions(k, 3);

length = uniformRegions(k, 4);

decompressedImage(startRow:(startRow+length-1), startCol:(startCol+length-1)) = value;

end

decompressedImage = imresize(decompressedImage, [rows, cols], 'nearest');

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[compressedData, ~, predictionErrors] = pixelPositionBasedCompressionWithDecompression(img);

decompressed\_img = decompress(compressedData, size(img, 1), size(img, 2));

bits\_per\_error = 8;

bits\_per\_region = 64;

compressed\_size\_bits = numel(predictionErrors) \* bits\_per\_error + ...

numel(compressedData.uniformRegions) \* bits\_per\_region;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [compressedData, uniformRegions, predictionErrors] = pixelPositionBasedCompressionWithDecompression(inputImage)

img = double(inputImage);

[rows, cols] = size(img);

predictionErrors = zeros(rows, cols);

isType1 = mod((1:rows)' + (1:cols), 2);

uniformRegions = [];

[uniformRegions, uniformMask] = detectUniformRegions(img);

img(uniformMask) = NaN;

for i = 2:rows-1

for j = 2:cols-1

if isnan(img(i, j))

continue;

end

if isType1(i, j) == 1

predictedValue = floor((4\*img(i-1, j) + 4\*img(i, j-1) + img(i+1, j) + img(i, j+1)) / 10);

else

predictedValue = floor((img(i-1, j) + img(i, j-1) + img(i+1, j) + img(i, j+1) + ...

2\*img(i-1, j-1) + 2\*img(i-1, j+1) + 2\*img(i+1, j-1) + 2\*img(i+1, j+1)) / 12);

end

predictionErrors(i, j) = img(i, j) - predictedValue;

end

end

predictionErrors(isnan(predictionErrors)) = 0;

predictionErrors = predictionErrors(predictionErrors ~= 0);

compressedData = arithmeticCoding(predictionErrors, uniformRegions);

compressedData.predictionErrors = predictionErrors;

end

function errorValue = handleBorders(img, i, j, rows, cols)

if i == 1 && j == 1

errorValue = img(i, j);

elseif i == 1

if j > 1

errorValue = img(i, j) - img(i, j-1);

else

errorValue = img(i, j);

end

elseif j == 1

if i > 1

errorValue = img(i, j) - img(i-1, j);

else

errorValue = img(i, j);

end

elseif i == rows || j == cols

errorValue = img(i, j) - floor((img(i-1, j) + img(i, j-1)) / 2);

else

errorValue = img(i, j);

end

end

function [uniformRegions, uniformMask] = detectUniformRegions(img)

uniformMask = false(size(img));

uniformRegions = [];

regionCount = 0;

threshold = 10;

block\_size = 8;

for i = 1:block\_size:size(img, 1) - block\_size + 1

for j = 1:block\_size:size(img, 2) - block\_size + 1

block = img(i:i+block\_size-1, j:j+block\_size-1);

if range(block(:)) < threshold

uniformMask(i:i+block\_size-1, j:j+block\_size-1) = true;

regionCount = regionCount + 1;

uniformRegions(regionCount, :) = [mean(block(:)), i, j, block\_size];

end

end

end

end

function compressedData = arithmeticCoding(predictionErrors, uniformRegions)

predictionErrors = predictionErrors(:);

nonZeroErrors = predictionErrors(predictionErrors ~= 0);

if isempty(nonZeroErrors)

compressedData.encodedValue = 0;

compressedData.uniformRegions = uniformRegions;

compressedData.predictionErrors = [];

return;

end

minVal = floor(min(nonZeroErrors));

maxVal = ceil(max(nonZeroErrors));

range = minVal:maxVal;

freq = histcounts(nonZeroErrors, [range, maxVal + 1]);

if all(freq == 0)

freq = ones(size(freq));

end

cumFreq = [0; cumsum(freq(:))];

if length(cumFreq) <= 1

error('Invalid cumulative frequency array. Check input data.');

end

low = 0;

high = 1;

for k = 1:length(nonZeroErrors)

symbol = nonZeroErrors(k) - minVal + 1;

symbol = max(1, min(symbol, length(cumFreq)-1));

rangeWidth = high - low;

high = low + rangeWidth \* cumFreq(symbol + 1) / cumFreq(end);

low = low + rangeWidth \* cumFreq(symbol) / cumFreq(end);

end

compressedData.encodedValue = (low + high) / 2;

compressedData.uniformRegions = uniformRegions;

compressedData.predictionErrors = nonZeroErrors;

end

function decompressedImage = decompress(compressedData, rows, cols)

decompressedImage = zeros(rows, cols);

uniformRegions = compressedData.uniformRegions;

for k = 1:size(uniformRegions, 1)

value = uniformRegions(k, 1);

startRow = uniformRegions(k, 2);

startCol = uniformRegions(k, 3);

length = uniformRegions(k, 4);

decompressedImage(startRow:(startRow+length-1), startCol:(startCol+length-1)) = value;

end

decompressedImage = imresize(decompressedImage, [rows, cols], 'nearest');

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

[compressedData, uniformRegions] = pixelPositionBasedCompressionWithDecompression(img);

decompressed\_img = decompress(compressedData, size(img, 1), size(img, 2));

times(idx) = toc;

catch ME

warning('Error on image %d: %s', idx, ME.message);

times(idx) = NaN;

end

end

avg\_time = nanmean(times);

mem\_info = memory;

mem\_usage = mem\_info.MemUsedMATLAB / 1e6;

fprintf('Pixel-Position-Based Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', avg\_time);

fprintf('Avg Memory Used: %.2f MB\n', mem\_usage);

function [compressedData, uniformRegions] = pixelPositionBasedCompressionWithDecompression(inputImage)

img = double(inputImage);

[rows, cols] = size(img);

predictionErrors = zeros(rows, cols);

isType1 = mod((1:rows)' + (1:cols), 2);

uniformRegions = [];

[uniformRegions, uniformMask] = detectUniformRegions(img);

img(uniformMask) = NaN;

for i = 1:rows

for j = 1:cols

if isnan(img(i, j)), continue; end

if i == 1 || j == 1 || i == rows || j == cols

predictionErrors(i, j) = handleBorders(img, i, j, rows, cols);

else

if isType1(i, j) == 1

predictedValue = floor((img(i-1, j-1) + img(i-1, j+1)) / 2);

else

predictedValue = floor((img(i-1, j) + img(i, j-1) + img(i, j+1) + img(i+1, j)) / 4);

end

predictionErrors(i, j) = img(i, j) - predictedValue;

end

end

end

compressedData = arithmeticCoding(predictionErrors, uniformRegions);

end

function errorValue = handleBorders(img, i, j, rows, cols)

if i == 1

if j == 1

errorValue = img(i, j);

else

errorValue = img(i, j) - img(i, j-1);

end

elseif j == 1

errorValue = img(i, j) - img(i-1, j);

elseif i == rows || j == cols

errorValue = img(i, j) - floor((img(i-1, j) + img(i, j-1)) / 2);

else

errorValue = img(i, j);

end

end

function [uniformRegions, uniformMask] = detectUniformRegions(img)

uniformMask = false(size(img));

uniformRegions = [];

regionCount = 0;

for i = 1:size(img, 1)

if all(img(i, :) == img(i, 1))

uniformMask(i, :) = true;

regionCount = regionCount + 1;

uniformRegions(regionCount, :) = [img(i, 1), i, 1, size(img, 2)];

end

end

for j = 1:size(img, 2)

if all(img(:, j) == img(1, j))

uniformMask(:, j) = true;

regionCount = regionCount + 1;

uniformRegions(regionCount, :) = [img(1, j), 1, j, size(img, 1)];

end

end

end

function compressedData = arithmeticCoding(predictionErrors, uniformRegions)

predictionErrors = predictionErrors(:);

minVal = floor(min(predictionErrors(:)));

maxVal = ceil(max(predictionErrors(:)));

range = minVal:maxVal;

freq = histcounts(predictionErrors, [range, maxVal + 1]);

if sum(freq) == 0, error('Frequency array is empty'); end

cumFreq = [0; cumsum(freq(:))];

low = 0; high = 1;

for k = 1:length(predictionErrors)

symbol = predictionErrors(k) - minVal + 1;

symbol = max(1, min(symbol, length(cumFreq)-1));

rangeWidth = high - low;

high = low + rangeWidth \* cumFreq(symbol + 1) / cumFreq(end);

low = low + rangeWidth \* cumFreq(symbol) / cumFreq(end);

end

compressedData.encodedValue = (low + high) / 2;

compressedData.uniformRegions = uniformRegions;

end

function decompressedImage = decompress(compressedData, rows, cols)

decompressedImage = zeros(rows, cols);

uniformRegions = compressedData.uniformRegions;

for k = 1:size(uniformRegions, 1)

value = uniformRegions(k, 1);

startRow = uniformRegions(k, 2);

startCol = uniformRegions(k, 3);

length = uniformRegions(k, 4);

decompressedImage(startRow:(startRow+length-1), startCol:(startCol+length-1)) = value;

end

decompressedImage = imresize(decompressedImage, [rows, cols], 'nearest');

end

**A.1.9 Medical Image Compression with Thresholding (MICT)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = imresize(img, [256, 256]);

[LL, LH, HL, HH] = dwt2(double(img), 'db9');

[rows\_LL, cols\_LL] = size(LL);

[rows\_LH, cols\_LH] = size(LH);

[rows\_HL, cols\_HL] = size(HL);

[rows\_HH, cols\_HH] = size(HH);

SC = zeros(rows\_LL + rows\_HL, cols\_LL + cols\_LH);

max\_iterations = 50;

iteration = 0;

while true

iteration = iteration + 1;

LL\_threshold = find\_initial\_threshold(LL);

LH\_threshold = find\_initial\_threshold(LH);

HL\_threshold = find\_initial\_threshold(HL);

HH\_threshold = find\_initial\_threshold(HH);

SC1 = zeros(size(SC));

SC1(1:rows\_LL, 1:cols\_LL) = LL .\* (LL >= LL\_threshold);

SC1(1:rows\_LH, cols\_LL+1:cols\_LL+cols\_LH) = LH .\* (LH >= LH\_threshold);

SC1(rows\_LL+1:rows\_LL+rows\_HL, 1:cols\_HL) = HL .\* (HL >= HL\_threshold);

SC1(rows\_LL+1:rows\_LL+rows\_HH, cols\_LL+1:cols\_LL+cols\_HH) = HH .\* (HH >= HH\_threshold);

SC = SC + SC1;

decompressed\_img = idwt2(SC(1:rows\_LL, 1:cols\_LL), ...

SC(1:rows\_LH, cols\_LL+1:cols\_LL+cols\_LH), ...

SC(rows\_LL+1:rows\_LL+rows\_HL, 1:cols\_HL), ...

SC(rows\_LL+1:rows\_LL+rows\_HH, cols\_LL+1:cols\_LL+cols\_HH), 'db9');

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

psnr\_value = calculate\_psnr(double(img), decompressed\_img);

if psnr\_value >= 36 || iteration >= max\_iterations

break;

end

LL\_threshold = update\_threshold(LL, LL\_threshold);

LH\_threshold = update\_threshold(LH, LH\_threshold);

HL\_threshold = update\_threshold(HL, HL\_threshold);

HH\_threshold = update\_threshold(HH, HH\_threshold);

end

[encodedData, counts] = arithmetic\_encode(SC);

decompressed\_img = mict\_decompression(encodedData, counts, size(SC));

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), uint8(img));

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), uint8(img));

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(encodedData) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function threshold = find\_initial\_threshold(subband)

[counts, values] = hist(subband(:), unique(subband));

[~, idx] = max(counts);

threshold = values(idx);

end

function updated\_threshold = update\_threshold(subband, current\_threshold)

remaining\_values = subband(subband < current\_threshold);

if isempty(remaining\_values)

updated\_threshold = current\_threshold;

else

[counts, values] = hist(remaining\_values(:), unique(remaining\_values));

[~, idx] = max(counts);

updated\_threshold = values(idx);

end

end

function psnr\_value = calculate\_psnr(original, reconstructed)

if size(original) ~= size(reconstructed)

reconstructed = imresize(reconstructed, size(original), 'nearest');

end

mse = mean((original(:) - reconstructed(:)).^2);

psnr\_value = 10 \* log10(255^2 / mse);

end

function [encoded, counts] = arithmetic\_encode(data)

data = double(data(:));

symbols = unique(data);

data\_mapped = arrayfun(@(x) find(symbols == x, 1), data);

counts = histc(data\_mapped, 1:length(symbols));

encoded = arithenco(data\_mapped, counts);

end

function decoded = arithmetic\_decode(encodedData, counts, imgSize)

decoded = arithdeco(encodedData, counts, prod(imgSize));

decoded = reshape(decoded, imgSize);

end

function decompressed\_img = mict\_decompression(encodedData, counts, imgSize)

decoded = arithmetic\_decode(encodedData, counts, imgSize);

halfRows = floor(size(decoded, 1) / 2);

halfCols = floor(size(decoded, 2) / 2);

cA = decoded(1:halfRows, 1:halfCols);

cH = decoded(1:halfRows, halfCols+1:end);

cV = decoded(halfRows+1:end, 1:halfCols);

cD = decoded(halfRows+1:end, halfCols+1:end);

decompressed\_img = idwt2(cA, cH, cV, cD, 'db9');

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = imresize(img, [256, 256]);

[LL, LH, HL, HH] = dwt2(double(img), 'db9');

SC = zeros(size(LL) + size(HL));

max\_iterations = 50;

iteration = 0;

while true

iteration = iteration + 1;

LL\_threshold = find\_initial\_threshold(LL);

LH\_threshold = find\_initial\_threshold(LH);

HL\_threshold = find\_initial\_threshold(HL);

HH\_threshold = find\_initial\_threshold(HH);

SC(1:size(LL,1), 1:size(LL,2)) = LL .\* (LL >= LL\_threshold);

SC(1:size(LH,1), size(LL,2)+1:end) = LH .\* (LH >= LH\_threshold);

SC(size(LL,1)+1:end, 1:size(HL,2)) = HL .\* (HL >= HL\_threshold);

SC(size(LL,1)+1:end, size(LL,2)+1:end) = HH .\* (HH >= HH\_threshold);

decompressed\_img = idwt2(SC(1:size(LL,1), 1:size(LL,2)), ...

SC(1:size(LH,1), size(LL,2)+1:end), ...

SC(size(LL,1)+1:end, 1:size(HL,2)), ...

SC(size(LL,1)+1:end, size(LL,2)+1:end), 'db9');

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

psnr\_value = calculate\_psnr(double(img), decompressed\_img);

if psnr\_value >= 36 || iteration >= max\_iterations

break;

end

LL\_threshold = update\_threshold(LL, LL\_threshold);

LH\_threshold = update\_threshold(LH, LH\_threshold);

HL\_threshold = update\_threshold(HL, HL\_threshold);

HH\_threshold = update\_threshold(HH, HH\_threshold);

end

[encodedData, counts] = arithmetic\_encode(SC);

decompressed\_img = mict\_decompression(encodedData, counts, size(SC));

compressed\_size\_bits = numel(encodedData) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function threshold = find\_initial\_threshold(subband)

[counts, values] = hist(subband(:), unique(subband));

[~, idx] = max(counts);

threshold = values(idx);

end

function updated\_threshold = update\_threshold(subband, current\_threshold)

remaining\_values = subband(subband < current\_threshold);

if isempty(remaining\_values)

updated\_threshold = current\_threshold;

else

[counts, values] = hist(remaining\_values(:), unique(remaining\_values));

[~, idx] = max(counts);

updated\_threshold = values(idx);

end

end

function psnr\_value = calculate\_psnr(original, reconstructed)

mse = mean((original(:) - reconstructed(:)).^2);

psnr\_value = 10 \* log10(255^2 / mse);

end

function [encoded, counts] = arithmetic\_encode(data)

data = double(data(:));

symbols = unique(data);

data\_mapped = arrayfun(@(x) find(symbols == x, 1), data);

counts = histc(data\_mapped, 1:length(symbols));

encoded = arithenco(data\_mapped, counts);

end

function decoded = arithmetic\_decode(encodedData, counts, imgSize)

decoded = arithdeco(encodedData, counts, prod(imgSize));

decoded = reshape(decoded, imgSize);

end

function decompressed\_img = mict\_decompression(encodedData, counts, imgSize)

decoded = arithmetic\_decode(encodedData, counts, imgSize);

halfRows = floor(size(decoded, 1) / 2);

halfCols = floor(size(decoded, 2) / 2);

cA = decoded(1:halfRows, 1:halfCols);

cH = decoded(1:halfRows, halfCols+1:end);

cV = decoded(halfRows+1:end, 1:halfCols);

cD = decoded(halfRows+1:end, halfCols+1:end);

decompressed\_img = idwt2(cA, cH, cV, cD, 'db9');

end

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

try

tic;

run\_MICT\_compression(img);

times(idx) = toc;

catch ME

warning(['Error on image ', num2str(idx), ': ', ME.message]);

times(idx) = NaN;

end

end

avg\_time = nanmean(times);

mem\_info = memory;

mem\_usage\_MB = mem\_info.MemUsedMATLAB / 1e6;

fprintf('\nMICT Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', avg\_time);

fprintf('Avg Memory Used: %.2f MB\n', mem\_usage\_MB);

function run\_MICT\_compression(img)

img = imresize(img, [256, 256]);

img = double(img);

[LL, LH, HL, HH] = dwt2(img, 'db9');

[rows\_LL, cols\_LL] = size(LL);

[rows\_LH, cols\_LH] = size(LH);

[rows\_HL, cols\_HL] = size(HL);

[rows\_HH, cols\_HH] = size(HH);

SC = zeros(rows\_LL + rows\_HL, cols\_LL + cols\_LH);

max\_iterations = 50;

iteration = 0;

while true

iteration = iteration + 1;

LL\_threshold = find\_initial\_threshold(LL);

LH\_threshold = find\_initial\_threshold(LH);

HL\_threshold = find\_initial\_threshold(HL);

HH\_threshold = find\_initial\_threshold(HH);

SC1 = zeros(size(SC));

SC1(1:rows\_LL, 1:cols\_LL) = LL .\* (LL >= LL\_threshold);

SC1(1:rows\_LH, cols\_LL+1:cols\_LL+cols\_LH) = LH .\* (LH >= LH\_threshold);

SC1(rows\_LL+1:rows\_LL+rows\_HL, 1:cols\_HL) = HL .\* (HL >= HL\_threshold);

SC1(rows\_LL+1:rows\_LL+rows\_HH, cols\_LL+1:cols\_LL+cols\_HH) = HH .\* (HH >= HH\_threshold);

SC = SC + SC1;

reconstructed = idwt2(SC(1:rows\_LL, 1:cols\_LL), ...

SC(1:rows\_LH, cols\_LL+1:cols\_LL+cols\_LH), ...

SC(rows\_LL+1:rows\_LL+rows\_HL, 1:cols\_HL), ...

SC(rows\_LL+1:rows\_LL+rows\_HH, cols\_LL+1:cols\_LL+cols\_HH), 'db9');

reconstructed = imresize(reconstructed, size(img), 'nearest');

psnr\_val = calculate\_psnr(img, reconstructed);

if psnr\_val >= 36 || iteration >= max\_iterations

break;

end

LL\_threshold = update\_threshold(LL, LL\_threshold);

LH\_threshold = update\_threshold(LH, LH\_threshold);

HL\_threshold = update\_threshold(HL, HL\_threshold);

HH\_threshold = update\_threshold(HH, HH\_threshold);

end

[encodedData, counts] = arithmetic\_encode(SC);

mict\_decompression(encodedData, counts, size(SC));

end

function threshold = find\_initial\_threshold(subband)

abs\_coeffs = abs(subband(:));

sorted\_coeffs = sort(abs\_coeffs, 'descend');

top10percent = round(0.1 \* length(sorted\_coeffs));

threshold = mean(sorted\_coeffs(1:top10percent));

end

function new\_threshold = update\_threshold(subband, previous\_threshold)

delta = 0.05 \* previous\_threshold;

new\_threshold = max(0, previous\_threshold - delta);

end

function psnr\_val = calculate\_psnr(original, reconstructed)

mse = mean((original(:) - reconstructed(:)).^2);

if mse == 0

psnr\_val = Inf;

else

max\_pixel = max(original(:));

psnr\_val = 10 \* log10(max\_pixel^2 / mse);

end

end

function [encoded, counts] = arithmetic\_encode(matrix)

symbols = unique(matrix(:));

counts = histc(matrix(:), symbols);

symbols = double(symbols(:));

counts = double(counts(:));

pmf = counts / sum(counts);

stream = matrix(:);

encoded = arithmetic\_coder(pmf, stream, symbols);

end

function encoded = arithmetic\_coder(pmf, stream, symbols)

symbol\_map = containers.Map(symbols, 1:length(symbols));

low = 0.0;

high = 1.0;

for k = 1:length(stream)

idx = symbol\_map(stream(k));

cumulative = [0; cumsum(pmf)];

range = high - low;

high = low + range \* cumulative(idx + 1);

low = low + range \* cumulative(idx);

end

encoded = (low + high) / 2;

end

function reconstructed = mict\_decompression(encoded, counts, size\_matrix)

symbols = 1:length(counts);

pmf = counts / sum(counts);

len = prod(size\_matrix);

reconstructed = arithmetic\_decoder(encoded, pmf, symbols, len);

reconstructed = reshape(reconstructed, size\_matrix);

end

function stream = arithmetic\_decoder(encoded, pmf, symbols, len)

cumulative = [0; cumsum(pmf)];

low = 0.0;

high = 1.0;

value = encoded;

stream = zeros(len, 1);

for k = 1:len

range = high - low;

scaled\_value = (value - low) / range;

idx = find(cumulative <= scaled\_value, 1, 'last') - 1;

if idx == 0

idx = 1;

end

stream(k) = symbols(idx);

high = low + range \* cumulative(idx + 1);

low = low + range \* cumulative(idx);

end

end

**A.1.10 Frequency Domain Decomposition**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = double(img);

dct\_image = dct2(img);

[rows, cols] = size(dct\_image);

z\_scanned = zigzag(dct\_image(:));

reshaped\_dct = reshape(z\_scanned, rows, cols);

high\_bits = floor(reshaped\_dct / 256);

low\_bits = mod(reshaped\_dct, 256);

mask\_high = high\_bits > 0;

mask\_low = low\_bits > 0;

combined\_high = high\_bits .\* mask\_high;

combined\_low = low\_bits .\* mask\_low;

if mod(rows, 2) ~= 0

combined\_high = padarray(combined\_high, [1, 0], 'post');

combined\_low = padarray(combined\_low, [1, 0], 'post');

rows = size(combined\_high, 1);

end

if mod(cols, 2) ~= 0

combined\_high = padarray(combined\_high, [0, 1], 'post');

combined\_low = padarray(combined\_low, [0, 1], 'post');

cols = size(combined\_high, 2);

end

row\_block\_sizes = [floor(rows / 2), rows - floor(rows / 2)];

col\_block\_sizes = [cols];

row\_blocks = mat2cell(combined\_high, row\_block\_sizes, col\_block\_sizes);

col\_blocks = mat2cell(combined\_low, row\_block\_sizes, col\_block\_sizes);

sub\_images = {row\_blocks{1,1}, row\_blocks{2,1}, col\_blocks{1,1}, col\_blocks{2,1}};

reconstructed\_high = cell2mat(row\_blocks);

reconstructed\_low = cell2mat(col\_blocks);

reconstructed\_high = reconstructed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_low = reconstructed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

coeff\_loss = dct\_image - (reconstructed\_high + reconstructed\_low);

compressed\_high = jpeg\_xl\_encode(combined\_high);

decompressed\_high = jpeg\_xl\_decode(compressed\_high);

compressed\_low = compress\_net\_encode(combined\_low);

decompressed\_low = compress\_net\_decode(compressed\_low, size(combined\_low));

compressed\_loss = compress\_net\_encode(coeff\_loss(:));

decompressed\_loss = compress\_net\_decode(compressed\_loss, size(coeff\_loss));

decompressed\_high = double(decompressed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2)));

decompressed\_low = decompressed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

decompressed\_loss = decompressed\_loss(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_img = idct2(decompressed\_loss + decompressed\_high + decompressed\_low);

if size(reconstructed\_img) ~= size(img)

reconstructed\_img = imresize(reconstructed\_img, size(img), 'nearest');

end

profile = double(reconstructed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(reconstructed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

end

cnr\_values = [cnr\_values, cnr];

ssim\_value = ssim(uint8(reconstructed\_img), uint8(img));

ssim\_values = [ssim\_values, ssim\_value];

original\_bits = numel(img) \* 8;

compressed\_size = numel(compressed\_high) + numel(compressed\_low) + numel(compressed\_loss);

cr\_value = original\_bits / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function z = zigzag(matrix)

[rows, cols] = size(matrix);

z = zeros(rows \* cols, 1);

index = 1;

for i = 1:rows + cols - 1

if mod(i, 2) == 0

for x = 1:rows

y = i - x + 1;

if y > 0 && y <= cols

z(index) = matrix(x, y);

index = index + 1;

end

end

else

for y = 1:cols

x = i - y + 1;

if x > 0 && x <= rows

z(index) = matrix(x, y);

index = index + 1;

end

end

end

end

end

function compressed = jpeg\_xl\_encode(data)

temp\_file = 'temp\_high\_bits.png';

imwrite(uint8(data), temp\_file);

fid = fopen(temp\_file, 'r');

compressed = fread(fid, '\*uint8');

fclose(fid);

end

function decompressed = jpeg\_xl\_decode(compressed)

temp\_file = 'temp\_high\_bits\_decoded.png';

fid = fopen(temp\_file, 'w');

fwrite(fid, compressed, 'uint8');

fclose(fid);

decompressed = imread(temp\_file);

end

function compressed = compress\_net\_encode(data)

quantized\_data = round(data / 4) \* 4;

symbols = unique(quantized\_data);

counts = histc(quantized\_data(:), symbols);

probabilities = counts / sum(counts);

dict = huffmandict(symbols, probabilities);

compressed = huffmanenco(quantized\_data(:), dict);

save('compress\_net\_dict.mat', 'dict', 'symbols', '-v7.3');

end

function decompressed = compress\_net\_decode(compressed, original\_size)

load('compress\_net\_dict.mat', 'dict');

decompressed = huffmandeco(compressed, dict);

if numel(decompressed) ~= prod(original\_size)

error('Decompressed data size mismatch: Expected %d elements, got %d.', prod(original\_size), numel(decompressed));

end

decompressed = reshape(decompressed, original\_size);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

compression\_ratios = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = double(img);

dct\_image = dct2(img);

[rows, cols] = size(dct\_image);

z\_scanned = zigzag(dct\_image(:));

reshaped\_dct = reshape(z\_scanned, rows, cols);

high\_bits = floor(reshaped\_dct / 256);

low\_bits = mod(reshaped\_dct, 256);

mask\_high = high\_bits > 0;

mask\_low = low\_bits > 0;

combined\_high = high\_bits .\* mask\_high;

combined\_low = low\_bits .\* mask\_low;

if mod(rows, 2) ~= 0

combined\_high = padarray(combined\_high, [1, 0], 'post');

combined\_low = padarray(combined\_low, [1, 0], 'post');

rows = size(combined\_high, 1);

end

if mod(cols, 2) ~= 0

combined\_high = padarray(combined\_high, [0, 1], 'post');

combined\_low = padarray(combined\_low, [0, 1], 'post');

cols = size(combined\_high, 2);

end

row\_block\_sizes = [floor(rows / 2), rows - floor(rows / 2)];

col\_block\_sizes = [cols];

row\_blocks = mat2cell(combined\_high, row\_block\_sizes, col\_block\_sizes);

col\_blocks = mat2cell(combined\_low, row\_block\_sizes, col\_block\_sizes);

reconstructed\_high = cell2mat(row\_blocks);

reconstructed\_low = cell2mat(col\_blocks);

reconstructed\_high = reconstructed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_low = reconstructed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

coeff\_loss = dct\_image - (reconstructed\_high + reconstructed\_low);

compressed\_high = jpeg\_xl\_encode(combined\_high);

decompressed\_high = jpeg\_xl\_decode(compressed\_high);

compressed\_low = compress\_net\_encode(combined\_low);

decompressed\_low = compress\_net\_decode(compressed\_low, size(combined\_low));

compressed\_loss = compress\_net\_encode(coeff\_loss(:));

decompressed\_loss = compress\_net\_decode(compressed\_loss, size(coeff\_loss));

decompressed\_high = double(decompressed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2)));

decompressed\_low = decompressed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

decompressed\_loss = decompressed\_loss(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_img = idct2(decompressed\_loss + decompressed\_high + decompressed\_low);

if size(reconstructed\_img) ~= size(img)

reconstructed\_img = imresize(reconstructed\_img, size(img), 'nearest');

end

original\_size\_MB = numel(img) \* 8 / (1024 \* 1024);

compressed\_size\_bits = (numel(compressed\_high) + numel(compressed\_low) + numel(compressed\_loss)) \* 8;

compressed\_size\_MB = compressed\_size\_bits / (8 \* 1024 \* 1024);

CR = original\_size\_MB / compressed\_size\_MB;

compression\_ratios(idx) = CR;

fprintf('Image %d: Original Size: %.2f MB, Compressed Size: %.2f MB, CR: %.2f\n', ...

idx, original\_size\_MB, compressed\_size\_MB, CR);

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

average\_CR = mean(compression\_ratios);

fprintf('Average Compression Ratio (CR): %.2f\n', average\_CR);

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function z = zigzag(matrix)

[rows, cols] = size(matrix);

z = zeros(rows \* cols, 1);

index = 1;

for i = 1:rows + cols - 1

if mod(i, 2) == 0

for x = 1:rows

y = i - x + 1;

if y > 0 && y <= cols

z(index) = matrix(x, y);

index = index + 1;

end

end

else

for y = 1:cols

x = i - y + 1;

if x > 0 && x <= rows

z(index) = matrix(x, y);

index = index + 1;

end

end

end

end

end

function compressed = jpeg\_xl\_encode(data)

temp\_file = 'temp\_high\_bits.png';

imwrite(uint8(data), temp\_file);

fid = fopen(temp\_file, 'r');

compressed = fread(fid, '\*uint8');

fclose(fid);

end

function decompressed = jpeg\_xl\_decode(compressed)

temp\_file = 'temp\_high\_bits\_decoded.png';

fid = fopen(temp\_file, 'w');

fwrite(fid, compressed, 'uint8');

fclose(fid);

decompressed = imread(temp\_file);

end

function compressed = compress\_net\_encode(data)

quantized\_data = round(data / 4) \* 4;

symbols = unique(quantized\_data);

counts = histc(quantized\_data(:), symbols);

probabilities = counts / sum(counts);

dict = huffmandict(symbols, probabilities);

compressed = huffmanenco(quantized\_data(:), dict);

save('compress\_net\_dict.mat', 'dict', 'symbols', '-v7.3');

end

function decompressed = compress\_net\_decode(compressed, original\_size)

load('compress\_net\_dict.mat', 'dict');

decompressed = huffmandeco(compressed, dict);

if numel(decompressed) ~= prod(original\_size)

error('Decompressed data size mismatch: Expected %d elements, got %d.', prod(original\_size), numel(decompressed));

end

decompressed = reshape(decompressed, original\_size);

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

tic;

run\_Frequency\_Domain\_compression(img);

times(idx) = toc;

catch ME

warning(['Error on image ', num2str(idx), ': ', ME.message]);

times(idx) = NaN;

end

end

avg\_time = nanmean(times);

mem\_info = memory;

avg\_mem\_MB = mem\_info.MemUsedMATLAB / 1e6;

fprintf('Frequency Domain Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', avg\_time);

fprintf('Avg Memory Used: %.2f MB\n', avg\_mem\_MB);

function run\_Frequency\_Domain\_compression(img)

img = double(img);

dct\_image = dct2(img);

[rows, cols] = size(dct\_image);

z\_scanned = zigzag(dct\_image(:));

reshaped\_dct = reshape(z\_scanned, rows, cols);

high\_bits = floor(reshaped\_dct / 256);

low\_bits = mod(reshaped\_dct, 256);

mask\_high = high\_bits > 0;

mask\_low = low\_bits > 0;

combined\_high = high\_bits .\* mask\_high;

combined\_low = low\_bits .\* mask\_low;

if mod(rows, 2) ~= 0

combined\_high = padarray(combined\_high, [1, 0], 'post');

combined\_low = padarray(combined\_low, [1, 0], 'post');

rows = size(combined\_high, 1);

end

if mod(cols, 2) ~= 0

combined\_high = padarray(combined\_high, [0, 1], 'post');

combined\_low = padarray(combined\_low, [0, 1], 'post');

cols = size(combined\_high, 2);

end

row\_block\_sizes = [floor(rows / 2), rows - floor(rows / 2)];

col\_block\_sizes = [cols];

row\_blocks = mat2cell(combined\_high, row\_block\_sizes, col\_block\_sizes);

col\_blocks = mat2cell(combined\_low, row\_block\_sizes, col\_block\_sizes);

reconstructed\_high = cell2mat(row\_blocks);

reconstructed\_low = cell2mat(col\_blocks);

reconstructed\_high = reconstructed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_low = reconstructed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

coeff\_loss = dct\_image - (reconstructed\_high + reconstructed\_low);

compressed\_high = jpeg\_xl\_encode(combined\_high);

decompressed\_high = jpeg\_xl\_decode(compressed\_high);

compressed\_low = compress\_net\_encode(combined\_low);

decompressed\_low = compress\_net\_decode(compressed\_low, size(combined\_low));

compressed\_loss = compress\_net\_encode(coeff\_loss(:));

decompressed\_loss = compress\_net\_decode(compressed\_loss, size(coeff\_loss));

decompressed\_high = double(decompressed\_high(1:size(dct\_image, 1), 1:size(dct\_image, 2)));

decompressed\_low = decompressed\_low(1:size(dct\_image, 1), 1:size(dct\_image, 2));

decompressed\_loss = decompressed\_loss(1:size(dct\_image, 1), 1:size(dct\_image, 2));

reconstructed\_img = idct2(decompressed\_loss + decompressed\_high + decompressed\_low);

if ~isequal(size(reconstructed\_img), size(img))

reconstructed\_img = imresize(reconstructed\_img, size(img), 'nearest');

end

end

function z = zigzag(matrix)

[rows, cols] = size(matrix);

z = zeros(rows \* cols, 1);

index = 1;

for i = 1:rows + cols - 1

if mod(i, 2) == 0

for x = 1:rows

y = i - x + 1;

if y > 0 && y <= cols

z(index) = matrix(x, y);

index = index + 1;

end

end

else

for y = 1:cols

x = i - y + 1;

if x > 0 && x <= rows

z(index) = matrix(x, y);

index = index + 1;

end

end

end

end

end

function compressed = jpeg\_xl\_encode(data)

temp\_file = 'temp\_jpegxl.png';

imwrite(uint8(data), temp\_file);

fid = fopen(temp\_file, 'r');

compressed = fread(fid, '\*uint8');

fclose(fid);

end

function decompressed = jpeg\_xl\_decode(compressed)

temp\_file = 'temp\_jpegxl\_decoded.png';

fid = fopen(temp\_file, 'w');

fwrite(fid, compressed, 'uint8');

fclose(fid);

decompressed = imread(temp\_file);

end

function compressed = compress\_net\_encode(data)

quantized\_data = round(data / 4) \* 4;

symbols = unique(quantized\_data);

counts = histc(quantized\_data(:), symbols);

probabilities = counts / sum(counts);

dict = huffmandict(symbols, probabilities);

compressed = huffmanenco(quantized\_data(:), dict);

save('compress\_net\_dict.mat', 'dict', 'symbols', '-v7.3');

end

function decompressed = compress\_net\_decode(compressed, original\_size)

load('compress\_net\_dict.mat', 'dict');

decompressed = huffmandeco(compressed, dict);

if numel(decompressed) ~= prod(original\_size)

error('Decompressed data size mismatch');

end

decompressed = reshape(decompressed, original\_size);

end

**Appendix A.2: Unexplored Compression Techniques**

**A.2.1 Modified Singular Value Decomposition (MSVD)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

pixel\_resolution = 72;

pixel\_size = 25.4 / pixel\_resolution;

results\_msvd\_k10 = [];

blockSize = 64;

rank = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

padRows = blockSize \* ceil(rows / blockSize) - rows;

padCols = blockSize \* ceil(cols / blockSize) - cols;

imgPadded = padarray(img, [padRows, padCols], 'replicate', 'post');

compressed\_img = zeros(size(imgPadded));

compressed\_size = 0;

for i = 1:blockSize:size(imgPadded, 1)

for j = 1:blockSize:size(imgPadded, 2)

block = imgPadded(i:i+blockSize-1, j:j+blockSize-1);

[U, S, V] = svd(double(block));

max\_rank = min(size(U, 2), rank);

Uk = U(:, 1:max\_rank);

Sk = S(1:max\_rank, 1:max\_rank);

Vk = V(:, 1:max\_rank);

compressed\_size = compressed\_size + numel(Uk) + numel(Sk) + numel(Vk);

compressed\_block = Uk \* Sk \* Vk';

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = compressed\_block;

end

end

compressed\_img\_cropped = compressed\_img(1:rows, 1:cols);

original\_size = numel(img) \* 8;

cr = original\_size / compressed\_size;

[fwhm, cnr, ssim\_val] = calculate\_compression\_metrics(img, compressed\_img\_cropped);

fwhm\_mm = fwhm \* pixel\_size;

results\_msvd\_k10 = [results\_msvd\_k10; idx, fwhm\_mm, cnr, ssim\_val, cr];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_val, cr);

end

fprintf('Average FWHM: %.2f mm\n', mean(results\_msvd\_k10(:,2)));

fprintf('Average CNR: %.2f\n', mean(results\_msvd\_k10(:,3)));

fprintf('Average SSIM: %.2f\n', mean(results\_msvd\_k10(:,4)));

fprintf('Average CR: %.2f\n', mean(results\_msvd\_k10(:,5)));

diary off;

function [fwhm, cnr, ssim\_val] = calculate\_compression\_metrics(original\_img, compressed\_img)

profile = double(compressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm = right\_idx - left\_idx;

reconstruction\_error = abs(double(original\_img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = NaN;

ssim\_val = NaN;

return;

end

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_val = ssim(uint8(compressed\_img), original\_img);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

blockSize = 64;

rank = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

padRows = blockSize \* ceil(rows / blockSize) - rows;

padCols = blockSize \* ceil(cols / blockSize) - cols;

imgPadded = padarray(img, [padRows, padCols], 'replicate', 'post');

compressed\_img = zeros(size(imgPadded));

compressed\_size\_bits = 0;

for i = 1:blockSize:size(imgPadded, 1)

for j = 1:blockSize:size(imgPadded, 2)

block = imgPadded(i:i+blockSize-1, j:j+blockSize-1);

[U, S, V] = svd(double(block));

max\_rank = min(size(U, 2), rank);

Uk = U(:, 1:max\_rank);

Sk = S(1:max\_rank, 1:max\_rank);

Vk = V(:, 1:max\_rank);

compressed\_size\_bits = compressed\_size\_bits + (numel(Uk) + numel(Sk) + numel(Vk)) \* 8;

compressed\_block = Uk \* Sk \* Vk';

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = compressed\_block;

end

end

compressed\_img\_cropped = compressed\_img(1:rows, 1:cols);

original\_size\_bits = numel(img) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

blockSize = 64;

rank = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[rows, cols] = size(img);

padRows = blockSize \* ceil(rows / blockSize) - rows;

padCols = blockSize \* ceil(cols / blockSize) - cols;

imgPadded = padarray(img, [padRows, padCols], 'replicate', 'post');

compressed\_img = zeros(size(imgPadded));

tic;

for i = 1:blockSize:size(imgPadded, 1)

for j = 1:blockSize:size(imgPadded, 2)

block = double(imgPadded(i:i+blockSize-1, j:j+blockSize-1));

[U, S, V] = svd(block);

k = min(rank, size(U,2));

Uk = U(:, 1:k);

Sk = S(1:k, 1:k);

Vk = V(:, 1:k);

compressed\_block = Uk \* Sk \* Vk';

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = compressed\_block;

end

end

times(idx) = toc;

end

exec\_time = mean(times);

mem\_info = memory;

mem\_usage = mem\_info.MemUsedMATLAB / 1e6;

fprintf('\nMSVD Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', exec\_time);

fprintf('Avg Memory Used: %.2f MB\n', mem\_usage);

**A.2.2 Modified Fast Fractal Image Compression (MFFI)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

blockSize = 16;

overlap = 4;

domainBlocks = createDomainBlocks(img, blockSize, overlap);

compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap);

profile = double(compressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_pixels = right\_idx - left\_idx;

fwhm\_mm = fwhm\_pixels \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(compressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;% If no valid data for either region, return default values to avoid NaN

ssim\_value = ssim(uint8(compressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_signal = std(signal\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(compressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(find(compressed\_img ~= 0)) \* 8;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function domainBlocks = createDomainBlocks(image, blockSize, overlap)

[rows, cols] = size(image);

step = blockSize - overlap;

domainBlocks = {};

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

domainBlocks{end+1} = image(i:i+blockSize-1, j:j+blockSize-1);

end

end

end

function compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap)

[rows, cols] = size(img);

step = blockSize - overlap;

compressed\_img = zeros(size(img));

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

rangeBlock = img(i:i+blockSize-1, j:j+blockSize-1);

[bestMatch, bestIndex] = findBestMatch(rangeBlock, domainBlocks);

[S, O] = calculateAffineTransformation(rangeBlock, bestMatch);

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = S \* bestMatch + O;

end

end

end

function [bestMatch, bestIndex] = findBestMatch(rangeBlock, domainBlocks)

rangeBlock = double(rangeBlock);

minDist = Inf;

bestMatch = [];

bestIndex = -1;

for i = 1:length(domainBlocks)

domainBlock = double(domainBlocks{i});

dist = norm(rangeBlock - domainBlock, 'fro');

if dist < minDist

minDist = dist;

bestMatch = domainBlock;

bestIndex = i;

end

end

end

function [S, O] = calculateAffineTransformation(rangeBlock, domainBlock)

meanRange = mean(rangeBlock(:));

meanDomain = mean(domainBlock(:));

S = (meanRange - meanDomain) / std(domainBlock(:));

O = meanRange - S \* meanDomain;

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

blockSize = 16;

overlap = 4;

domainBlocks = createDomainBlocks(img, blockSize, overlap);

compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap);

compressed\_size\_bits = numel(find(compressed\_img ~= 0)) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function domainBlocks = createDomainBlocks(image, blockSize, overlap)

[rows, cols] = size(image);

step = blockSize - overlap;

domainBlocks = {};

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

domainBlocks{end+1} = image(i:i+blockSize-1, j:j+blockSize-1);

end

end

end

function compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap)

[rows, cols] = size(img);

step = blockSize - overlap;

compressed\_img = zeros(size(img));

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

rangeBlock = img(i:i+blockSize-1, j:j+blockSize-1);

[bestMatch, bestIndex] = findBestMatch(rangeBlock, domainBlocks);

[S, O] = calculateAffineTransformation(rangeBlock, bestMatch);

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = S \* bestMatch + O;

end

end

end

function [bestMatch, bestIndex] = findBestMatch(rangeBlock, domainBlocks)

rangeBlock = double(rangeBlock);

minDist = Inf;

bestMatch = [];

bestIndex = -1;

for i = 1:length(domainBlocks)

domainBlock = double(domainBlocks{i});

dist = norm(rangeBlock - domainBlock, 'fro');

if dist < minDist

minDist = dist;

bestMatch = domainBlock;

bestIndex = i;

end

end

end

function [S, O] = calculateAffineTransformation(rangeBlock, domainBlock)

meanRange = mean(rangeBlock(:));

meanDomain = mean(domainBlock(:));

S = (meanRange - meanDomain) / std(domainBlock(:));

O = meanRange - S \* meanDomain;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

times = zeros(num\_images, 1);

blockSize = 16;

overlap = 4;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

domainBlocks = createDomainBlocks(img, blockSize, overlap);

compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap);

times(idx) = toc;

catch ME

warning(['Error processing image ', num2str(idx), ': ', ME.message]);

times(idx) = NaN;

end

end

avg\_time = nanmean(times);

mem\_info = memory;

avg\_mem = mem\_info.MemUsedMATLAB / 1e6;

fprintf('MFFI Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', avg\_time);

fprintf('Avg Memory Used: %.2f MB\n', avg\_mem);

function domainBlocks = createDomainBlocks(image, blockSize, overlap)

[rows, cols] = size(image);

step = blockSize - overlap;

domainBlocks = {};

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

domainBlocks{end+1} = image(i:i+blockSize-1, j:j+blockSize-1);

end

end

end

function compressed\_img = fractalCompression(img, domainBlocks, blockSize, overlap)

[rows, cols] = size(img);

step = blockSize - overlap;

compressed\_img = zeros(size(img));

for i = 1:step:rows - blockSize + 1

for j = 1:step:cols - blockSize + 1

rangeBlock = img(i:i+blockSize-1, j:j+blockSize-1);

[bestMatch, ~] = findBestMatch(rangeBlock, domainBlocks);

[S, O] = calculateAffineTransformation(rangeBlock, bestMatch);

compressed\_img(i:i+blockSize-1, j:j+blockSize-1) = S \* bestMatch + O;

end

end

end

function [bestMatch, bestIndex] = findBestMatch(rangeBlock, domainBlocks)

rangeBlock = double(rangeBlock);

minDist = Inf;

bestMatch = [];

bestIndex = -1;

for i = 1:length(domainBlocks)

domainBlock = double(domainBlocks{i});

dist = norm(rangeBlock - domainBlock, 'fro');

if dist < minDist

minDist = dist;

bestMatch = domainBlock;

bestIndex = i;

end

end

end

function [S, O] = calculateAffineTransformation(rangeBlock, domainBlock)

meanRange = mean(rangeBlock(:));

meanDomain = mean(domainBlock(:));

stdDomain = std(domainBlock(:));

if stdDomain == 0

S = 1;

O = meanRange - meanDomain;

else

S = (meanRange - meanDomain) / stdDomain;

O = meanRange - S \* meanDomain;

end

end

**A.2.3 Block Partitioning Embedded Coding (BPEC)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

blockSize = 64;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[cA, cH, cV, cD] = dwt2(img, 'bior4.4');

imgSize = size(img);

[encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize);

decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize);

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(encodedData);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize)

waveletCoeffs = [cA, cH; cV, cD];

[rows, cols] = size(waveletCoeffs);

numBlocks = ceil(rows / blockSize) \* ceil(cols / blockSize);

maxbp = zeros(numBlocks, 1);

encodedData = [];

blockIdx = 1;

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), ...

col:min(col+blockSize-1, cols));

maxbp(blockIdx) = floor(log2(max(abs(block(:)))) + 1e-10);

blockIdx = blockIdx + 1;

end

end

maxBitPlane = max(maxbp);

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), ...

col:min(col+blockSize-1, cols));

encodedBlock = encode\_bitplane(block, p);

encodedData = [encodedData, encodedBlock];

end

end

end

end

function decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize)

rows = imgSize(1);

cols = imgSize(2);

decompressedCoeffs = zeros(rows, cols);

dataIdx = 1;

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

blockRows = min(row+blockSize-1, rows) - row + 1;

blockCols = min(col+blockSize-1, cols) - col + 1;

[decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, [blockRows, blockCols]);

decompressedCoeffs(row:(row+blockRows-1), col:(col+blockCols-1)) = decodedBlock;

end

end

end

halfRows = floor(rows / 2);

halfCols = floor(cols / 2);

cA = decompressedCoeffs(1:halfRows, 1:halfCols);

cH = decompressedCoeffs(1:halfRows, halfCols+1:min(halfCols\*2, cols));

cV = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), 1:halfCols);

cD = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), halfCols+1:min(halfCols\*2, cols));

decompressed\_img = idwt2(cA, cH, cV, cD, 'bior4.4');

end

function encodedBlock = encode\_bitplane(block, p)

sigMask = (abs(block) >= 2^p) & (abs(block) < 2^(p+1));

encodedBlock = sigMask(:)';

end

function [decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, blockSize)

decodedBlock = zeros(blockSize);

for idx = 1:numel(decodedBlock)

if dataIdx > numel(encodedData)

break;

end

if encodedData(dataIdx) == 1

decodedBlock(idx) = 2^p;

end

dataIdx = dataIdx + 1;

end

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

blockSize = 64;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[cA, cH, cV, cD] = dwt2(img, 'bior4.4');

imgSize = size(img);

[encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize);

decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize);

if ~isequal(size(decompressed\_img), size(img))

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

original\_size\_bits = numel(img) \* 8;

compressed\_size\_bits = numel(encodedData) \* 8;

cr\_values(idx) = original\_size\_bits / compressed\_size\_bits;

original\_size\_MB = original\_size\_bits / (8 \* 1024 \* 1024);

compressed\_size\_MB = compressed\_size\_bits / (8 \* 1024 \* 1024);

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize)

waveletCoeffs = [cA, cH; cV, cD];

[rows, cols] = size(waveletCoeffs);

numBlocks = ceil(rows / blockSize) \* ceil(cols / blockSize);

maxbp = zeros(numBlocks, 1);

encodedData = [];

blockIdx = 1;

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), ...

col:min(col+blockSize-1, cols));

maxbp(blockIdx) = floor(log2(max(abs(block(:)))) + 1e-10);

blockIdx = blockIdx + 1;

end

end

maxBitPlane = max(maxbp);

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), ...

col:min(col+blockSize-1, cols));

encodedBlock = encode\_bitplane(block, p);

encodedData = [encodedData, encodedBlock];

end

end

end

end

function decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize)

rows = imgSize(1);

cols = imgSize(2);

decompressedCoeffs = zeros(rows, cols);

dataIdx = 1;

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

blockRows = min(row+blockSize-1, rows) - row + 1;

blockCols = min(col+blockSize-1, cols) - col + 1;

[decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, [blockRows, blockCols]);

decompressedCoeffs(row:(row+blockRows-1), col:(col+blockCols-1)) = decodedBlock;

end

end

end

halfRows = floor(rows / 2);

halfCols = floor(cols / 2);

cA = decompressedCoeffs(1:halfRows, 1:halfCols);

cH = decompressedCoeffs(1:halfRows, halfCols+1:min(halfCols\*2, cols));

cV = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), 1:halfCols);

cD = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), halfCols+1:min(halfCols\*2, cols));

decompressed\_img = idwt2(cA, cH, cV, cD, 'bior4.4');

end

function encodedBlock = encode\_bitplane(block, p)

sigMask = (abs(block) >= 2^p) & (abs(block) < 2^(p+1));

encodedBlock = sigMask(:)';

end

function [decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, blockSize)

decodedBlock = zeros(blockSize);

for idx = 1:numel(decodedBlock)

if dataIdx > numel(encodedData)

break;

end

if encodedData(dataIdx) == 1

decodedBlock(idx) = 2^p;

end

dataIdx = dataIdx + 1;

end

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

pixel\_size = 0.3528;

blockSize = 64;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

mem\_info = memory;

mem\_before = mem\_info.MemUsedMATLAB;

tic;

[cA, cH, cV, cD] = dwt2(img, 'bior4.4');

imgSize = size(img);

[encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize);

decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize);

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = (mem\_info.MemUsedMATLAB - mem\_before) / 1e6;

catch

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('\nBPEC Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function [encodedData, maxBitPlane] = bpec\_encode(cA, cH, cV, cD, blockSize)

waveletCoeffs = [cA, cH; cV, cD];

[rows, cols] = size(waveletCoeffs);

numBlocks = ceil(rows / blockSize) \* ceil(cols / blockSize);

maxbp = zeros(numBlocks, 1);

encodedData = [];

blockIdx = 1;

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), col:min(col+blockSize-1, cols));

maxbp(blockIdx) = floor(log2(max(abs(block(:))) + 1e-10));

blockIdx = blockIdx + 1;

end

end

maxBitPlane = max(maxbp);

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

block = waveletCoeffs(row:min(row+blockSize-1, rows), col:min(col+blockSize-1, cols));

encodedBlock = encode\_bitplane(block, p);

encodedData = [encodedData, encodedBlock];

end

end

end

end

function decompressed\_img = bpec\_decode(encodedData, maxBitPlane, blockSize, imgSize)

rows = imgSize(1);

cols = imgSize(2);

decompressedCoeffs = zeros(rows, cols);

dataIdx = 1;

for p = maxBitPlane:-1:0

for row = 1:blockSize:rows

for col = 1:blockSize:cols

blockRows = min(row+blockSize-1, rows) - row + 1;

blockCols = min(col+blockSize-1, cols) - col + 1;

[decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, [blockRows, blockCols]);

decompressedCoeffs(row:(row+blockRows-1), col:(col+blockCols-1)) = decodedBlock;

end

end

end

halfRows = floor(rows / 2);

halfCols = floor(cols / 2);

cA = decompressedCoeffs(1:halfRows, 1:halfCols);

cH = decompressedCoeffs(1:halfRows, halfCols+1:min(halfCols\*2, cols));

cV = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), 1:halfCols);

cD = decompressedCoeffs(halfRows+1:min(halfRows\*2, rows), halfCols+1:min(halfCols\*2, cols));

decompressed\_img = idwt2(cA, cH, cV, cD, 'bior4.4');

end

function encodedBlock = encode\_bitplane(block, p)

sigMask = (abs(block) >= 2^p) & (abs(block) < 2^(p+1));

encodedBlock = sigMask(:)';

end

function [decodedBlock, dataIdx] = decode\_bitplane(encodedData, dataIdx, p, blockSize)

decodedBlock = zeros(blockSize);

for idx = 1:numel(decodedBlock)

if dataIdx > numel(encodedData)

break;

end

if encodedData(dataIdx) == 1

decodedBlock(idx) = 2^p;

end

dataIdx = dataIdx + 1;

end

end

**A.2.4 Highly Scalable - Block Partitioning Embedded Coding (HS-BPEC)**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

blockSize = 64;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

M = 5;

[coeffs, sizes] = wavedec2(img, M, 'bior4.4');

imgSize = size(img);

[encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M);

decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M);

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(encodedData);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M)

encodedData = [];

resolutionMarkers = [];

maxBitPlane = floor(log2(max(abs(coeffs(:)))) + 1e-10);

for r = M:-1:0

subbands = extract\_subbands(coeffs, sizes, r);

[encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane);

encodedData = [encodedData, encodedSubbands];

resolutionMarkers = [resolutionMarkers; markers];

end

end

function decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M)

coeffs = zeros(sum(prod(sizes, 2)), 1);

dataIdx = 1;

for r = M:-1:1

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(r, :), blockSize, sizes(r+1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, r);

end

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(1, :), blockSize, sizes(1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, 0);

decompressed\_img = waverec2(coeffs, sizes, 'bior4.4');

end

function subbands = extract\_subbands(coeffs, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = coeffs(startIdx:endIdx);

subbandSizes = sizes(r+1, :);

subbands = {reshape(subbandData, subbandSizes)};

end

function coeffs = insert\_subbands(coeffs, subbands, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = subbands{1};

subbandData = subbandData(:);

if length(subbandData) ~= (endIdx - startIdx + 1)

error('Subband size mismatch during insertion at resolution level %d.', r);

end

coeffs(startIdx:endIdx) = subbandData;

end

function [encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane)

encodedSubbands = [];

markers = [];

blockData = subbands{1};

for p = maxBitPlane:-1:0

significantBits = blockData >= 2^p & blockData < 2^(p+1);

encodedSubbands = [encodedSubbands, significantBits(:)'];

end

markers = maxBitPlane;

end

function [decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, markers, blockSize, sizes, dataIdx)

maxbp = markers;

blockSizeData = sizes;

blockData = zeros(blockSizeData);

for p = maxbp:-1:0

bitData = encodedData(dataIdx:dataIdx + numel(blockData) - 1);

dataIdx = dataIdx + numel(blockData);

bitData = reshape(bitData, size(blockData));

blockData = blockData + bitData \* 2^p;

end

decodedSubbands = {blockData};

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

blockSize = 64;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

M = 5;

[coeffs, sizes] = wavedec2(img, M, 'bior4.4');

imgSize = size(img);

[encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M);

decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M);

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

original\_size\_bits = numel(img) \* 8;

compressed\_size\_bits = numel(encodedData);

original\_size\_MB = original\_size\_bits / (8 \* 1024 \* 1024);

compressed\_size\_MB = compressed\_size\_bits / (8 \* 1024 \* 1024);

cr\_values(idx) = original\_size\_bits / compressed\_size\_bits;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M)

encodedData = [];

resolutionMarkers = [];

maxBitPlane = floor(log2(max(abs(coeffs(:)))) + 1e-10);

for r = M:-1:0

subbands = extract\_subbands(coeffs, sizes, r);

[encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane);

encodedData = [encodedData, encodedSubbands];

resolutionMarkers = [resolutionMarkers; markers];

end

end

function decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M)

coeffs = zeros(sum(prod(sizes, 2)), 1);

dataIdx = 1;

for r = M:-1:1

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(r, :), blockSize, sizes(r+1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, r);

end

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(1, :), blockSize, sizes(1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, 0);

decompressed\_img = waverec2(coeffs, sizes, 'bior4.4');

end

function subbands = extract\_subbands(coeffs, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = coeffs(startIdx:endIdx);

subbandSizes = sizes(r+1, :);

subbands = {reshape(subbandData, subbandSizes)};

end

function coeffs = insert\_subbands(coeffs, subbands, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = subbands{1};

subbandData = subbandData(:);

if length(subbandData) ~= (endIdx - startIdx + 1)

error('Subband size mismatch during insertion at resolution level %d.', r);

end

coeffs(startIdx:endIdx) = subbandData;

end

function [encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane)

encodedSubbands = [];

markers = [];

blockData = subbands{1};

for p = maxBitPlane:-1:0

significantBits = blockData >= 2^p & blockData < 2^(p+1);

encodedSubbands = [encodedSubbands, significantBits(:)'];

end

markers = maxBitPlane;

end

function [decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, markers, blockSize, sizes, dataIdx)

maxbp = markers;

blockSizeData = sizes;

blockData = zeros(blockSizeData);

for p = maxbp:-1:0

bitData = encodedData(dataIdx:dataIdx + numel(blockData) - 1);

dataIdx = dataIdx + numel(blockData);

bitData = reshape(bitData, size(blockData));

blockData = blockData + bitData \* 2^p;

end

decodedSubbands = {blockData};

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usage = zeros(num\_images, 1);

blockSize = 64;

M = 5;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3, img = rgb2gray(img); end

mem\_start = memory;

tic;

[coeffs, sizes] = wavedec2(img, M, 'bior4.4');

imgSize = size(img);

[encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M);

decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M);

if ~isequal(size(decompressed\_img), size(img))

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

exec\_times(idx) = toc;

mem\_end = memory;

mem\_usage(idx) = (mem\_end.MemUsedMATLAB - mem\_start.MemUsedMATLAB) / 1e6;

end

fprintf('HS-BPEC Compression Timing and Memory Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', mean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', mean(mem\_usage));

function [encodedData, resolutionMarkers] = hs\_bpec\_encode(coeffs, sizes, blockSize, M)

encodedData = [];

resolutionMarkers = [];

maxBitPlane = floor(log2(max(abs(coeffs(:)))) + 1e-10);

for r = M:-1:0

subbands = extract\_subbands(coeffs, sizes, r);

[encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane);

encodedData = [encodedData, encodedSubbands];

resolutionMarkers = [resolutionMarkers; markers];

end

end

function decompressed\_img = hs\_bpec\_decode(encodedData, resolutionMarkers, sizes, blockSize, imgSize, M)

coeffs = zeros(sum(prod(sizes, 2)), 1);

dataIdx = 1;

for r = M:-1:1

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(r, :), blockSize, sizes(r+1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, r);

end

[decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, resolutionMarkers(1, :), blockSize, sizes(1, :), dataIdx);

coeffs = insert\_subbands(coeffs, decodedSubbands, sizes, 0);

decompressed\_img = waverec2(coeffs, sizes, 'bior4.4');

end

function subbands = extract\_subbands(coeffs, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = coeffs(startIdx:endIdx);

subbandSizes = sizes(r+1, :);

subbands = {reshape(subbandData, subbandSizes)};

end

function coeffs = insert\_subbands(coeffs, subbands, sizes, r)

startIdx = sum(prod(sizes(1:r, :), 2)) + 1;

endIdx = sum(prod(sizes(1:r+1, :), 2));

subbandData = subbands{1};

subbandData = subbandData(:);

coeffs(startIdx:endIdx) = subbandData;

end

function [encodedSubbands, markers] = encode\_resolution\_level(subbands, blockSize, maxBitPlane)

encodedSubbands = [];

markers = [];

blockData = subbands{1};

for p = maxBitPlane:-1:0

significantBits = blockData >= 2^p & blockData < 2^(p+1);

encodedSubbands = [encodedSubbands, significantBits(:)'];

end

markers = maxBitPlane;

end

function [decodedSubbands, dataIdx] = decode\_resolution\_level(encodedData, markers, blockSize, sizes, dataIdx)

maxbp = markers;

blockData = zeros(sizes);

for p = maxbp:-1:0

bitData = encodedData(dataIdx:dataIdx + numel(blockData) - 1);

dataIdx = dataIdx + numel(blockData);

bitData = reshape(bitData, size(blockData));

blockData = blockData + bitData \* 2^p;

end

decodedSubbands = {blockData};

end

**A.2.5 Improved Lossless Image Compression**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

compressedData = repmat(struct('compressedStream', [], 'dict', [], 'positionZeros', [], 'positionNegatives', [], 'negValueStorage', []), 1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[predictedImage, predictionErrors] = linear\_prediction(img);

[lowFreq, highFreq, positionZeros, positionNegatives] = integer\_wavelet\_transform(predictionErrors);

compressedData(idx) = forced\_huffman\_encoding(lowFreq, highFreq, positionZeros, positionNegatives);

decompressed\_img = decompress(compressedData(idx), size(img));

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(compressedData(idx).compressedStream);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [predictedImage, predictionErrors] = linear\_prediction(inputImage)

[rows, cols] = size(inputImage);

predictedImage = zeros(rows, cols);

predictionErrors = zeros(rows, cols);

for i = 2:rows-1

for j = 2:cols-1

predictedValue = round((inputImage(i-1, j-1) + inputImage(i-1, j) + ...

inputImage(i-1, j+1) + inputImage(i, j-1) + ...

inputImage(i, j+1)) / 5);

predictedImage(i, j) = predictedValue;

predictionErrors(i, j) = double(inputImage(i, j)) - predictedValue;

end

end

end

function [lowFreq, highFreq, positionZeros, positionNegatives] = integer\_wavelet\_transform(predictionErrors)

[rows, cols] = size(predictionErrors);

numLevels = 6;

lowFreq = predictionErrors;

highFreq = zeros(size(predictionErrors));

for level = 1:numLevels

[lowFreq, highFreq] = legall53\_transform(lowFreq);

end

positionZeros = find(lowFreq == 0);

positionNegatives = find(lowFreq < 0);

lowFreq(lowFreq < 0) = abs(lowFreq(lowFreq < 0));

end

function [lowFreq, highFreq] = legall53\_transform(data)

[rows, cols] = size(data);

rowTransformed = zeros(size(data));

for r = 1:rows

[low, high] = legall53\_1d(data(r, :));

rowTransformed(r, :) = [low, high];

end

colTransformed = zeros(size(rowTransformed));

for c = 1:cols

[low, high] = legall53\_1d(rowTransformed(:, c)');

colTransformed(:, c) = [low, high]';

end

halfRows = floor(rows / 2);

halfCols = floor(cols / 2);

lowFreq = colTransformed(1:halfRows, 1:halfCols);

highFreq = colTransformed(halfRows+1:end, halfCols+1:end);

end

function [low, high] = legall53\_1d(data)

n = length(data);

low = zeros(1, ceil(n / 2));

high = zeros(1, floor(n / 2));

for i = 1:length(high)

high(i) = data(2 \* i) - floor((data(max(2 \* i - 1, 1)) + data(min(2 \* i + 1, n))) / 2);

end

for i = 1:length(low)

low(i) = data(2 \* i - 1) + floor((high(max(i - 1, 1)) + high(min(i + 1, length(high)))) / 4);

end

end

function compressedData = forced\_huffman\_encoding(lowFreq, highFreq, positionZeros, positionNegatives)

dataStream = [lowFreq(:); highFreq(:)];

dataStream = dataStream(dataStream ~= 0);

allSymbols = unique(dataStream);

minVal = min(allSymbols);

maxVal = max(allSymbols);

allSymbols = minVal:maxVal;

allProbabilities = histcounts(dataStream, [allSymbols, max(allSymbols)+1], 'Normalization', 'probability');

if sum(allProbabilities) == 0

allProbabilities(:) = 1 / numel(allProbabilities);

else

allProbabilities = allProbabilities / sum(allProbabilities);

end

dict = huffmandict(allSymbols, allProbabilities);

dictSymbols = [dict{:,1}];

missingSymbols = setdiff(dataStream, dictSymbols);

if ~isempty(missingSymbols)

warning('Huffman dictionary missing symbols: %s', mat2str(missingSymbols));

extraSymbols = missingSymbols(:);

extraProbabilities = repmat(1e-6, size(extraSymbols));

dict = huffmandict([dictSymbols, extraSymbols], [allProbabilities, extraProbabilities]);

end

paddedStream = [dataStream; min(allSymbols) \* ones(70000, 1)];

compressedStream = huffmanenco(paddedStream, dict);

compressedData = struct('compressedStream', {compressedStream}, ...

'dict', {dict}, ...

'positionZeros', {positionZeros}, ...

'positionNegatives', {positionNegatives}, ...

'negValueStorage', {dataStream(positionNegatives)});

end

function decompressedImage = decompress(singleCompressedData, originalSize)

decodedStream = huffmandeco(singleCompressedData.compressedStream, singleCompressedData.dict);

lowFreq = zeros(originalSize);

highFreq = zeros(originalSize);

lowFreq(singleCompressedData.positionZeros) = 0;

if numel(decodedStream) >= numel(singleCompressedData.positionNegatives)

lowFreq(singleCompressedData.positionNegatives) = -singleCompressedData.negValueStorage;

else

error('Mismatch: Decoded stream size is smaller than positionNegatives.');

end

decompressedImage = lowFreq + highFreq;

decompressedImage = uint8(decompressedImage);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

compressedData = repmat(struct('compressedStream', [], 'dict', [], 'positionZeros', [], 'positionNegatives', [], 'negValueStorage', []), 1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[predictedImage, predictionErrors] = linear\_prediction(img);

[lowFreq, highFreq, positionZeros, positionNegatives] = integer\_wavelet\_transform(predictionErrors);

compressedData(idx) = forced\_huffman\_encoding(lowFreq, highFreq, positionZeros, positionNegatives, idx);

decompressed\_img = decompress(compressedData(idx), size(img));

compressed\_size\_bits = numel(compressedData(idx).compressedStream) \* 8;

CR = (numel(img) \* 8) / compressed\_size\_bits;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [predictedImage, predictionErrors] = linear\_prediction(inputImage)

[rows, cols] = size(inputImage);

predictedImage = zeros(rows, cols);

predictionErrors = zeros(rows, cols);

for i = 2:rows-1

for j = 2:cols-1

predictedValue = round((inputImage(i-1, j-1) + inputImage(i-1, j) + ...

inputImage(i-1, j+1) + inputImage(i, j-1) + ...

inputImage(i, j+1)) / 5);

predictedImage(i, j) = predictedValue;

predictionErrors(i, j) = double(inputImage(i, j)) - predictedValue;

end

end

end

function [lowFreq, highFreq, positionZeros, positionNegatives] = integer\_wavelet\_transform(predictionErrors)

[rows, cols] = size(predictionErrors);

numLevels = 6;

lowFreq = predictionErrors;

highFreq = zeros(size(predictionErrors));

for level = 1:numLevels

[lowFreq, highFreq] = legall53\_transform(lowFreq);

end

positionZeros = find(lowFreq == 0);

positionNegatives = find(lowFreq < 0);

lowFreq(lowFreq < 0) = abs(lowFreq(lowFreq < 0));

end

function [lowFreq, highFreq] = legall53\_transform(data)

[rows, cols] = size(data);

rowTransformed = zeros(size(data));

for r = 1:rows

[low, high] = legall53\_1d(data(r, :));

rowTransformed(r, :) = [low, high];

end

colTransformed = zeros(size(rowTransformed));

for c = 1:cols

[low, high] = legall53\_1d(rowTransformed(:, c)');

colTransformed(:, c) = [low, high]';

end

halfRows = floor(rows / 2);

halfCols = floor(cols / 2);

lowFreq = colTransformed(1:halfRows, 1:halfCols);

highFreq = colTransformed(halfRows+1:end, halfCols+1:end);

end

function [low, high] = legall53\_1d(data)

n = length(data);

low = zeros(1, ceil(n / 2));

high = zeros(1, floor(n / 2));

for i = 1:length(high)

high(i) = data(2 \* i) - floor((data(max(2 \* i - 1, 1)) + data(min(2 \* i + 1, n))) / 2);

end

for i = 1:length(low)

low(i) = data(2 \* i - 1) + floor((high(max(i - 1, 1)) + high(min(i + 1, length(high)))) / 4);

end

end

function compressedData = forced\_huffman\_encoding(lowFreq, highFreq, positionZeros, positionNegatives, idx)

dataStream = [lowFreq(:); highFreq(:); zeros(1000, 1)];

allSymbols = unique(dataStream);

minVal = min(allSymbols);

maxVal = max(allSymbols);

allSymbols = minVal:maxVal;

allProbabilities = histcounts(dataStream, [allSymbols, max(allSymbols)+1], 'Normalization', 'probability');

if sum(allProbabilities) == 0

allProbabilities(:) = 1 / numel(allProbabilities);

else

allProbabilities = allProbabilities / sum(allProbabilities);

end

dict = huffmandict(allSymbols, allProbabilities);

dictSymbols = [dict{:,1}];

missingSymbols = setdiff(dataStream, dictSymbols);

if ~isempty(missingSymbols)

warning('Huffman dictionary missing symbols: %s', mat2str(missingSymbols));

extraSymbols = missingSymbols(:);

extraProbabilities = repmat(1e-6, size(extraSymbols));

dict = huffmandict([dictSymbols, extraSymbols], [allProbabilities, extraProbabilities]);

end

paddingValue = min(allSymbols);

paddedStream = [dataStream; paddingValue \* ones(5000, 1)];

compressedStream = huffmanenco(paddedStream, dict);

compressedData = struct('compressedStream', {compressedStream}, ...

'dict', {dict}, ...

'positionZeros', {positionZeros}, ...

'positionNegatives', {positionNegatives}, ...

'negValueStorage', {dataStream(positionNegatives)});

end

function decompressedImage = decompress(singleCompressedData, originalSize)

decodedStream = huffmandeco(singleCompressedData.compressedStream, singleCompressedData.dict);

lowFreq = zeros(originalSize);

highFreq = zeros(originalSize);

lowFreq(singleCompressedData.positionZeros) = 0;

if numel(decodedStream) >= numel(singleCompressedData.positionNegatives)

lowFreq(singleCompressedData.positionNegatives) = -singleCompressedData.negValueStorage;

else

error('Mismatch: Decoded stream size is smaller than positionNegatives.');

end

decompressedImage = lowFreq + highFreq;

decompressedImage = uint8(decompressedImage);

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = double(img);

tic;

[~, predErrors] = linear\_prediction(img);

encoded = predErrors(:) + 128;

encoded(encoded > 255) = 255;

encoded(encoded < 0) = 0;

symbols = uint8(encoded);

u = unique(symbols);

dict = huffmandict(u, ones(length(u), 1) / length(u));

stream = huffmanenco(symbols, dict);

decoded = huffmandeco(stream, dict);

decoded = double(decoded) - 128;

recon = reshape(decoded, size(img));

for i = 1:size(recon,1)

for j = 2:size(recon,2)

recon(i,j) = recon(i,j) + recon(i,j-1);

end

end

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch ME

fprintf('Error at image %d: %s\n', idx, ME.message);

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('Improved Lossless Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function [predictedImage, predictionErrors] = linear\_prediction(img)

img = double(img);

[rows, cols] = size(img);

predictedImage = zeros(rows, cols);

predictionErrors = zeros(rows, cols);

for i = 1:rows

for j = 1:cols

if j == 1

predicted = 0;

else

predicted = img(i, j-1);

end

predictedImage(i, j) = predicted;

predictionErrors(i, j) = img(i, j) - predicted;

end

end

end

function [transformed, posZeros, posNegatives] = integer\_wavelet\_transform(errors)

transformed = errors;

for l = 1:6

transformed = legall53\_transform(transformed);

end

posZeros = (transformed == 0);

posNegatives = (transformed < 0);

end

function out = legall53\_transform(img)

[rows, cols] = size(img);

out = zeros(rows, cols);

for i = 1:rows

out(i,:) = legall53\_1d(img(i,:));

end

for j = 1:cols

out(:,j) = legall53\_1d(out(:,j)')';

end

end

function y = legall53\_1d(x)

N = length(x);

if mod(N,2) ~= 0

x(end+1) = x(end);

N = N + 1;

end

odd = x(1:2:end);

even = x(2:2:end);

if length(odd) < 2

s = odd;

d = zeros(size(even));

else

d = even - floor((odd(1:end-1) + odd(2:end))/2);

d(end+1) = d(end);

s = odd + floor(([d(1), d(1:end-1)] + d)/4);

end

y = zeros(1, N);

y(1:2:end) = s(1:min(end,length(1:2:end)));

y(2:2:end) = d(1:min(end,length(2:2:end)));

end

function compressedData = forced\_huffman\_encoding(transformed, posZeros, posNegatives)

data = transformed(:) + 128;

data(data > 255) = 255;

data(data < 0) = 0;

symbols = uint8(data);

u = unique(symbols);

dict = huffmandict(u, ones(length(u), 1) / length(u));

compressedStream = huffmanenco(symbols, dict);

compressedData.stream = compressedStream;

compressedData.dict = dict;

compressedData.size = size(transformed);

compressedData.posZeros = posZeros;

compressedData.posNegatives = posNegatives;

end

function decompressed\_img = decompress(compressedData)

decoded = huffmandeco(compressedData.stream, compressedData.dict);

decoded = double(decoded) - 128;

expected\_len = prod(compressedData.size);

if length(decoded) < expected\_len

decoded(end+1:expected\_len) = 0;

elseif length(decoded) > expected\_len

decoded = decoded(1:expected\_len);

end

decoded = reshape(decoded, compressedData.size);

decoded(compressedData.posZeros) = 0;

decoded(compressedData.posNegatives) = -abs(decoded(compressedData.posNegatives));

decompressed\_img = decoded;

end

**A.2.6 Novel High-Frequency Encoding**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

blockSize = 16;

quantizationFactor = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[compressedData, lookupTable] = compressImage(img, blockSize, quantizationFactor);

decompressed\_img = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, size(img));

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = (numel(compressedData.minimizedArray) + numel(compressedData.diffDC)) \* 32;

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [compressedData, lookupTable] = compressImage(inputImage, blockSize, quantizationFactor)

[rows, cols] = size(inputImage);

inputImage = double(inputImage);

dctBlocks = cell(floor(rows/blockSize), floor(cols/blockSize));

for i = 1:blockSize:rows-blockSize+1

for j = 1:blockSize:cols-blockSize+1

block = inputImage(i:i+blockSize-1, j:j+blockSize-1);

dctBlocks{(i-1)/blockSize+1, (j-1)/blockSize+1} = dct2(block);

end

end

quantizedBlocks = cellfun(@(block) round(block ./ quantizationFactor), dctBlocks, 'UniformOutput', false);

DCArray = cellfun(@(block) block(1, 1), quantizedBlocks);

ACMatrix = cell2mat(cellfun(@(block) block(2:end, :), quantizedBlocks, 'UniformOutput', false));

reducedArray = ACMatrix(ACMatrix ~= 0);

[minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray);

diffDC = diff(DCArray);

minimizedArrayEncoded = arithmeticEncode(minimizedArray);

diffDCEncoded = arithmeticEncode(diffDC);

compressedData.minimizedArray = minimizedArrayEncoded;

compressedData.diffDC = diffDCEncoded;

end

function [minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray)

M = 1.5 \* max(reducedArray);

K1 = rand();

K2 = K1 + M + 1;

K3 = M \* (K1 + K2);

lookupTable = struct('K1', K1, 'K2', K2, 'K3', K3);

minimizedArray = arrayfun(@(i) K1 \* reducedArray(i) + K2 \* reducedArray(i+1) + K3 \* reducedArray(i+2), 1:3:length(reducedArray)-2);

end

function decompressedImage = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, imgSize)

K1 = lookupTable.K1;

K2 = lookupTable.K2;

K3 = lookupTable.K3;

minimizedArray = arithmeticDecode(compressedData.minimizedArray);

reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3);

numBlocks = floor(imgSize(1) / blockSize) \* floor(imgSize(2) / blockSize);

rowsAC = blockSize - 1;

colsAC = blockSize;

totalACRows = numBlocks \* rowsAC;

ACMatrix = zeros(totalACRows, colsAC);

totalElements = min(length(reducedArray), numel(ACMatrix));

ACMatrix(1:totalElements) = reducedArray(1:totalElements);

diffDC = arithmeticDecode(compressedData.diffDC);

DCArray = cumsum(diffDC);

decompressedImage = zeros(imgSize);

blockIndex = 1;

for i = 1:blockSize:imgSize(1)-blockSize+1

for j = 1:blockSize:imgSize(2)-blockSize+1

if blockIndex > length(DCArray)

break;

end

block = zeros(blockSize, blockSize);

block(1, 1) = DCArray(blockIndex);

startRow = (blockIndex-1) \* rowsAC + 1;

endRow = startRow + rowsAC - 1;

if endRow <= size(ACMatrix, 1)

block(2:end, :) = reshape(ACMatrix(startRow:endRow, :), rowsAC, colsAC);

end

decompressedImage(i:i+blockSize-1, j:j+blockSize-1) = idct2(block \* quantizationFactor);

blockIndex = blockIndex + 1;

end

end

end

function encodedArray = arithmeticEncode(inputArray)

encodedArray = inputArray;

end

function decodedArray = arithmeticDecode(encodedArray)

decodedArray = encodedArray;

end

function reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3)

reducedArray = zeros(1, length(minimizedArray) \* 3);

for i = 1:length(minimizedArray)

reducedArray((i-1)\*3+1) = (minimizedArray(i) - K2 - K3) / K1;

end

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

blockSize = 16;

quantizationFactor = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[compressedData, lookupTable] = compressImage(img, blockSize, quantizationFactor);

decompressed\_img = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, size(img));

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

compressed\_size\_bits = (numel(compressedData.minimizedArray) + numel(compressedData.diffDC)) \* 32;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [compressedData, lookupTable] = compressImage(inputImage, blockSize, quantizationFactor)

[rows, cols] = size(inputImage);

inputImage = double(inputImage);

dctBlocks = cell(floor(rows/blockSize), floor(cols/blockSize));

for i = 1:blockSize:rows-blockSize+1

for j = 1:blockSize:cols-blockSize+1

block = inputImage(i:i+blockSize-1, j:j+blockSize-1);

dctBlocks{(i-1)/blockSize+1, (j-1)/blockSize+1} = dct2(block);

end

end

quantizedBlocks = cellfun(@(block) round(block ./ quantizationFactor), dctBlocks, 'UniformOutput', false);

DCArray = cellfun(@(block) block(1, 1), quantizedBlocks);

ACMatrix = cell2mat(cellfun(@(block) block(2:end, :), quantizedBlocks, 'UniformOutput', false));

reducedArray = ACMatrix(ACMatrix ~= 0);

[minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray);

diffDC = diff(DCArray);

minimizedArrayEncoded = arithmeticEncode(minimizedArray);

diffDCEncoded = arithmeticEncode(diffDC);

compressedData.minimizedArray = minimizedArrayEncoded;

compressedData.diffDC = diffDCEncoded;

end

function [minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray)

M = 1.5 \* max(reducedArray);

K1 = rand();

K2 = K1 + M + 1;

K3 = M \* (K1 + K2);

lookupTable = struct('K1', K1, 'K2', K2, 'K3', K3);

minimizedArray = arrayfun(@(i) K1 \* reducedArray(i) + K2 \* reducedArray(i+1) + K3 \* reducedArray(i+2), 1:3:length(reducedArray)-2);

end

function decompressedImage = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, imgSize)

K1 = lookupTable.K1;

K2 = lookupTable.K2;

K3 = lookupTable.K3;

minimizedArray = arithmeticDecode(compressedData.minimizedArray);

reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3);

numBlocks = floor(imgSize(1) / blockSize) \* floor(imgSize(2) / blockSize);

rowsAC = blockSize - 1;

colsAC = blockSize;

totalACRows = numBlocks \* rowsAC;

ACMatrix = zeros(totalACRows, colsAC);

totalElements = min(length(reducedArray), numel(ACMatrix));

ACMatrix(1:totalElements) = reducedArray(1:totalElements);

diffDC = arithmeticDecode(compressedData.diffDC);

DCArray = cumsum(diffDC);

decompressedImage = zeros(imgSize);

blockIndex = 1;

for i = 1:blockSize:imgSize(1)-blockSize+1

for j = 1:blockSize:imgSize(2)-blockSize+1

if blockIndex > length(DCArray)

break;

end

block = zeros(blockSize, blockSize);

block(1, 1) = DCArray(blockIndex);

startRow = (blockIndex-1) \* rowsAC + 1;

endRow = startRow + rowsAC - 1;

if endRow <= size(ACMatrix, 1)

block(2:end, :) = reshape(ACMatrix(startRow:endRow, :), rowsAC, colsAC);

end

decompressedImage(i:i+blockSize-1, j:j+blockSize-1) = idct2(block \* quantizationFactor);

blockIndex = blockIndex + 1;

end

end

end

function encodedArray = arithmeticEncode(inputArray)

encodedArray = inputArray;

end

function decodedArray = arithmeticDecode(encodedArray)

decodedArray = encodedArray;

end

function reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3)

reducedArray = zeros(1, length(minimizedArray) \* 3);

for i = 1:length(minimizedArray)

reducedArray((i-1)\*3+1) = (minimizedArray(i) - K2 - K3) / K1;

end

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

blockSize = 16;

quantizationFactor = 10;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

try

tic;

[compressedData, lookupTable] = compressImage(img, blockSize, quantizationFactor);

decompressed\_img = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, size(img));

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch ME

fprintf('Error at image %d: %s\n', idx, ME.message);

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('Novel High-Frequency Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function [compressedData, lookupTable] = compressImage(inputImage, blockSize, quantizationFactor)

[rows, cols] = size(inputImage);

inputImage = double(inputImage);

dctBlocks = cell(floor(rows/blockSize), floor(cols/blockSize));

for i = 1:blockSize:rows-blockSize+1

for j = 1:blockSize:cols-blockSize+1

block = inputImage(i:i+blockSize-1, j:j+blockSize-1);

dctBlocks{(i-1)/blockSize+1, (j-1)/blockSize+1} = dct2(block);

end

end

quantizedBlocks = cellfun(@(block) round(block ./ quantizationFactor), dctBlocks, 'UniformOutput', false);

DCArray = cellfun(@(block) block(1, 1), quantizedBlocks);

ACMatrix = cell2mat(cellfun(@(block) block(2:end, :), quantizedBlocks, 'UniformOutput', false));

reducedArray = ACMatrix(ACMatrix ~= 0);

[minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray);

diffDC = diff([0 DCArray(:)']);

minimizedArrayEncoded = arithmeticEncode(minimizedArray);

diffDCEncoded = arithmeticEncode(diffDC);

compressedData.minimizedArray = minimizedArrayEncoded;

compressedData.diffDC = diffDCEncoded;

end

function [minimizedArray, lookupTable] = encodeHighFrequencies(reducedArray)

M = 1.5 \* max(reducedArray);

K1 = rand();

K2 = K1 + M + 1;

K3 = M \* (K1 + K2);

lookupTable = struct('K1', K1, 'K2', K2, 'K3', K3);

minimizedArray = arrayfun(@(i) K1 \* reducedArray(i) + K2 \* reducedArray(i+1) + K3 \* reducedArray(i+2), 1:3:length(reducedArray)-2);

end

function decompressedImage = decompressImage(compressedData, lookupTable, blockSize, quantizationFactor, imgSize)

K1 = lookupTable.K1;

K2 = lookupTable.K2;

K3 = lookupTable.K3;

minimizedArray = arithmeticDecode(compressedData.minimizedArray);

reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3);

numBlocks = floor(imgSize(1) / blockSize) \* floor(imgSize(2) / blockSize);

rowsAC = blockSize - 1;

colsAC = blockSize;

totalACRows = numBlocks \* rowsAC;

ACMatrix = zeros(totalACRows, colsAC);

totalElements = min(length(reducedArray), numel(ACMatrix));

ACMatrix(1:totalElements) = reducedArray(1:totalElements);

diffDC = arithmeticDecode(compressedData.diffDC);

DCArray = cumsum(diffDC);

decompressedImage = zeros(imgSize);

blockIndex = 1;

for i = 1:blockSize:imgSize(1)-blockSize+1

for j = 1:blockSize:imgSize(2)-blockSize+1

if blockIndex > length(DCArray)

break;

end

block = zeros(blockSize, blockSize);

block(1, 1) = DCArray(blockIndex);

startRow = (blockIndex-1) \* rowsAC + 1;

endRow = startRow + rowsAC - 1;

if endRow <= size(ACMatrix, 1)

block(2:end, :) = reshape(ACMatrix(startRow:endRow, :), rowsAC, colsAC);

end

decompressedImage(i:i+blockSize-1, j:j+blockSize-1) = idct2(block \* quantizationFactor);

blockIndex = blockIndex + 1;

end

end

end

function encodedArray = arithmeticEncode(inputArray)

encodedArray = inputArray;

end

function decodedArray = arithmeticDecode(encodedArray)

decodedArray = encodedArray;

end

function reducedArray = binarySearchReconstruction(minimizedArray, K1, K2, K3)

reducedArray = zeros(1, length(minimizedArray) \* 3);

for i = 1:length(minimizedArray)

reducedArray((i-1)\*3+1) = (minimizedArray(i) - K2 - K3) / K1;

end

end

**A.2.7 3D DCT-based compression**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

blockSize = 8;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[reconstructed\_img, encodedData, Q] = dct3d\_compression(img, blockSize);

reconstructed\_img = guided\_deblocking(reconstructed\_img);

if size(reconstructed\_img) ~= size(img)

reconstructed\_img = imresize(reconstructed\_img, size(img), 'nearest');

end

profile = double(reconstructed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(reconstructed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(reconstructed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(reconstructed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(encodedData);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [reconstructed\_img, encodedData, Q] = dct3d\_compression(img, blockSize)

[height, width] = size(img);

paddedHeight = ceil(height / blockSize) \* blockSize;

paddedWidth = ceil(width / blockSize) \* blockSize;

paddedImg = zeros(paddedHeight, paddedWidth);

paddedImg(1:height, 1:width) = img;

numBlocksX = paddedWidth / blockSize;

numBlocksY = paddedHeight / blockSize;

blocks3D = zeros(4, 4, 4, numBlocksX, numBlocksY);

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = paddedImg((by-1)\*blockSize+1:by\*blockSize, (bx-1)\*blockSize+1:bx\*blockSize);

R = block(1:2:end, 1:2:end);

G1 = block(1:2:end, 2:2:end);

G2 = block(2:2:end, 1:2:end);

B = block(2:2:end, 2:2:end);

blocks3D(:, :, 1, bx, by) = R;

blocks3D(:, :, 2, bx, by) = G1;

blocks3D(:, :, 3, bx, by) = G2;

blocks3D(:, :, 4, bx, by) = B;

end

end

dctBlocks = zeros(size(blocks3D));

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = squeeze(blocks3D(:, :, :, bx, by));

dctBlocks(:, :, :, bx, by) = butterfly\_dct3(block);

end

end

Q = ga\_optimize\_quantization(dctBlocks);

quantizedBlocks = round(dctBlocks ./ Q);

zigzagIndices = [

1, 2, 6, 7, 15, 16, 28, 29, ...

3, 5, 8, 14, 17, 27, 30, 43, ...

4, 9, 13, 18, 26, 31, 42, 44, ...

10, 12, 19, 25, 32, 41, 45, 54, ...

11, 20, 24, 33, 40, 46, 53, 55, ...

21, 23, 34, 39, 47, 52, 56, 60, ...

22, 35, 38, 48, 51, 57, 59, 61, ...

36, 37, 49, 50, 58, 62, 63, 64];

encodedData = [];

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = quantizedBlocks(:, :, :, bx, by);

reshapedBlock = reshape(block, [64, 1]);

zigzagged = reshapedBlock(zigzagIndices);

encodedData = [encodedData; zigzagged(:)];

end

end

decodedBlocks = quantizedBlocks .\* Q;

reconstructed\_img = zeros(paddedHeight, paddedWidth);

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = squeeze(decodedBlocks(:, :, :, bx, by));

spatialBlock = zeros(blockSize, blockSize);

spatialBlock(1:2:end, 1:2:end) = block(:, :, 1);

spatialBlock(1:2:end, 2:2:end) = block(:, :, 2);

spatialBlock(2:2:end, 1:2:end) = block(:, :, 3);

spatialBlock(2:2:end, 2:2:end) = block(:, :, 4);

reconstructed\_img((by-1)\*blockSize+1:by\*blockSize, (bx-1)\*blockSize+1:bx\*blockSize) = spatialBlock;

end

end

reconstructed\_img = reconstructed\_img(1:height, 1:width);

end

function D = butterfly\_dct3(block)

[N, ~, ~] = size(block);

C = dctmtx(N);

D = zeros(N, N, N);

for i = 1:N

for j = 1:N

D(:, i, j) = C \* squeeze(block(:, i, j));

end

end

for i = 1:N

for j = 1:N

D(i, :, j) = (C \* squeeze(D(i, :, j))')';

end

end

for i = 1:N

for j = 1:N

D(i, j, :) = C \* squeeze(D(i, j, :));

end

end

end

function Q = ga\_optimize\_quantization(dctBlocks)

numGenerations = 50;

populationSize = 20;

mutationRate = 0.1;

crossoverRate = 0.8;

lambda = 0.5;

population = initialize\_population(populationSize);

for gen = 1:numGenerations

fitness = evaluate\_population(population, dctBlocks, lambda);

parents = select\_parents(population, fitness);

offspring = perform\_crossover(parents, crossoverRate);

offspring = perform\_mutation(offspring, mutationRate);

population = offspring;

end

[~, bestIndex] = max(evaluate\_population(population, dctBlocks, lambda));

Q = reshape(population(bestIndex, :), [4, 4, 4]);

end

function population = initialize\_population(populationSize)

population = randi([8, 64], populationSize, 4\*4\*4);

end

function fitness = evaluate\_population(population, dctBlocks, lambda)

[blockDim1, blockDim2, blockDim3, numBlocksX, numBlocksY] = size(dctBlocks);

numIndividuals = size(population, 1);

fitness = zeros(numIndividuals, 1);

for i = 1:numIndividuals

Q = reshape(population(i, :), [blockDim1, blockDim2, blockDim3]);

expandedQ = repmat(Q, [1, 1, 1, numBlocksX, numBlocksY]);

quantized = round(dctBlocks ./ expandedQ);

distortion = mean((dctBlocks(:) - (quantized(:) .\* expandedQ(:))).^2);

rate = nnz(quantized);

fitness(i) = -(lambda \* distortion + (1 - lambda) \* rate);

end

end

function parents = select\_parents(population, fitness)

numParents = size(population, 1);

fitness = fitness - min(fitness) + 1;

probabilities = fitness / sum(fitness);

cumulativeProb = cumsum(probabilities);

parents = zeros(size(population));

for i = 1:numParents

r = rand;

selected = find(cumulativeProb >= r, 1, 'first');

parents(i, :) = population(selected, :);

end

end

function offspring = perform\_crossover(parents, crossoverRate)

numParents = size(parents, 1);

offspring = parents;

for i = 1:2:numParents-1

if rand < crossoverRate

crossoverPoint = randi(size(parents, 2) - 1);

offspring(i, crossoverPoint+1:end) = parents(i+1, crossoverPoint+1:end);

offspring(i+1, crossoverPoint+1:end) = parents(i, crossoverPoint+1:end);

end

end

end

function mutated = perform\_mutation(offspring, mutationRate)

numIndividuals = size(offspring, 1);

numGenes = size(offspring, 2);

mutated = offspring;

for i = 1:numIndividuals

for j = 1:numGenes

if rand < mutationRate

mutated(i, j) = randi([8, 64]);

end

end

end

end

function img = guided\_deblocking(img)

thresholdSMSDS = 5;

lambda = 0.1;

[height, width] = size(img);

deblockedImg = img;

blockSize = 8;

for x = blockSize:blockSize:width-1

for y = blockSize:blockSize:height-1

boundaryRegion = extract\_boundary\_region(img, x, y, blockSize);

smsds = compute\_smsds(boundaryRegion);

if smsds > thresholdSMSDS

continue;

end

deblockedRegion = apply\_frequency\_filter(boundaryRegion, lambda);

deblockedImg = update\_boundary\_region(deblockedImg, deblockedRegion, x, y, blockSize);

end

end

img = deblockedImg;

end

function boundaryRegion = extract\_boundary\_region(img, x, y, blockSize)

[height, width] = size(img);

xStart = max(1, x-blockSize+1);

xEnd = min(width, x+blockSize);

yStart = max(1, y-blockSize+1);

yEnd = min(height, y+blockSize);

boundaryRegion = img(yStart:yEnd, xStart:xEnd);

end

function smsds = compute\_smsds(boundaryRegion)

freqCoeffs = fft2(boundaryRegion);

freqCoeffs = abs(freqCoeffs).^2;

smsds = sum(abs(diff(freqCoeffs(:))));

end

function filteredRegion = apply\_frequency\_filter(boundaryRegion, lambda)

freqCoeffs = fft2(boundaryRegion);

[height, width] = size(boundaryRegion);

[u, v] = meshgrid(1:width, 1:height);

ridgeFilter = 1 ./ (1 + lambda \* (u.^2 + v.^2));

filteredFreqCoeffs = freqCoeffs .\* ridgeFilter;

filteredRegion = real(ifft2(filteredFreqCoeffs));

end

function img = update\_boundary\_region(img, deblockedRegion, x, y, blockSize)

[height, width] = size(img);

xStart = max(1, x-blockSize+1);

xEnd = min(width, x+blockSize);

yStart = max(1, y-blockSize+1);

yEnd = min(height, y+blockSize);

deblockedXSize = xEnd - xStart + 1;

deblockedYSize = yEnd - yStart + 1;

deblockedRegion = deblockedRegion(1:deblockedYSize, 1:deblockedXSize);

img(yStart:yEnd, xStart:xEnd) = deblockedRegion;

end

function D = dct3(block)

D = dct(dct(dct(block, [], 1), [], 2), [], 3);

end

function block = idct3(D)

block = idct(idct(idct(D, [], 1), [], 2), [], 3);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

blockSize = 8;

for idx = 1:num\_images

rng(0);

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[reconstructed\_img, encodedData, Q] = dct3d\_compression(img, blockSize);

reconstructed\_img = guided\_deblocking(reconstructed\_img);

if size(reconstructed\_img) ~= size(img)

reconstructed\_img = imresize(reconstructed\_img, size(img), 'nearest');

end

compressed\_size\_bits = numel(encodedData) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [reconstructed\_img, encodedData, Q] = dct3d\_compression(img, blockSize)

[height, width] = size(img);

paddedHeight = ceil(height / blockSize) \* blockSize;

paddedWidth = ceil(width / blockSize) \* blockSize;

paddedImg = zeros(paddedHeight, paddedWidth);

paddedImg(1:height, 1:width) = img;

numBlocksX = paddedWidth / blockSize;

numBlocksY = paddedHeight / blockSize;

blocks3D = zeros(4, 4, 4, numBlocksX, numBlocksY);

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = paddedImg((by-1)\*blockSize+1:by\*blockSize, (bx-1)\*blockSize+1:bx\*blockSize);

R = block(1:2:end, 1:2:end);

G1 = block(1:2:end, 2:2:end);

G2 = block(2:2:end, 1:2:end);

B = block(2:2:end, 2:2:end);

blocks3D(:, :, 1, bx, by) = R;

blocks3D(:, :, 2, bx, by) = G1;

blocks3D(:, :, 3, bx, by) = G2;

blocks3D(:, :, 4, bx, by) = B;

end

end

dctBlocks = zeros(size(blocks3D));

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = squeeze(blocks3D(:, :, :, bx, by));

dctBlocks(:, :, :, bx, by) = butterfly\_dct3(block);

end

end

Q = ga\_optimize\_quantization(dctBlocks);

Q = Q \* 2;

quantizedBlocks = round(dctBlocks ./ Q);

encodedData = [];

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = quantizedBlocks(:, :, :, bx, by);

reshapedBlock = reshape(block, [64, 1]);

encodedData = [encodedData; reshapedBlock(:)];

end

end

decodedBlocks = quantizedBlocks .\* Q;

reconstructed\_img = zeros(paddedHeight, paddedWidth);

for bx = 1:numBlocksX

for by = 1:numBlocksY

block = squeeze(decodedBlocks(:, :, :, bx, by));

reconstructed\_block = imresize(block(:, :, 1), [blockSize, blockSize], 'nearest');

reconstructed\_img((by-1)\*blockSize+1:by\*blockSize, (bx-1)\*blockSize+1:bx\*blockSize) = reconstructed\_block;

end

end

reconstructed\_img = reconstructed\_img(1:height, 1:width);

end

function D = butterfly\_dct3(block)

[N, ~, ~] = size(block);

C = dctmtx(N);

D = zeros(N, N, N);

for i = 1:N

for j = 1:N

D(:, i, j) = C \* squeeze(block(:, i, j));

end

end

for i = 1:N

for j = 1:N

D(i, :, j) = (C \* squeeze(D(i, :, j))')';

end

end

for i = 1:N

for j = 1:N

D(i, j, :) = C \* squeeze(D(i, j, :));

end

end

end

function Q = ga\_optimize\_quantization(dctBlocks)

numGenerations = 50;

populationSize = 20;

mutationRate = 0.1;

crossoverRate = 0.8;

lambda = 0.5;

population = initialize\_population(populationSize);

for gen = 1:numGenerations

fitness = evaluate\_population(population, dctBlocks, lambda);

parents = select\_parents(population, fitness);

offspring = perform\_crossover(parents, crossoverRate);

offspring = perform\_mutation(offspring, mutationRate);

population = offspring;

end

[~, bestIndex] = max(evaluate\_population(population, dctBlocks, lambda));

Q = reshape(population(bestIndex, :), [4, 4, 4]);

end

function population = initialize\_population(populationSize)

population = randi([8, 64], populationSize, 4\*4\*4);

end

function fitness = evaluate\_population(population, dctBlocks, lambda)

[blockDim1, blockDim2, blockDim3, numBlocksX, numBlocksY] = size(dctBlocks);

numIndividuals = size(population, 1);

fitness = zeros(numIndividuals, 1);

for i = 1:numIndividuals

Q = reshape(population(i, :), [blockDim1, blockDim2, blockDim3]);

expandedQ = repmat(Q, [1, 1, 1, numBlocksX, numBlocksY]);

quantized = round(dctBlocks ./ expandedQ);

distortion = mean((dctBlocks(:) - (quantized(:) .\* expandedQ(:))).^2);

rate = nnz(quantized);

fitness(i) = -(lambda \* distortion + (1 - lambda) \* rate);

end

end

function parents = select\_parents(population, fitness)

numParents = size(population, 1);

fitness = fitness - min(fitness) + 1;

probabilities = fitness / sum(fitness);

cumulativeProb = cumsum(probabilities);

parents = zeros(size(population));

for i = 1:numParents

r = rand;

selected = find(cumulativeProb >= r, 1, 'first');

parents(i, :) = population(selected, :);

end

end

function offspring = perform\_crossover(parents, crossoverRate)

numParents = size(parents, 1);

offspring = parents;

for i = 1:2:numParents-1

if rand < crossoverRate

crossoverPoint = randi(size(parents, 2) - 1);

offspring(i, crossoverPoint+1:end) = parents(i+1, crossoverPoint+1:end);

offspring(i+1, crossoverPoint+1:end) = parents(i, crossoverPoint+1:end);

end

end

end

function mutated = perform\_mutation(offspring, mutationRate)

numIndividuals = size(offspring, 1);

numGenes = size(offspring, 2);

mutated = offspring;

for i = 1:numIndividuals

for j = 1:numGenes

if rand < mutationRate

mutated(i, j) = randi([8, 64]);

end

end

end

end

function img = guided\_deblocking(img)

thresholdSMSDS = 5;

lambda = 0.1;

[height, width] = size(img);

deblockedImg = img;

blockSize = 8;

for x = blockSize:blockSize:width-1

for y = blockSize:blockSize:height-1

boundaryRegion = extract\_boundary\_region(img, x, y, blockSize);

smsds = compute\_smsds(boundaryRegion);

if smsds > thresholdSMSDS

continue;

end

deblockedRegion = apply\_frequency\_filter(boundaryRegion, lambda);

deblockedImg = update\_boundary\_region(deblockedImg, deblockedRegion, x, y, blockSize);

end

end

img = deblockedImg;

end

function boundaryRegion = extract\_boundary\_region(img, x, y, blockSize)

[height, width] = size(img);

xStart = max(1, x-blockSize+1);

xEnd = min(width, x+blockSize);

yStart = max(1, y-blockSize+1);

yEnd = min(height, y+blockSize);

boundaryRegion = img(yStart:yEnd, xStart:xEnd);

end

function smsds = compute\_smsds(boundaryRegion)

freqCoeffs = fft2(boundaryRegion);

freqCoeffs = abs(freqCoeffs).^2;

smsds = sum(abs(diff(freqCoeffs(:))));

end

function filteredRegion = apply\_frequency\_filter(boundaryRegion, lambda)

freqCoeffs = fft2(boundaryRegion);

[height, width] = size(boundaryRegion);

[u, v] = meshgrid(1:width, 1:height);

ridgeFilter = 1 ./ (1 + lambda \* (u.^2 + v.^2));

filteredFreqCoeffs = freqCoeffs .\* ridgeFilter;

filteredRegion = real(ifft2(filteredFreqCoeffs));

end

function img = update\_boundary\_region(img, deblockedRegion, x, y, blockSize)

[height, width] = size(img);

xStart = max(1, x-blockSize+1);

xEnd = min(width, x+blockSize);

yStart = max(1, y-blockSize+1);

yEnd = min(height, y+blockSize);

deblockedXSize = xEnd - xStart + 1;

deblockedYSize = yEnd - yStart + 1;

deblockedRegion = deblockedRegion(1:deblockedYSize, 1:deblockedXSize);

img(yStart:yEnd, xStart:xEnd) = deblockedRegion;

end

function D = dct3(block)

D = dct(dct(dct(block, [], 1), [], 2), [], 3);

end

function block = idct3(D)

block = idct(idct(idct(D, [], 1), [], 2), [], 3);

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

tic;

compressed = compress\_and\_reconstruct(img);

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('3D DCT-Based Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function reconstructed = compress\_and\_reconstruct(img)

img = double(imresize(img, [256 256]));

img\_3d = repmat(img, [1, 1, 3]);

dct3d = butterfly\_dct3(img\_3d);

quant\_params = ga\_optimize\_quantization(dct3d);

quantized = round(dct3d ./ quant\_params);

filtered = guided\_deblocking(quantized);

dequantized = filtered .\* quant\_params;

reconstructed = idct3(dequantized);

reconstructed = reconstructed(:, :, 1);

end

function dct\_out = butterfly\_dct3(img3d)

[h, w, d] = size(img3d);

dct\_out = zeros(h, w, d);

for i = 1:h

for j = 1:w

dct\_out(i,j,:) = dct(squeeze(img3d(i,j,:)));

end

end

for i = 1:h

for k = 1:d

dct\_out(i,:,k) = dct(dct\_out(i,:,k));

end

end

for j = 1:w

for k = 1:d

dct\_out(:,j,k) = dct(dct\_out(:,j,k));

end

end

end

function optimized = ga\_optimize\_quantization(dct\_coeffs)

population = initialize\_population(10, size(dct\_coeffs));

for gen = 1:5

fitness = evaluate\_population(population, dct\_coeffs);

parents = select\_parents(population, fitness);

offspring = perform\_crossover(parents);

population = perform\_mutation(offspring);

end

optimized = population{1};

end

function population = initialize\_population(n, size3d)

population = cell(1, n);

for i = 1:n

population{i} = rand(size3d);

end

end

function scores = evaluate\_population(population, original)

scores = zeros(1, numel(population));

for i = 1:numel(population)

diff = abs(original - population{i});

scores(i) = sum(diff(:));

end

end

function selected = select\_parents(population, fitness)

[~, idx] = sort(fitness);

selected = population(idx(1:2));

end

function offspring = perform\_crossover(parents)

offspring = cell(1, 2);

[h, w, d] = size(parents{1});

cross\_point = round([h, w, d]/2);

child1 = parents{1};

child2 = parents{2};

child1(1:cross\_point(1), :, :) = parents{2}(1:cross\_point(1), :, :);

child2(:, 1:cross\_point(2), :) = parents{1}(:, 1:cross\_point(2), :);

offspring{1} = child1;

offspring{2} = child2;

end

function mutated = perform\_mutation(offspring)

mutated = cell(size(offspring));

for i = 1:numel(offspring)

noise = randn(size(offspring{i})) \* 0.01;

mutated{i} = offspring{i} + noise;

end

end

function deblocked = guided\_deblocking(quantized)

mask = extract\_boundary(quantized);

smoothed = apply\_frequency\_filter(quantized);

deblocked = update\_boundary(smoothed, mask);

end

function mask = extract\_boundary(img3d)

grad\_x = diff(img3d, 1, 2);

grad\_y = diff(img3d, 1, 1);

grad\_z = diff(img3d, 1, 3);

mask = abs(padarray(grad\_x, [0 1 0], 'post')) + ...

abs(padarray(grad\_y, [1 0 0], 'post')) + ...

abs(padarray(grad\_z, [0 0 1], 'post'));

end

function score = compute\_smsds(original, deblocked)

score = sum((original(:) - deblocked(:)).^2) / numel(original);

end

function filtered = apply\_frequency\_filter(data)

H = fspecial('gaussian', [3 3], 0.5);

[~, ~, d] = size(data);

filtered = data;

for i = 1:d

filtered(:,:,i) = imfilter(data(:,:,i), H, 'replicate');

end

end

function updated = update\_boundary(smoothed, mask)

updated = smoothed;

updated(mask > 0.5) = smoothed(mask > 0.5);

end

function dct3d = dct3(block)

dct3d = block;

for k = 1:size(dct3d,3)

dct3d(:,:,k) = dct2(dct3d(:,:,k));

end

dct3d = permute(dct3d, [3 2 1]);

for k = 1:size(dct3d,3)

dct3d(:,:,k) = dct2(dct3d(:,:,k));

end

dct3d = permute(dct3d, [3 2 1]);

end

function idct3d = idct3(block)

idct3d = block;

for k = 1:size(idct3d,3)

idct3d(:,:,k) = idct2(idct3d(:,:,k));

end

idct3d = permute(idct3d, [3 2 1]);

for k = 1:size(idct3d,3)

idct3d(:,:,k) = idct2(idct3d(:,:,k));

end

idct3d = permute(idct3d, [3 2 1]);

end

**A.2.8 3D DCT-Based Compression with Advanced Optimization**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.png')); ...

dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize] = compress3D\_DCT(img);

decompressed\_img = decompress3D\_DCT(compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize);

if size(decompressed\_img) ~= size(img)

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

profile = double(decompressed\_img(round(end/2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(img) - double(decompressed\_img));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

ssim\_value = ssim(uint8(decompressed\_img), img);

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

ssim\_value = ssim(uint8(decompressed\_img), img);

end

cnr\_values = [cnr\_values, cnr];

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(compressedData);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize] = compress3D\_DCT(img)

imgSize = size(img);

[LL, LH, HL, HH] = dwt2(img, 'haar');

LH\_reconstructed = idwt2(zeros(size(LL)), LH, zeros(size(HL)), zeros(size(HH)), 'haar');

LH\_dft = fft2(LH\_reconstructed);

LH\_mean = mean(abs(LH\_dft(:)));

cubeSize = (LH\_mean < 0.5) \* 8 + (LH\_mean >= 0.5) \* 4;

padRows = mod(-size(img, 1), cubeSize);

padCols = mod(-size(img, 2), cubeSize);

paddedImg = padarray(img, [padRows, padCols], 'post');

paddedImgSize = size(paddedImg);

cubes = {};

idx = 1;

for i = 1:cubeSize:paddedImgSize(1)

for j = 1:cubeSize:paddedImgSize(2)

block = paddedImg(i:min(i+cubeSize-1, paddedImgSize(1)), ...

j:min(j+cubeSize-1, paddedImgSize(2)));

block = double(block);

if size(block, 1) < cubeSize || size(block, 2) < cubeSize

block = padarray(block, [cubeSize - size(block, 1), cubeSize - size(block, 2)], 'post');

end

if numel(block) == cubeSize^2

block = repmat(block, 1, 1, cubeSize);

end

if numel(block) == cubeSize^3

cubes{idx} = reshape(block, [cubeSize, cubeSize, cubeSize]);

idx = idx + 1;

end

end

end

if isempty(cubes)

error('Cube formation failed: No valid cubes generated.');

end

for k = 1:length(cubes)

cubes{k} = dctn(cubes{k});

end

quantMatrix = @(i, j, k) 3 + 6\*i + 1\*j + k;

for k = 1:length(cubes)

cubes{k} = round(cubes{k} ./ quantMatrix(1:cubeSize, 1:cubeSize, 1:cubeSize));

end

zigzagOrder = generateZigZagOrder(cubeSize);

scannedCubes = [];

for k = 1:length(cubes)

cube = cubes{k};

scannedCubes = [scannedCubes; cube(zigzagOrder)];

end

symbols = unique(scannedCubes);

prob = histcounts(scannedCubes, [symbols(:); max(symbols)+1]) / numel(scannedCubes);

huffmanDict = huffmandict(symbols, prob);

compressedData = huffmanenco(scannedCubes, huffmanDict);

end

function img = decompress3D\_DCT(compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize)

scannedCubes = huffmandeco(compressedData, huffmanDict);

expectedElements = (paddedImgSize(1) / cubeSize) \* (paddedImgSize(2) / cubeSize) \* cubeSize^3;

if numel(scannedCubes) ~= expectedElements

error('Decompression failed: scannedCubes size mismatch.');

end

zigzagOrder = generateZigZagOrder(cubeSize);

numCubes = numel(scannedCubes) / (cubeSize^3);

cubes = cell(1, numCubes);

for k = 1:numCubes

startIdx = (k-1)\*cubeSize^3 + 1;

endIdx = k\*cubeSize^3;

scannedBlock = scannedCubes(startIdx:endIdx);

cube = zeros(cubeSize^3, 1);

cube(zigzagOrder) = scannedBlock;

cubes{k} = reshape(cube, [cubeSize, cubeSize, cubeSize]) .\* quantMatrix(1:cubeSize, 1:cubeSize, 1:cubeSize);

end

paddedImg = zeros(paddedImgSize);

idx = 1;

for i = 1:cubeSize:paddedImgSize(1)

for j = 1:cubeSize:paddedImgSize(2)

paddedImg(i:i+cubeSize-1, j:j+cubeSize-1) = cubes{idx}(:, :, 1);

idx = idx + 1;

end

end

img = paddedImg(1:imgSize(1), 1:imgSize(2));

end

function zigzagOrder = generateZigZagOrder(cubeSize)

[x, y, z] = ndgrid(1:cubeSize, 1:cubeSize, 1:cubeSize);

linearIndices = sub2ind([cubeSize, cubeSize, cubeSize], x(:), y(:), z(:));

layerSums = x(:) + y(:) + z(:);

[~, sortIdx] = sort(layerSums);

zigzagOrder = linearIndices(sortIdx);

end

function output = dctn(input)

dims = ndims(input);

output = input;

for dim = 1:dims

output = dct(output, [], dim);

end

end

function output = idctn(input)

dims = ndims(input);

output = input;

for dim = 1:dims

output = idct(output, [], dim);

end

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.png')); ...

dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

original\_size = numel(img) \* 8;

[compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize] = compress3D\_DCT(img);

decompressed\_img = decompress3D\_DCT(compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize);

if ~isequal(size(decompressed\_img), size(img))

decompressed\_img = imresize(decompressed\_img, size(img), 'nearest');

end

compressed\_size = numel(compressedData);

cr\_value = original\_size / compressed\_size;

cr\_values(idx) = cr\_value;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size \* 8, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size \* 8, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Compression Ratio: %.2f\n', mean(cr\_values));

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize] = compress3D\_DCT(img)

imgSize = size(img);

[LL, LH, HL, HH] = dwt2(img, 'haar');

LH\_reconstructed = idwt2(zeros(size(LL)), LH, zeros(size(HL)), zeros(size(HH)), 'haar');

LH\_dft = fft2(LH\_reconstructed);

LH\_mean = mean(abs(LH\_dft(:)));

cubeSize = (LH\_mean < 0.5) \* 8 + (LH\_mean >= 0.5) \* 4;

padRows = mod(-size(img, 1), cubeSize);

padCols = mod(-size(img, 2), cubeSize);

paddedImg = padarray(img, [padRows, padCols], 'post');

paddedImgSize = size(paddedImg);

cubes = {};

idx = 1;

for i = 1:cubeSize:paddedImgSize(1)

for j = 1:cubeSize:paddedImgSize(2)

block = paddedImg(i:min(i+cubeSize-1, paddedImgSize(1)), ...

j:min(j+cubeSize-1, paddedImgSize(2)));

block = double(block);

if size(block, 1) < cubeSize || size(block, 2) < cubeSize

block = padarray(block, [cubeSize - size(block, 1), cubeSize - size(block, 2)], 'post');

end

if numel(block) == cubeSize^2

block = repmat(block, 1, 1, cubeSize);

end

if numel(block) == cubeSize^3

cubes{idx} = reshape(block, [cubeSize, cubeSize, cubeSize]);

idx = idx + 1;

end

end

end

if isempty(cubes)

error('Cube formation failed: No valid cubes generated.');

end

for k = 1:length(cubes)

cubes{k} = dctn(cubes{k});

end

quantMatrix = @(i, j, k) 3 + 6\*i + 1\*j + k;

for k = 1:length(cubes)

cubes{k} = round(cubes{k} ./ quantMatrix(1:cubeSize, 1:cubeSize, 1:cubeSize));

end

zigzagOrder = generateZigZagOrder(cubeSize);

scannedCubes = [];

for k = 1:length(cubes)

cube = cubes{k};

scannedCubes = [scannedCubes; cube(zigzagOrder)];

end

symbols = unique(scannedCubes);

prob = histcounts(scannedCubes, [symbols(:); max(symbols)+1]) / numel(scannedCubes);

huffmanDict = huffmandict(symbols, prob);

compressedData = huffmanenco(scannedCubes, huffmanDict);

end

function img = decompress3D\_DCT(compressedData, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize)

scannedCubes = huffmandeco(compressedData, huffmanDict);

expectedElements = (paddedImgSize(1) / cubeSize) \* (paddedImgSize(2) / cubeSize) \* cubeSize^3;

if numel(scannedCubes) ~= expectedElements

error('Decompression failed: scannedCubes size mismatch.');

end

zigzagOrder = generateZigZagOrder(cubeSize);

numCubes = numel(scannedCubes) / (cubeSize^3);

cubes = cell(1, numCubes);

for k = 1:numCubes

startIdx = (k-1)\*cubeSize^3 + 1;

endIdx = k\*cubeSize^3;

scannedBlock = scannedCubes(startIdx:endIdx);

cube = zeros(cubeSize^3, 1);

cube(zigzagOrder) = scannedBlock;

cubes{k} = reshape(cube, [cubeSize, cubeSize, cubeSize]) .\* quantMatrix(1:cubeSize, 1:cubeSize, 1:cubeSize);

end

paddedImg = zeros(paddedImgSize);

idx = 1;

for i = 1:cubeSize:paddedImgSize(1)

for j = 1:cubeSize:paddedImgSize(2)

paddedImg(i:i+cubeSize-1, j:j+cubeSize-1) = cubes{idx}(:, :, 1);

idx = idx + 1;

end

end

img = paddedImg(1:imgSize(1), 1:imgSize(2));

end

function zigzagOrder = generateZigZagOrder(cubeSize)

[x, y, z] = ndgrid(1:cubeSize, 1:cubeSize, 1:cubeSize);

linearIndices = sub2ind([cubeSize, cubeSize, cubeSize], x(:), y(:), z(:));

layerSums = x(:) + y(:) + z(:);

[~, sortIdx] = sort(layerSums);

zigzagOrder = linearIndices(sortIdx);

end

function output = dctn(input)

dims = ndims(input);

output = input;

for dim = 1:dims

output = dct(output, [], dim);

end

end

function output = idctn(input)

dims = ndims(input);

output = input;

for dim = 1:dims

output = idct(output, [], dim);

end

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = im2double(imresize(img, [256 256]));

tic;

[compressedStream, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize] = compress3D\_DCT(img);

decompressed\_img = decompress3D\_DCT(compressedStream, huffmanDict, quantMatrix, cubeSize, imgSize, paddedImgSize);

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch ME

fprintf('Error at image %d: %s\n', idx, ME.message);

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('3D DCT-Advanced Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function [stream, dict, quantMatrix, cubeSize, imgSize, paddedSize] = compress3D\_DCT(img)

cubeSize = 8;

imgSize = size(img);

paddedImg = padarray(img, mod(-size(img), cubeSize), 'post');

paddedSize = size(paddedImg);

numBlocks = (paddedSize(1) / cubeSize) \* (paddedSize(2) / cubeSize);

cube = zeros(cubeSize, cubeSize, numBlocks);

count = 1;

for i = 1:cubeSize:paddedSize(1)

for j = 1:cubeSize:paddedSize(2)

cube(:,:,count) = paddedImg(i:i+cubeSize-1, j:j+cubeSize-1);

count = count + 1;

end

end

dct\_cube = dctn(cube);

[I, J, K] = ndgrid(1:cubeSize, 1:cubeSize, 1:size(cube,3));

quantMatrix = 1 + (I + J + K - 3);

q\_cube = round(dct\_cube ./ quantMatrix);

data = q\_cube(:) + 128;

data(data > 255) = 255;

data(data < 0) = 0;

symbols = uint8(data);

uniqueVals = unique(symbols);

prob = ones(length(uniqueVals),1) / length(uniqueVals);

dict = huffmandict(uniqueVals, prob);

stream = huffmanenco(symbols, dict);

end

function decompressed = decompress3D\_DCT(stream, dict, quantMatrix, cubeSize, imgSize, paddedSize)

decoded = huffmandeco(stream, dict);

decoded = double(decoded) - 128;

totalElements = prod(paddedSize);

if length(decoded) < totalElements

decoded(end+1:totalElements) = 0;

elseif length(decoded) > totalElements

decoded = decoded(1:totalElements);

end

cubeCount = paddedSize(1)/cubeSize \* paddedSize(2)/cubeSize;

q\_cube = reshape(decoded, cubeSize, cubeSize, cubeCount);

dequant = q\_cube .\* quantMatrix;

idct\_cube = idctn(dequant);

reconstructed = zeros(paddedSize);

count = 1;

for i = 1:cubeSize:paddedSize(1)

for j = 1:cubeSize:paddedSize(2)

reconstructed(i:i+cubeSize-1, j:j+cubeSize-1) = idct\_cube(:,:,count);

count = count + 1;

end

end

decompressed = reconstructed(1:imgSize(1), 1:imgSize(2));

end

function order = generateZigZagOrder(N)

order = zeros(N, N);

index = 1;

for s = 1:2\*N - 1

if mod(s, 2) == 0

for i = max(1, s - N + 1):min(N, s)

j = s - i + 1;

order(i, j) = index;

index = index + 1;

end

else

for j = max(1, s - N + 1):min(N, s)

i = s - j + 1;

order(i, j) = index;

index = index + 1;

end

end

end

order = order(:);

end

function y = dctn(x)

y = x;

for k = 1:size(x,3)

y(:,:,k) = dct2(y(:,:,k));

end

y = permute(y, [3 2 1]);

for k = 1:size(y,3)

y(:,:,k) = dct2(y(:,:,k));

end

y = permute(y, [3 2 1]);

end

function y = idctn(x)

y = x;

for k = 1:size(y,3)

y(:,:,k) = idct2(y(:,:,k));

end

y = permute(y, [3 2 1]);

for k = 1:size(y,3)

y(:,:,k) = idct2(y(:,:,k));

end

y = permute(y, [3 2 1]);

end

**A.2.9 Novel JPEG Post-Processing**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

quantMatrix = [16 11 10 16 24 40 51 61;

12 12 14 19 26 58 60 55;

14 13 16 24 40 57 69 56;

14 17 22 29 51 87 80 62;

18 22 37 56 68 109 103 77;

24 35 55 64 81 104 113 92;

49 64 78 87 103 121 120 101;

72 92 95 98 112 100 103 99];

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

originalImage = img;

[reconstructedImage, encodedData] = post\_process\_artifact\_reduction(double(originalImage), quantMatrix);

profile = double(reconstructedImage(round(end / 2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

reconstruction\_error = abs(double(originalImage) - double(reconstructedImage));

dynamic\_threshold = mean(reconstruction\_error(:)) + 2 \* std(reconstruction\_error(:));

signal\_region = reconstruction\_error(reconstruction\_error > dynamic\_threshold);

background\_region = reconstruction\_error(reconstruction\_error <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

else

mu\_signal = mean(signal\_region(:));

mu\_background = mean(background\_region(:));

std\_background = std(background\_region(:));

cnr = abs(mu\_signal - mu\_background) / std\_background;

end

cnr\_values = [cnr\_values, cnr];

ssim\_value = ssim(uint8(reconstructedImage), uint8(originalImage));

ssim\_values = [ssim\_values, ssim\_value];

if exist('encodedData', 'var') && ~isempty(encodedData)

original\_size = numel(originalImage) \* 8;

compressed\_size = numel(encodedData);

cr\_value = original\_size / compressed\_size;

else

cr\_value = NaN;

end

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values, 'omitnan'));

fprintf('Average CNR: %.2f\n', mean(cnr\_values, 'omitnan'));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values, 'omitnan'));

fprintf('Average CR: %.2f\n', mean(cr\_values, 'omitnan'));

function [finalImage, encodedData] = post\_process\_artifact\_reduction(img, quantMatrix)

shifts = [-1, 0, 1];

decompressedImages = zeros([size(img), numel(shifts)^2]);

count = 1;

for dx = shifts

for dy = shifts

shiftedImg = circshift(img, [dx, dy]);

[encodedData, huffmanDict, imgSize, paddedSize] = compress\_image(shiftedImg, quantMatrix);

decompressedImg = decompress\_image(encodedData, huffmanDict, imgSize, paddedSize, quantMatrix);

decompressedImages(:, :, count) = circshift(decompressedImg, [-dx, -dy]);

count = count + 1;

end

end

finalImage = mean(decompressedImages, 3);

finalImage = uint8(finalImage);

end

function [encodedData, huffmanDict, imgSize, paddedSize] = compress\_image(img, quantMatrix)

imgSize = size(img);

paddedSize = ceil(imgSize / 8) \* 8;

paddedImage = padarray(img, paddedSize - imgSize, 'replicate', 'post');

blocks = mat2cell(paddedImage, ...

repmat(8, 1, size(paddedImage, 1) / 8), ...

repmat(8, 1, size(paddedImage, 2) / 8));

blocks = cellfun(@(block) block - 128, blocks, 'UniformOutput', false);

dctBlocks = cellfun(@(block) dct2(block), blocks, 'UniformOutput', false);

quantBlocks = cellfun(@(block) round(block ./ quantMatrix), dctBlocks, 'UniformOutput', false);

zigzagBlocks = cellfun(@zigzag\_scan, quantBlocks, 'UniformOutput', false);

allCoefficients = cell2mat(zigzagBlocks(:));

symbols = unique(allCoefficients);

probabilities = histcounts(allCoefficients, [symbols; max(symbols) + 1]) / numel(allCoefficients);

huffmanDict = huffmandict(symbols, probabilities);

encodedData = huffmanenco(allCoefficients(:), huffmanDict);

end

function zigzagArray = zigzag\_scan(block)

zigzagOrder = [

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64

];

zigzagArray = block(zigzagOrder(:));

end

function reconstructedImage = decompress\_image(encodedData, huffmanDict, imgSize, paddedSize, quantMatrix)

decodedCoefficients = huffmandeco(encodedData, huffmanDict);

numBlocks = (paddedSize(1) / 8) \* (paddedSize(2) / 8);

reshapedCoefficients = reshape(decodedCoefficients, [], numBlocks);

quantBlocks = cellfun(@(col) inverse\_zigzag\_scan(col), num2cell(reshapedCoefficients, 1), 'UniformOutput', false);

quantBlocks = reshape(quantBlocks, paddedSize(1) / 8, paddedSize(2) / 8);

dequantBlocks = cellfun(@(block) block .\* quantMatrix, quantBlocks, 'UniformOutput', false);

idctBlocks = cellfun(@(block) idct2(block), dequantBlocks, 'UniformOutput', false);

reconstructedBlocks = cellfun(@(block) block + 128, idctBlocks, 'UniformOutput', false);

reconstructedImage = cell2mat(reconstructedBlocks);

reconstructedImage = reconstructedImage(1:imgSize(1), 1:imgSize(2));

reconstructedImage = uint8(reconstructedImage);

end

function block = inverse\_zigzag\_scan(zigzagArray)

zigzagOrder = [

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64

];

block = zeros(8, 8);

block(zigzagOrder(:)) = zigzagArray;

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.png')); ...

dir(fullfile('\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

quantMatrix = [16 11 10 16 24 40 51 61;

12 12 14 19 26 58 60 55;

14 13 16 24 40 57 69 56;

14 17 22 29 51 87 80 62;

18 22 37 56 68 109 103 77;

24 35 55 64 81 104 113 92;

49 64 78 87 103 121 120 101;

72 92 95 98 112 100 103 99];

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

originalImage = img;

[reconstructedImage, encodedData] = post\_process\_artifact\_reduction(double(originalImage), quantMatrix);

original\_size\_MB = (numel(originalImage) \* 8) / (1024 \* 1024);

compressed\_size\_MB = numel(encodedData) \* 8 / (1024 \* 1024);

if compressed\_size\_MB > 0

cr\_value = original\_size\_MB / compressed\_size\_MB;

else

cr\_value = NaN;

end

cr\_values(idx) = cr\_value;

compressed\_size\_bits = numel(encodedData) \* 8;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [finalImage, encodedData] = post\_process\_artifact\_reduction(img, quantMatrix)

shifts = [-1, 0, 1];

decompressedImages = zeros([size(img), numel(shifts)^2]);

count = 1;

for dx = shifts

for dy = shifts

shiftedImg = circshift(img, [dx, dy]);

[encodedData, huffmanDict, imgSize, paddedSize] = compress\_image(shiftedImg, quantMatrix);

decompressedImg = decompress\_image(encodedData, huffmanDict, imgSize, paddedSize, quantMatrix);

decompressedImages(:, :, count) = circshift(decompressedImg, [-dx, -dy]);

count = count + 1;

end

end

finalImage = mean(decompressedImages, 3);

finalImage = uint8(finalImage);

end

function [encodedData, huffmanDict, imgSize, paddedSize] = compress\_image(img, quantMatrix)

imgSize = size(img);

paddedSize = ceil(imgSize / 8) \* 8;

paddedImage = padarray(img, paddedSize - imgSize, 'replicate', 'post');

blocks = mat2cell(paddedImage, ...

repmat(8, 1, size(paddedImage, 1) / 8), ...

repmat(8, 1, size(paddedImage, 2) / 8));

blocks = cellfun(@(block) block - 128, blocks, 'UniformOutput', false);

dctBlocks = cellfun(@(block) dct2(block), blocks, 'UniformOutput', false);

quantBlocks = cellfun(@(block) round(block ./ quantMatrix), dctBlocks, 'UniformOutput', false);

zigzagBlocks = cellfun(@zigzag\_scan, quantBlocks, 'UniformOutput', false);

allCoefficients = cell2mat(zigzagBlocks(:));

symbols = unique(allCoefficients);

probabilities = histcounts(allCoefficients, [symbols; max(symbols) + 1]) / numel(allCoefficients);

huffmanDict = huffmandict(symbols, probabilities);

encodedData = huffmanenco(allCoefficients(:), huffmanDict);

end

function zigzagArray = zigzag\_scan(block)

zigzagOrder = [

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64

];

zigzagArray = block(zigzagOrder(:));

end

function reconstructedImage = decompress\_image(encodedData, huffmanDict, imgSize, paddedSize, quantMatrix)

decodedCoefficients = huffmandeco(encodedData, huffmanDict);

numBlocks = (paddedSize(1) / 8) \* (paddedSize(2) / 8);

reshapedCoefficients = reshape(decodedCoefficients, [], numBlocks);

quantBlocks = cellfun(@(col) inverse\_zigzag\_scan(col), num2cell(reshapedCoefficients, 1), 'UniformOutput', false);

quantBlocks = reshape(quantBlocks, paddedSize(1) / 8, paddedSize(2) / 8);

dequantBlocks = cellfun(@(block) block .\* quantMatrix, quantBlocks, 'UniformOutput', false);

idctBlocks = cellfun(@(block) idct2(block), dequantBlocks, 'UniformOutput', false);

reconstructedBlocks = cellfun(@(block) block + 128, idctBlocks, 'UniformOutput', false);

reconstructedImage = cell2mat(reconstructedBlocks);

reconstructedImage = reconstructedImage(1:imgSize(1), 1:imgSize(2));

reconstructedImage = uint8(reconstructedImage);

end

function block = inverse\_zigzag\_scan(zigzagArray)

zigzagOrder = [

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64

];

block = zeros(8, 8);

block(zigzagOrder(:)) = zigzagArray;

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

img = im2double(imresize(img, [256 256]));

tic;

compressed = compress\_image(img);

decompressed = decompress\_image(compressed);

final\_img = post\_process\_artifact\_reduction(decompressed);

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch ME

fprintf('Error at image %d: %s\n', idx, ME.message);

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('JPEG Post-Processing Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function enhanced = post\_process\_artifact\_reduction(img)

H = fspecial('average', [3 3]);

smoothed = imfilter(img, H, 'replicate');

enhanced = img + 0.2 \* (smoothed - img);

end

function compressed = compress\_image(img)

blockSize = 8;

[h, w] = size(img);

numBlocksH = floor(h / blockSize);

numBlocksW = floor(w / blockSize);

blocks = zeros(blockSize, blockSize, numBlocksH \* numBlocksW);

count = 1;

for i = 1:numBlocksH

for j = 1:numBlocksW

block = img((i-1)\*blockSize+1:i\*blockSize, (j-1)\*blockSize+1:j\*blockSize);

dct\_block = dct2(block);

blocks(:,:,count) = dct\_block;

count = count + 1;

end

end

zigzag = zigzag\_scan(blocks);

symbols = round(zigzag(:) + 128);

symbols(symbols > 255) = 255;

symbols(symbols < 0) = 0;

symbols = uint8(symbols);

u = unique(symbols);

dict = huffmandict(u, ones(length(u), 1) / length(u));

stream = huffmanenco(symbols, dict);

compressed.stream = stream;

compressed.dict = dict;

compressed.size = size(blocks);

end

function zz = zigzag\_scan(blocks)

[m, n, p] = size(blocks);

order = reshape([

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64], [m\*n 1]);

zz = zeros(length(order), p);

for k = 1:p

flat = reshape(blocks(:,:,k), [], 1);

zz(:,k) = flat(order);

end

end

function img = decompress\_image(compressed)

total\_elements = prod(compressed.size);

decoded = huffmandeco(compressed.stream, compressed.dict);

decoded = double(decoded(1:total\_elements)) - 128;

zz = reshape(decoded, [], compressed.size(3));

blocks = inverse\_zigzag\_scan(zz, compressed.size(1), compressed.size(2));

[h\_blocks, w\_blocks, ~] = size(blocks);

blockSize = h\_blocks;

side = sqrt(compressed.size(3));

img = zeros(blockSize \* side);

count = 1;

for i = 1:side

for j = 1:side

img((i-1)\*blockSize+1:i\*blockSize, (j-1)\*blockSize+1:j\*blockSize) = idct2(blocks(:,:,count));

count = count + 1;

end

end

end

function blocks = inverse\_zigzag\_scan(zz, m, n)

order = reshape([

1 2 6 7 15 16 28 29;

3 5 8 14 17 27 30 43;

4 9 13 18 26 31 42 44;

10 12 19 25 32 41 45 54;

11 20 24 33 40 46 53 55;

21 23 34 39 47 52 56 61;

22 35 38 48 51 57 60 62;

36 37 49 50 58 59 63 64], [m\*n 1]);

blocks = zeros(m, n, size(zz, 2));

for k = 1:size(zz, 2)

flat = zeros(m \* n, 1);

flat(order) = zz(:,k);

blocks(:,:,k) = reshape(flat, m, n);

end

end

**A.2.10 New Prediction-Based Compression**

Image Quality Evaluation

imageFiles = [dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.png')); ...

dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

fwhm\_values = [];

cnr\_values = [];

ssim\_values = [];

cr\_values = [];

pixel\_size = 0.3528;

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

[encodedData, reconstructedImage, row, col] = prediction\_based\_compression(img);

profile = double(reconstructedImage(round(end / 2), :));

max\_val = max(profile);

half\_max = max\_val / 2;

left\_idx = find(profile >= half\_max, 1, 'first');

right\_idx = find(profile >= half\_max, 1, 'last');

fwhm\_mm = (right\_idx - left\_idx) \* pixel\_size;

fwhm\_values = [fwhm\_values, fwhm\_mm];

dynamic\_threshold = mean(img(:));

signal\_region = img(img > dynamic\_threshold);

background\_region = img(img <= dynamic\_threshold);

if isempty(signal\_region) || isempty(background\_region)

cnr = 0;

else

mu\_signal = mean(double(signal\_region(:)));

mu\_background = mean(double(background\_region(:)));

std\_background = std(double(background\_region(:)));

cnr = abs(mu\_signal - mu\_background) / std\_background;

end

cnr\_values = [cnr\_values, cnr];

ssim\_value = ssim(uint8(reconstructedImage), img);

ssim\_values = [ssim\_values, ssim\_value];

original\_size = numel(img) \* 8;

compressed\_size = numel(encodedData);

cr\_value = original\_size / compressed\_size;

cr\_values = [cr\_values, cr\_value];

fprintf('Image %d: FWHM: %.2f mm, CNR: %.2f, SSIM: %.2f, CR: %.2f\n', ...

idx, fwhm\_mm, cnr, ssim\_value, cr\_value);

end

fprintf('Average FWHM: %.2f mm\n', mean(fwhm\_values));

fprintf('Average CNR: %.2f\n', mean(cnr\_values));

fprintf('Average SSIM: %.2f\n', mean(ssim\_values));

fprintf('Average CR: %.2f\n', mean(cr\_values));

function [encodedData, reconstructedImage, row, col] = prediction\_based\_compression(img)

[row, col] = size(img);

MV = max(max(img));

Mat = double(img);

reducedMatrix = Mat(:, 1:2:end);

savedColumns = Mat(:, 2:2:end);

E = reducedMatrix - [zeros(size(reducedMatrix, 1), 1), reducedMatrix(:, 1:end-1)];

errorVector = E(:);

[uniqueValues, ~, idxMap] = unique(errorVector);

frequencies = histcounts(idxMap, numel(uniqueValues));

validIndices = frequencies > 0;

symbols = uniqueValues(validIndices)';

frequencies = frequencies(validIndices);

if numel(symbols) <= 1

warning('Only one unique symbol found. Huffman coding skipped.');

encodedData = [];

reconstructedImage = uint8(reducedMatrix);

return;

end

probabilities = frequencies / sum(frequencies);

huffmanDict = huffmandict(symbols, probabilities);

encodedData = huffmanenco(errorVector, huffmanDict);

decodedErrors = huffmandeco(encodedData, huffmanDict);

decodedErrors = reshape(decodedErrors, size(E));

decodedMatrix = zeros(size(reducedMatrix));

decodedMatrix(:, 1) = reducedMatrix(:, 1);

for c = 2:size(reducedMatrix, 2)

decodedMatrix(:, c) = decodedMatrix(:, c - 1) + decodedErrors(:, c);

end

reconstructedImage = zeros(row, col);

reconstructedImage(:, 1:2:end) = decodedMatrix;

reconstructedImage(:, 2:2:end) = savedColumns;

reconstructedImage = uint8(reconstructedImage);

end

Transmission Performance Evaluation

imageFiles = [dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.png')); ...

dir(fullfile('\\\\stafffiles.win.canberra.edu.au\\homes$\\s443807\\My Documents\\Matlab\\10 images\\Images', '\*.jpg'))];

num\_images = min(10, length(imageFiles));

bandwidth\_3G = 384 \* 1024;

bandwidth\_4G = 10 \* 1024 \* 1024;

chunk\_size = 1024;

transmission\_times\_3G = zeros(1, num\_images);

transmission\_times\_4G = zeros(1, num\_images);

cr\_values = zeros(1, num\_images);

for idx = 1:num\_images

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

original\_size\_bits = numel(img) \* 8;

original\_size\_MB = original\_size\_bits / (8 \* 1024 \* 1024);

[encodedData, ~, ~, ~] = prediction\_based\_compression(img);

compressed\_size\_bits = numel(encodedData) \* 8;

compressed\_size\_MB = compressed\_size\_bits / (8 \* 1024 \* 1024);

cr\_values(idx) = original\_size\_bits / compressed\_size\_bits;

transmission\_times\_3G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_3G, chunk\_size);

transmission\_times\_4G(idx) = simulate\_transfer(compressed\_size\_bits, bandwidth\_4G, chunk\_size);

fprintf('Image %d: Transfer Time (3G): %.2f sec, Transfer Time (4G): %.2f sec\n', ...

idx, transmission\_times\_3G(idx), transmission\_times\_4G(idx));

end

fprintf('Average Transfer Time (3G): %.2f sec\n', mean(transmission\_times\_3G));

fprintf('Average Transfer Time (4G): %.2f sec\n', mean(transmission\_times\_4G));

function time = simulate\_transfer(data\_size\_bits, bandwidth\_bps, chunk\_size)

chunk\_size\_bits = chunk\_size \* 8;

num\_chunks = ceil(data\_size\_bits / chunk\_size\_bits);

time\_per\_chunk = chunk\_size\_bits / bandwidth\_bps;

time = num\_chunks \* time\_per\_chunk;

end

function [encodedData, reconstructedImage, row, col] = prediction\_based\_compression(img)

[row, col] = size(img);

Mat = double(img);

reducedMatrix = Mat(:, 1:2:end);

savedColumns = Mat(:, 2:2:end);

E = reducedMatrix - [zeros(size(reducedMatrix, 1), 1), reducedMatrix(:, 1:end-1)];

errorVector = E(:);

[uniqueValues, ~, idxMap] = unique(errorVector);

frequencies = histcounts(idxMap, numel(uniqueValues));

validIndices = frequencies > 0;

symbols = uniqueValues(validIndices)';

frequencies = frequencies(validIndices);

if numel(symbols) <= 1

warning('Only one unique symbol found. Huffman coding skipped.');

encodedData = [];

reconstructedImage = uint8(reducedMatrix);

return;

end

probabilities = frequencies / sum(frequencies);

huffmanDict = huffmandict(symbols, probabilities);

encodedData = huffmanenco(errorVector, huffmanDict);

decodedErrors = huffmandeco(encodedData, huffmanDict);

decodedErrors = reshape(decodedErrors, size(E));

decodedMatrix = zeros(size(reducedMatrix));

decodedMatrix(:, 1) = reducedMatrix(:, 1);

for c = 2:size(reducedMatrix, 2)

decodedMatrix(:, c) = decodedMatrix(:, c - 1) + decodedErrors(:, c);

end

reconstructedImage = zeros(row, col);

reconstructedImage(:, 1:2:end) = decodedMatrix;

reconstructedImage(:, 2:2:end) = savedColumns;

reconstructedImage = uint8(reconstructedImage);

end

Timing and Memory Profiling

clear; clc;

addpath(genpath(pwd));

image\_dir = '\\stafffiles.win.canberra.edu.au\homes$\s443807\My Documents\Matlab\10 images\Images';

imageFiles = [dir(fullfile(image\_dir, '\*.png')); dir(fullfile(image\_dir, '\*.jpg'))];

num\_images = min(10, length(imageFiles));

exec\_times = zeros(num\_images, 1);

mem\_usages = zeros(num\_images, 1);

for idx = 1:num\_images

try

img = imread(fullfile(imageFiles(idx).folder, imageFiles(idx).name));

if size(img, 3) == 3

img = rgb2gray(img);

end

tic;

[encodedData, reconstructedImage] = prediction\_based\_compression(img);

exec\_times(idx) = toc;

mem\_info = memory;

mem\_usages(idx) = mem\_info.MemUsedMATLAB / 1e6;

catch ME

fprintf('Error at image %d: %s\n', idx, ME.message);

exec\_times(idx) = NaN;

mem\_usages(idx) = NaN;

end

end

fprintf('Prediction-Based Compression Timing Benchmark:\n');

fprintf('Avg Time per Image: %.4f seconds\n', nanmean(exec\_times));

fprintf('Avg Memory Used: %.2f MB\n', nanmean(mem\_usages));

function [encodedData, reconstructedImage] = prediction\_based\_compression(img)

[row, col] = size(img);

Mat = double(img);

reducedMatrix = Mat(:, 1:2:end);

savedColumns = Mat(:, 2:2:end);

E = reducedMatrix - [zeros(size(reducedMatrix, 1), 1), reducedMatrix(:, 1:end-1)];

errorVector = E(:);

[uniqueValues, ~, idxMap] = unique(errorVector);

frequencies = histcounts(idxMap, numel(uniqueValues));

validIndices = frequencies > 0;

symbols = uniqueValues(validIndices)';

frequencies = frequencies(validIndices);

if numel(symbols) <= 1

encodedData = [];

reconstructedImage = uint8(reducedMatrix);

return;

end

probabilities = frequencies / sum(frequencies);

huffmanDict = huffmandict(symbols, probabilities);

encodedData = huffmanenco(errorVector, huffmanDict);

decodedErrors = huffmandeco(encodedData, huffmanDict);

decodedErrors = reshape(decodedErrors, size(E));

decodedMatrix = zeros(size(reducedMatrix));

decodedMatrix(:, 1) = reducedMatrix(:, 1);

for c = 2:size(reducedMatrix, 2)

decodedMatrix(:, c) = decodedMatrix(:, c - 1) + decodedErrors(:, c);

end

reconstructedImage = zeros(row, col);

reconstructedImage(:, 1:2:end) = decodedMatrix;

reconstructedImage(:, 2:2:end) = savedColumns;

reconstructedImage = uint8(reconstructedImage);

end