Image Compression based on Non-Parametric Sampling in Noisy Environments (Compression using Holes and DCT)

Table of Contents

Authors: 19G01: Kishan Narotam (717 931) & Nitesh Nana (720 064)

Clear workspace and command window

```
clear all
clc
close all
```

Step 1: Loading an Image

We are going to load the image into MATLAB, asking for the name of the file. The user input is typed in and converted into a string. When the file is read into MATLAB, it's read in as a 3 dimensional matrix. We display the image for manual debugging.

```
fileName = uigetfile('*.*');
uploadedImage = imread(fileName);

% Display the uploaded image. The axis do not show, so we set the
% visibility to be on in order to see the pixels
figure('units','normalized','outerposition',[0 0 1 1])
subplot(1,3,1)
imshow(uploadedImage);
title(strcat('Original image: ', fileName));
axis = gca;
axis.Visible = 'On';

tic
```

Step 2: Convert the image to grayscale

Dealing with a 3 dimensional matrix is a challenge, as the third dimension is the colour map. So converting the image to grayscale will make the uploaded image into a 2 dimensional array.

```
grayImage = rgb2gray(uploadedImage);
%grayImage = uploadedImage;

% Display the uploaded image into grayscale. Again, the axis does not
 show,
% so we set the visibility to be on.
subplot(1,3,2)
imshow(grayImage);
title(strcat('Grayscale image: ', fileName));
axis = gca;
axis.Visible = 'On';
imwrite(grayImage, 'OriginalImage.gif')
```

Step 3: Divide the image into the domain pool

First we need to know the height and the width of the image. From this point, when "image" is used it refers to the converted grayscale image and NOT the original colour image.

```
[heightOfImage, widthOfImage] = size(grayImage);

% Time to determine the number of 8x8 squares in the domain pool
blocksAcross = widthOfImage/8;
blocksDown = heightOfImage/8;
totalNumberOfBlocks = blocksAcross * blocksDown;
fprintf('Total number of blocks in the domain pool: %d \n',
 totalNumberOfBlocks);

% Going row by row, the blocks in the domain pool are indexed from 1
 to the
% totalNumberOfBlocks. This is done by nested for loops. One way to
 test it
% is to subtract an indexed block from/to the corresponding pixels
 from
% imageToGray. If the resulting matrix from this subtraction is all
 0s,
% then the pixels are indexed correctly.
blocks = cell(1, totalNumberOfBlocks);

blockIndex = 1;

for yIndex = 1:blocksDown
    for xIndex = 1:blocksAcross
        if (xIndex <= ((blocksAcross*8)-8))
            blocks{blockIndex} = grayImage(((8*yIndex)-7):(8*yIndex),
 ((8*xIndex)-7):(8*xIndex));
            if (blockIndex < totalNumberOfBlocks + 1)
                blockIndex = blockIndex + 1;
            end
```

```
            end
        end
    end
```

Step 4: Creating the holes in each small block in the domain pool

With the image now split into the domain pool, we have an array of 8x8 matrices, with a size of totalNumberOfBlocks. The first part of this step is to move to the center square at position

```
for q = 1:totalNumberOfBlocks

    % Each 8x8 block within the domain pool will always start at (1,1)
    % being the top left pixel and (8,8) being the bottom right pixel.
 Each
    % 8x8 block is treated independently. For this reason, each
 block's
    % small center square will start at (4,4) - its respective top
 left
    % pixel.
    block2x2 = blocks{q}(4:5,4:5);

    % The average of the 4 blocks is calculated so that it can be used
 as
    % the point for the Chebychev check.
    average = mean(block2x2, 'all');

    % Since the starting sqaure is only 2x2, this loop need only run
 twice.
    counter = 1;
    for i = 1:2
        for j = 1:2
            temp1(counter) = pdist([block2x2(i,j);
 average], 'chebychev');
            counter = counter + 1;
        end
    end

    % Now we check if the Chebychev distance between each pixel and
 the
    % average of the square is less than 5. If they all are less than
 5,
    % the square size is increased from 2x2 to 4x4. If one value is 5
 or
    % more, no hole is created and the next block in the domain pool
 is
    % checked.
    if (all(temp1 < 6))
        block4x4 = blocks{q}(3:6, 3:6);

        average = mean(block4x4, 'all');
```

```
        % With the square size for the hole now increasing to a 4x4
size,
        % all 16 pixels are checked to see if a larger hole can be
created.
        % The reason the smaller hole is not created first is due to
the
        % hole (all values of 0), it changes the value of the average
of
        % the entire square. Only if this 4x4 square cannot be a hole,
will
        % it create the smaller hole.
        counter = 1;
        for i = 1:4
            for j = 1:4
                temp2(counter) = pdist([block4x4(i,j);
average], 'chebychev');
                counter = counter + 1;
            end
        end
        if (all(temp2 < 6))
            block6x6 = blocks{q}(2:7, 2:7);
            average = mean(block6x6, 'all');

            counter = 1;
            for i = 1:6
                for j = 1:6
                    temp3(counter) = pdist([block6x6(i,j);
average], 'chebychev');
                    counter = counter + 1;
                end
            end

            if (all(temp3 < 6))
                blocks{q}(2:7, 2:7) = 0;

            else
                blocks{q}(3:6, 3:6) = 0;
            end


        else
            blocks{q}(4:5, 4:5) = 0;
        end

    end

end

% As a way of recontructing the image, we take each indexed block from
 the
% domain pool and combine it into 1 large 2D array that is the
 grayscale
% image with holes present.
bIndex = 1;
```

```
for yIndex = 1:blocksDown
    for xIndex = 1:blocksAcross
        holesImage((8*yIndex)-7:(yIndex*8), (8*xIndex)-7:(xIndex*8)) =
 blocks{bIndex};
        bIndex = bIndex+1;
    end
end

% The axis do not show, so we set the visibility to be on in order to
 see
% the pixels.
subplot(1,3,3)
imshow(holesImage)
title(strcat('Grayscale image with holes: ', fileName));
axis = gca;
axis.Visible = 'On';
```

Step 5: Compressing the image using a known technique

Using known techniques, we perform the compression of the chosen image. First we start by performing th dct on the image itself so that we can see where the majority of the intensity is.

```
intensityImage = dct2(holesImage);
figure('units','normalized','outerposition',[0 0 1 1])
subplot(2,3,1)
imshow(intensityImage);
title(strcat('Image showing the intensity (amplitude) of the image
 for: ', fileName));
axis = gca;
axis.Visible = 'On';

% We create an input dialog box so we can get the compression depth
 for the
% image. The number will be between 1 and 8, and will determine how
 much
% compression will take place.

compressionDepth = '7';
testString = strcat(' Compression depth = ', compressionDepth);
compressionDepth = str2double(compressionDepth);

% Now to perform the actual dct compression on 8x8 blocks, we can
 create
% empty domain pools. This will be for the quantized image and idct
 and
% final compressed image.

quantizedBlocks = cell(1, totalNumberOfBlocks);
idctBlocks = cell(1, totalNumberOfBlocks);
compressedBlocks = cell(1, totalNumberOfBlocks);
```

```
for i = 1:totalNumberOfBlocks
    f = blocks{i};
    dctTemp = dct2(f);
    quantizedBlocks{i} = dctTemp;
    idctTemp = idct2(dctTemp);
    blocks{i} = idctTemp;

    dctTemp(8:-1:compressionDepth+1, :) = 0;
    dctTemp(:, 8:-1:compressionDepth+1) = 0;
    idctBlocks{i} = dctTemp;
    idctTemp = idct2(dctTemp);
    compressedBlocks{i} = idctTemp;

end


% We reconstruct the quantized blocks into an image
bIndex = 1;
for yIndex = 1:blocksDown
    for xIndex = 1:blocksAcross
        quantizedImage((8*yIndex)-7:(yIndex*8), (8*xIndex)-7:
(xIndex*8)) = quantizedBlocks{bIndex};
        bIndex = bIndex+1;
    end
end


subplot(2,2,2)
imshow(quantizedImage)
title(strcat('Quantized DCT of: ', fileName))
axis = gca;
axis.Visible = 'On';

% We reconstruct the compressed Image
bIndex = 1;
for yIndex = 1:blocksDown
    for xIndex = 1:blocksAcross
        compressedImage((8*yIndex)-7:(yIndex*8), (8*xIndex)-7:
(xIndex*8)) = compressedBlocks{bIndex};
        bIndex = bIndex+1;
    end
end

subplot(2,2,3)
imshow(holesImage)
title(strcat('Image before compression: ', fileName))
axis = gca;
axis.Visible = 'On';
% imwrite(holesImage, 'abc.png');

compressedImage255 = compressedImage/255;
```

```
subplot(2,2,4)
imshow(compressedImage255)
title(strcat('Image after compression: ', fileName, testString))
axis = gca;
axis.Visible = 'On';
% imwrite(compressedImage255, 'abc2.png');
```

Step 6: Encoding the image using Run-Length Ecoding

```
for i = 1:totalNumberOfBlocks
   compressedBlocks{i} = uint8(compressedBlocks{i});
end

% The algorithm for run-length encoding was modified and adapted from
 an
% implmentation that was found on the MathWorks File Exchange
 database. The
% following comment block is the license to use the code after being
% modified.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
% Copyright (c) 2011, Abdulrahman Siddiq
% All rights reserved.
%
% Redistribution and use in source and binary forms, with or without
% modification, are permitted provided that the following conditions
 are
% met:
%
%     * Redistributions of source code must retain the above copyright
%       notice, this list of conditions and the following disclaimer.
%     * Redistributions in binary form must reproduce the above
 copyright
%       notice, this list of conditions and the following disclaimer
 in
%       the documentation and/or other materials provided with the
 distribution
%
% THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 "AS IS"
% AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 TO, THE
% IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 PURPOSE
% ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 CONTRIBUTORS BE
% LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
% CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
% SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 BUSINESS
```

```
% INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
 IN
% CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 OTHERWISE)
% ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 OF THE
% POSSIBILITY OF SUCH DAMAGE.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%

% The code is modified for the use of this compression technique.

encodedValues = cell(1, totalNumberOfBlocks*8);
encodedCount = cell(1, totalNumberOfBlocks*8);
encoderCounter = 1;

for i = 1:totalNumberOfBlocks

    encodingBlock = double(compressedBlocks{i});

    for j = 1:8

        encodingRow = encodingBlock(j, 1:8);
        index = 1;
        encodedValues{encoderCounter}(index) = encodingRow(1);
        encodedCount{encoderCounter}(index) = 1;

        for k = 2:length(encodingRow)
            if (encodingRow(k-1) == encodingRow(k))
                encodedCount{encoderCounter}(index) =
 encodedCount{encoderCounter}(index)+1;
            else
                index = index + 1;
                encodedValues{encoderCounter}(index) = encodingRow(k);
                encodedCount{encoderCounter}(index) = 1;
            end

        end
        encoderCounter = encoderCounter + 1;

    end


end
```

Step 7: Introducing Errors

The error introduction is done in a completely random way. There are two options, which are both done on a bit level. The probability is based on a "coin flip" where should a random value should be chosen, that specific pixel will be affected.

```
list = {'Ones Compliment', 'Individual Bit Flip'};
```

```
[ErrorMode, rf] = listdlg('PromptString', 'Select a
 method', 'SelectionMode', 'single', 'ListString', list);

probInput = inputdlg('Choose the probability:', 'Enter the value for
 probability', [1 70]);
probInput = str2double(probInput);
probInput = (probInput/100) * 1000;
probInput = 1000 - probInput


% The first error mode does a 1s compliment of the chosen pixel.
% Essentially it takes the pixel value, converts to binary and using
 the ~
% on MATLAB automatically inverst the values. It is then converted
 back to
% a decimal number.
if (ErrorMode == 1)
    fprintf('Ones compliment chosen');
    for i = 1:length(encodedValues)
        sizeOfValue = length(encodedValues{i});
        probability = randi([1 1000]);
        if (probability > probInput)
            if (sizeOfValue == 1)

                temp = encodedValues{i};
                temp = decimalToBinaryVector(temp);
                invertedTemp = ~temp;
                invertedTemp = double(invertedTemp);
                encodedValues{i} =
 binaryVectorToDecimal(invertedTemp, 'MSBFirst');

            else
                position = randi([1 length(encodedValues{i})]);
                temp = encodedValues{i}(position);
                temp = decimalToBinaryVector(temp);
                invertedTemp = ~temp;
                invertedTemp = double(invertedTemp);
                encodedValues{i}(position) =
 binaryVectorToDecimal(invertedTemp, 'MSBFirst');

            end
        end

    end

% The second error mode is dependent on the random number generated
 between
% 1 and 8. this determines which bit will be flipped.
elseif (ErrorMode == 2)
    fprintf('Bit flip chosen');

    for i = 1:length(encodedValues)
        sizeOfValue = length(encodedValues{i});
        probability = randi([1 1000]);
```

```
        if (probability > probInput)
            if (sizeOfValue == 1)
                temp = encodedValues{i};
                temp = decimalToBinaryVector(temp);
                invertedTemp = temp;
                randomBit = randi([1 length(temp)]);
                invertedTemp(randomBit) = ~invertedTemp(randomBit);
                invertedTemp = double(invertedTemp);
                encodedValues{i} =
 binaryVectorToDecimal(invertedTemp, 'MSBFirst');

            else
                position = randi([1 length(encodedValues{i})]);
                temp = encodedValues{i}(position);
                temp = decimalToBinaryVector(temp);
                invertedTemp = temp;
                randomBit = randi([1 length(temp)]);
                invertedTemp(randomBit) = ~invertedTemp(randomBit);
                invertedTemp = double(invertedTemp);
                encodedValues{i}(position) =
 binaryVectorToDecimal(invertedTemp, 'MSBFirst');

            end
        end

    end

end
```

Step 8: Decoding the Image using Run-Length Decoding

```
% The algorithm for run-length decoding was modified and adapted from
 an
% implmentation that was found on the MathWorks File Exchange
 database. The
% following comment block is the license to use the code after being
% modified.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%
% Copyright (c) 2011, Abdulrahman Siddiq
% All rights reserved.
%
% Redistribution and use in source and binary forms, with or without
% modification, are permitted provided that the following conditions
 are
% met:
%
%     * Redistributions of source code must retain the above copyright
%       notice, this list of conditions and the following disclaimer.
```

```
%     * Redistributions in binary form must reproduce the above
 copyright
%       notice, this list of conditions and the following disclaimer
 in
%       the documentation and/or other materials provided with the
 distribution
%
% THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 "AS IS"
% AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 TO, THE
% IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 PURPOSE
% ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 CONTRIBUTORS BE
% LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
% CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
% SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 BUSINESS
% INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
 IN
% CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 OTHERWISE)
% ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 OF THE
% POSSIBILITY OF SUCH DAMAGE.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%

% The code is modified for the use of this compression technique.

decodedImage = cell(1, totalNumberOfBlocks);
decoderCounter = 1;

for i = 1:totalNumberOfBlocks

    for row = 1:8
        decodedRow = [];
        for j = 1:length(encodedValues{decoderCounter})
            decodedRow = [decodedRow encodedValues{decoderCounter}
(j)*ones(1,encodedCount{decoderCounter}(j))];
        end
        decoderCounter = decoderCounter + 1;
        decodedImage{i}(row, :) = decodedRow;
    end

end
```

Step 9: Filling in the holes

Filling the holes is based around the algorithm used to create the holes as well as research done on various techniques when holes were used. the idea is to fill the hole using surrounding blocks to get an image as accurate as possible. So when filling a specific hole pixel it uses the pixels directly above, directly left, and directly above-left of the hole pixel. It calculates the average of the 3 values and moves to the next pixel. Since the image does not transmit a specifc marker for the holes, it searches for the holes the same way it creates the holes.

```
filledImage = cell(1, totalNumberOfBlocks);
for i = 1:totalNumberOfBlocks
    filledImage{i} = double(decodedImage{i});
end


for q = 1:totalNumberOfBlocks

    block2x2 = filledImage{q}(4:5,4:5);

    average = mean(block2x2, 'all');

    counter = 1;
    for i = 1:2
        for j = 1:2
            temp1(counter) = pdist([block2x2(i,j);
 average], 'chebychev');
            counter = counter + 1;
        end
    end

    if (all(temp1 < 6))
        block4x4 = filledImage{q}(3:6, 3:6);

        average = mean(block4x4, 'all');

        counter = 1;
        for i = 1:4
            for j = 1:4
                temp2(counter) = pdist([block4x4(i,j);
 average], 'chebychev');
                counter = counter + 1;
            end
        end
        if (all(temp2 < 6))
            block6x6 = filledImage{q}(2:7, 2:7);
            average = mean(block6x6, 'all');

            counter = 1;
            for i = 1:6
                for j = 1:6
                    temp3(counter) = pdist([block6x6(i,j);
 average], 'chebychev');
                    counter = counter + 1;
                end
            end

            if (all(temp3 < 6))
                for rowPoint = 2:7
```

```
                    for colPoint = 2:7
                        filledImage{q}(rowPoint,colPoint) = ...
                            (filledImage{q}(rowPoint-1,colPoint-1) ...
                            + filledImage{q}(rowPoint-1,colPoint) ...
                            + filledImage{q}(rowPoint,colPoint-1))/3;
                    end
                end

            else
                for rowPoint = 3:6
                    for colPoint = 3:6
                        filledImage{q}(rowPoint,colPoint) = ...
                            (filledImage{q}(rowPoint-1,colPoint-1) ...
                            + filledImage{q}(rowPoint-1,colPoint) ...
                            + filledImage{q}(rowPoint,colPoint-1))/3;
                    end
                end
            end


        else
            for rowPoint = 4:5
                for colPoint = 4:5
                    filledImage{q}(rowPoint,colPoint) = ...
                        (filledImage{q}(rowPoint-1,colPoint-1) ...
                        + filledImage{q}(rowPoint-1,colPoint) ...
                        + filledImage{q}(rowPoint,colPoint-1))/3;
                end
            end

        end

    end

end


bIndex = 1;
for yIndex = 1:blocksDown
    for xIndex = 1:blocksAcross
        reconstructedImage((8*yIndex)-7:(yIndex*8), (8*xIndex)-7:
(xIndex*8)) = filledImage{bIndex};
        bIndex = bIndex+1;
    end
end

reconstructedImage255 = reconstructedImage/255;

figure('units','normalized','outerposition',[0 0 1 1])
imshow(reconstructedImage255)
title(strcat('Reconstructed Image with 50% Error Probability (Bit
 Flip): ', fileName))
axis = gca;
axis.Visible = 'On';
```

```
%imwrite(reconstructedImage,'CompressedImage.gif');

toc

CompressionRatio
```

Published with MATLAB® R2019a