Florida State University Libraries

Electronic Theses, Treatises and Dissertations

The Graduate School

2017

Fixed-Point Implementation of Discrete Hirschman Transform

Rajesh Thomas

FLORIDA STATE UNIVERSITY
FAMU-FSU COLLEGE OF ENGINEERING

FIXED-POINT IMPLEMENTATION OF DISCRETE HIRSCHMAN TRANSFORM

By
RAJESH THOMAS

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science.

2017

Rajesh Thomas defended this thesis on November 17, 2017.

The members of the supervisory committee were:

Victor DeBrunner

Professor Directing Thesis

Linda DeBrunner

Committee Member

Bruce Harvey

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

In dedication to my caring parents, loving brother
and dear grandmother.

ACKNOWLEDGMENTS

I take this opportunity to acknowledge and thank everyone who were of great inspiration for me and whose support helped me complete this thesis.

First of all, I sincerely thank my advisor Dr. Victor DeBrunner for his inspiration, guidance and support all throughout my academic career in Florida State University. I extend my thanks to professors Dr. Linda DeBrunner and Dr. Bruce Harvey for being members of my supervisory committee and for their valuable feedback.

I convey my special thanks to the Florida State University and the Department of Electrical Engineering in the FAMU-FSU College of Engineering for providing me with the educational opportunity and resources throughout my Master's education. At this juncture, I express my gratitude to Ms. Melissa Jackson for her help and support from the very first day. I also thank Dr. Petru Andrei, Mr. Donte Ford and all my faculty and teachers for their help, guidance and knowledge provided to me.

I thank my parents and brother for their encouragement, guidance and wisdom throughout my life, especially during my Master's education.

TABLE OF CONTENTS

List of Figures	vi
Abstract	vii
1. INTRODUCTION	1
2. BACKGROUND STUDY	2
2.1 Discrete Fourier Transform.....	2
2.2 Hirschman Optimal Transform	5
2.3 Fixed-Point Arithmetic	6
3. DISCRETE HIRSCHMAN TRANSFORM	7
3.1 Coding.....	7
3.1.1 Sine and Cosine Calculation	8
3.1.2 DHT Computation.....	13
3.1.3 Inverse DHT.....	20
4. HIRSCHMAN COSINE TRANSFORM.....	21
5. RESULTS AND DISCUSSION	24
5.1 Discrete Hirschman Transform.....	25
5.2 Hirschman Cosine Transform	31
6. CONCLUSION AND FUTURE WORK	38
APPENDICES	39
A. C CODES FOR DHT AND IDHT.....	39
B. MATLAB CODES FOR DHT AND IDHT	47
C. C CODES FOR HCT AND IHCT	49
D. MATLAB CODES FOR HCT AND IHCT	57
E. MATLAB CODES FOR TESTING	59
References	63
Biographical Sketch	64

LIST OF FIGURES

1	Waveform of the D note produced by a flute. The plot is zoomed-in along the time axis in order to show the detailed shape of the wave.	3
2	Fourier Transform of a flute's sound	4
3	Signs of trigonometric functions in the four quadrants.	12
4	Points around the unit circle for a 12-point DFT	12
5	Error in the twiddle factors in dB. The graph on the left is zoomed-in and shown on the right to highlight the error for DFT lengths from 8 to 128.....	25
6	Mean Square Error of the Discrete Hirschman Transform in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.....	26
7	Mean Square Error between the processed image and the original image in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.....	27
8	Median Error of the pixel values after computing the forward and inverse DHT.....	28
9	Maximum Absolute Error of the pixel values after taking the forward and inverse DHT.	28
10	Visual comparison on the 'cameraman.tif' image. The image on the bottom was processed using DHT and IDHT with $K = 128$ and $L = 2$	29
11	Visual comparison on the 'office.tif' image. The image on the bottom was processed using DHT and IDHT with $K = 128$ and $L = 4$	30
12	Error in the transform matrix elements of HCT.	32
13	Mean Square Error of the HCT in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.	32
14	Mean Square Error between the processed image and the original image in dB.	33
15	Median Error of the pixel values after computing the forward and inverse HCT.	34
16	Maximum Absolute Error of the pixel values after taking the forward and inverse HCT.	34
17	Visual comparison on the 'cameraman.tif' image. The image on the bottom was processed using HCT and IHCT with $K = 128$ and $L = 2$	35
18	Visual comparison on the 'office.tif' image. The image on the bottom was processed using HCT and IHCT with $K = 128$ and $L = 4$	36

ABSTRACT

Digital Signal Processing (DSP) performs a very important role in various applications of electrical engineering like communications and signal enhancement. In many situations one finds that the DSP hardware available are fixed point processors. In these situations, it is necessary to perform DSP with high accuracy using the least amount of hardware resources. This thesis looks into an approach to calculate the two dimensional Discrete Hirschman Transform (DHT), the inverse DHT, the Hirschman Cosine Transform (HCT) and the inverse HCT using fixed-point hardware. The complex coefficients required for the transform are calculated beforehand and saved as vectors. Special attention has been given to minimize errors due to scaling. The processed image and the original image does not show significant difference even for DFT or DCT length of 128. Mean square errors of -37 dB for the DHT and -40 dB for the HCT could be obtained for DFT and DCT lengths of 128.

CHAPTER 1

INTRODUCTION

Signal processing is concerned with the representation, transformation and manipulation of signals and the information they contain [12]. Transforms like the Discrete Fourier Transform (DFT) are mathematical tools which aids in signal processing. DFT is very important in various applications like communications, noise reduction, data compression and sound, radar and image processing [5][6]. Therefore, the accurate and efficient computation of DFT also holds a prominent role while implementing Digital Signal Processing (DSP) for real world applications. Thus, various techniques and algorithms like the Fast Fourier Transform (FFT) exist to aid DFT computation. Similarly, other transforms like the Discrete Cosine Transform (DCT) and Wavelet Transform are used in various applications. One such transform is the Hirschman Optimal Transform (HOT) introduced in [1].

This thesis focuses on the computation of two dimensional Discrete Hirschman Transform (DHT) using fixed point arithmetic. The DHT is a generalization of the HOT and can be computed with the aid of DFT. Fixed point computing is important because it is computationally less demanding compared to the floating-point arithmetic. This enables better performance on low-power and portable digital signal processors like the MSP430 microcontroller board from Texas Instruments. Moreover, some work has also been put in developing the computation of the Hirschman Cosine Transform (HCT) which is based on the DCT.

The C programming language is the language of choice in this thesis. The C programming language offers various benefits like use of pointers, simplicity in programming and support on various platforms and DSP chips. In C, the fixed-width integer data type `int16_t` can be used to store the 16 bit numbers. The programs have been developed and tested using the MatLab software. The programs used in this thesis are given in the appendix.

CHAPTER 2

BACKGROUND STUDY

The roots of the Discrete Hirschman Transform (DHT) can be found in the Discrete Fourier Transform (DFT). The DFT has a very long a rich history which is discussed in [9]. Additionally, from [8], it is clear that a particular version of the Fast Fourier Transform (FFT) algorithm was discovered by the famous German mathematician Carl Friedrich Gauss around the year 1805. He used this algorithm to interpolate the measurements of the orbits of the asteroids Juno and Pallas. However, this FFT algorithm is often attributed to the modern re-discoverers Cooley and Tucky [7]. Because of the relationship DHT has with DFT [1], we shall first focus our attention on the DFT.

2.1 Discrete Fourier Transform

The Fourier Transform is a mathematical tool that provides an alternate representation of a signal in terms of its frequency content. Almost all signals can be represented in the form of a waveform - sound waves, stock exchange graphs, a plot of atmospheric pressure versus distance from a point, etc. It is also a fact that every signal can be represented as a sum of sinusoidal functions of varying frequency and amplitude. The Fourier Transform allows us to see the amplitude and frequency of these sinusoids. This is often called as the frequency domain representation of a signal, and the original signal is said to be in the time domain. Often it is more helpful to view a signal in the frequency domain. Thus, the Fourier Transform is roughly equivalent to splitting a signal into its constituent parts.

For example, if an alien is trying to learn about ice cream, he might find it helpful to learn about the constituents that make an ice cream. Hypothetically, if the alien were to perform a Fourier Transform on ice cream, he would find its ingredients, namely milk, cream, sugar, flavoring, coloring and other additives along with their concentrations. With this data, it will be easier for the alien to conduct his research on ice cream. Similarly, if we perform a Fourier Transform on the note of a flute, it will show the frequencies present in the note, along with their relative strength.

Figure 1 shows part of the sound waveform of a musical D note produced by a flute. It is quite difficult to obtain a lot of useful information from the waveform except the approximate time period and fundamental frequency of the wave. From the figure, one can observe that the wave repeats almost every 0.002 seconds. This makes the fundamental frequency slightly greater than 500 Hz (frequency is the inverse of the time period). We can also see that the waveform approximately resembles a sinusoidal function. The information provided by the fundamental frequency does not account for the richness of the flute's music. Figure 2 shows the Discrete Fourier Transform of the flute's sound. (represented by Figure 1). The plot shown in Figure 2 is symmetrical about the zero Hertz line and for the time being we only need to consider the positive frequencies. It is seen that there is a very strong presence of a frequency at 586 Hz which is the exact fundamental frequency and is close to the approximate value we obtained earlier. There are also other frequencies present which are integer multiples of the fundamental frequency (called harmonics), but with diminishing magnitude. The presence of all these frequencies create the rich sound we hear from a musical instrument. Thus, the Fourier Transform enabled us to gain more information about the sound produced by a flute - in particular the frequencies present and their magnitudes. The magnitude shows the relative strength of the frequency

Figure 1: Waveform of the D note produced by a flute. The plot is zoomed-in along the time axis in order to show the detailed shape of the wave.

Figure 2: Fourier Transform of a flute's sound

The continuous time Fourier Transform and the Inverse Fourier Transform in one dimension can be calculated using the equations Eq. (1) and Eq. (2) respectively [4]. The Inverse Fourier Transform is used to get back the time domain signal from the frequency domain signal. Here $\Omega = 2\pi f$, where f is the frequency and $X(\Omega)$ is the transform of the time domain signal $x(t)$.

$$X(\Omega) = \int_{t=-\infty}^{\infty} x(t)e^{-j\Omega t} dt \quad (1)$$

$$x(t) = \frac{1}{2\pi} \int_{\Omega=-\infty}^{\infty} X(\Omega)e^{j\Omega t} d\Omega \quad (2)$$

In practice, almost all signals used today are discrete time signals. Therefore, the Discrete Fourier Transform (DFT) and the Inverse Discrete Fourier Transform (IDFT) are used, which deals with discrete signals both in the time and frequency domains. The DFT and IDFT in one dimension are given by the equations (3) and (4) respectively. In these formulae, N is the number of samples present in each of the signals $x(n)$ and $X(k)$. The transform in Figure 2 was obtained using DFT.

$$X(k) = \text{DFT}(x(n)) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn} \quad (3)$$

$$x(n) = \text{IDFT}(X(k)) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k) e^{j\frac{2\pi}{N}nk} \quad (4)$$

2.2 Hirschman Optimal Transform

Entropy is the most fundamental measure in information theory [10]. Some signals have equal entropies in the time signal and its Fourier transform. In [1], the authors have identified these signals and have found that they form an orthonormal basis which gives rise to an orthogonal transform. This transform can optimally pack a finite-duration discrete-time signal, called Hirschman Optimal Transform (HOT). The HOT for one dimensional signals is given by Eq. (5) and the inverse transform is given by Eq. (6). These equations have been derived in [1]. The length of the signal is N and $N = K^2$.

$$X(Kr + L) = \frac{1}{\sqrt{K}} \sum_{n=0}^{K-1} x(Kn + L) e^{-j\frac{2\pi}{K}nr} \quad , 0 \leq r, L \leq K - 1 \quad (5)$$

$$x(Kn + L) = \frac{1}{\sqrt{K}} \sum_{r=0}^{K-1} X(Kr + L) e^{j\frac{2\pi}{K}nr} \quad , 0 \leq n, L \leq K - 1 \quad (6)$$

On a given input, HOT requires a smaller length of DFT when compared to the actual DFT. The DFT on a signal of length N requires a DFT length of N . On the other hand, the HOT of an input of length N only requires a DFT length of N . Moreover, as given in [11], HOT performs better than DFT and DCT (Discrete Cosine Transform) on the ability to separate frequencies that are close to each other and to resolve additive white noise. Furthermore, in spectral estimation, FIR (Finite Impulse Response) convolution and compressed sensing, HOT has better performance when compared to DFT [2]. Also, on Field Programmable Gate Arrays (FPGAs) simulations show that HOT has superiority over DFT in terms of computational speed, space requirements and rounding error performance [2].

2.3 Fixed-Point Arithmetic

Fixed-point arithmetic is where the number of digits before and after the radix point (decimal point) is fixed by the programmer. This is in contrast with floating point arithmetic where the position of the radix point can change depending on the value of the number represented. Fixed-point computing is often preferred in mobile DSP applications when the power source is a battery pack. Fixed-point processors have simpler logic circuits when compared with floating-point processors [3]. Thus, fixed-point processors are cheaper, smaller and consume lesser power. The Nintendo DS hand-held gaming console used fixed-point processing. Programs written using fixed-point code often run faster with fewer number of processor clock cycles. Moreover, the round off errors are more predictable and offer greater control to the programmer in fixed-point arithmetic.

The disadvantages of using fixed-point arithmetic include greater time and effort to develop the code. Moreover, floating-point data types offer greater dynamic range with equivalent number of bits. Therefore, proper scaling should be done to avoid overflow and quantization errors [3]. However, in many circumstances, the advantages outweigh the disadvantages.

In this thesis 16 bit fixed-point numbers are used. The fixed width integer data type `int16_t` is used to store these fixed-point numbers. Out of the 16 bits, one bit is used to represent the sign of the number and the rest 15 bits are used for storing the magnitude.

CHAPTER 3

DISCRETE HIRSCHMAN TRANSFORM

The Discrete Hirschman Transform (DHT) is a generalization of the Hirschman Optimal Transform. The dimensions of the input need not be a perfect square number. Additionally, the number of sampling points and the up-sampling rate are not required to be equal. When the up-sampling rate is one, the DHT equals the DFT and when the up-sampling rate equals the length of the input, the result obtained after the transform equals the input.

3.1 Coding

Techniques used for computing DFT can be used for DHT, since they are related. This idea has been explored in [1]. The fundamental idea that needs to be remembered while developing the code is to minimize memory usage and computational load.

The first step is to make sure that the user has made the right inputs. Since the user has the flexibility to choose the up-sampling rate (L) and the number of sampling points (K) in the DFT, the size of the input square matrix (N) must be equal to the product of the up-sampling rate and the number of sampling point (i.e., $N = K \times L$). Moreover, regarding the code developed for this thesis, further restrictions have been imposed for improving performance. The first restriction is that the input must be a square matrix (i.e., equal number of rows and columns). Secondly, the minimum value of K must be 4, and K must be a multiple of 4.

Moreover, in this thesis, the fixed-point computations are done using 16 bit signed integers. The question can be asked why 16 bits were chosen. Most DSP hardware available today use 16 bit processors. Also, 16 bit computations are preferred because the precision obtained from 8 bits is often too less and 32 bit computations require more hardware resources. Thus since 16 bits lie in between, it has the benefit from both extremes. Additionally, using 8 bits would not be practical for higher DFT lengths like 64 and 128 since the 8 bit representation of the angles around the unit circle would have values that are too close to each other.

3.1.1 Sine and Cosine Calculation

The next step is to calculate and store the sine and cosine of the angles required for DFT computation and store them in arrays. This trade-off has the advantage of lesser computational load in calculating the sines and cosines at the expense of taking up a little memory for storing the sine and cosine vectors. The length of each vector is the same as the number of sampling points in the DFT, or in other words, the length will be ‘K’ for a K-point DFT. The angles used for the computation will be $m * 2\pi/K$ where m ranges from 0 to $K - 1$. The sines and cosines are calculated by calling the function `sin_calc()` in the code to perform the task. This function uses the Taylor series expansion for sine which is given in Eq. (7). The cosine values are calculated using the formula $\cos(x) = \sin(\pi/2 - x)$.

$$\sin(x) = \sum_{i=0}^m \frac{(-1)^i x^{2i+1}}{2i+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{(-1)^m x^{2m+1}}{(2m+1)!} \quad (7)$$

In this formula, the larger the value of m , the more accurate the value of $\sin(x)$ will be. But since the focus is on fixed point computation, it is not necessary to calculate beyond a certain number of terms, given that each successive term in the above formula contributes less compared to its previous term. To further improve the accuracy of the calculation, x is kept as small as possible. This is done by performing computations such that x will have values only between 0 and $\pi/2$. The general code for computing $\sin(x)$ using n bit signed integers is given in Algorithm 1.

It can be found that only five terms are required to calculate $\sin(x)$ if $0 < x < \pi/2$ when the calculation is done using 16 bit signed integers. The contribution from the sixth term onward is zero. The sixth term from Eq. (7) is $x^{11}/11!$. As a 16 bit signed integer, this term can be written as given in Eq. (8).

$$t_6 = \frac{x^{11}}{11!} \times 16384 \quad (8)$$

Substituting $x = \pi/2$ in Eq. (8), one gets Eq. (9).

$$t_6 = \frac{(\pi/2)^{11}}{11!} \times 16384 = 0.059 \quad (9)$$

Since only the whole number part is considered, it can be seen that the contribution from the sixth term is zero. Higher terms yield smaller values than t_6 . From a programming stand point, only three terms are required if x is less than $\pi/4$, and four terms for values above that. If x is greater than 80° (or $4\pi/9$ rad), the contribution from the fifth term turns out to be approximately one. So, one is added to the calculated value if $x > 80^\circ$.

In the code, the Taylor series formula is implemented in such a way that overflow is avoided by following a multiplication with a division as shown in Algorithm 1. Thus, most of the divisions performed here serve the purpose of bit shifts to keep bit growth in check. After substituting $n = 16$ in Algorithm 1, one gets the code given in Algorithm 2, which is used for calculating the sine of an input. It should be noted that the code given in Algorithm 2 has been optimized to run specifically for 16 bit data.

This paragraph explains the working of the code given in Algorithm 2. As an example, take $x^3/3!$ which is the second term in Eq. (7). For the fixed-point implementation used in the code, x and the variable storing $\sin(x)$ are 16 bit signed integer values. Thus, it is implied that there is one sign bit and 15 magnitude bits. The C code for computing the second term is $x * x / 32768 * x / 8192 / 6$. In C, arithmetic operations are evaluated from left to right. The result of the multiplication $x * x$ is a 30 bit value. But this result is immediately divided by 32,768, which is the same as right shifting by 15 bits since $2^{15} = 32,768$. Thus, after performing $x * x / 32768$, the result is 15 bits, which is again multiplied by the 15 bit value x . This brings the result back to 30 bits. The resultant is divided by $8192 = 2^{13}$ and is further divided by 6, which is a 3 bit value. So now the net result is under 16 bits and can be stored as a 16 bit signed integer. The final number 6 is obtained directly from the formula where the denominator of the third term is $3! = 6$. The 32,768 and 8192 are actually two 16,384s (2^{14}). This is to compensate for the bit growth introduced by x . It can be asked why specifically 2^{14} was chosen. The answer is explained in the following paragraph. In short, the reason is because the largest value that x will have is $25,736$ which is equal to $2^{14} * \pi/2$.

Algorithm 1: Generalized code for finding the sine of a value using n bits.

```
Declare  $n$  bit integers  $x$ ,  $sum$  and  $temp$ ;  
Initialize  $x$  with the angle whose sine is to be calculated in  $n$  bits;  
Initialize variable  $c = 6$ ;  
 $temp = x * x / 2^n * x / 2^n * x / 2^{n-1} * x / 2^{n-3} / 120$ ;  
 $sum = x - x * x / 2^n * x / 2^{n-2} / 6 + temp$ ;  
Repeat until  $temp$  equals zero  
{  
     $temp = -temp * x / 2^{n-1} * x / 2^{n-1} / c / (c + 1)$ ;  
     $sum = sum + temp$ ;  
     $c = c + 2$ ;  
}  
Print the value of  $sum$  as the sine of the input  $x$ ;
```

Algorithm 2: Function `sin_calc()` used for calculating sine and cosine of an input.

```
int16_t sin_calc(int16_t x)  
{  
    int16_t sine, temp;  
    temp = x*x / 32768 * x / 32768 * x / 16384 * x / 4096 / 120;  
    sine = x - x*x / 32768 * x / 8192 / 6 + temp;  
  
    if (x > pi_by2 / 2) // 45 degrees  
    {  
        temp = -1 * temp * x / 16384 * x / 16384 / 42;  
        sine = sine + temp;  
  
        if (x > pi_by2 * 8/9) // 80 degrees  
            sine = sine + 1;  
    }  
    return sine;  
}
```

The constant π plays an important role in finding the angles used for the sine and cosine computation, because all angles used in the transform are a fractional multiple of π as shown in Figure 4. The fixed-point value of $\pi/2$ is stored as a 16 bit signed integer, and the largest number that can be stored in such a memory is 32,767 (which is equal to $2^{15} - 1$). Since $\pi/2 = 1.5708$, one bit is required to store the whole number part (= '1'), and the rest of the 14 bits in the signed integer can be used to store the fractional part (= 0.5708). Thus, in order to fit

$\pi/2$ in this memory, one has to perform $(\pi/2) \ll 14$, where the operator $\ll$ represents bitwise left shift. This operation is the same as doing $(\pi/2) * 2^{14}$ and is equal to 25,736. In the same manner, 1.0 will equal $1.0 * 2^{14} = 16,384$. This explains the reason why 2^{14} was used to avoid bit growth. Also, since the range of a sinusoid is between -1 and +1, in the fixed-point code the range is between -16,384 and 16,384.

Initially during the code's development, π was represented in 16 bits as 25,736 which is obtained by rounding $\pi * 2^{13}$ to the nearest integer. Here two bits are required for representing the whole number part (i.e., 3) and the rest of 13 bits for the fractional part in π (i.e., 0.14159...). Using this logic, the value of '1' would have been 8192 which is equal to 2^{13} and all values of sine and cosine will range between -8192 and 8192. In this circumstance if we assume that the length of the DFT is a multiple of four, we can gain one more bit by representing $\pi/2$ as 25,736, which is currently used in the code.

As mentioned previously, in the code, the function `sin_calc()` is used for calculating the sines and cosines of angles. This function takes a 16 bit signed integer value as the input argument, and returns another 16 bit signed integer as output. For Eq. (7) to converge fast using the least number of terms, the value of x should be as small as possible. Using various trigonometric formulae, the range of x can be kept between 0 and $\pi/2$. It can be observed that for the elements in the DFT matrix, the points on the unit circle are symmetrical on either side of the real axis. Thus, for values between π and 2π , the elements between 0 and π are mirrored after changing the sign for sines. Figure 3 shows the signs of trigonometric functions in the various quadrants. If a restriction is imposed such that the length of the DFT is a multiple of four, one more symmetry can be found about the imaginary axis. In these circumstances, the sines and cosines need to be calculated only for the first quadrant of the complex plane. For the second quadrant (i.e., for angles between $\pi/2$ and π), the sine and cosine values are mirrored but the cosines are taken with the opposite sign (Figure 3). It has already been explained above for values in the third and fourth quadrants. Figure 4 shows the plot for a 12-point DFT around the unit circle where the angle between consecutive points with respect to the origin is $2\pi/12 = \pi/6$ rad = 30° . As an example, for the point whose angle is $2\pi/12$ from the positive real axis, the complex value will be $e^{j2\pi/12} = \cos(2\pi/12) + j \sin(2\pi/12) = 0.866 + j 0.5$. As already explained, using a 16 bit signed integer data type, the complex value will be multiplied by 2^{14} .

Figure 3: Signs of trigonometric functions in the four quadrants.

Figure 4: Points around the unit circle for a 12-point DFT

For calculating the cosine value, the `sin_calc()` function itself is used. The trigonometric formula $\cos(x) = \sin(\pi/2 - x)$; $\forall x \in (0, \pi/2)$ is used to calculate the cosine values using the sine function. But in practice, as a 16 bit signed integer in the code, it is found that the formula $\cos(x) = 1 + \text{sin_calc}(\pi/2 - x - 8)$ yields better results. Also, if x is more than 40° , 5 is added to the value of $\cos(x)$. This modified formula was found by trial and error method.

3.1.2 DHT Computation

After obtaining the sine and cosine vectors, the next step is the calculation of DHT. The DHT computation is usually applied for images. Since 8-bit images have pixel values between 0 and 255, the average value of pixels will have a positive value which can lead to bit overflow. Therefore, the first step is to eliminate this positive bias by subtracting 128 from the input image. Thus, the input now has pixel values between -128 and 127, which enables the average to be centered around zero. When the inverse DHT is taken, 128 is added to the final result to get back the original image.

Similar to the 2D Fourier Transform, a two-dimensional DHT computation can be thought of as doing two separate 1D DHT computations. This is done by passing the row or column of the input to a function that calculates the DHT of a one-dimensional array. In the code, this function is named `dht_1D()`. First, columns are extracted from the input matrix and sent to the function. Next the rows are extracted from the new matrix and are again sent to the function as 1D arrays to obtain the output matrix. This completes the 2D DHT of the input matrix. This logic is derived from the transform equation $Y = H \cdot X \cdot H^T$, where Y is the transform of the input matrix X , H is the transformation matrix and H^T is the transpose of H . On analyzing the equation, it can be seen that the row-wise and column-wise operations are done separately on the input matrix. If the first multiplication is given as $Y_1 = H \cdot X$, it's seen that the rows of H are multiplied by the columns of X . In other words, Y_1 contains the 1D DHT of the columns of X . The same process is repeated for the rows of Y_1 using $Y = Y_1 \cdot H^T$ to get the final result Y .

As mentioned above, the function to calculate the DHT of 1D arrays is used to find the DHT of the whole input matrix. Since the DHT transform matrix is closely related to the DFT transform matrix, the elements from the DFT matrix are used to calculate the DHT. The DFT transform matrix of size 3 is given in Eq. (10).

$$W = \begin{bmatrix} 1 & 1 & 1 \\ 1 & e^{-j\frac{2\pi}{3}} & e^{-j\frac{4\pi}{3}} \\ 1 & e^{-j\frac{4\pi}{3}} & e^{-j\frac{8\pi}{3}} \end{bmatrix} \quad (10)$$

The DHT transform matrix for a 3-point DFT with an up-sampling of 3 is given in Eq. (11) which can be used to calculate 9-point DHT. Instead of using this matrix, this 9-point DHT can also be calculated using just the DFT with the set of equations in Eq. (12). Here y_i represents the outputs and x_i represents the inputs. The inputs and outputs have 9 elements.

$$W = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 \\ 0 & 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 \\ 0 & 0 & 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} \\ 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} & 0 & 0 \\ 0 & 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} & 0 \\ 0 & 0 & 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} \end{bmatrix} \quad (11)$$

$$\begin{aligned} \begin{bmatrix} y_0 \\ y_3 \\ y_6 \end{bmatrix} &= \text{DFT} \left(\begin{bmatrix} x_0 \\ x_3 \\ x_6 \end{bmatrix} \right) \\ \begin{bmatrix} y_1 \\ y_4 \\ y_7 \end{bmatrix} &= \text{DFT} \left(\begin{bmatrix} x_1 \\ x_4 \\ x_7 \end{bmatrix} \right) \\ \begin{bmatrix} y_2 \\ y_5 \\ y_8 \end{bmatrix} &= \text{DFT} \left(\begin{bmatrix} x_2 \\ x_5 \\ x_8 \end{bmatrix} \right) \end{aligned} \quad (12)$$

The relationship between DHT and DFT can be seen by observing Eq. (10) and Eq. (11). The length of the DFT determines the elements of the DHT transform matrix, and the up-sampling rate determines the spacing between the non-zero elements. As mentioned previously, for the Discrete Hirschman Transform of an input matrix of size N , using a K -point DFT with an upsampling rate of L , the relationship $N = K \times L$ must be satisfied. For the example given in Eq. (11), $K = 3$ and $L = 2$. Therefore, the elements are similar to the 3-point DFT transform matrix (since $K = 3$), and the number of zeros between consecutive non-zero elements is $L-1$ (which

equals 2 in Eq. (11)). If the up-sampling rate was 2, there would have been one zero between each non-zero element and the size of the matrix H would have been 6×6 .

For DFT of length K , every element in the transformation matrix will be the complex exponential of an angle which is an integer multiple of $-2\pi/K$. Generalizing, if the twiddle factor ω is defined as $\omega = e^{-j\frac{2\pi}{K}}$, the Fourier Transform matrix (also known as twiddle factor matrix) for length K is shown in Eq. (13). The elements of the twiddle factor matrix are the K^{th} complex roots of unity. Due to the periodicity of sinusoidal functions, the complex exponentials are also periodic with a period 2π , which is also evident from Figure 4. Therefore, every angle used in Eq. (13) can be found in the set of angles $2\pi m/K \ \forall m \in [0, K-1]$, where m is an integer. As an example, for $K = 4$ the DFT matrix is given in Eq. (14), where $\omega = e^{-j\frac{2\pi}{4}}$.

$$W = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \dots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & & \omega^{K-1} \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & & \omega^{2(K-1)} \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & & \omega^{3(K-1)} \\ \vdots & & & & \ddots & \vdots \\ \omega^0 & \omega^{K-1} & \omega^{2(K-1)} & \omega^{3(K-1)} & \dots & \omega^{(K-1)(K-1)} \end{bmatrix} \quad (13)$$

$$W = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^0 & \omega^2 \\ \omega^0 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \quad (14)$$

Thanks to this property, while coding, instead of having a transformation matrix of size $K \times K$, we only need a one-dimensional array to store K values. In practice, since the elements in the transformation matrix are complex values, we need two arrays each of length K , one for storing the real values and the other for storing the imaginary values. In this article, we shall call these arrays of length K as transformation arrays. In the code the transformation arrays are `co[]` and `si[]` which are respectively used for storing the cosine (real) and sine (imaginary) values of the transformation matrix. In the Discrete Hirschman Transform matrix, at least half of the elements will be zeros (except in the circumstances where the up-sampling rate is one, which then becomes

equivalent to DFT). While calculating the transform, the multiplications involving the zero elements can be avoided.

In order to obtain these benefits, the elements from the transformation arrays are to be called in the right order so as to mimic the elements of the transformation matrix. From Eq. (14), it can be seen that the exponent of the elements of the second matrix is the remainder obtained after dividing the exponents of the elements of the first matrix by four. In the C programming language, the modulo operator (%) is used to find the remainder. Generalizing using this technique, Eq. (13) can also be written as given in Eq. (15). Since the elements of the DFT transformation matrix for a given value of K are known, one can use this knowledge to find the index of the corresponding element in the transformation array using the modulo operator.

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{1\%K} & \omega^{2\%K} & \omega^{3\%K} & & \omega^{K-1} \\ 1 & \omega^{2\%K} & \omega^{4\%K} & \omega^{6\%K} & & \omega^{K-2} \\ 1 & \omega^{3\%K} & \omega^{6\%K} & \omega^{9\%K} & & \vdots \\ \vdots & & & & \ddots & \omega^2 \\ 1 & \omega^{K-1} & \omega^{K-2} & \dots & \omega^2 & \omega^1 \end{bmatrix} \quad (15)$$

For the DHT transformation matrix, there might be a few zeros between the elements. The code can be made to run faster by not using the elements that correspond with zeros in the transformation matrix. On studying the example given in Eq. (16), this point can be made more clear. This equation is of the form $Y = HX$, where we are finding the one dimensional Discrete Hirschman Transform Y , of the vector X using the transformation matrix H . For calculating y_0 , we only need x_0, x_3 and x_6 . Likewise, for y_4 we only need the inputs x_1, x_4 and x_7 because the other elements are multiplied with zeros. The equations in Eq. (12) also show this. Thus, while writing the C code, only the elements that do not correspond to zeros are used in the computation.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 \\ 0 & 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 \\ 0 & 0 & 1 & 0 & 0 & e^{-j\frac{2\pi}{3}} & 0 & 0 & e^{-j\frac{4\pi}{3}} \\ 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} & 0 & 0 \\ 0 & 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} & 0 \\ 0 & 0 & 1 & 0 & 0 & e^{-j\frac{4\pi}{3}} & 0 & 0 & e^{-j\frac{8\pi}{3}} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} \quad (16)$$

Moreover, it should also be noted that these computations are done on complex numbers. Therefore, the computation for the real and imaginary parts are to be done separately in the C code. For instance, consider the complex multiplication in Eq. (17).

$$Z = (a + jb)(c + jd) \quad (17)$$

$$= (ac - bd) + j(ad + bc) \quad (18)$$

From Eq. (18), it is implied that the real part, $\text{real}(Z) = ac - bd$ and the imaginary part, $\text{imag}(Z) = ad + bc$. In the code, one matrix is used to store the real values and another to store the imaginary values. While performing these mathematical operations, the program does not distinguish between real and imaginary numbers [5]. The concept of real and imaginary numbers exists in the mind of the programmer, and the values in the matrices are used accordingly when required for a particular computation. The lines of code in the inner for loop given in the code snippet in Algorithm 3 performs multiplication on complex numbers.

Furthermore, in the code, all values represented in Eq. (16) are 16 bit signed integers with 15 magnitude bits. When two 15 bit integers are multiplied the result will have a size of 30 bits. Since the input X and the transformation matrix H are 15 bit values, each element of the output array Y needs to be 30 bits long (which can be stored using 32 bit signed integers). This is called bit growth. However, using 32 bit signed integers require more memory to store the matrix and also require more computational resources for further processing. Moreover, if bit growth is allowed while performing these multiplications on two dimensional matrices, the output might

lead to 64 bit numbers. Therefore, to restrict bit growth, the output matrix Y must also to be 16 bits long. Thus, it follows that the product of two 15 bit numbers ought to be divided by 2^{15} so that the result is also 15 bits. In addition, one has to understand that the H matrix actually contains sines and cosines and the greatest absolute value of any given element in H will be one. However, as explained before, 14 bits are used to represent the sine and cosine elements in the code. Therefore, in the code, the greatest value of any element in H will be 2^{14} (which is equivalent to the floating point 1.0). Thus, these 14 bits needs to be eliminated in order to make the matrices X and Y the same bit length. As a result, the product of H and X is divided by $2^{14} = 16384$. In this circumstance, since the number 2^{14} is used to scale the product due to memory constraints, it is called as the scaling factor.

For instance, consider the following example. Consider the multiplication $\cos(\pi/3) * 120$ which yields the answer 60. In the code, $\pi/3$ in 16 bits is represented as 17,157 which is obtained from $25,736 * 2/3$. It has been already explained that 25,736 is the 16 bit representation of $\pi/2$ and as it is known $(\pi/2) * (2/3) = \pi/3$. From the code, we get the cosine of $\pi/3$ as 8187. In order to avoid bit growth, the computation $\cos(\pi/3) * 120$ is given in Eq. (19) which uses the scaling factor. The result obtained from Eq. (19) is very close to the actual result of 60. If the scaling factor of 16,384, is not used, the result can get very huge and may not leave margins for performing further computations. These tiny differences are difficult to avoid and lead to the errors discussed in the 'Results' section.

$$\begin{aligned} \cos(\pi/3) * 120/16384 &= 8187 * 120/16384 \\ &= 59 \end{aligned} \tag{19}$$

However, in practice, it has been found that the error in the result can be reduced by using a smaller scaling factor. Experimentally, it is found that a scaling factor of $2^{12} = 4096$ produces a much better result compared to 2^{14} for values of $K \in \{4,8,16,32,64,128\}$. For greater values of K , the scaling factor must also be increased. Likewise, for lesser values of K , processing can be done with a smaller scaling factor. Thus, there is a relationship between the length of the DFT and the scaling factor. Using a smaller scaling factor can avoid under-utilization of memory. This could partly be due to the fact that the inputs used for testing this code are 8-bit images.

Algorithm 3: Function dht_1D() used for calculating 1-dimensional DHT of an input

```
void dht_1D(int16_t *xre, int16_t *xim, int16_t *tre, int16_t *tim, int16_t div) // x is input, t is output
{
    int16_t i, j, p, r, theta;
    for (i = 0; i < n; i++)
    {
        tre[i] = 0; // initializing the output matrix to be 0
        tim[i] = 0;
        r = i/l;
        p = i%l;
        for (j = 0; j < k; j++) // loop performing complex matrix multiplication and scaling
        {
            theta = (j*r) % k;
            tre[i] += (xre[l*j+p]*co[theta] + xim[l*j+p]*si[theta])/4096/div;
            tim[i] += (xim[l*j+p]*co[theta] - xre[l*j+p]*si[theta])/4096/div;
        }
    }
    return;
}
```

The under-utilization of memory is also evident in Eq. (19). To store the number 59, only 5 bits are required. But if it is stored as a 16 bit integer, 10 bits are not utilized. Thus, in this example if the scaling factor was smaller, greater precision could have been obtained by avoiding under-utilization of bits.

There are other techniques to deal with overflow apart from scaling. Using saturation arithmetic is one such technique. Assume the result of a mathematical operation exceeded the maximum value that can be stored using the given data type (or the given bit length). Now, one choice is to replace the result with the greatest value that can be stored using the data type. This technique is called saturation, since the memory gets saturated after overflow. The advantage of this technique is that it is easy to implement. But while dealing with matrices, the values of the adjacent elements can change relative to each other. This can lead to erroneous results especially if the replaced number had a large difference from the saturated result.

The code snippet given in Algorithm 3 shows the calculation of one-dimensional DHT using the function dht_1D(). If the input is a 2-D matrix, the dht_1D() function is to be run separately along the rows and columns of the input to find the two-dimensional DHT.

3.1.3 Inverse DHT

Finding the inverse DHT (IDHT) is very similar to finding the DHT. In order to understand this, first let us talk about the inverse DFT. If Y is the 2D DFT of X , we can write in the inverse DFT as $X = \overline{W} \cdot Y \cdot (\overline{W})^T$, where $\overline{W}$ represents the complex conjugate of the Fourier Transformation matrix given in Eq. (13) and $(\overline{W})^T$ is the transpose of $\overline{W}$. In Eq. (17), substitute the second term with its complex conjugate, $\overline{(c + jd)} = c - jd$. With this substitution, Eq. (17) yields the result $Z = (ac + bd) + j(-ad + bc)$. The only difference with Eq. (17) is that the terms containing d has changed the sign.

Similarly, in the C code, the only essential difference of the IDHT when compared with the DHT is that the sign of the terms containing sine values are changed in the function given in Algorithm 3. Thus, the inner 'for' loop in the code snippet Algorithm 3 is changed to the code snippet in Algorithm 4.

Additionally, scaling and shifting is to be done to account for the changes introduced in the C code during DHT computation. It has been mentioned that a scaling factor of 2^{12} was used in the DHT computation instead of 2^{14} , which is also shown in the code snippet Algorithm 3. Thus, there is a deviation of $2^{14}/2^{12} = 2^2$ from the actual scaling value. However, every element in the input passes through the `dht_1D()` function twice. Thus, a total deviation of 2^4 is introduced, which needs to be divided from the input during the inverse DHT computation.

Moreover, at the end of the inverse DHT calculation, 128 has to be added to the real part of the output to compensate for the subtraction by 128 introduced at the beginning of the DHT computation. This enables the elements of the output matrix to be in the range between 0 and 255.

Algorithm 4: Code snippet showing the changed signs for calculating the inverse DHT. Only the inner 'for' loop is shown.

```
for (j = 0; j < k; j++) // loop performing complex matrix multiplication and scaling
{
    theta = (j*r) % k;
    tre[i] += (xre[l*j+p]*co[theta] - xim[l*j+p]*si[theta])/16384/div;
    tim[i] += (xim[l*j+p]*co[theta] + xre[l*j+p]*si[theta])/16384/div;
}
```

CHAPTER 4

HIRSCHMAN COSINE TRANSFORM

The Hirschman Cosine Transform (HCT) is a derivative of the Discrete Cosine Transform (DCT). Following the steps of the Discrete Hirschman Transform (DHT), the transform matrices of most transforms can be made sparser by adding zeros between the elements of the transform matrices. Thus, from a programming stand point, the relationship between the HCT and DCT is similar to the relationship between DHT and DFT. For the HCT, K represents the length of the DCT and L represents the up-sampling.

The 1D equations for DCT and inverse DCT are given in Eq. (20) and Eq. (21) respectively. The DCT transform matrix for a 3-point DCT is given in Eq. 22. Equation 23 gives the transform matrix for the 6-point HCT with $K = 3$ and $L = 2$.

$$X(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x(n) \cos\left(\frac{\pi}{2N} (2n+1)k\right) \quad (20)$$

$$x(n) = \frac{X(0)}{\sqrt{N}} + \sqrt{\frac{2}{N}} \sum_{k=1}^{N-1} X(k) \cos\left(\frac{\pi}{2N} (2n+1)k\right) \quad (21)$$

$$W = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \cos\left(\frac{\pi}{6}\right) & \cos\left(\frac{\pi}{6} \cdot 3\right) & \cos\left(\frac{\pi}{6} \cdot 5\right) \\ \cos\left(\frac{\pi}{6} \cdot 2\right) & \cos\left(\frac{\pi}{6} \cdot 6\right) & \cos\left(\frac{\pi}{6} \cdot 10\right) \end{bmatrix} \quad (22)$$

$$H = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \cos\left(\frac{\pi}{6}\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 3\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 5\right) & 0 \\ 0 & \cos\left(\frac{\pi}{6}\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 3\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 5\right) \\ \cos\left(\frac{\pi}{6} \cdot 2\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 6\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 10\right) & 0 \\ 0 & \cos\left(\frac{\pi}{6} \cdot 2\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 6\right) & 0 & \cos\left(\frac{\pi}{6} \cdot 10\right) \end{bmatrix} \quad (23)$$

The coding for the HCT is very similar to the DHT. But there are some notable differences. First of all, it can be seen from Eq. 20 that the angles used for computing the cosines are a multiple of $\frac{\pi}{2N}$. Thus, there are a total of $4N$ points around the unit circle. (Since $\frac{\pi}{2N} = \frac{2\pi}{4N}$, we can see that there 2π radians are to be divided into $4N$ points). Therefore, the transform array needs to have a length of $4N$. As discussed in the DHT coding section, the transform array stores the elements of the transform matrix. The second difference is that there are no imaginary values involved in the HCT.

Moreover, from Eq. 22 we see that the first row of the transform matrix is $\frac{1}{\sqrt{2}}$. Therefore, we should define $\frac{1}{\sqrt{2}}$ as a 16 bit signed integer. Like the DHT code, all values in the HCT code scale with a factor of $2^{14} = 16,384$. Therefore, the integer 11,585 represents $\frac{1}{\sqrt{2}}$ as a 16 bit signed integer since $\frac{1}{\sqrt{2}} * 2^{14} = 11,585$.

Most of the other aspects of HCT computation remains the same as the DHT. The function `sin_calc()` is reused in the HCT code to find the elements of the transform matrix. Since K is restricted to be a multiple of 4, symmetry is used to reduce the number of computations required to fill the transform array. The 2D HCT is calculated by taking the 1D HCT along the columns followed by the 1D HCT along the rows. The code snippet given in Algorithm 5 shows the function `hct_1D()` used to calculate the 1D HCT. The code snippet in Algorithm 6 shows the inverse 1D computation.

Algorithm 5: Function `hct_1D()` used for calculating 1-dimensional HCT of an input vector

```
void hct_1D(int16_t *xre, int16_t *tre, int16_t div) // xre is input, tre is output
{
    int16_t i, j, p, r, theta;
    int16_t scl = 4096;
    for (i = 0; i < l; i++)
    {
        tre[i] = 0;
        for (j = 0; j < k; j++) // loop iterating through the first row of the transform matrix
            tre[i] += xre[j*l+i]*sqrt1by2/scl/div;
    }
    for (i = l; i < n; i++)
    {
        tre[i] = 0;
        r = i/l;
        p = i%l;
        for (j = 0; j < k; j++) // loop for rest of the elements
        {
            theta = ((2*j+1)*r) % (4*k);
            tre[i] += (xre[l*j+p]*co[theta])/scl/div;
        }
    }
    return;
}
```

Algorithm 6: Function `ihct_1D()` used for calculating 1-dimensional IHCT of an input vector

```
void ihct_1D(int16_t *xre, int16_t *tre, int16_t div) // xre is input, tre is output
{
    int16_t i, j, p, r, theta;
    for (i = 0; i < n; i++)
    {
        r = 2*(i/l)+1;
        p = i%l;
        tre[i] = xre[p]*sqrt1by2/fac/div; // first line of the transform matrix
        for (j = 1; j < k; j++)
        {
            theta = (j*r) % (4*k);
            tre[i] += (xre[l*j+p]*co[theta])/fac/div; // multiplication and scaling
        }
    }
    return;
}
```

CHAPTER 5

RESULTS AND DISCUSSION

The platform used for coding and testing is the MatLab software. The fixed-point computation is performed using C programming language run on MatLab. The C programs are compiled using Microsoft Visual Studio 2015 compiler. The ‘mex’ command in MatLab is used to compile the C programs. Testing is done using codes written in the MatLab programming language. The values obtained from the MatLab programming language are 64 bit double-precision floating-point data and are assumed to have infinite precision for testing purposes.

For testing, the floating-point values from MatLab are compared against the fixed-point values from the C code. The code for testing is also written using the MatLab programming language. For comparing the error, Mean Square Error (MSE), Median Error and Maximum Absolute Error are used. The MatLab function ‘immse’ is used to find the MSE. The mean square error for a two-dimensional matrix is calculated using Eq. (24), where x_{ij} and y_{ij} represent the elements of two complex matrices. Equation (25) shows the error in decibels (dB), where MSE is the mean square error calculated using Eq. (24). Equations (26) and (27) show the formula for median and maximum absolute errors respectively.

$$\text{MSE} = \frac{1}{N^2} \sum_{i=1}^N \sum_{j=1}^N |x_{ij} - y_{ij}|^2 \quad (24)$$

$$\text{MSE(dB)} = 10 \log_{10}(\text{MSE}) \quad (25)$$

$$E_{\text{median}} = \text{median}(|x_{ij} - y_{ij}|); \quad 1 \leq i \leq N, 1 \leq j \leq N \quad (26)$$

$$E_{\text{MAE}} = \max(|x_{ij} - y_{ij}|); \quad 1 \leq i \leq N, 1 \leq j \leq N \quad (27)$$

The error is found by testing the code on randomly generated matrices. Images usually have similar pixel values for pixels that are close to each other. But random matrices do not have such a pattern. Moreover, there is almost no limit to the set of random matrices that can be generated. Thus, if the code works on random matrices, there can be confidence that the code will work on real images also.

5.1 Discrete Hirschman Transform

Figure 5 shows the mean square error (MSE) between the twiddle factors obtained from the fixed-point C code and the 64 bit floating-point values from MatLab. If the transformation equation is $Y = W \cdot X \cdot W^T$, Figure 5 shows the error between the fixed and floating-point values of W , where W is the transform matrix. In the figures, the number beside each point represents the length of DFT used in calculating the MSE. It can be seen from the figure that the error associated with the 4-point DFT is almost negligible compared with the other points. For a 4-point DFT, the twiddle factors are $1, j, -1$ and $-j$ which correspond to the complex exponents of the angles $0, \pi/2, \pi$ and $3\pi/2$ around the unit circle. In the code, the values for these points are not calculated, but are instead assigned directly. Moreover, the sine and cosine of these angles are exactly 0 or ± 1 which is 0 or $\pm 2^{14}$ as a 16 bit integer in the code. Thus, the error associated with the twiddle factors of the 4-point DFT are comparable to the 'eps' value in MatLab which is about -313 dB. In MatLab, eps is the smallest positive number that can be represented using double-precision and is approximately 2.2204×10^{-16} or -313 dB. The error associated with higher DFT lengths are more, because the twiddle factors need to be calculated using the `sin_calc()` function.

Figure 5: Error in the twiddle factors in dB. The graph on the left is zoomed-in and shown on the right to highlight the error for DFT lengths from 8 to 128.

Figure 6 shows the MSE of the fixed-point DHT in dB scale. If $Y = \text{DHT}(X)$, Figure 6 shows the error between the fixed-point and floating-point values of Y obtained from the C and MatLab codes respectively. The numbers on the plot represents the length of the DFT used in finding the error. This graph shows the error obtained from fifty randomly generated 256×256 matrices, with each element in the matrix having an integer value in the range $[0, 255]$. These matrices perform the role of random 8-bit grayscale images. In Figure 6, the absence of the length 4 DFT can be noted. This absence is due to the fact that there is no error associated with the 4-point DFT. In dB scale, an error of 0 equals negative infinity decibels and MatLab does not plot this value. It can also be noted from Figure 6 that the difference between the greatest and least error for a given value of K is quite small, about 0.05 dB.

Matrices of size 256×256 were preferred for testing because the maximum length of the DFT is 128. Therefore, a length of 256 can allow the up-sampling rate to be at least 2. Moreover, using greater lengths would result in code to run slower during testing. However, the C codes written are generalized to the maximum extent possible and will run on images of size greater than 256. An example image of size 512×512 is shown at end of the Results.

Figure 6: Mean Square Error of the Discrete Hirschman Transform in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.

Figure 7 shows the MSE between the processed image and original image in dB scale for fifty randomly generated matrices. These matrices perform the function of random images as explained the previous paragraph. Also, in this context, the term ‘processed image’ refers to the 8-bit image obtained after taking the DHT and inverse DHT of an input. If $Z = \text{IDHT}(\text{DHT}(X))$, Figure 7 shows the error between X and Z where the computation is done using the fixed-point C code. Thus, every pixel value is compared and the plot shows the error in pixel values of Z with respect to X . Like in Figure 6, in Figure 7 the error associated with 4-point DFT is $-\infty$ in dB scale.

Similarly, Figure 8 shows the median error and Figure 9 shows the maximum absolute error between the processed and original images. The Y-axis of these plots are not in dB scale and directly show the error in pixel values. Since 8-bit images are used, each pixel can have values between 0 and 255. From Figure 8 it can be seen that even for DFT length of 128, the median value of the error between corresponding pixels is 3. From Figure 9, it is clear that even for the worst-case scenario, the greatest deviation from the actual pixel value is about 15. It is also worth noting that the least absolute error is zero for all values of K .

Figure 7: Mean Square Error between the processed image and the original image in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.

Figure 8: Median of the pixel values after computing the forward and inverse DHT.

Figure 9: Maximum Absolute Error of the pixel values after taking the forward and inverse DHT.

Original Image (256x256)

Processed Image ($K=128$, $L=2$)

Figure 10: Visual comparison on the 'cameraman.tif' image. The image on the bottom was processed using DHT and IDHT with $K = 128$ and $L = 2$.

Original Image (512x512)

Processed Image ($K=128$, $L=4$)

Figure 11: Visual comparison on the ‘office.tif’ image. The image on the bottom was processed using DHT and IDHT with $K = 128$ and $L = 4$.

Figure 10 shows the processing on an actual image. The image used is the 'cameraman.tif' image which is available in the MatLab software. It is a 256×256 grayscale image. The image on the top is the original and one at the bottom is obtained after taking the forward and inverse DHT on the original image using $K = 128$ and $L = 2$. The worst output is obtained when $K = 128$. Therefore, the quality of the output image gets better for smaller values of K . Figure 11 shows the visual comparison on a 512×512 image. The image on the top was obtained by editing the 'office_5.jpg' file from MatLab. The color image 'office_5.jpg' was cropped and converted to greyscale using the MatLab function 'rgb2gray' to get the 'office.tif' image. The image at the bottom in Figure 11 was processed using $K = 128$ and $L = 4$.

5.2 Hirschman Cosine Transform

Figure 12 shows the mean square error (MSE) between the transform matrix elements for the HCT obtained from the fixed-point C code and the 64 bit floating-point values from MatLab. If the transformation equation is $Y = W \cdot X \cdot W^T$, Figure 12 shows the error between the fixed and floating-point values of W , where W is the transform matrix. In the figure, the number beside each point represents the length of DCT. The Y-axis shows the error associated with each length of DCT.

Figure 13 shows the MSE of the fixed-point HCT in dB scale. If $Y = \text{HCT}(X)$, Figure 6 shows the error between the fixed-point and floating-point values of Y obtained from the C and MatLab codes respectively. The numbers on the plot represents the length of the DCT used in finding the error. This graph shows the error obtained from fifty randomly generated 256×256 matrices, with each element in the matrix having an integer value in the range [0, 255]. These matrices perform the role of random 8-bit grayscale images. It can also be noted from Figure 13 that the difference between the greatest and least error for a given value of K is quite small, about 0.08 dB. The error of the HCT and DHT is similar for the forward transforms. HCT shows a slightly better error performance.

Figure 12: Error in the transform matrix elements of HCT.

Figure 13: Mean Square Error of the HCT in dB. The figure on the right is zoomed in on the point $K=128$, to highlight the range of greatest and least MSE. The dot shows the average MSE.

Figure 14 shows the MSE between the processed image and original image in dB scale for fifty randomly generated image matrices. These matrices perform the function of random images as explained the previous paragraph. As mentioned before, in this context, the term ‘processed image’ refers to the 8-bit image obtained after taking the HCT and inverse HCT (IHCT) of an input. If $Z = \text{IHCT}(\text{HCT}(X))$, Figure 14 shows the error between X and Z . Here, both the forward and the inverse transforms are computed using the fixed-point C code. Every pixel is compared to obtain the plot shown in Figure 14, which shows the error in pixel values of Z with respect to X .

Like the MSE from the previous paragraph, Figure 15 shows the median error and Figure 16 shows the maximum absolute error between the processed and original images. The Y-axis of these plots are not in dB scale and directly show the error in pixel values. Since 8-bit images are used, each pixel can have values between 0 and 255. From Figure 15 we can see that even for DCT length of 128, the median value of the error between corresponding pixels is 2. From Figure 16, it is clear that even for the worst-case scenario, the greatest deviation from the actual pixel value is about 10. It is also worth noting that the least absolute error is zero for all values of K . Comparing figures 7 through 9 and figures 14 through 16, one can see that the error from the HCT is lower than the DHT.

Figure 14: Mean Square Error between the processed image and the original image in dB.

Figure 15: Median Error of the pixel values after computing the forward and inverse HCT.

Figure 16: Maximum Absolute Error of the pixel values after taking the forward and inverse HCT.

Original Image (256x256)

Processed Image ($K=128, L=2$)

Figure 17: Visual comparison on the 'cameraman.tif' image. The image on the bottom was processed using HCT and IHCT with $K = 128$ and $L = 2$.

Original Image (512x512)

Processed Image ($K=128$, $L=4$)

Figure 18: Visual comparison on the ‘office.tif’ image. The image on the bottom was processed using HCT and IHCT with $K = 128$ and $L = 4$.

Figure 17 shows the processing on an actual image. The image used is the 'cameraman.tif' image which is available in the MatLab software. It is a 256×256 grayscale image. The image on the top is the original and one at the bottom is obtained after taking the forward and inverse HCT on the original image using $K = 128$ and $L = 2$. The worst output is obtained when $K = 128$. Therefore, the quality of the output image gets better for smaller values of K . Figure 18 shows the visual comparison on a 512×512 image. The image on the top was obtained by editing the 'office_5.jpg' file from MatLab. The image at the bottom in Figure 18 was processed using $K = 128$ and $L = 4$.

CHAPTER 6

CONCLUSION AND FUTURE WORK

From the data shown in the results, it can be seen that reasonably accurate processing can be obtained using the method given. As the length of the DFT or DCT decreases, the accuracy increases. The maximum error during testing was found to be 15 for the DHT and 10 for the HCT. These numbers are the deviation of the pixel values from the original image. The errors obtained are low when compared to the maximum pixel value of 255. For most images, such a low error is most likely to go undetected using the human eye in normal circumstances. The C programs used for the implementation of DHT is given in the Appendix.

Further work can be done to improve the work done in this thesis. Fast algorithms similar to the FFT can be implemented on the DHT since DHT can be implemented using FFTs. On Field Programmable Gate Arrays (FPGAs), the speed can be increased by using the parallel processing capabilities of FPGA boards. Moreover, the scaling factor is correlated with the length of the DFT or DCT. Therefore, dynamic allocation of scaling factor can be implemented to improve the error performance.

APPENDIX A

C CODES FOR DHT AND IDHT

A.1 C Code for DHT

```

/*=====
*2-D Discrete Hirschman Transform (DHT), using no float
* for input length n=kdft*upsamp
* kdft point DFT with upsampling by upsamp times
*
* y = dht2_fixed(x,kdft,upsamp)
* this function call returns the complex matrix y, which is
* the DHT of input matrix x using kdft point DFT at an
* upsampling rate of upsamp. x must be a square matrix and
* its maximum value must be less than 256 (ie., each element
* of x must be of 8 bits).
*=====*/
#include <mex.h>
#include <stdlib.h>
#include <inttypes.h>
const int16_t pi2 = 25736; // 25736 = round((pi/2)*2^14); ie., pi/2 in 15
bits (1 bit for whole no. part)
const int16_t fac = 16384; // 16384 = 2^14 (for dividing the fixed point
result to floating point)
size_t m, n; // size_t is kind of like an unsigned data type used for
indexing, string length, etc.
int16_t k, l;
int16_t *si, *co; // sine and cosine vectors
int16_t sin_calc(int16_t); // for calculating sine of an angle
void dht2_begin(int16_t **, int16_t **); // begin 2D DHT calculation
void dht_1D(int16_t *, int16_t *, int16_t *, int16_t *, int16_t); // DHT of
1D vector
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *inre, *inim, *trans_re, *trans_im;
    int16_t i, j;
    int16_t **re, **im;
    if(nrhs != 3)
        mexErrMsgTxt("Three inputs required.");
    if(nlhs > 1)
        mexErrMsgTxt("Function returns only one output.");
    m = mxGetM(prhs[0]); // no. of rows in input
    n = mxGetN(prhs[0]); // no. of columns in input
    inre = mxGetPr(prhs[0]); // read inputs (real values)
    inim = mxGetPi(prhs[0]);
    k = (int16_t)mxGetScalar(prhs[1]);
    l = (int16_t)mxGetScalar(prhs[2]); // make sure the first input argument is
    scalar
    if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) || m < 2 || m != n )
        mexErrMsgTxt("First argument must be a real square matrix of type
double.");

```

```

    if( !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
mxGetNumberOfElements(prhs[1]) != 1 )
    mexErrMsgTxt("Size of the DFT (the second argument) must be a scalar.");
    if( !mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||
mxGetNumberOfElements(prhs[2]) != 1 )
    mexErrMsgTxt("Upsampling rate (the third argument) must be a scalar.");
    if( k < 1 || l < 1 )
    mexErrMsgTxt("Size of DFT and Upsampling rate must be positive values.");
    if( k < 4 || k % 4 != 0 )
    mexErrMsgTxt("The size of DFT must be a positive multiple of 4."); //
checking for power of 2
    if( k*l != m )
    mexErrMsgTxt("Size of DFT times Upsampling rate must equal no. of rows of
input matrix.");
    re = (int16_t **)mxMalloc(m * sizeof(int16_t *));
    im = (int16_t **)mxMalloc(m * sizeof(int16_t *));
    for(i=0; i<m; i++)
    {
        re[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));
        im[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));
        for(j=0; j<n; j++) // converting double input to int16_t
        {
            re[i][j] = (int16_t)inre[j*n+i] - 128; // converting matlab column order
to c row order
        }
    }
    dht2_begin(re, im); // calculate DHT of input
    plhs[0] = mxCreateDoubleMatrix((mwSize)m, (mwSize)n, mxCOMPLEX);
    trans_re = mxGetPr(plhs[0]);
    trans_im = mxGetPi(plhs[0]);
    for(i=0; i<m; i++) // converting int16_t values to double
    {
        for(j=0; j<n; j++)
        {
            trans_re[j*n+i] = (double)(re[i][j] + 128);
            trans_im[j*n+i] = (double)im[i][j];
        }
        mxFree(re[i]); // freeing memory allocated by mxMalloc
        mxFree(im[i]);
    }
    mxFree(re);
    mxFree(im);
    return;
}
void dht2_begin(int16_t **yre, int16_t **yim)
{
    si = (int16_t *)mxMalloc(k * sizeof(int16_t));
    co = (int16_t *)mxMalloc(k * sizeof(int16_t));
    int16_t i, j, j1, j2, c, z, arg;
    int16_t *inpre, *inpim, *outre, *outim; // vectors for storing rows and
columns
    inpre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    inpim = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outim = (int16_t *)mxMalloc(m * sizeof(int16_t));
    arg = pi*2*4/k;
    j1 = k/4;

```



```

j2 = k/2;
c = 1;
// calculating sine and cosine vectors
si[0] = 0; co[0] = fac; // fac corresponds to 1.0 in float data type
si[j1] = fac; co[j1] = 0; // at pi/2
si[j2] = 0; co[j2] = -fac; // at pi
for (i = 1; i < j1; i++) // theta = 0 to pi/2
{
    z = arg*i;
    si[i] = sin_calc(z);
    co[i] = 1 + sin_calc(piby2 - z - 8); // cos(x) = sin(pi/2-x)
    if ( z > piby2*4/9) // 40 degrees
        co[i] = co[i] + 5;
}
for (i = j1+1; i < j2; i++) // theta = pi/2 to pi
{
    si[i] = si[j1 - c]; //sin_calc(2*piby2 - z); // sin(x) = sin(pi-x)
    co[i] = -co[j1 - c]; //-sin_calc(z - piby2); // cos(x) = sin(pi/2-x) = -
sin(x-pi/2)
    c++;
}
c = 1;
for (i = j2+1; i < k; i++) // theta = +pi to 2pi (ie., -pi to 0)
{
    si[i] = -si[j2 - c];
    co[i] = co[j2 - c];
    c++;
} // end of sine and cosine calculation
//-----
for (i = 0; i < m; i++) // for purely real inputs
    inpim[i] = 0;
for (i = 0; i < n; i++) // calculating DHT along columns
{
    for (j = 0; j < m; j++)
        inpre[j] = yre[j][i];
    dht_1D(inpre, inpim, outre, outim, 2); // DHT of 1D vector
}
for (j = 0; j < m; j++)
{
    yre[j][i] = outre[j];
    yim[j][i] = outim[j];
}
}
for (i = 0; i < m; i++) // calculating DHT along rows
{
    for (j = 0; j < n; j++)
    {
        inpre[j] = yre[i][j];
        inpim[j] = yim[i][j];
    }
    dht_1D(inpre, inpim, outre, outim, k); // divide by k
    for (j = 0; j < n; j++)
    {
        yre[i][j] = outre[j];
        yim[i][j] = outim[j];
    }
}
}

```

```

    mxFree(co); mxFree(si); // freeing memory allocated by mxMalloc
    mxFree(inpre); mxFree(inpim);
    mxFree(outr); mxFree(outim);
    return;
}
// complex DHT of 1D array
void dht_1D(int16_t *xre, int16_t *xim, int16_t *tre, int16_t *tim, int16_t
div)
{
    int16_t i, j, p, r, theta;
    for (i = 0; i < n; i++)
    {
        tre[i] = 0;
        tim[i] = 0;
        r = i/l;
        p = i%l;
        for (j = 0; j < k; j++) // loop performing multiplication and scaling
        {
            theta = (j*r) % k;
            tre[i] += (xre[l*j+p]*co[theta] + xim[l*j+p]*si[theta])/4096/div;
            tim[i] += (xim[l*j+p]*co[theta] - xre[l*j+p]*si[theta])/4096/div;
        }
    }
    return;
}
// function to calculate sin(theta)
// using Taylor series expansion of sin(x) [= x - x^3/3! + x^5/5! - x^7/7! +
....]
int16_t sin_calc(int16_t x)
{
    int16_t sine, temp;
    temp = x*x / 32768 * x / 32768 * x / 16384 * x / 4096 / 120;
    sine = x - x*x / 32768 * x / 8192 / 6 + temp;
    if (x > pi/2) // pi/4 radians
    {
        temp = -1 * temp * x / 16384 * x / 16384 / 42;
        sine = sine + temp;
        if (x > pi/2 * 8/9) // 80 degrees
            sine = sine + 1;
    }
    return sine;
}

```

A.2 C Code for Inverse DHT

```

/*=====
*2-D Inverse Discrete Hirschman Transform (IDHT), using no float
* for input length n=kdft*upsamp * kdft point DFT with upsampling
* by upsamp times
*
* y = idht2_fixed(x,kdft,upsamp)
* this function call returns the real
* matrix y, which is the IDHT of input matrix x using kdft
* point DFT at an upsampling rate of upsamp. x must be a

```

```

* square matrix and each element must be at the most 15 bits wide.
*=====*/
#include <mex.h>
#include <stdlib.h>
#include <inttypes.h>
const int16_t pi2 = 25736; // 25736 = round((pi/2)*2^14); ie., pi/2 in 15
bits (1 bit for whole no. part)
const int16_t fac = 16384; // 16384 = 2^14 (for dividing the fixed point
result to floating point)
size_t m, n; // size_t is kind of like an unsigned data type used for
indexing, string length, etc.
int16_t k, l;
int16_t *si, *co; // sine and cosine vectors
int16_t sin_calc(int16_t); // for calculating sine of an angle
void dht2_begin(int16_t **, int16_t **); // begin 2D DHT calculation
void dht_1D(int16_t *, int16_t *, int16_t *, int16_t *, int16_t); // DHT of
1D vector
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double *inre, *inim, *trans_re, *trans_im;
    int16_t i, j;
    int16_t **re, **im;
    if(nrhs != 3)
        mexErrMsgTxt("Three inputs required.");
    if(nlhs > 1)
        mexErrMsgTxt("Function returns only one output.");
    m = mxGetM(prhs[0]); // no. of rows in input
    n = mxGetN(prhs[0]); // no. of columns in input
    inre = mxGetPr(prhs[0]); // read inputs (real values)
    inim = mxGetPi(prhs[0]);
    k = (int16_t)mxGetScalar(prhs[1]);
    l = (int16_t)mxGetScalar(prhs[2]); // make sure the first input argument is
scalar
    if( !mxIsDouble(prhs[0]) || m < 2 || m != n )
        mexErrMsgTxt("First argument must be a square matrix of type double.");
    if( !mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
mxGetNumberOfElements(prhs[1]) != 1 )
        mexErrMsgTxt("Size of the DFT (the second argument) must be a scalar.");
    if( !mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||
mxGetNumberOfElements(prhs[2]) != 1 )
        mexErrMsgTxt("Upsampling rate (the third argument) must be a scalar.");
    if( k < 1 || l < 1 )
        mexErrMsgTxt("Size of DFT and Upsampling rate must be positive values.");
    if( k < 4 || k % 4 != 0 )
        mexErrMsgTxt("The size of DFT must be a positive multiple of 4."); //
checking for power of 2
    if( k*l != m )
        mexErrMsgTxt("Size of DFT times Upsampling rate must equal no. of rows of
input matrix.");
    re = (int16_t *)mxMalloc(m * sizeof(int16_t *));
    im = (int16_t *)mxMalloc(m * sizeof(int16_t *));
    for(i=0; i<m; i++)
    {
        re[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));
        im[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));
        for(j=0; j<n; j++) // converting double input to int16_t
        {

```

```

    re[i][j] = (int16_t)inre[j*n+i]; // converting matlab column order to c
row order
    im[i][j] = (int16_t)inim[j*n+i];
}
}
dht2_begin(re, im); // calculate DHT of input
plhs[0] = mxCreateDoubleMatrix((mwSize)m, (mwSize)n, mxCOMPLEX);
trans_re = mxGetPr(plhs[0]); trans_im = mxGetPi(plhs[0]);
for(i=0; i<m; i++) // converting int16_t values to double
{
    for(j=0; j<n; j++)
    {
        trans_re[j*n+i] = (double)(re[i][j] + 128);
        trans_im[j*n+i] = (double)im[i][j];
    }
    mxFree(re[i]); // freeing memory allocated by mxMalloc mxFree(im[i]);
}
mxFree(re); mxFree(im); return;
}
void dht2_begin(int16_t **yre, int16_t **yim)
{
    si = (int16_t *)mxMalloc(k * sizeof(int16_t));
    co = (int16_t *)mxMalloc(k * sizeof(int16_t));
    int16_t i, j, j1, j2, c, z, arg;
    int16_t *inpre, *inpim, *outre, *outim; // vectors for storing rows and
columns
    inpre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    inpim = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outim = (int16_t *)mxMalloc(m * sizeof(int16_t));
    arg = pi*2*4/k;
    j1 = k/4;
    j2 = k/2;
    c = 1;
    // calculating sine and cosine vectors
    si[0] = 0; co[0] = fac; // fac corresponds to 1.0 in float data type
    si[j1] = fac; co[j1] = 0; // at pi/2
    si[j2] = 0; co[j2] = -fac; // at pi
    for (i = 1; i < j1; i++) // theta = 0 to pi/2
    {
        z = arg*i;
        si[i] = sin_calc(z);
        co[i] = 1 + sin_calc(pi*2 - z - 8); // cos(x) = sin(pi/2-x)
        if ( z > pi*2*4/9) // 40 degrees
            co[i] = co[i] + 5;
    }
    for (i = j1+1; i < j2; i++) // theta = pi/2 to pi
    {
        si[i] = si[j1 - c]; //sin_calc(2*pi*2 - z); // sin(x) = sin(pi-x)
        co[i] = -co[j1 - c]; //-sin_calc(z - pi*2); // cos(x) = sin(pi/2-x) = -
sin(x-pi/2)
        c++;
    }
    c = 1;
    for (i = j2+1; i < k; i++) // theta = +pi to 2pi (ie., -pi to 0)
    {
        si[i] = -si[j2 - c];

```

```

    co[i] = co[j2 - c];
    c++;
} // end of sine and cosine calculation
//-----
for (i = 0; i < n; i++) // calculating DHT along columns
{
    for (j = 0; j < m; j++)
    {
        inpre[j] = yre[j][i];
        inpim[j] = yim[j][i];
    }
    dht_1D(inpre, inpim, outre, outim, 2); // DHT of 1D vector
    for (j = 0; j < m; j++)
    {
        yre[j][i] = outre[j];
        yim[j][i] = outim[j];
    }
}
for (i = 0; i < m; i++) // calculating DHT along rows
{
    for (j = 0; j < n; j++)
    {
        inpre[j] = yre[i][j];
        inpim[j] = yim[i][j];
    }
    dht_1D(inpre, inpim, outre, outim, k); // divide by k
    for (j = 0; j < n; j++)
    {
        yre[i][j] = outre[j]/4;
        yim[i][j] = outim[j]/4;
    }
}
mxFree(co);
mxFree(si);
// freeing memory allocated by mxMalloc
mxFree(inpre); mxFree(inpim);
mxFree(outre); mxFree(outim);
return;
}
// complex DHT of 1D array
void dht_1D(int16_t *xre, int16_t *xim, int16_t *tre, int16_t *tim, int16_t
div)
{
    int16_t i, j, p, r, theta;
    for (i = 0; i < n; i++)
    {
        tre[i] = 0; tim[i] = 0;
        r = i/l;
        p = i%l;
        for (j = 0; j < k; j++)
        {
            theta = (j*r) % k;
            tre[i] += (xre[l*j+p]*co[theta] - xim[l*j+p]*si[theta])/fac/div;
            tim[i] += (xim[l*j+p]*co[theta] + xre[l*j+p]*si[theta])/fac/div;
        }
    }
}
return;

```

```

}
// function to calculate sin(theta)
// using Taylor series expansion of sin(x) [= x - x^3/3! + x^5/5! - x^7/7! +
....]
int16_t sin_calc(int16_t x)
{
    int16_t sine, temp;
    temp = x*x / 32768 * x / 32768 * x / 16384 * x / 4096 / 120;
    sine = x - x*x / 32768 * x / 8192 / 6 + temp;
    if (x > pi_by2 / 2) // pi/4 radians
    {
        temp = -1 * temp * x / 16384 * x / 16384 / 42;
        sine = sine + temp;
        if (x > pi_by2 * 8/9) // 80 degrees
            sine = sine + 1;
    }
    return sine;
}

```

APPENDIX B

MATLAB CODES FOR DHT AND IDHT

B.1 MatLab Code for DHT

```
%Two-dimensional Discrete Hirschman Transform (DHT)
% using floating point.
%
% y = dht2_float(x,kdft,upsamp);
% this function call returns the matrix y, which is
% the 2-D DHT of the square matrix x using kdft point
% DFT at an upsampling rate of upsamp.
% If the matrix x contains n rows (or columns), the condition
% n=kdft*upsamp must be satisfied.

function y = dht2_float(x,kdft,upsamp)
    [m,n] = size(x);
    x = x - 128;
    if m ~= n
        error('The input must be a square matrix');
    end
    if length(kdft) ~= 1 || length(upsamp) ~= 1
        error(['The second and third arguments must ',...
            'be scalars']);
    end
    if m ~= kdft*upsamp
        error(['Size of the DFT times the upsampling ',...
            'rate must equal the rows of the input matrix.']);
    end
    w = dftmtx(kdft); % w stores the DFT transform matrix
    h = zeros(m);
    t = 1:upsamp:m;
    for i=0:upsamp-1 % populating the DHT transform matrix
        h(t+i,t+i) = w;
    end
    y = h*x*h;
    y = y/kdft;
end
```

B.2 MatLab code for IDHT

```
%Two-dimensional Inverse Discrete Hirschman Transform (DHT)
% using floating point.
%
% y = idht2_float(x,kdft,upsamp);
% this function call returns the matrix y, which is
% the 2-D IDHT of the square matrix x using kdft point
% IDFT at an upsampling rate of upsamp.
% If the matrix x contains n rows (or columns), the condition
% n=kdft*upsamp must be satisfied.
```

```

function y = idht2_float(x,kdft,upsamp)
    [m,n] = size(x);
    if m ~= n
        error('The input must be a square matrix');
    end
    if length(kdft) ~= 1 || length(upsamp) ~= 1
        error(['The second and third arguments must ',...
            'be scalars']);
    end
    if m ~= kdft*upsamp
        error(['Size of the DFT times the upsampling ',...
            'rate must equal the rows of the input matrix.']);
    end
    w = conj(dftmtx(kdft)); % w stores the DFT transform matrix
    h = zeros(m);
    t = 1:upsamp:m;
    for i=0:upsamp-1 % populating the DHT transform matrix
        h(t+i,t+i) = w;
    end
    y = h*x*h;
    y = y/kdft;
    y = y + 128;
end

```


APPENDIX C

C CODES FOR HCT AND IHCT

C.1 C Code for HCT

```
/*=====
*2-D Hirschman Cosine Transform (DCT), using fixed-point
* for input length n=kdct*upsamp
* kdct point DCT with upsampling by upsamp times
*
* y = hct2_fixed(x,kdct,upsamp)
* this function call returns the matrix y, which is
* the HCT of input matrix x using kdft point DCT at an
* upsampling rate of upsamp. x must be a square matrix and
* its maximum value must be less than 256 (ie., each element
* of x must be of 8 bits).
*=====*/

// for compiling this file:
// mex D:\Libraries\Documents\MATLAB\Thesis\hct2_fixed.c

#include <mex.h>
#include <stdlib.h>
#include <inttypes.h>

const int16_t piby2 = 25736; // 25736 = round((pi/2)*2^14); ie., pi/2 in 15
bits (1 bit for whole no. part)
const int16_t fac = 16384; // 16384 = 2^14 (for dividing the fixed point
result to floating point)
const int16_t sqrt1by2 = 11585; // 11585 = 2^14/sqrt(2); ie., sqrt(1/2) if 1
-> 16384
size_t m, n; // size_t is kind of like an unsigned data type used for
indexing, string length, etc.
int16_t k, l;
int16_t *co; // sine and cosine vectors

int16_t sin_calc(int16_t); // for calculating sine of an
angle
void hct2_begin(int16_t **); // begin 2D DHT calculation
void hct_1D(int16_t *, int16_t *, int16_t); // DHT of 1D vector

void mexFunction(int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
    double *inp, *trans_re;
    int16_t i, j;
    int16_t **re;

    if(nrhs != 3)
        mexErrMsgTxt("Three inputs required.");
    if(nlhs > 1)
        mexErrMsgTxt("Function returns only one output.");
```

```

m = mxGetM(prhs[0]); // no. of rows in input
n = mxGetN(prhs[0]); // no. of columns in input

inp = mxGetPr(prhs[0]); // read inputs (real values)
k = (int16_t)mxGetScalar(prhs[1]);
l = (int16_t)mxGetScalar(prhs[2]);

// make sure the first input argument is scalar
if( !mxIsDouble(prhs[0]) ||
    mxIsComplex(prhs[0]) ||
    m < 2 || m != n )
    mexErrMsgTxt("First argument must be a real square matrix of type
double.");
if( !mxIsDouble(prhs[1]) ||
    mxIsComplex(prhs[1]) ||
    mxGetNumberOfElements(prhs[1]) != 1 )
    mexErrMsgTxt("Size of the DFT (the second argument) must be a
scalar.");
if( !mxIsDouble(prhs[2]) ||
    mxIsComplex(prhs[2]) ||
    mxGetNumberOfElements(prhs[2]) != 1 )
    mexErrMsgTxt("Upsampling rate (the third argument) must be a
scalar.");
if( k < 1 || l < 1 )
    mexErrMsgTxt("Size of DFT and Upsampling rate must be positive
values.");
if( k < 4 || k % 4 != 0 )
    mexErrMsgTxt("The size of DFT must be a positive multiple of 4.");
if( k*l != m )
    mexErrMsgTxt("Size of DFT times Upsampling rate must equal no. of
rows of input matrix.");

re = (int16_t *)mxMalloc(m * sizeof(int16_t *));

for(i=0; i<m; i++)
{
    re[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));

    for(j=0; j<n; j++) // converting double input to uint8_t
    {
        re[i][j] = (int16_t)inp[j*n+i] - 128; // converting matlab column
order to c row order
    }
}

hct2_begin(re); // calculate DHT of input

plhs[0] = mxCreateDoubleMatrix((mwSize)m, (mwSize)n, mxREAL);
trans_re = mxGetPr(plhs[0]);

for(i=0; i<m; i++) // converting int16_t values to double
{
    for(j=0; j<n; j++)
    {
        trans_re[j*n+i] = (double)re[i][j];
    }
}

```

```

    }
    mxFree(re[i]); // freeing memory allocated by mxMalloc
}
mxFree(re);

return;
}

void hct2_begin(int16_t **yre)
{
    co = (int16_t *)mxMalloc(4*k * sizeof(int16_t));

    int16_t i, j, c=1, z, arg;
    int16_t *inpre, *outre; // vectors for storing rows and columns

    inpre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outre = (int16_t *)mxMalloc(m * sizeof(int16_t));

    arg = pi*2/k;
    co[0] = fac; // fac corresponds to 1.0 in float data type
    co[k] = 0; // corresponding to pi/2
    co[2*k] = -fac; // corresponding to pi

    // calculating cosine vectors
    for (i = 1; i < k; i++) // theta = 0 to pi/2
    {
        z = arg*i;
        co[i] = 1 + sin_calc(pi*2 - z - 8); // cos(x) = sin(pi/2-x)
        if (z > pi*2*4/9) // 40 degrees
            co[i] = co[i] + 5;
    }

    for (i = k+1; i < 2*k; i++) // theta = pi/2 to pi
    {
        co[i] = -co[k - c];
        c++;
    }

    c = 1;
    for (i = 2*k+1; i < 4*k; i++) // theta = +pi to 2pi (ie., -pi to 0)
    {
        co[i] = co[2*k - c];
        c++;
    } // end of cosine calculation

    //-----
    for (i = 0; i < n; i++) // calculating DHT along columns
    {
        for (j = 0; j < m; j++)
            inpre[j] = yre[j][i];

        hct_1D(inpre, outre, 1); // DHT of 1D vector
        for (j = 0; j < m; j++)
        {
            yre[j][i] = outre[j];
        }
    }
}

```

```

    }

    for (i = 0; i < m; i++) // calculating DHT along rows
    {
        for (j = 0; j < n; j++)
            inpre[j] = yre[i][j];

        hct_1D(inpre, outre, 2*k/2); // divide by k/2
        for (j = 0; j < n; j++)
        {
            yre[i][j] = outre[j];
        }
    }
    mxFree(co); // freeing memory allocated by mxMalloc
    mxFree(inpre);
    mxFree(outre);
    return;
}

// DCT of 1D array
void hct_1D(int16_t *xre, int16_t *tre, int16_t div)
{
    int16_t i, j, p, r, theta;
    int16_t scl = 4096;
    for (i = 0; i < l; i++)
    {
        tre[i] = 0;
        for (j = 0; j < k; j++)
            tre[i] += xre[j*l+i]*sqrtlby2/scl/div;
    }
    for (i = l; i < n; i++)
    {
        tre[i] = 0;
        r = i/l;
        p = i%l;
        for (j = 0; j < k; j++)
        {
            theta = ((2*j+1)*r) % (4*k);
            tre[i] += (xre[l*j+p]*co[theta])/scl/div;
        }
    }
    return;
}

// function to calculate sin(theta)
// using Taylor series expansion of sin(x) [= x - x^3/3! + x^5/5! - x^7/7! +
....]
// [https://en.wikipedia.org/wiki/Sine#Series_definition]
int16_t sin_calc(int16_t x)
{
    int16_t sine, temp;
    temp = x*x / 32768 * x / 32768 * x / 16384 * x / 4096 / 120;
    sine = x - x*x / 32768 * x / 8192 / 6 + temp;

    if (x > pi*2 / 2) // pi/4 radians
    {
        temp = -1 * temp * x / 16384 * x / 16384 / 42;
    }
}

```

```

        sine = sine + temp;

        if (x > pi*2 * 8/9) // 80 degrees
            sine = sine + 1;
    }

    return sine;
}

```

C.2 C code for IHCT

```

/*=====
*2-D Inverse Hirschman Cosine Transform (IHCT), using
* fixed-point for input length n=kdct*upsamp
* kcft point DCT with upsampling by upsamp times
*
* y = ihct2_fixed(x,kdct,upsamp)
* this function call returns the matrix y, which is
* the inverse HCT of input matrix x using kcft point DCT at an
* upsampling rate of upsamp. x must be a square matrix and
* its maximum value must be less than 256 (ie., each element
* of x must be of 8 bits).
*=====*/

// for compiling this file:
// mex D:\Libraries\Documents\MATLAB\Thesis\ihct2_fixed.c

#include <mex.h>
#include <stdlib.h>
#include <inttypes.h>

const int16_t pi*2 = 25736; // 25736 = round((pi/2)*2^14); ie., pi/2 in 15
bits (1 bit for whole no. part)
const int16_t fac = 16384; // 16384 = 2^14 (for dividing the fixed point
result to floating point)
const int16_t sqrt1by2 = 11585; // 11585 = 2^14/sqrt(2); ie., sqrt(1/2) if 1
-> 16384
size_t m, n; // size_t is kind of like an unsigned data type used for
indexing, string length, etc.
int16_t k, l;
int16_t *co; // sine and cosine vectors

int16_t sin_calc(int16_t); // for calculating sine of an
angle
void idct2_begin(int16_t **); // begin 2D DHT calculation
void idct_1D(int16_t *, int16_t *, int16_t); // DHT of 1D vector

void mexFunction(int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
    double *inp, *trans_re;
    int16_t i, j;
    int16_t **re;

```

```

if(nrhs != 3)
    mexErrMsgTxt("Three inputs required.");
if(nlhs > 1)
    mexErrMsgTxt("Function returns only one output.");

m = mxGetM(prhs[0]); // no. of rows in input
n = mxGetN(prhs[0]); // no. of columns in input

inp = mxGetPr(prhs[0]); // read inputs (real values)
k = (int16_t)mxGetScalar(prhs[1]);
l = (int16_t)mxGetScalar(prhs[2]);

// make sure the first input argument is scalar
if( !mxIsDouble(prhs[0]) ||
    mxIsComplex(prhs[0]) ||
    m < 2 || m != n )
    mexErrMsgTxt("First argument must be a real square matrix of type
double.");
if( !mxIsDouble(prhs[1]) ||
    mxIsComplex(prhs[1]) ||
    mxGetNumberOfElements(prhs[1]) != 1 )
    mexErrMsgTxt("Size of the DFT (the second argument) must be a
scalar.");
if( !mxIsDouble(prhs[2]) ||
    mxIsComplex(prhs[2]) ||
    mxGetNumberOfElements(prhs[2]) != 1 )
    mexErrMsgTxt("Upsampling rate (the third argument) must be a
scalar.");
if( k < 1 || l < 1 )
    mexErrMsgTxt("Size of DFT and Upsampling rate must be positive
values.");
if( k < 4 || k % 4 != 0 )
    mexErrMsgTxt("The size of DFT must be a positive multiple of 4.");
if( k*l != m )
    mexErrMsgTxt("Size of DFT times Upsampling rate must equal no. of
rows of input matrix.");

re = (int16_t **)mxMalloc(m * sizeof(int16_t *));

for(i=0; i<m; i++)
{
    re[i] = (int16_t *)mxMalloc(n * sizeof(int16_t));

    for(j=0; j<n; j++) // converting double input to uint8_t
    {
        re[i][j] = (int16_t)inp[j*n+i]; // converting matlab column order
to c row order
    }
}

ihct2_begin(re); // calculate DHT of input

plhs[0] = mxCreateDoubleMatrix((mwSize)m, (mwSize)n, mxREAL);
trans_re = mxGetPr(plhs[0]);

```

```

for(i=0; i<m; i++) // converting int16_t values to double
{
    for(j=0; j<n; j++)
    {
        trans_re[j*n+i] = (double)(re[i][j] + 128);
    }
    mxFree(re[i]); // freeing memory allocated by mxMalloc
}
mxFree(re);
return;
}

void ihct2_begin(int16_t **yre)
{
    co = (int16_t *)mxMalloc(4*k * sizeof(int16_t));

    int16_t i, j, c=1, z, arg;
    int16_t *inpre, *outre; // vectors for storing rows and columns

    inpre = (int16_t *)mxMalloc(m * sizeof(int16_t));
    outre = (int16_t *)mxMalloc(m * sizeof(int16_t));

    arg = pi/2/k;
    co[0] = fac; // fac corresponds to 1.0 in float data type
    co[k] = 0; // corresponding to pi/2
    co[2*k] = -fac; // corresponding to pi

    // calculating cosine vectors
    for (i = 1; i < k; i++) // theta = 0 to pi/2
    {
        z = arg*i;
        co[i] = 1 + sin_calc(pi/2 - z - 8); // cos(x) = sin(pi/2-x)
        if (z > pi/4) // 45 degrees
            co[i] = co[i] + 5;
    }

    for (i = k+1; i < 2*k; i++) // theta = pi/2 to pi
    {
        co[i] = -co[k - c];
        c++;
    }

    c = 1;
    for (i = 2*k+1; i < 4*k; i++) // theta = +pi to 2pi (ie., -pi to 0)
    {
        co[i] = co[2*k - c];
        c++;
    } // end of cosine calculation

    //-----

    for (i = 0; i < n; i++) // calculating IDCT along rows
    {
        for (j = 0; j < m; j++)
            inpre[j] = yre[j][i];

        ihct_1D(inpre, outre, 2); // DHT of 1D vector
    }
}

```

```

        for (j = 0; j < m; j++)
            yre[j][i] = outre[j];
    }

    for (i = 0; i < m; i++) // calculating IDCT along columns
    {
        for (j = 0; j < n; j++)
            inpre[j] = yre[i][j];

        ihct_1D(inpre, outre, k/2); // divide by k/2
        for (j = 0; j < n; j++)
            yre[i][j] = outre[j]/4;
    }
    mxFree(co); // freeing memory allocated by mxMalloc
    mxFree(inpre);
    mxFree(outre);
    return;
}

// DCT of 1D array
void ihct_1D(int16_t *xre, int16_t *tre, int16_t div)
{
    int16_t i, j, p, r, theta;
    for (i = 0; i < n; i++)
    {
        r = 2*(i/1)+1;
        p = i%1;
        tre[i] = xre[p]*sqrt1by2/fac/div;
        for (j = 1; j < k; j++)
        {
            theta = (j*r) % (4*k);
            tre[i] += (xre[1*j+p]*co[theta])/fac/div;
        }
    }
    return;
}

// function to calculate sin(theta)
// using Taylor series expansion of sin(x) [= x - x^3/3! + x^5/5! - x^7/7! +
....]
// [https://en.wikipedia.org/wiki/Sine#Series_definition]
int16_t sin_calc(int16_t z)
{
    int16_t sine, temp;
    temp = z*z / 32768 * z / 32768 * z / 16384 * z / 4096 / 120;
    sine = z - z*z / 32768 * z / 8192 / 6 + temp;

    if (z > pi/2 // pi/4 radians
    {
        temp = -1 * temp * z / 16384 * z / 16384 / 42;
        sine = sine + temp;

        if (z > pi/2 * 80 / 90 // 80 degrees
            sine = sine + 1;
        }
    }
    return sine;
}

```


APPENDIX D

MATLAB CODES FOR HCT AND IHCT

D.1 MatLab Code for HCT

```
%Two-dimensional Hirschman Cosine Transform (HCT)
% using floating point.
%
% y = dht2_float(x,kdct,upsamp);
% this function call returns the matrix y, which is
% the 2-D DHT of the square matrix x using kdct point
% DCT at an upsampling rate of upsamp.
% If the matrix x contains n rows (or columns), the condition
% n=kdct*upsamp must be satisfied.

function [y] = hct2_float(x,kdct,upsamp)
    [m,n] = size(x);
    x = x - 128;
    if m ~= n
        error('The input must be a square matrix');
    end
    if length(kdct) ~= 1 || length(upsamp) ~= 1
        error(['The second and third arguments must ',...
            'be scalars']);
    end
    if m ~= kdct*upsamp
        error(['Size of the DFT times the upsampling ',...
            'rate must equal the rows of the input matrix.']);
    end
    w = dctmtx(kdct); % w stores the DCT transform matrix
    h = zeros(m);
    t = 1:upsamp:m;
    for i=0:upsamp-1 % populating the HCT transform matrix
        h(t+i,t+i) = w;
    end
    y = h*x*h';
end
```

D.2 MatLab Code for IHCT

```
%Two-dimensional Inverse Discrete Cosine Transform (IHCT)
% using floating point.
%
% y = ihct2_float(x,kdct,upsamp);
% this function call returns the matrix y, which is
% the 2-D DHT of the square matrix x using kdct point
% DCT at an upsampling rate of upsamp.
% If the matrix x contains n rows (or columns), the condition
% n=kdct*upsamp must be satisfied.

function [y] = ihct2_float(x,kdct,upsamp)
    [m,n] = size(x);
    if m ~= n
        error('The input must be a square matrix');
    end
    if length(kdct) ~= 1 || length(upsamp) ~= 1
        error(['The second and third arguments must ',...
            'be scalars']);
    end
    if m ~= kdct*upsamp
        error(['Size of the DFT times the upsampling ',...
            'rate must equal the rows of the input matrix.']);
    end
    w = dctmtx(kdct); % w stores the DCT transform matrix
    h = zeros(m);
    t = 1:upsamp:m;
    for i=0:upsamp-1 % populating the IHCT transform matrix
        h(t+i,t+i) = w;
    end
    y = h'*x*h;
    y = y + 128;
end
```

APPENDIX E

MATLAB CODES FOR TESTING

E.1 MatLab Code for Forward DHT and HCT Error Computation

```
% 2D DHT or HCT error calculation
% uses dht2_float.m and dht2_fixed.c files for DHT.
% uses hct2_float.m and hct2_fixed.c files for HCT.
%
% sel has value 1 or 2.
% if sel = 1, calculates MSE of the DHT transformed outputs
% if sel = 2, calculates MSE of the HCT transformed outputs

function [] = trans_error(sel)

m = 256; % size of input square matrix
iter = 10; % number of iterations
kmin = 4; % minimum value of k-point DFT
kmax = 128; % maximum value of k-point DFT

range = log2(kmax/kmin)+1;
avgmse = zeros(range,1); % average mean square error
maxmse = avgmse;
minmse = inf*ones(range,1);

%rng shuffle;
for i = 1:iter
    x = randi([0,255],m); % random integers between 0 and 255 (m*m image)
    k = kmin;
    j = 1;
    while(k <= kmax)
        l = m/k;
        if sel == 1 % DHT error
            yfix = dht2_fixed(x,k,l)/8;
            yflt = dht2_float(x,k,l);
        elseif sel == 2 % HCT error
            yfix = dct2_fixed(x,k,l)/8;
            yflt = dct2_float(x,k,l);
        end
        mse = immse(yfix, yflt); % calculate MSE
        avgmse(j) = avgmse(j) + mse;

        if mse < minmse(j) % minimum error
            minmse(j) = mse;
        end
        if mse > maxmse(j) % maximum error
            maxmse(j) = mse;
        end

        j = j+1;
        k = k*2;
    end
end
end
```

```

avgmse = avgmse/iter/(m*m);
avgdb = 10*log10(avgmse);
mindb = 10*log10(minmse/(m*m));
mindb = avgdb - mindb;
maxdb = 10*log10(maxmse/(m*m));
maxdb = maxdb - avgdb;
xaxis = (2.^(2:range+1))';

if sel == 1
    labelx = 'Length of DFT';
elseif sel == 2
    labelx = 'Length of DCT';
end

figure;
errorbar(xaxis,avgdb,mindb,maxdb, '.');
title('Mean Square Error (dB)');
ylabel('MSE (dB)'); xlabel(labelx);
text(xaxis,avgdb+0.5,num2cell(xaxis));
grid on;

end

```

E.2 MatLab Code for Processed Image Error Computation for DHT and HCT

```

% Total error on transform and inverse transform
% If sel = 1, calculates mean square error on DHT
% If sel = 2, calculates mean square error on DCT

function [] = img_error(sel)

m = 256; % size of input square matrix
iter = 10; % number of iterations
kmin = 4; % minimum value of k-point DFT
kmax = 128; % maximum value of k-point DFT

range = log2(kmax/kmin)+1;

avgmse = zeros(range,3); % mean sqmre error
maxerr = zeros(range,3); % maximum absolute error
mederr = zeros(range,3); % average median error

avgmse(:,2) = inf*ones(range,1);
maxerr(:,2) = inf*ones(range,1);
mederr(:,2) = inf*ones(range,1);

for i = 1:iter
    x = randi([0,255],m); % random integers between 0 and 255 (m*m image)
    k = kmin;
    j = 1;
    while(k <= kmax)
        l = m/k;

```

```

    if (sel == 1)
        y = dht2_fixed(x,k,1);
        z = idht2_fixed(y,k,1);
    elseif (sel == 2)
        y = dct2_fixed(x,k,1);
        z = idct2_fixed(y,k,1);
    end

    mse = immse(x, z); % calculate MSE
    avgmse(j,1) = avgmse(j,1) + mse;

    if mse < avgmse(j,2)
        avgmse(j,2) = mse; % minimum MSE
    end
    if mse > avgmse(j,3) % maximum MSE
        avgmse(j,3) = mse;
    end

    err = abs(x-z); % absolute error
    merr1 = max(max(err)); % maximum absolute error
    merr2 = median(err(:)); % median error
    maxerr(j,1) = maxerr(j,1) + merr1;
    mederr(j) = mederr(j) + merr2;

    if merr1 < maxerr(j,2) % least MAE
        maxerr(j,2) = merr1;
    end
    if merr1 > maxerr(j,3) % greatest MAE
        maxerr(j,3) = merr1;
    end

    if merr2 < mederr(j,2) % least Median error
        mederr(j,2) = merr2;
    end
    if merr2 > mederr(j,3) % greatest median error
        mederr(j,3) = merr2;
    end

    j = j+1;
    k = k*2;
end
end

if (sel == 1)
    labelx = 'Length of DFT';
elseif (sel == 2)
    labelx = 'Length of DCT';
end

avgmse = avgmse/(m*m);
avgmse(:,1) = avgmse(:,1)/iter;
maxerr(:,1) = maxerr(:,1)/iter;
mederr(:,1) = mederr(:,1)/iter;
avgdb = 10*log10(avgmse);
xaxis = (2.^(2:range+1))';

y = avgdb(:,1);

```

```

ymin = y - avgdb(:,2);
ymax = avgdb(:,3) - y;

figure;
errorbar(xaxis,y,ymin,ymax, '.'); % plot MSE
title('Range of Mean Square Error (dB)');
ylabel('MSE (dB)'); xlabel(labelx);
text(xaxis+0.5,y+0.3,num2cell(xaxis));
grid on;

y = maxerr(:,1);
ymin = y - maxerr(:,2);
ymax = maxerr(:,3) - y;

figure;
errorbar(xaxis,y,ymin,ymax, '.'); % plot MAE
title('Range of Maximum Absolute Error');
ylabel('Maximum Absolute Error'); xlabel(labelx);
text(xaxis+0.5,y+0.2,num2cell(xaxis));
grid on;

y = mederr(:,1);
ymin = y - mederr(:,2);
ymax = mederr(:,3) - y;

figure;
errorbar(xaxis,y,ymin,ymax, '.'); % plot Median error
title('Median Error');
ylabel('Median Error'); xlabel(labelx);
text(xaxis+0.5,y+0.1,num2cell(xaxis));
grid on;

end

```

REFERENCES

- [1] Tomasz Przebinda, Victor DeBrunner, and Murad Ozaydin, "The Optimal Transform for the Discrete Hirschman Uncertainty Principle", *IEEE Transactions on Information Theory*, vol. 47, no. 5, pp. 2086-2090, Jul 2001.
- [2] Peng Xi, and Victor DeBrunner, "Hardware Implementation of a Series of Transform Matrices Based on Discrete Hirschman Transform", *2016 50th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2016, pp. 229-232.
- [3] "Benefits of Using Fixed-Point Hardware", *Mathworks.com*, 2017. [Online]. Available: <https://www.mathworks.com/help/fixedpoint/gs/benefits-of-fixed-point-hardware.html>. [Accessed: 20- Nov- 2017].
- [4] M. M. Levy, "Fourier Transform Analysis", *Journal of the British Institution of Radio Engineers*, vol. 6, issue 6, December 1946
- [5] P. Kabal, and B. Sayar, "Performance of Fixed-point FFT's: Rounding and Scaling Considerations", *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Japan, April 1986.
- [6] Wei-Hsin Chang, and Truong Q. Nguyen, "On the Fixed-Point Accuracy Analysis of FFT Algorithms", *IEEE Transactions on Signal Processing*, vol. 56, issue 10, October 2008.
- [7] James W. Cooley, and John W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", in *Mathematics of Computation* vol. 19, no. 90, pp. 297-301, April 1965.
- [8] M. Heideman, D. Johnson, and C. Burrus, "Gauss and the History of the Fast Fourier Transform", *IEEE ASSP Magazine*, vol. 1, issue 4, pp. 14-21, October 1984.
- [9] J. Cooley, P. Lewis, and P. Welch, "Historical notes on the fast Fourier transform", *IEEE Transactions on Audio and Electroacoustics*, vol. 15, issue 2, June 1967.
- [10] Kirandeep Ghuman, "Some Theory and an Experiment on the Fundamentals of Hirschman Uncertainty", Ph.D. dissertation, Dept. Elect. and Comp. Eng., Florida State Univ., Tallahassee, 2015.
- [11] G. Liu and V. DeBrunner, "High-resolution non-parametric spectral estimation using the Hirschman optimal transform," *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Kyoto, 2012, pp. 3721-3724.
- [12] Alan V. Oppenheim and Ronald W. Schaffer, "Introduction", in *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, Inc., 1999.

BIOGRAPHICAL SKETCH

Rajesh Thomas was born in Trivandrum, Kerala, India. He has a great interest in science and technology and developed the passion for science from a very young age itself. His parents were very motivational and helped him set up a small physics and chemistry lab at their home when he was studying in elementary school. He obtained his Bachelor's Degree from the Vellore Institute of Technology (VIT), Vellore, India in Electronics and Communication Engineering in 2015.

Rajesh continued his educational career in the Florida State University (FSU) where he got joined the Master's Program in Electrical Engineering at the FAMU-FSU College of Engineering. His areas of interest include Image Processing, Digital Signal Processing (DSP) and DSP with FPGAs. He has been working as a Teaching Assistant (TA) in the Department of Electrical and Computer Engineering at FSU for four semesters during 2016 and 2017. As a TA, he has worked in Digital Logic Design and Microcontroller based System Design laboratory courses. Currently he is working as a TA for the Digital Logic Design laboratory course.