



EERI 413

PRACTICAL 1

IMAGE PROCESSING

C.R van Zyl

21492204

Prof. W.C Venter

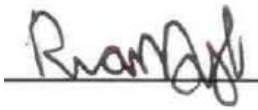
21/05/2012

Potchefstroom
2012

DECLARATION

I C.R van Zyl declare that this report is a presentation of my own original work. Whenever contributions of others are involved, every effort was made to indicate this clearly, with due reference to the literature. No part of this work has been submitted in the past, or is being submitted, for a degree of examination at any other university or course.

Signed on this 21st day of May 2012, in Potchefstroom.

A handwritten signature in dark ink, appearing to read 'Carel Ruan van Zyl', is written over a horizontal line.

Carel Ruan van Zyl

TABLE OF CONTENTS

| | |
|---------------------------------------|----|
| DECLARATION | 2 |
| 1. INTRODUCTION | 6 |
| 2. GOAL..... | 7 |
| 3. METHOD | 8 |
| 3.1 FFT | 8 |
| 3.2 Isolating Different Colours | 9 |
| 3.3 Edge Detection..... | 10 |
| 3.4 Low-Pass Filter..... | 11 |
| 4. RESULTS | 12 |
| 4.2 Assignment 1..... | 12 |
| 4.3 Assignment 2..... | 13 |
| 5. CONCLUSION..... | 15 |
| 6. REFERENCES | 16 |

LIST OF FIGURES

| | |
|--|----|
| Figure 1: Read Image Method..... | 8 |
| Figure 2: Greyscale Implementation of RGB..... | 9 |
| Figure 3: Red Implementation of RGB | 10 |
| Figure 4: Smooth Method..... | 11 |

LIST OF TABLES

Table 1: Practical Goals7

1. INTRODUCTION

Numerous algorithms exist which can be implemented in software in order to process digital signals. One of the most powerful algorithms in the field of DSP (Digital Signal Processing) is the DFT (Discrete Fourier Transform). The purpose of the DFT is to decompose a sequence into components of different frequencies. Computation of the DFT is only possible if the input signal is discrete. Since naturally occurring signals are continuous, it is common practice to first sample the signal before computing its DFT. Digital signal processing is more easily applied in the frequency domain. Computing the input signal's DFT is consequently the first step in most signal processing tasks. It is, however, often too slow in terms of processing time to compute the DFT of a signal directly from the definition. In practical applications we use the FFT (Fast Fourier Transform), which is a mathematical technique which greatly reduces the number of operations required to compute a DFT.

Image processing is a form of signal processing where the input is an image (which is already sampled). Images are typically treated as a two-dimensional signal, where its intensity at any point is a function of two spatial variables [1]. An image can be decomposed into its frequency representation by applying a FFT algorithm. It should be noted that the FFT can only be applied on square images whose size is an integer power of 2 (ex. 256x256). The calculation of the FFT makes effective use of the separability property of the Fourier Transform. Separability means that the Fourier Transform is calculated in two stages; the rows are first transformed by means of a 1D FFT, then the data is transformed in columns, again using a 1D FFT. Since the computational cost of a 1D DFT of N points is $O(N \log(N))$, the cost (by separability) for the 2D FFT is $O(N^2 \log(N))$ whereas the computational cost of the 2D DFT is $O(N^3)$ [2]. The result of the aforementioned discussion is that a FFT takes much less time to be computed in comparison with a standard DFT algorithm.

2. GOAL

The goal of the practical is to develop software capable of demonstration the principles of digital signal processing in the Visual Studio environment. Input signals are assumed to be digital images in either .bmp or .jpg formats. The practical is divided into two assignments, as shown in Table 1:

| Assignment 1 | Assignment 2 |
|---|--|
| Display a black and white image | Display a colour image |
| Determine and display the DFT of the image | Display the RGB (Red, Green, and Blue) components of the image |
| Remove high frequency components of the image by filtering (Low-pass filter) | Determine and display the edges of the red, green, and blue components present in the original image (Edge Detection) |
| Calculate the inverse DFT and display it alongside the original image in order to demonstrate the effect of filtering | Merge the red, green, and blue component images wherein the edges have been detected, into one image and display the edges that are found in the original colour image |

Table 1: Practical Goals

Table 1 is the basis on which a GUI (Graphical User Interface) was designed for the software. Each image was placed alongside its comparable counterpart (where applicable) in order to properly demonstrate the software.

3. METHOD

Under this section several important methods used to complete the practical will be discussed. Code snippets will be presented and discussed.

3.1 FFT

The student made use of the most common FFT implementation, the Cooley-Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1 N_2$ into smaller DFTs of sizes N_1 and N_2 . The Cooley-Tukey algorithm was chosen because of the large amount of literature on the subject which made it easier to implement. [3]

Firstly the target image is read into a 2D array of integers. Each element within the 2D array is accessible by means of a pointer variable defined as imagePointer. The function populates the array by making use of int variables 'i' and 'j' to represent the rows and columns respectively.

Figure 1 shows how the image was read into the 2D array as described above:

```
private void ReadImage()
{
    int i, j;
    GreyImage = new int[Width, Height]; //i = row, j = column
    Bitmap image = Obj;
    BitmapData bitmapData = image.LockBits(new Rectangle(0, 0, image.Width, image.Height),
        ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

    unsafe
    {
        byte* imagePointer = (byte*)bitmapData.Scan0; //Pointer to matrix elements

        for (i = 0; i < bitmapData.Height; i++)
        {
            for (j = 0; j < bitmapData.Width; j++)
            {
                GreyImage[j, i] = (int)((imagePointer[0] + imagePointer[1] + imagePointer[2])/3.0);
                imagePointer += 4;
            }

            imagePointer += bitmapData.Stride - (bitmapData.Width*4);
        }
    }
    image.UnlockBits(bitmapData);

    return;
}
```

Figure 1: Read Image Method

It is furthermore worth noting that each pixel is represented by 4 bytes of data. Now that the image is in matrix-form, we can apply mathematical operations on them.

Complete source code is appended to this document and can be found in Appendix A.

3.2 Isolating Different Colours

In solving the problem to isolate the RGB (Red, Green, and Blue) colours contained in the original image, a function called RGB was written. RGB accepts two parameters, a 24-bit Bitmap image, and a string that specifies what the user wishes to extract from the Bitmap. Three colour-based components can be extracted: Greyscale, Red, Green, or Blue. The function generically (before even knowing the user's intended purpose for it) prepares the imported image for 2D matrix access, i.e. each pixel of the image becomes accessible through software. Next the function reads the contents of the image matrix and modifies the individual pixels according to user input. Below is a description of each of the available options, supported by code snippets:

Greyscale

Figure 2 shows a code snippet applicable to the 'Greyscale' operation:

```
if (colour == "Greyscale")
{
    p[0] = p[1] = p[2] = (byte)(0.299 * red + 0.587 * green + 0.114 * blue);
}
```

Figure 2: Greyscale Implementation of RGB

As can be seen from Figure 2, the software masks out contributions of the respective primary colour components contained in the image, without sacrificing quality. The result is a perfect greyscale version of the original image. Note: p[0], p[1], p[2] refers to the blue, green, and red components respectively.

Red

Figure 3 shows a code snippet applicable to the 'Red' operation of the RGB function:

```
else if (colour == "Red")
{
    p[2] = (byte)(1 * red);

    p[0] = 0;
    p[1] = 0;
}
```

Figure 3: Red Implementation of RGB

From Figure 3 one can see that the software masks out contributions from the green and blue spectrum contained in the original image. The red component is, however, held intact as to implement the RGB colour isolation.

The remaining options (green and blue) follow a similar procedure and will not be discussed.

3.3 Edge Detection

Difference edge detection was used to handle the edge detection aspect of the practical. Difference edge detection detects the difference between pairs of pixels. It calculates the highest difference between a pixel and its eight neighbours. A threshold was included which allows softer edges (small differences) to be forced down to black, when employed only prominent edges as visible. The difference edge detection technique described here was used to determine the red, green, and blue edges contained in the original image as well as the RGB edge detection assignment. Note: It is also possible to make use of convolution to implement edge detection.

3.4 Low-Pass Filter

In solving the problem of the low-pass image filter, a convolution filter was chosen. The implemented convolution filter works by moving a 3 x 3 matrix through the 2D image matrix. As the matrix is moved over the image matrix, the current index of the filter is given by the middle entry of the matrix, i.e. S_{21} . The filter then calculates a relative weight of the pixel in question and the eight surrounding it. The total value of the matrix is then divided by a factor, and an offset is added to the end value. Since the image is not changed by passing the matrix over the pixels, the convolution matrix is an identity matrix. A function called Smooth was written to apply the low-pass filter. 'Smooth' essentially spreads the weight of each pixel over the surrounding area, creating a blurred image. [5]

Figure 4 shows the 'Smooth' method:

```
public static bool Smooth(Bitmap b, int nWeight)
{
    ConvMatrix m = new ConvMatrix();
    m.SetAll(1);
    m.Pixel = nWeight;
    m.Factor = nWeight + 10;

    return BitmapFilter.Conv3x3(b, m);
}
```

Figure 4: Smooth Method

The 'Smooth' method accepts a target variable of type Bitmap and a integer that defines the aggressiveness of the filter, a larger number in the nWeight field causes more severe blurring. The return value of the method is passed through a standard 3 x 3 convolution function.

4. RESULTS

Under this section figures will be provided in order to demonstrate the proper operation of the developed software. It was considered appropriate to choose a image that has clear colour boundaries in order to demonstrate the software. The results will be presented under two seperate headings, namely Assignment 1 and 2 as described under Table 1, Section 2.

4.2 Assignment 1

Figure 5 shows the results of Assignment 1:

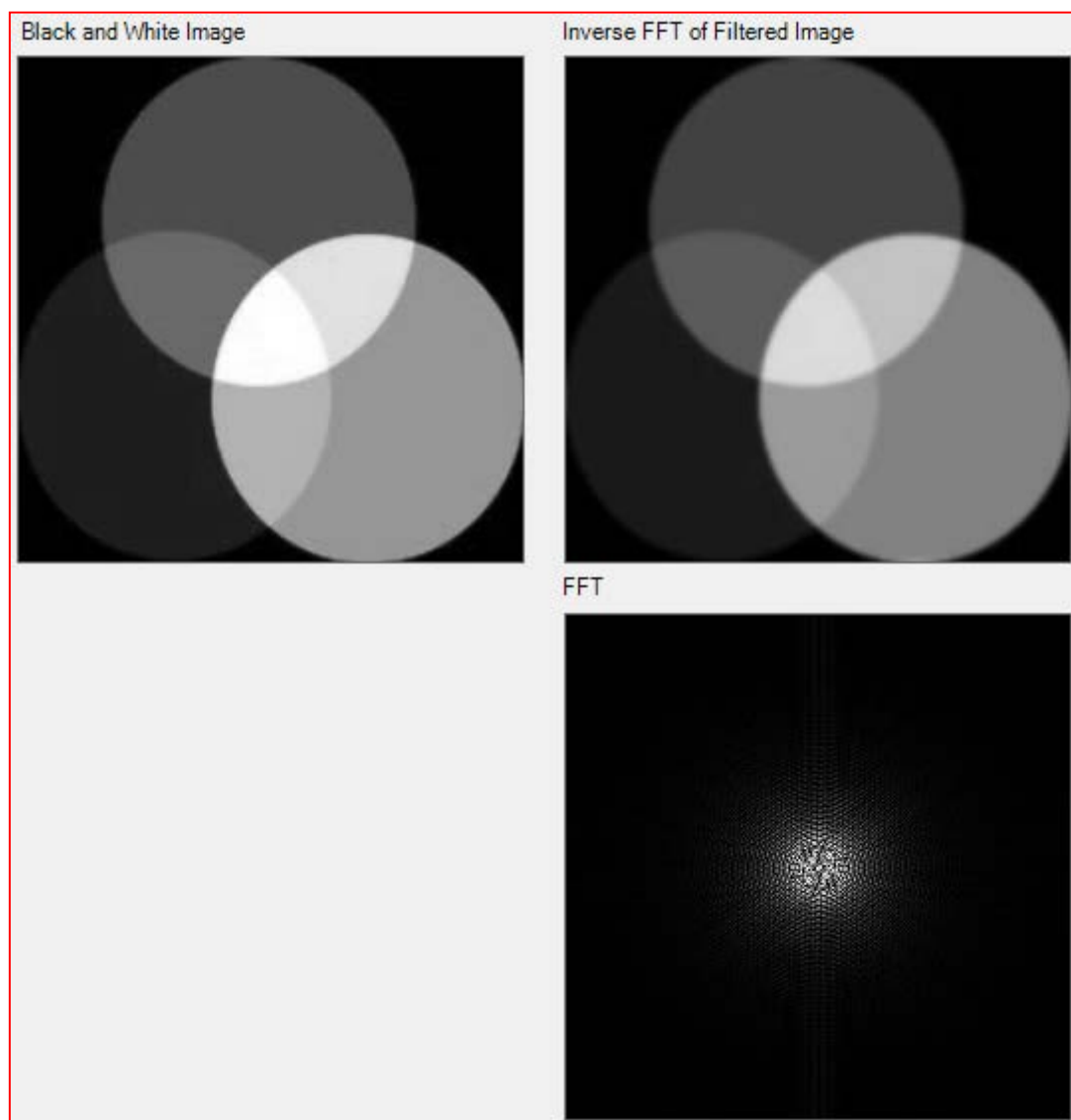


Figure 5: Results for Assignment 1

It can clearly be seen in Figure 5 that the filtered image (which was determined by calculating the inverse FFT) is a blurred version of the original (black and white) image. The Fourier spectrum is also shown.

4.3 Assignment 2

Figure 6 shows the results of Assignment 2:

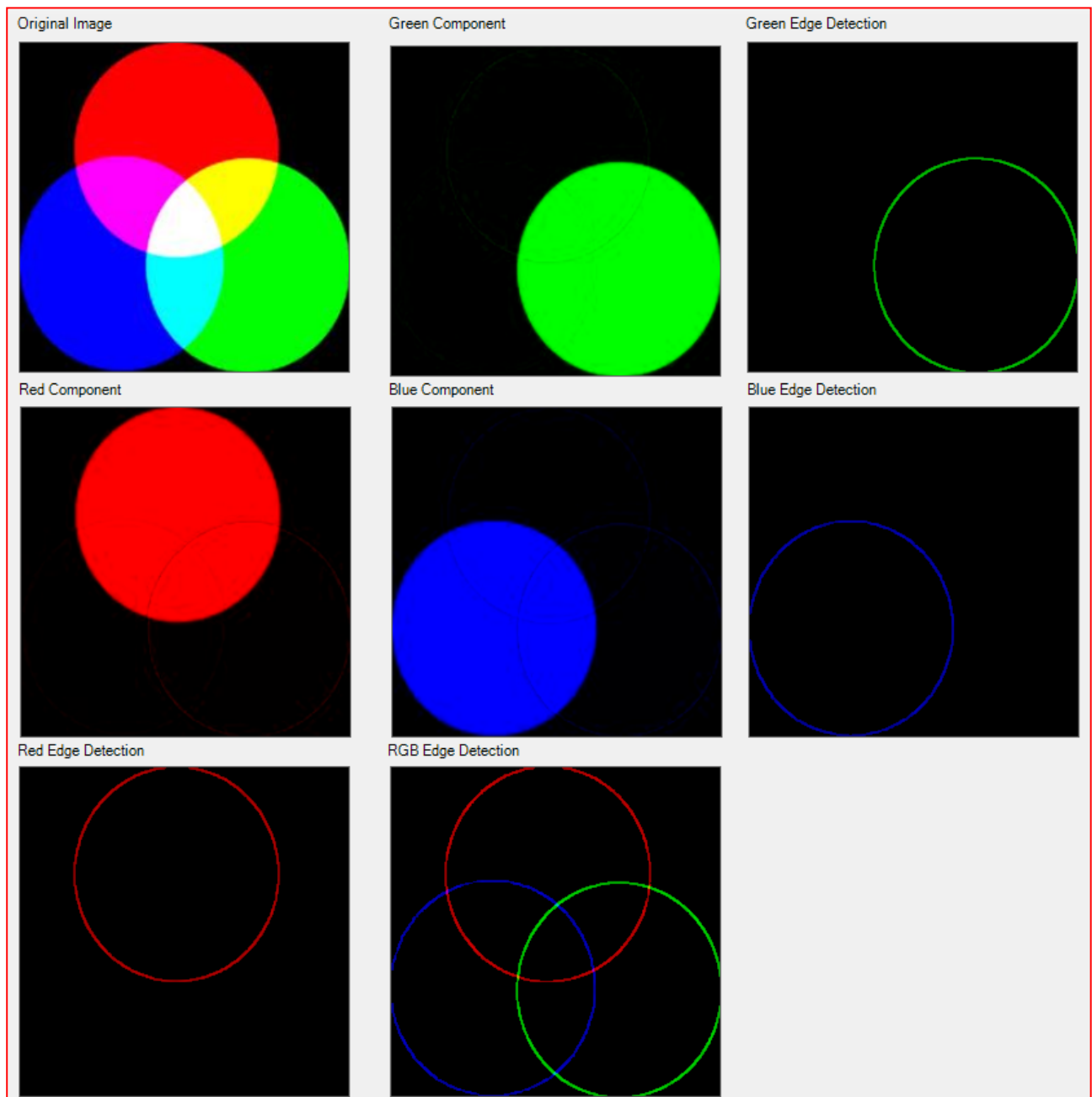


Figure 6: Results Assignment 2

As can be seen in Figure 6, the software perfectly distinguishes between the three primary colours. Edge detection also seems to be correct. The image used for this demonstration is very simple but it outlines the operation of the software.

5. CONCLUSION

The practical was completed successfully and a lot of insight was gained in the implementation of DSP principles from a software point of view. In future implementations the image processing toolbox of MATLAB will likely be used as opposed to directly programming the required mathematical equations. It was, however, insightful to observe commonly used image processing algorithms from first principles.

6. REFERENCES

- [1] S.K Mitra. *Digital Signal Processing: A Computer Based Approach*, 4th Edition. New York, NY: McGraw Hill, 2011, pp.20
- [2] M.S Nixon, A.S Aguado. *Feature Extraction and Image Processing*, 1st Edition. Woburn, MA: Newnes, 2002, pp. 50.
- [3] V.A Bharadi. "2D FFT of an Image in C#." Internet: <http://www.codeproject.com/Articles/44166/2D-FFT-of-an-Image-in-C>, Nov. 20, 2009 [May 16, 2012].
- [4] C. Graus. "Image Processing for Dummies with C# and GDI+ Part 3 – Edge Detection Filters." Internet: <http://www.codeproject.com/Articles/2056/Image-Processing-for-Dummies-with-C-and-GDI-Part-3>, Apr. 1, 2002 [May 16, 2012].
- [5] C.Graus. "Image Processing for Dummies with C# and GDI+ Part 2 – Convolution Filters". Internet: <http://www.codeproject.com/Articles/2008/Image-Processing-for-Dummies-with-C-and-GDI-Part-2>, Nov. 7, 2005 [May 16, 2012].

APPENDIX A

Under this section all source code relevant to the practical was included.