Cardiff School of Computer Science & Informatics

Final Report

Image Steganography

***Author**: Josh Preston*

***Student number:** C1737263*

***Supervisor**: Paul L Rosin*

***Moderator**: Yuhua Li*

***Module Number**: CM3203*

***Module Title**: One Semester Individual Project*

***Credits**: 40*

Acknowledgements

I would like to thank Paul, my supervisor, in helping me with this project when I did not understand certain aspects of it.

I would also like to thank my housemates who have had to tolerate me when programs did not work as intended.

Abstract

Steganography can be used to embed security information in images. This project has focused on images that have been binarised by a half-toning algorithm. These binarised images will be used to embed this information. Various half-toning and embedding algorithms have been implemented with each embedding algorithm tested on all half-toning algorithms. This report will compare and analyse how the choice of half-toning algorithm can influence each embedding algorithm.

Table of Contents

Table of figures

1. Introduction

This section aims to provide the reader what the project is about. An overview of steganography is introduced.

1.1. Steganography

Steganography is the art of concealing messages or information within objects. The word originates from the Greek words "*stegos*" which means "covered" and "graphia" which means "writing" (Krenn, 2004). Steganography, like cryptography is a means of providing secrecy by manipulating information to cipher or hide the existence of it (Bloisi and Iocchi, 2007). Secrecy by steganography is achieved by hiding the existence of information within an object, whereas it is achieved through cryptography by scrambling communication so that it cannot be understood (Johnson and Jaiodia, 1998). Cryptography can draw unwanted attention to the scrambled information, sometimes referred to as ciphertext, as it is in plain sight and could be easily decrypted if a weak encryption is used. Steganography is less likely to have attention drawn to it as information is hidden from view and essentially made "invisible" thus concealing the fact information has been sent in the first place (Johnson et al., 2000). A combination of the two methods would mean better overall secrecy of information as it would be both scrambled and hidden from view.

Throughout history, many methods have been used to hide information. One of the first documents describing steganography is from the Histories of Herodotus. A wax covered tablet had text written beneath the layer of wax. Demeratus wished to notify Sparta that Xerxes intended to invade Greece. This was done by scraping off the wax from a tablet, writing a message on the underlying wood, and then resealing the tablet with wax so that the message was hidden from view (Johnson et al., 2000). Another method was is a form of invisible writing which was made possible using invisible inks. This method was used during World War 1 and 2 and had success in sending hidden messages within seemingly innocent letters and made visible when heated (Zim, 1948). Null ciphers are also a form of steganography. This is where a message is transmitted, and a hidden message is within the "innocent" message. An example of an innocent message is: "Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issues affects pretext for embargo on by-products, ejecting suets and vegetable oils.". This was a message sent by a German Spy in World War 2 (Kahn, 1996). If every second letter was taken from each word, the sentence "Pershing sails from NY June 1".

There has been a steady increase in interest with steganography due to the widespread use of digital media and expansion in the internet (Fig.1). An ever-increasing number of images are being sent through the internet which provide a favourable environment for concealing hidden information. This is due the large number of pixels which make up images. These can be imperceptibly modified to encode a secret message (Fridrich, 2009).

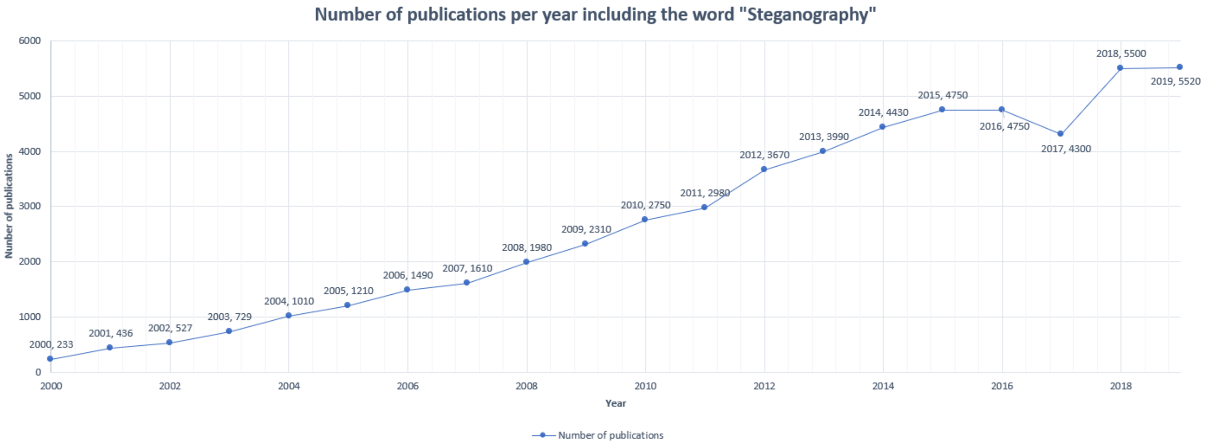


Figure 1 – Number of publications per year that include the word "Steganography" on Google Scholar.

1.2. Aims of the project

The aim for this project is to implement available algorithms that halftone images and to embed information within the halftoned images. I will be comparing and analysing different halftone algorithms. I will also be comparing and analysing different embedding algorithms and how the choice of halftoning algorithm can influence embedding success.

1.3. Intended audience

The intended audience for this report is for individuals seeking knowledge on different half-toning algorithms and steganographic algorithms which use halftoned images. The half-toning algorithms modify greyscale images to produce a binarised image. The steganographic algorithms use the binarised images to embed hidden information (text or image). The report will provide results and analysis of different combinations of halftone and steganographic algorithms to hide either text or images to see how each halftoning method influences embedding success.

1.4. Project scope

This project will consist of different halftoning and embedding algorithms. I originally planned to analyse and compare around 10 different embedding algorithms but due to research and implementation time, a lower number will be implemented.

I also originally planned to implement different attacks that will be used to extract embedded information within images but due to research and implementation time, this has changed to specific extraction programs for each embedding algorithm.

Early in the project, the Levenshtein Distance measure of success in extracted information was found to be inappropriate for use in this type of project. Instead, success has changed to visual quality and a simpler measure of percentage of extracted information.

A lower number of images used in the project has been reduced. I found the original plan of hundreds of images a little ambitious. The total number will be around 50 images.

The scope of the project is still largely the same as stated in the initial report but will have a lower number of different embedding algorithms and test images. Specific extraction programs will be implanted for each embedding algorithm rather than implementing various attacks.

1.5. Approach used in carrying out the project

The approach used in carrying out this project is the Agile methodology.

Planning and requirements for each halftoning and embedding algorithm implemented was done prior to the build phase. This was so I knew the deliverables required from the algorithm which resulted in a much clearer approach made to the design of the algorithms.

As I knew the required deliverables from the planning and requirements stage, the rough outline to each algorithm could be designed. This outline defining functions that will read images in, halftone them or embed them, and output the processed image.

From this rough outline the algorithms could then be built in the functions defined.

After each function was completed, it was tested to ensure it fully worked. If it did not work as intended, it was evaluated and required changes found during evaluation were then implemented. This part was repeated until the algorithm worked as intended.

2. Background

This section of the report aims to provide the reader the information needed to fully understand and appreciate the rest of the report. Other relevant work will be written about and explain why the project is addressing the problem identified.

2.1. Image processing

Image processing is any method that starts with a colour, or greyscale, image and applies operations that will return another image or an alteration of the original (Russ, 1990). There are a multitude of methods that are used in image processing with each one resulting in a differently processed image.

2.1.1. Half-toning

Halftoning, digital halftoning in this case as the project focuses on images, is the set of techniques used by a computer to display perceived greyscale images on devices that are only capable of displaying black and white pixels (Nielsen, 2005). It creates the illusion of a greyscale image using only black and white pixels. Varying densities of black pixels against white pixels create different perceived greyscales when viewed from a distance (Pitas, 2000). Historically, the halftoning technique has been used in the newspaper industry. Various sized black dots of ink, called halftones, are printed onto the paper to give the effect of a greyscale image. The size of the halftone was adjusted according to the local image intensity, known as analog halftoning (Meşe and Vaidyanathan, 2002).

2.1.2. Dithering

Dithering is a halftoning technique that is used to create the appearance of variable tone levels. It does this by controlling the spatial distribution of black and white pixels (Spaulding et al., 1997) rather than the size of the dot. Through doing this, different grey levels are perceived by local averaging (Nielsen, 2005).

2.1.2.1. Error diffusion dithering

Error diffusion is an approach that can be used to create a dithered image. An image is processed in a raster scanline pattern, left to right for every row in the image, and every pixel processed will be tested against a threshold. This threshold is usually half of the highest intensity (Deussen and Isenberg, 2013). An error known as the quantisation error is defined as the difference of the original greyscale images pixel value minus the result of thresholding the same pixel. The quantisation error is then distributed proportionately to neighbouring pixels, defined through a matrix, which have not yet been visited by the scanline method (Nielson, 2005). In Fig.2, this process is shown through a block diagram. The matrix defines which neighbouring pixels will have the error distributed to and defines what proportion of the error will be distributed (Knuth, 1987). Fig.3

demonstrates a simple matrix that shows the current pixel being scanned and which neighbouring pixels will have the proportionate error diffused to. If the value of the pixel currently being accessed is higher than the threshold, it will become white. If the value is not higher than the threshold, it will become black.

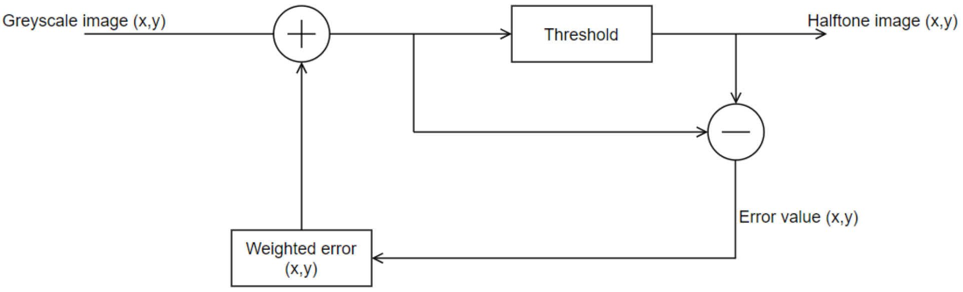


Figure 2 – A block diagram of the error diffusion algorithms implemented (Chung et al., 2011).

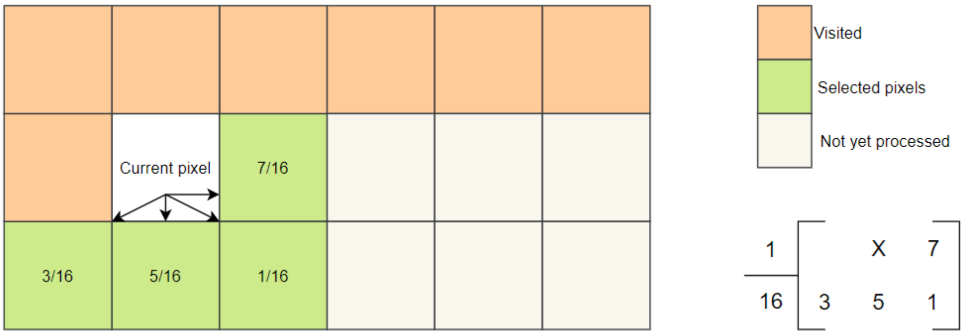
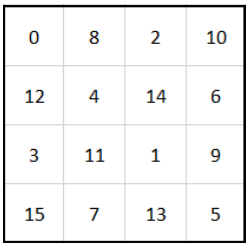


Figure 3 – A diagram demonstrating the proportion of the quantised error distributed to neighbouring pixels.

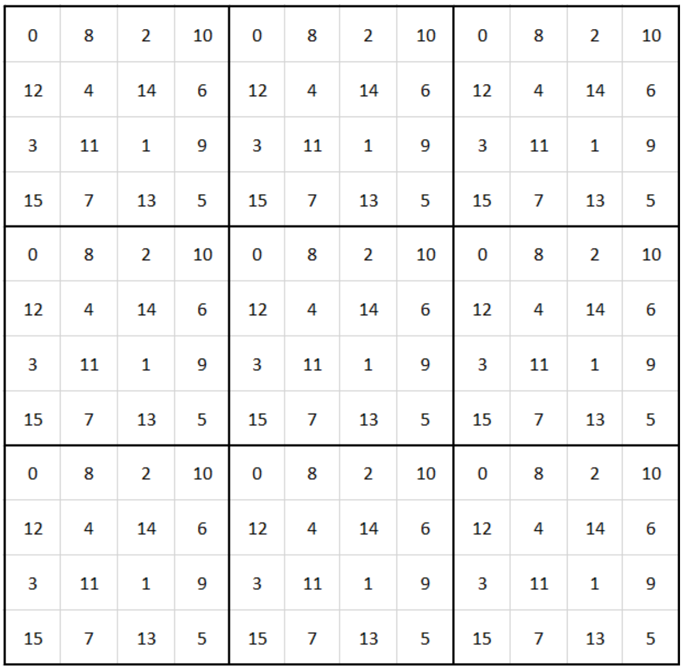
2.1.2.2. Ordered dithering

Ordered dithering is another approach that can be used to create dithered images. It is a much faster method in creating a dithered image as only one comparison is needed to halftone a pixel. This means that each pixel in the image can be processed simultaneously (Zhang, 1997). A threshold map matrix as shown in Fig.4 is defined and processes the input images corresponding pixel (x, y) against the number in the threshold map (x, y). If the images pixel value is higher than the threshold value, the halftone pixel colour will be white. Otherwise, it will be black. If, for instance, the image dimensions are larger than the threshold maps dimensions, the map will be ‘wrapped’ at the edges. This is shown in Fig.5 where the same matrix is positioned at the edges of one another so that the map has the same dimensions as the input image. Due to the structure of the matrix, different patterns are created depending on the greyscale value of the original image, see Fig.6.



0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

Figure 4 – A diagram demonstrating the 4x4 ordered matrix.



0	8	2	10	0	8	2	10	0	8	2	10
12	4	14	6	12	4	14	6	12	4	14	6
3	11	1	9	3	11	1	9	3	11	1	9
15	7	13	5	15	7	13	5	15	7	13	5
0	8	2	10	0	8	2	10	0	8	2	10
12	4	14	6	12	4	14	6	12	4	14	6
3	11	1	9	3	11	1	9	3	11	1	9
15	7	13	5	15	7	13	5	15	7	13	5
0	8	2	10	0	8	2	10	0	8	2	10
12	4	14	6	12	4	14	6	12	4	14	6
3	11	1	9	3	11	1	9	3	11	1	9
15	7	13	5	15	7	13	5	15	7	13	5

Figure 5 – A diagram demonstrating the 4x4 ordered matrix being wrapped at the edges.

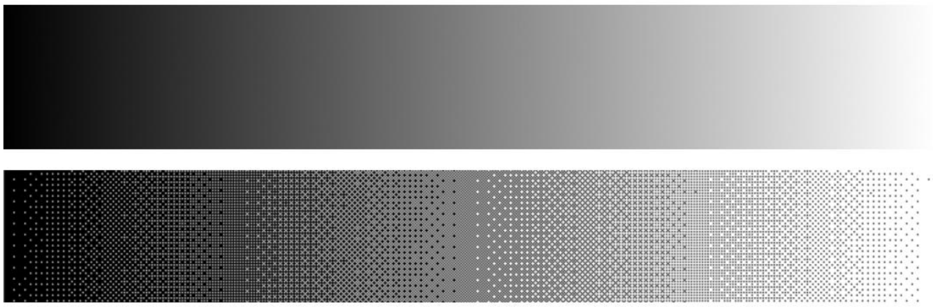


Figure 6 – Demonstration of the patterns created with ordered dithering.

2.1.3. Stippling

Stippling is an artistic technique that produces halftoned images. It does this by creating numerous small repetitive marks, known as stipples, to create an image (Kim et al., 2009). Each individual stipple is purposely placed so that a high quality black and white image is created with different perceived greyscales. An algorithm known as Lloyds algorithm controls where every stipple is placed with the creation or a Voronoi diagram. A Voronoi diagram is a diagram that has been segmented into areas that are determined from a set of points. These points can be determined by an input image (Kise et al., 1997) or they can be randomly generated. These points within the areas are then updated in their position to the centroid of the area (Secord, 2002). Different weights can be added to the centroid calculations so that areas of the diagram are more densely populated than others. These weights are determined by the input image. After this, a high quality halftoned image is created. Fig.7 demonstrates the black pixels having equal space where the input greyscale image had a uniform colour.

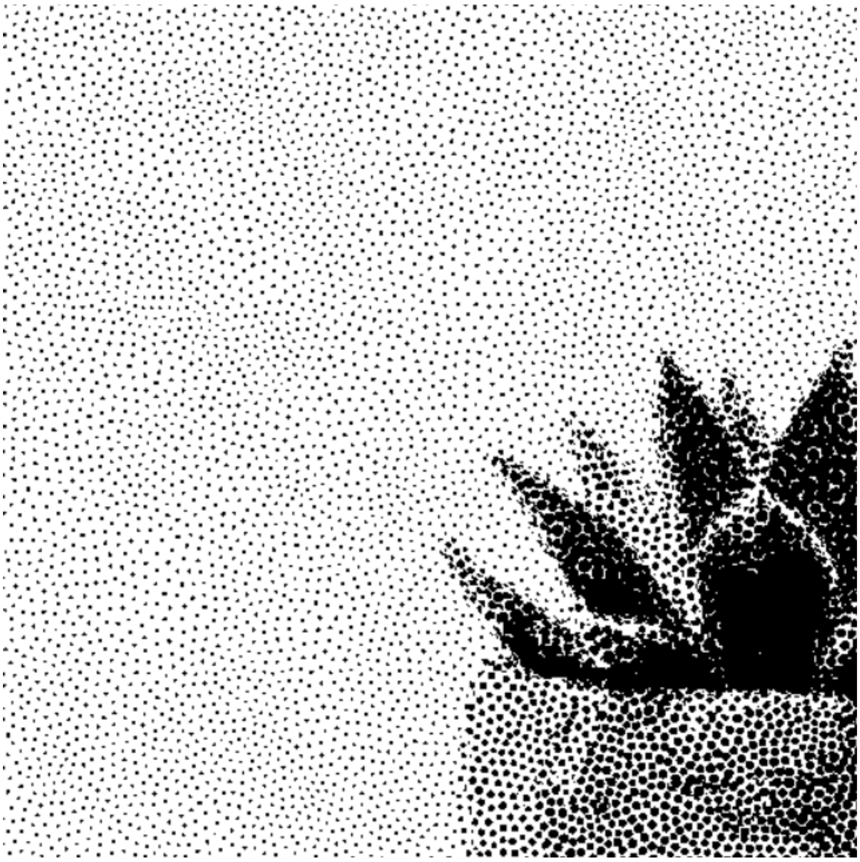


Figure 7 – Demonstration of equal spaced stipples from areas with similar colours in the original greyscale image

2.2. Image metrics

As this project focuses on image processing, image quality metrics are needed to determine the quality between the original image and the processed image. Any processing made to an image could cause a loss of information or of image quality (Horé and Ziou, 2010). These quality evaluation metrics can be split into two areas: objective and subjective. Objective evaluation metrics use a reference image against a processed image to give the quality between the two images. Subjective evaluation metrics do not use a reference image but instead are based on human judgement using a criterion. I have chosen two popular objective evaluation metrics as this project focuses on minimising any degradation of quality when halftoning and/or embedding information.

2.2.1. PSNR

Peak Signal-to-Noise Ratio (PSNR) is a metric that uses MSE, mean square error, which is computed by averaging the squared differences of the distorted processed pixels against the corresponding reference pixels of the original image. MSE is the most widely used and simplest full reference metric (Sara et al., 2019). PSNR is calculated from MSE in that it averages the values calculated using MSE with the peak signal-to-noise ratio (Wang et al., 2004) so that the peak noise is returned. This is essentially the amount of noise which has been added to an image as a result of halftoning and/or embedding.

2.2.2. SSIM

Structural Similarity, SSIM, is a metric developed by Wang et al. It is an improvement on PSNR in that it is correlated with the quality perception in the human visual system. SSIM uses three different factors in determining the quality of the processed image against the reference image. These are: luminance, contrast, and structure comparison. The luminance comparison returns the distance in the values calculated from the mean luminance of the processed and reference image. The contrast comparison returns the distance in the values calculated from the contrast values from the processed and reference image. Finally, the structure comparison returns the correlation coefficient between the processed and reference image (Horé and Ziou, 2010). The three of these factors combined result in the comparison of local patterns of pixel intensities that have been normalised for luminance and contrast (Wang et al., 2004).

2.3. Watermarking

Watermarking, digital watermarking in this case as the project focusses on images, is information that is imperceptibly and robustly embedded in an image (Hartung and Kuttler, 1999). A regular watermark in images usually contains information about the origin, status, or even the recipient of the image. Watermarks can also be used for copyright protection. Watermarks, for example, are printed into paper currency such as a £20 note. There are two properties to a watermark, the first being that the watermark is hidden from view during normal use and only becoming visible because of special viewing processes. The second is that a watermark carries information about the object in which it is hidden. In the case of a £20 note, it indicates the authenticity of the note rendering it legal (Cox et al., 2007). All watermarking methods share the same building blocks: a watermark embedding system and a watermark recovery system (Katzenbeisser and Petitcolas., 2000). It can be of any format such as a number, text, or an image.

2.4. Problem identified

There are many existing algorithms that either halftone an image or embed information into an image. This project aims to find which implemented halftoning technique and implemented embedding technique combined give the best results in terms of image quality after processing and the percentage of the embedded message extracted after the embedding process. Every image selected for testing in this project has been put into two categories: complex and non-complex. I aim to see if there is a correlation between halftoning and embedding methods giving better or poorer results depending on if an image is simple or complex. Each embedding algorithm has been specifically chosen or adapted to include a threshold, which is used to determine the strength at which the information is embedded at. Some embedding algorithms have also been adapted so that other halftoning methods can be used with them. I also aim to see if image quality is affected equally in simple and complex halftone images after embedding information with different thresholds.

There have been papers that have investigated the correlation between image complexity to the embedding capacity (Sajedi and Jamzad, 2010). There have also been algorithms that embed within complex areas (Ramani et al., 2007). But none, to my knowledge, have really tested algorithm combinations (halftoning and embedding) together with varying thresholds testing to see if complex and non-complex images perform better than one another in the same tests and same varying thresholds.

2.5. Data types and libraries implemented

This section is to explain any methods and tools the project utilises. Any methods or tools that are not common knowledge will be further explained below.

The project algorithms are written in Python version 3.8.2. Some require additional libraries to implement elements of the algorithm. All algorithms have the required import statements at the top of the file. All external libraries used: NumPy, OS, Time, PIL (Image, ImageFilter), Openpyxl, Sys, Binascii, Statistics, Collections (Counter), Traceback, Re, and OpenCV (cv2).

2.5.1.1. NumPy

Numpy (Numeric Python) arrays have been implemented into most algorithms for this project. This is due to the structure of image files and needing to make use of arrays that can store pixel values at corresponding positions. NumPy is an open-source add-on module that allow mathematical routines which are pre-compiled and fast (Shell, 2019).

2.5.1.2. OS

OS has been used to add external folders to the programs path. This is required when any external file outside of the programs folder is to be executed to obtain results from tests.

2.5.1.3. Openpyxl

Openpyxl is a Python library that can read/write Excel documents (Openpyxl, 2020). This library has been chosen as I required data from implemented algorithms to be stored in a manner that graphs can be created.

2.5.1.4. Binascii

Binascii is a Python module that contain methods to convert between binary and ascii text (Binascii). It is also capable of converting ascii text to binary. This module was chosen due to the embedding procedures implemented in the algorithms. Information, such as text, needs to be converted to binary so that it can be embedded into the image. Once embedded in binary format, it needs to become readable once extracted from the embedded image.

2.5.1.5. PIL - Image

Pillow is a popular image processing module in Python (Pillow). The “Image” module allows for images to be read, processed, and saved. This library has been chosen as I required images to be read for the algorithms implemented to process image data. Once processed by the algorithms, they had to also be saved which this library allows.

2.5.1.6. Opencv – CV2

Open Source Computer Vision (OpenCV) is a library that includes many computer vision algorithms. This library has been chosen as certain aspects of image manipulation is not possible with PIL and extra image processing libraries are needed to achieve the desired results.

3. Algorithm designs

In this section, the algorithms implemented will be described to give a clearer picture of how each of the chosen half-toning and embedding algorithms process images. Each of the halftoning and embedding algorithms implemented have been split into corresponding sections.

3.1. Halftoning

3.1.1. Error diffusion dithering

Error diffusion algorithms as described in section 2.1.2.1 use a matrix that contains information on which neighbouring pixels relative to the current pixel will have the error distributed and what proportion of the error will be distributed. Three different algorithms that all fall under the category of error diffusion have been chosen. These being: Floyd-Steinberg, Jarvis, and Sierra dithering. The basic error diffusion algorithm as proposed by Knuth works with all three of my chosen algorithms. This is because they all follow the same diffusion technique and only require a variation made to the matrix during the processing of each pixel.

3.1.1.1. Floyd-Steinberg

The Floyd-Steinberg algorithm uses a weighting matrix as seen in Fig. 3. This algorithm has some disadvantages such as: apparent direction due to the scanline order of the image, and artifacts close to edges (Nielson 2005). These are due to the small matrix size where the error is not propagated as much as other algorithms.

3.1.1.2. Jarvis

The Jarvis algorithm uses a weighting matrix as seen in Fig. 8. This algorithm is more powerful, and complex compared to Floyd-Steinberg's algorithm. This is due to the error being distributed to three times the number of neighbouring pixels than that in Floyd-Steinberg's and thus creating a smoother appearance to the image. The addition neighbouring pixels results in a higher execution time.

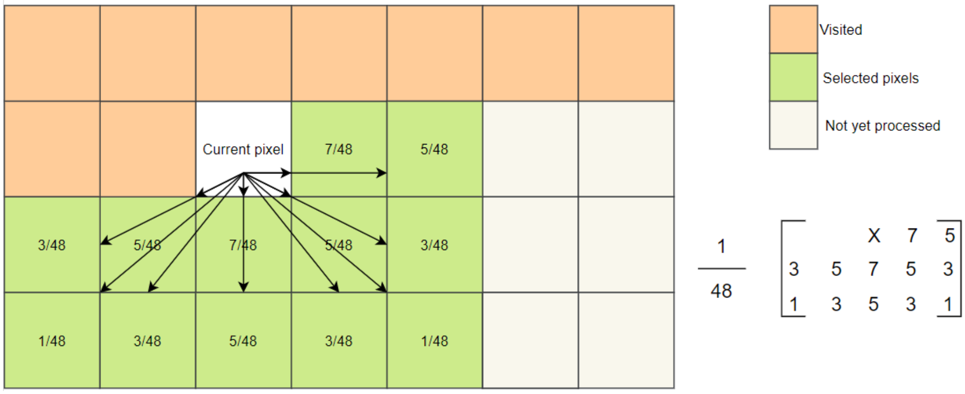
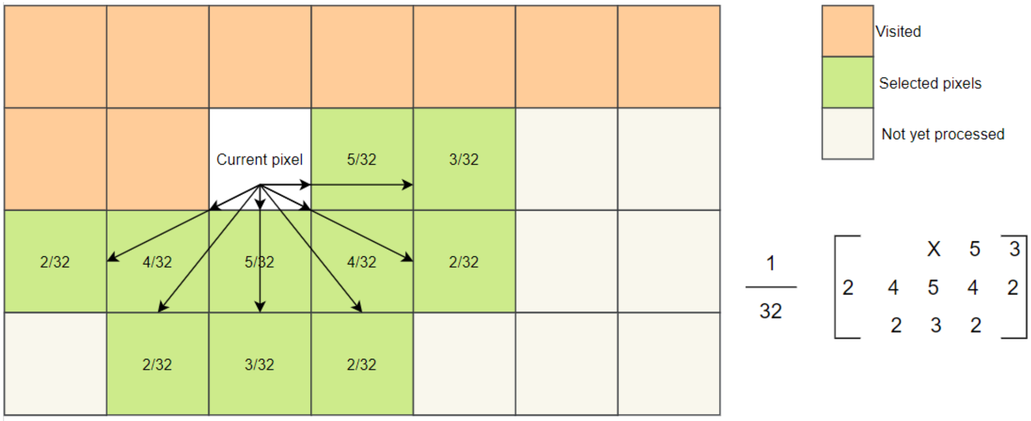


Figure 8 – A diagram demonstrating the proportion of the quantised error distributed to neighbouring pixels. Jarvis algorithm.

3.1.1.3. *Sierra*

The Sierra algorithm uses a weighting matrix as seen in Fig. 9. This algorithm is like Jarvis's algorithm but is less computationally intensive to run and in turn gives similar results with a lower execution time.



$$\frac{1}{32}\begin{bmatrix} & & X & 5 & 3 \\ 2 & 4 & 5 & 4 & 2 \\ & 2 & 3 & 2 & \end{bmatrix}$$

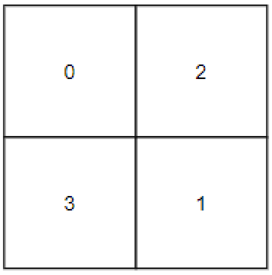
Figure 9 – A diagram demonstrating the proportion of the quantised error distributed to neighbouring pixels. Sierra algorithm.

3.1.2. Ordered dithering

Ordered dithering algorithms as described in section 2.1.2.2 also use a matrix but instead of defining neighbouring pixels to distribute errors to, it defines what threshold each pixel in an image is being tested against. Different sized matrices can be implemented which affect the quality of processed images. Again, I have decided to use three different matrix sizes that all can be used to create an ordered dither image. These are: 2x2, 4x4, and 8x8. The threshold values of the matrix are calculated by dividing each entry of the matrix by the total number of entries. For example, a matrix of size 4 (from a 2x2 matrix) would have all values within the matrix divided by 4 during the halftoning process.

3.1.2.1. *2x2 ordered dither*

The 2x2 ordered dither uses a 2x2 matrix with as seen in Fig 10. This matrix will require a 'wrapping' total of 256x256 when used with 512x512 images.



$$\frac{1}{4}\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

Figure 10 – A diagram demonstrating the 2x2 ordered matrix.

3.1.2.2. 4x4 ordered dither

The 4x4 ordered dither uses a 4x4 matrix with as seen in Fig 11. This matrix will require a 'wrapping' total of 128x128 when used with 512x512 images.

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

$$\frac{1}{16}\begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

Figure 11 – A diagram demonstrating the 4x4 ordered matrix.

3.1.2.3. 8x8 ordered dither

The 8x8 ordered dither uses a 8x8 matrix with as seen in Fig 12. This matrix will require a 'wrapping' total of 64x64 when used with 512x512 images.

0	48	12	60	3	51	15	63
32	16	44	28	35	19	47	31
8	56	4	52	11	59	7	55
40	24	36	20	43	27	39	23
2	50	14	62	1	49	13	61
34	18	46	30	33	17	45	29
10	58	6	54	9	57	5	53
42	26	38	22	41	25	37	21

$$\frac{1}{64}\begin{bmatrix} 0 & 48 & 12 & 60 & 3 & 51 & 15 & 63 \\ 32 & 16 & 44 & 28 & 35 & 19 & 47 & 31 \\ 8 & 56 & 4 & 52 & 11 & 59 & 7 & 55 \\ 40 & 24 & 36 & 20 & 43 & 27 & 39 & 23 \\ 2 & 50 & 14 & 62 & 1 & 49 & 13 & 61 \\ 34 & 18 & 46 & 30 & 33 & 17 & 45 & 29 \\ 10 & 58 & 6 & 54 & 9 & 57 & 5 & 53 \\ 42 & 26 & 38 & 22 & 41 & 25 & 37 & 21 \end{bmatrix}$$

Figure 12 – A diagram demonstrating the 8x8 ordered matrix.

3.1.3. Stippling

I have implemented an alternative to the stippling method as proposed by my supervisor Paul than to the one described in section 2.1.3. This is due to the complexity of implementing the described method in section 2.1.3 being too high and would result in a large amount of time dedicated to this one algorithm in this relatively short timeframe given. The method implemented splits a greyscale image into different groups depending on the value of the pixel. Every defined group has a separate reference image that represents a constant intensity which will be referenced as a gradient. Variations on the number of gradients used to create stippled images has been implemented, this is to see how the image quality is affected by reducing or increasing the number of groups an image is split into. The number of groups chosen are: 2, 4, 6, 8. The pixel locations in greyscale image will be replaced with the same pixel location in the gradient image (see Fig. 13).

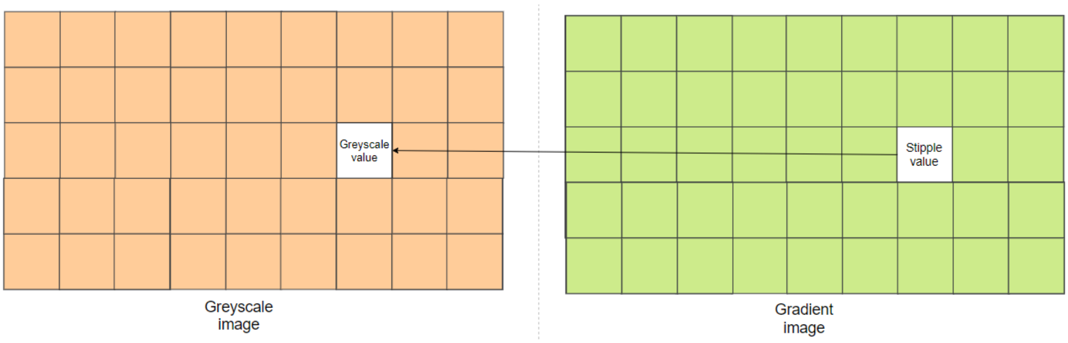


Figure 13 – How stipples are transferred to create a stippled image

3.2. Embedding

There are two main types of steganography for halftoned images. One type is where information is embedded during the halftone process and the other is where information is embedded in halftoned images (Pei and Guo, 2006), (Guo and Liu, 2011), and (Lien and Pei, 2009). Both types have been implemented for embedding text and image information.

As embedding processes occur on binary images, information is required to be in a binary format. This is because embedding processes need to keep images in a halftone format. When embedding text in an image, the familiar ASCII format will need to be processed into a binary format where 8 bits representing an ASCII character. When embedding image information, the image to embed will also need to be converted to a binary format. This is simply done by converting an image to black and white where 0’s represents black values and 1 (or 255) represents white values.

During the process of extracting information from an embedded image, the information will be extracted in its binary format as it is embedded in this format.

When extracting hidden text information from an image, the extracted binary information will need to be processed back into ASCII format for readability. An error in the extraction process could result in an incorrect character or part of an image being returned.

Each embedding algorithm has a threshold value which controls the strength at which information is embedded within an image. The strength is determined by how noticeable the embedded information will be after the embedding process.

3.2.1. Basic pixel embedding (text information)

This algorithm embeds text information within an image. It compares the message in a binary format against the halftoned image pixels in a raster scanline pattern. For every pixel, a comparison between the image pixel (black or white) and the message bit (0 or 1) is made. If the message bit to be embedded differs to the image pixel value, the image pixel is changed to match that of the message pixels value. The scanline pattern is determined by a step variable, this is a variable that determines the space between each processed pixel. Embedding continues until all message bits are embedded.

The algorithm extracts information in reverse to the embedding process. It uses the step variable to access pixels used to embed information to obtain the message in its binary format. After the image is processed, the algorithm processes the extracted binary information, 8 bits at a time, and converts it back to the original ASCII format.

The step variable in the algorithm will be used to determine the strength at which information is embedded in the image. A low step would result in a small distanced spaced between pixels and a high step would result in a larger distanced space between pixels.

3.2.2. Pixel density transition (text information)

This algorithm proposed by Lu et al. in theory improves upon the previous embedding algorithm. This improvement is due to selecting areas of the halftoned image which satisfy a specific criterion rather than embedding in any area like the previous method. It adopts the block-based method which processes images in blocks defined by a block size in a raster scanline pattern (see Fig.14).

Each block, size specified by block size, contains varying numbers of black and white pixels. These numbers can be used to create a density value for each block. The density value is calculated by the number of black pixels in the block. These densities can be changed with the addition or subtraction of a black pixel.

The algorithm starts by splitting the image up into blocks of a fixed size. This size is again, determined by the block size value. During this process, densities of the processed blocks is saved in the order of processing.

The algorithm selects the density which is close to the medium of all unique densities in the image. Blocks with this density are ideal to embed bits of information as they contain close to equal numbers of black and white pixels and are known as 'complex' blocks. Choosing a low or high density would mean choosing a block that is mainly black or mainly white and would affect the embedded images quality as changes to the block's values would be more perceptible to the human eye. Once the medium density is found, another will be chosen. This density is chosen between two possibilities, a density one higher than the medium or a density one lower than the medium. The lower the difference between the medium density and the possibilities will result in the lowest difference in being the second chosen. Once the second density has been chosen, there are now two densities that will be used to embed information into the image. The lower density of the two will be used to embed 0's and the higher will be used to embed 1's.

The algorithm then iterates through the image's blocks, size determined by block size, in a raster scanline pattern. If a block matches one of the two chosen densities it is then compared against the message bit to be embedded. If the message bit value corresponds with the density value, nothing is changed, and the scanline pattern process continues with the next message bit to embed being selected. For example, if the message bit to be embedded is a 0 and the density of the current block is the lower of the two densities (used to embed 0's) no change is needed. If it differs, then the block will have either a black pixel added to or subtracted from it. For example, if the message bit to be embedded is a 0 and the density of the current block is the higher of the two densities (used to embed 1's) then one of the black pixels in this block will be changed to white so that the density value of the current block is now the lower of the two densities (used to embed 0's). Embedding continues until all message bits are embedded.

The extraction process is very similar to the embedding process. The image is split into blocks, determined by the block size used to embed, and the same density selection process is used. The algorithm then iterates through the embedded image's blocks in a scanline pattern. If a block matches one of the chosen densities, it is then checked to see which density value it has. If the density is the lower one, then a 0 will be extracted. Otherwise, a 1 will be extracted as it is the higher one. This continues for the entire image until all blocks have been processed. After this, the algorithm processes the extracted information, 8 bits at a time, and converts the information into ASCII text. This will then reveal the original message.

The block size will be used to determine the strength at which information is embedded in the image. Changing the block size will result in a higher number of possible unique densities and a better choice in blocks that are very complex. This will result in the height of each unique density value being slightly lower and could eventually lead to an insufficient number of blocks used to embed the message in. An insufficient number of blocks will result in a lower portion of the message being embedded in an image.

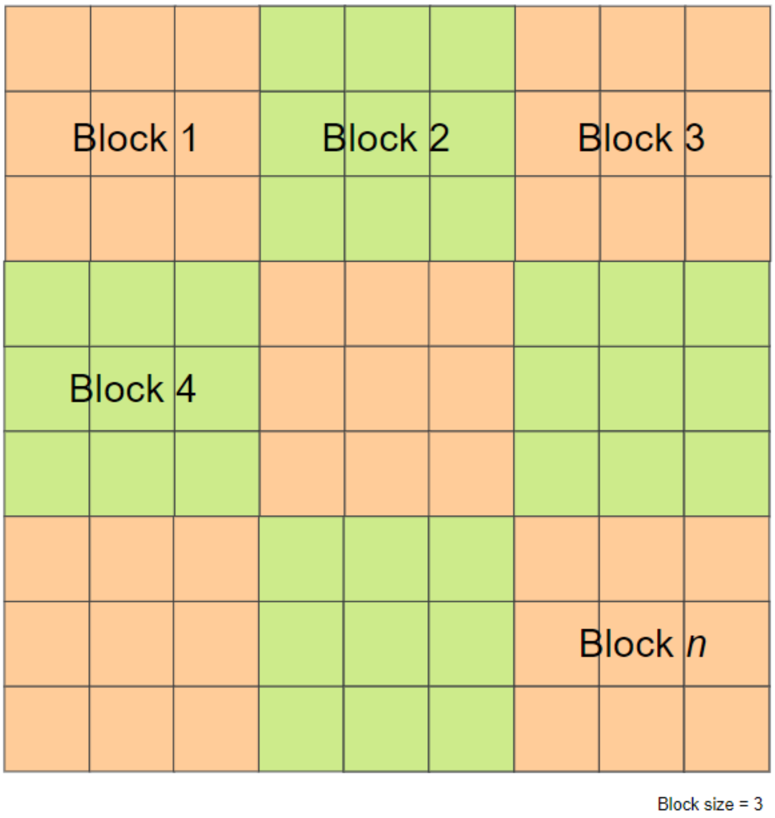


Figure 14 – Demonstration of how an image is split into blocks determined by the block size.

3.2.3. Greyscale pixel alteration (text information)

The algorithm as proposed by Mahdavi and Samavi uses Floyd Steinberg’s halftoning algorithm but can also use other error diffusion algorithms. This only satisfies one of the three halftoning methods implemented for this project. This algorithm has been modified in certain areas so that all three of the halftoning methods can be used with the proposed algorithm.

This algorithm is different to the other two algorithms above as it embeds information during the halftoning process. It does this by embedding data into the greyscale image and then halftoning the embedded greyscale image. Embedding data into the grayscale image is done so in groups (see Fig. 15), defined by a length.

These groups are processed in a raster scanline pattern by the algorithm. Each group is processed to return the number of white pixels in the group after the halftoning process is made. This number of white pixels is multiplied by two and taken away from the length. A group is classified as complex if the number of white pixels is within a threshold range. The threshold is defined by the length value multiplied by two and then divided by five. The range is created by adding and subtracting the threshold from zero. For example, a threshold of five would create a range of minus five to positive five with zero in the centre of the range. By applying the above operations to the group's white pixel count and threshold value, it is guaranteed that embedding does not occur in mainly black or white groups (Mahdavi and Samavi, 2015).

Similarly, as to the previous algorithm, embedding should only occur in groups that satisfy a complexity criterion. Embedding within a group that is mainly black or white would affect the embedded images quality as changes to pixel values in these groups would be more perceptible to the human eye.

If a group satisfies the complexity criteria, an error will be added to the greyscale values within the group. This is where the greyscale values are altered so that the halftoning process also embeds information and can then be extracted from the embedded halftoned image. The error is calculated by recursively adding or subtracting from the greyscale values so that the number of modulo 2(number of white pixels from the halftoned group) equals the message bit to be embedded. For example, a message bit of 1 would require the number of white pixels in a halftoned group to be an odd number as it returns a remainder of 1.

There is a choice of two possible errors that can be added to the greyscale values: addition or subtraction errors. These errors are compared to one another and the lowest error, one which changes the greyscale values the least, is chosen and added to the current group. If for any reason the error added to the greyscale values results in the group no longer being complex when halftoned (outside of the threshold range), the other error is added to the group.

The extraction process uses similar methods to the embedding process. The algorithm opens the embedded halftone image, processes groups of pixels using the same length as embedding and returns the number of white pixels within the group, same as embedding. If the group is complex, the message bit is calculated by modulo 2(number of white pixels within the group). This calculation will either return a 0 or a 1 depending on the value of white pixels within each group. Once the message bit is extracted and added to a list of extracted values, the next group is processed. This continues until all possible groups are processed. After this, the algorithm processes the extracted information, 8 bits at a time, and converts the information into ASCII text. This will then reveal the original message.

The group size will be used to determine the strength at which information is embedded in the image. Changes to the group size's length will in turn affect which groups are classified as complex as the length is used when operations are applied in the embedding process. A larger group size will result in a lower total number of groups message bits can be embedded to. A smaller group size will result in a larger total number of groups message bits can be embedded to.

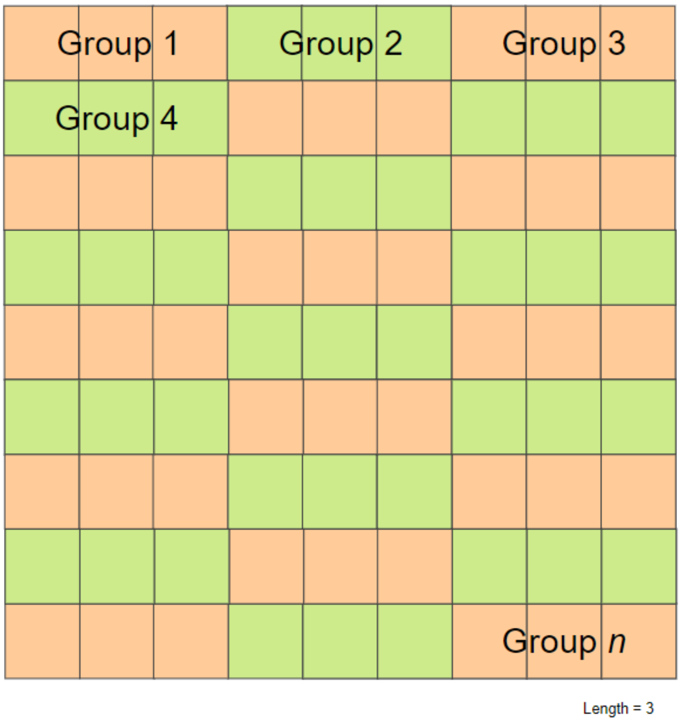


Figure 15 – Demonstration of how an image is split into groups determined by the length size.

3.2.4. Basic pixel embedding (image information)

This algorithm embeds image information within an image. It compares the image to embed pixels against the halftoned image pixels in a raster scanline pattern. For every pixel, a comparison between the image to embed pixel (black or white) and the halftoned image pixel (black or white) is made. If the image to embed pixel differs to the halftoned image pixel value, the halftoned image pixel is changed to match that of the image to embed pixels value. The scanline pattern is determined by a step variable, this is a variable that determines the space between each processed pixel. A low step would result in a small distanced spaced between pixels and a high step would result in a larger distanced space between pixels. Embedding continues until all pixels in both the image to embed and the halftoned image are processed.

There is no extraction process to this algorithm as the image to hide is embedded directly within the halftoned image by altering the halftoned image pixel values.

The step variable in the algorithm will be used to determine the strength at which information is embedded in the halftoned image. A closer step will result in the embedded pixels being close to one another and in theory, have a higher chance of being perceived. A higher step will result in the embedded pixels having a larger space between one another and in theory, have a lower change of being perceived.

3.2.5. Self-conjugate watermarking (image information)

The algorithm as proposed by Mahdavi and Samavi uses Jarvis's halftoning algorithm but can also be used with other error diffusion algorithms. This only satisfies one of the three halftoning methods implemented for this project. This algorithm has been modified in certain areas so that all three of the halftoning methods can be used with the proposed algorithm.

This algorithm is used to embed a hidden image in the bottom half of the test images. The hidden image is only made visible when the bottom half of the embedded halftone image is rotated 180 degrees and overlays the top half of the image. This satisfies the two properties described in section 2.3 as the embedded image is hidden during normal use and only made visible when a special viewing process is made. Once the image halves are overlaid, the hidden image is made visible by applying NXOR (Not Exclusive OR) operations to the top half pixel and the corresponding embedded bottom half pixel. NXOR operations return a black pixel if the two corresponding pixels are different and white if the two corresponding pixels are the same. The image to hide has the same dimensions as the bottom half. If for some reason they differ, the image to hide is resized to the bottom halves' dimensions. This is necessary due to the comparisons made during the embedding and extraction process.

There are two parts to the embedding process. The first being a normal, unaltered, halftoning process for the top half of the image. The second halftones the bottom half of the image but also considers the image to hide.

When halftoning the bottom half of the image, the raster scanline pattern is also taking place on the image to hide. With every pixel in the normal image, a temporary halftone value is created dependant on the greyscale value. In error diffusion algorithms, halftone values depend on whether a greyscale value is above or below a threshold. If a pixel with a value of 0 (black) is met on the image to hide, a comparison is made with the corresponding temporary halftone value and the top half of the image. In the other implemented halftone algorithms, the halftone value also depends on the greyscale value but may have a different threshold value. This comparison must consider that the extraction process involves the rotation of the embedded bottom half against the top half of 180 degrees (see Fig. 16). The comparison is to see if the bottom half halftone value is opposite to that of the top half halftone pixel value. It must be opposite as the NXOR operation during the extraction process returns a black pixel if two values do not match and white if they do. If this comparison is successful, and these two halftone values are different, the embedding operation continues. If unsuccessful, the bottom half pixel value requires an alteration. This alteration is made on the greyscale value so that the temporary halftone value is also changed. For example, a greyscale value of 130 (halftoned would be white) would need an alteration of -4 so that the new greyscale value would be 126 and become black when halftoned. This alteration is limited by a threshold value. No greyscale value may be altered more than this threshold as it would cause more noise than intended being added to the bottom half of the image. If the alteration value is below the threshold value, the alteration value is added to the greyscale value and when now halftoned, will embed the correct pixel colour so that the extraction process results in a black pixel being created from NXOR operations. This continues until the bottom half of the image is processed completely.

Once the bottom half of the image is halftoned, there will be a hidden image embedded within but not immediately perceivable. Again, the hidden image is extracted by rotating the bottom half of the image 180 degrees to overlay the top half of the image. The overlaid pixels have the XNOR operation applied and the resulting values from this process will reveal the hidden image.

The alteration threshold will be used to determine the strength at which information is embedded within the image. A lower threshold will result in a lower number of pixels being changed and the NXOR operations will extract a lower percentage of the original image to hide. A higher threshold will result in a higher number of pixels being changed and the NXOR operations will extract a higher percentage of the original image to hide.

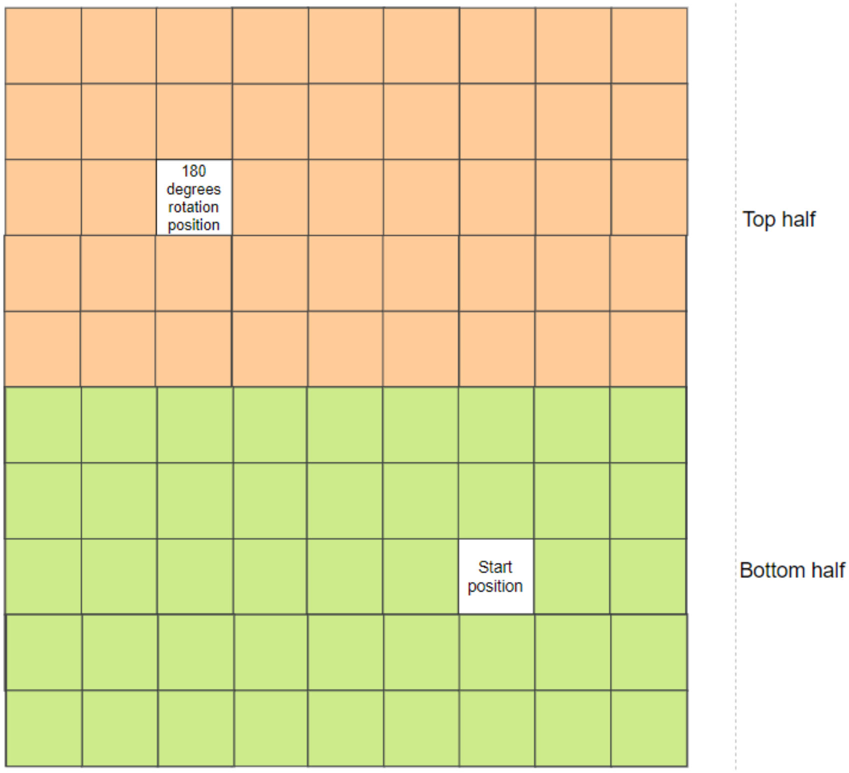


Figure 16 – Demonstration of 180 degrees comparison position.

3.3. Static architecture

The structure of this projects code has been partitioned into modules. As seen in the folder structure diagram (Appendix A), each section to the project has been split into 4 main folders. The folder "Images" contains the original, original to halftone, and embedded images. The folder "Half-toning" contains the halftoning programs under the categories in which they halftone an image. The folder "Embedding" contains the embedding programs under their corresponding names. The folder "Analysis" contains programs which are used to analyse any processed image. This structure has been used so that all elements to the project are easily found and all images that have been processed are in one section but divided into corresponding sub-sections.

3.4. How data flows through the project

As the structure of the projects code has been partitioned into modules, all data created is sent and stored within "Images". Within the folder there are three categories in which the images can be put under - these being "Original", "Basic Halftone", and "Embedded". All original images that are to be halftoned and/or embedded with information are stored within "Original". No data is sent to this folder during any execution of an algorithm implemented. The results of halftoning and/or embedding are either sent to "Basic Halftone" or "Embedded". During the execution of the halftone algorithms, images are saved within "Basic Halftone" within folders for the corresponding halftoning algorithm executed. During the execution of the embedding algorithms, images are saved within “Embedded” within folders for the corresponding embedding algorithm executed.

When each algorithm executed, data is sent to an Excel spreadsheet within the “Data” folder. This data includes: PSNR and SSIM values, execution timings (embed and extraction), and the success of information retrieval. Each program has been made so that this is an automatic process and no manual copy and pasting of data is required.

To calculate PSNR and SSIM values, images are sent directly to a program named “PSNRSSIM.py” within the “Analysis” folder. This program only returns the values of PSNR and SSIM to the program currently being executed and no images are saved during the execution of the “PSNRSSIM.py” program.

3.5. How algorithms are executed

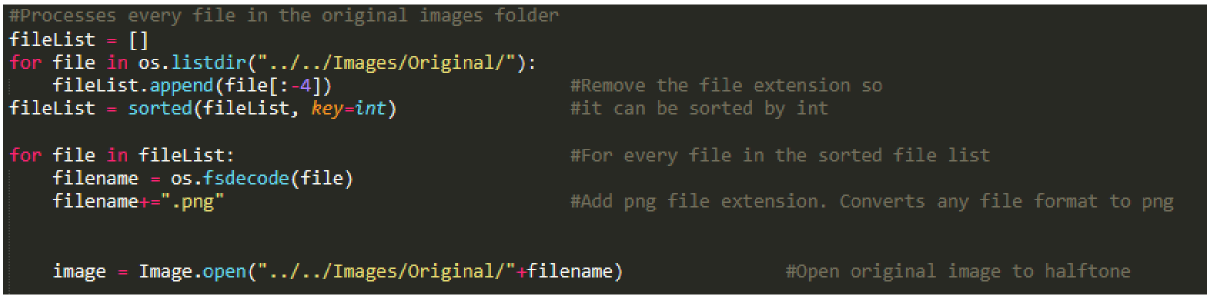
All algorithms, halftoning and embedding, are executed through the command prompt. As the project has been partitioned into modules, each algorithm is saved under corresponding folders. All halftoning and embedding algorithms are within their own folder and subdivided into subfolders. Each program can be executed by simply opening the file within the command prompt and all images that are within the corresponding images folder(s) needed are processed automatically. As the project is algorithmically based to process images, I did not feel a GUI was required to complete this task.

4. Implementation

In this section, algorithm implementations will be explained down to the code level. Any issues encountered during the implementations of algorithms will be written about and steps taken to mitigate the issue are explained.

4.1. Reading images into algorithms

All algorithms implemented have an input image from within the “Images” folder. This folder contains all original greyscale images to be halftoned, halftoned versions of the original images, and halftoned images embedded with information. Images, determined by the type of algorithm executed, are automatically read in one after the other after processing has taken place on the previous image. This process is done through a loop that iterates through a specified folder, in this case the original images folder (see Fig 17). Originally, the process read images in an undesirable order. It processed images in an order of 0,1,11,12,13,14,15,16,17,18,19,2,20, etc. This was acceptable until data was sent to an Excel sheet to be analysed. This order was not desirable in Excel as the first half of images were labelled as ‘complex’ and the second half were labelled as ‘not complex’. This hindered the viewing of results in graphs and other means of presenting data. To process images in the desired order, a list of all file names within the folder are stored and then sorted. This sorted list is then used to read images in the desired order.

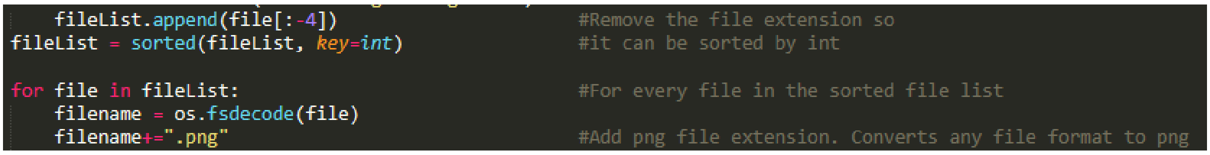


```
#Processes every file in the original images folder
fileList = []
for file in os.listdir("../../Images/Original/"):
    fileList.append(file[:-4])                      #Remove the file extension so
fileList = sorted(fileList, key=int)                #it can be sorted by int

for file in fileList:                               #For every file in the sorted file list
    filename = os.fsdecode(file)
    filename+=".png"                                #Add png file extension. Converts any file format to png


    image = Image.open("../../Images/Original/"+filename)                   #Open original image to halftone
```

Figure 17 – How each algorithm automatically processes every image in a specified folder.

4.2. Conversion of image types

Images for this project have been converted to or are .PNG filetype. This filetype is lossless which means it is not compressed and preserves all details of an image. An issue encountered during the project is with the use of JPEG filetypes. These types of images use a lossy compression which compress the image file when saved and causes some details of an image to be lost. Pillow (PIL) the image library used to read and save images in all implemented algorithms allows the conversion of JPEG to PNG. By replacing the filename’s extension to .PNG, the library automatically saves the image under the .PNG filetype (See Fig 18).

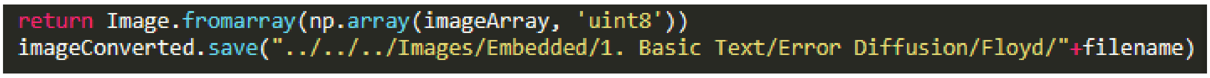


```
    fileList.append(file[:-4])                            #Remove the file extension so
fileList = sorted(fileList, key=int)                      #it can be sorted by int

for file in fileList:                                     #For every file in the sorted file list
    filename = os.fsdecode(file)
    filename+=".png"                                      #Add png file extension. Converts any file format to png
```

Figure 18 – How images are converted to the .PNG filetype format.

4.3. Saving of images

All images in the project are processed in numpy array form. These arrays cannot be directly saved as an image and require a conversion back to an image format. This is done with the PIL library as it supports conversion of numpy arrays to an image that can be saved. Once this process has finished, the PIL library is used once again to save the processed image. This is done by the '.save' function and a path to the desired folder can be specified. Figure 19 demonstrates both operations.



```
return Image.fromarray(np.array(imageArray, 'uint8'))
imageConverted.save("../../../Images/Embedded/1. Basic Text/Error Diffusion/Floyd/"+filename)
```

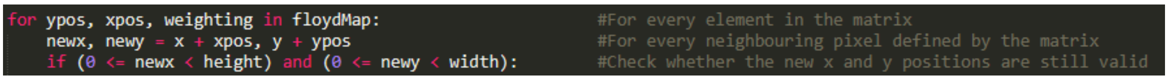
Figure 19 – How images are converted from numpy arrays and saved to a specific location

4.4. Halftoning

The halftoning algorithms structures have all been designed so that all methods under a type of halftoning process can be implemented by 'slotting' in the different methods values to the type of halftoning process.

4.4.1. Error diffusion dithering

What changes between the error diffusion algorithms is the map that includes information on the neighbouring pixels and the error weighting values relative to the current pixel being processed. Maps are sized differently with each error diffusion algorithm and neighbouring pixels relative to the current pixel being processed will still need to be within the image's dimensions. The method as proposed by Blidh, H solves this issue as the map's contents are iteratively added to the current coordinates (x and y positions) with the values that define neighbouring pixels (see Fig 20). Parts of the method have been altered so that the method can be used in line with the automated system implemented.

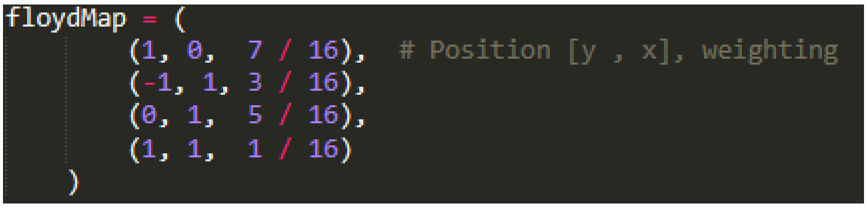


```
for ypos, xpos, weighting in floydMap:                  #For every element in the matrix
    newx, newy = x + xpos, y + ypos                     #For every neighbouring pixel defined by the matrix
    if (0 <= newx < height) and (0 <= newy < width):    #Check whether the new x and y positions are still valid
```

Figure 20 – Checking neighbouring pixel coordinates are still within image dimensions.

4.4.1.1. Error diffusion

As shown in Fig 3, the Floyd Steinberg error diffusion algorithm has four neighbouring pixels an error is distributed to. In Fig 21, this is implemented with a map size of four.

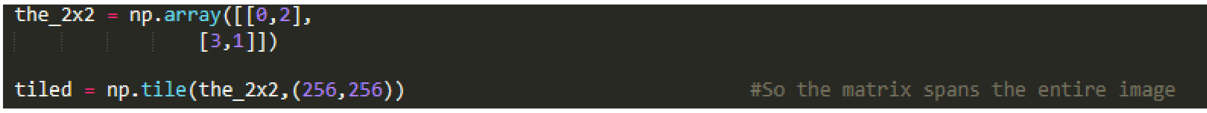


```
floydMap = (
        (1, 0,  7 / 16),  # Position [y , x], weighting
        (-1, 1, 3 / 16),
        (0, 1,  5 / 16),
        (1, 1,  1 / 16)
    )
```

Figure 21 – Map of Floyd Steinberg's neighbouring pixels and corresponding error distribution weightings

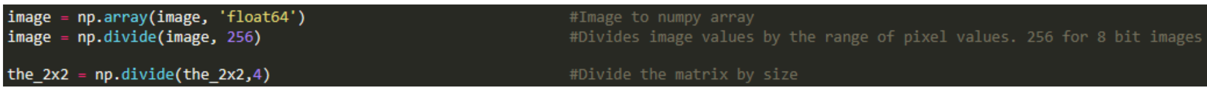
4.4.2. Ordered dithering

What changes between the ordered dithering algorithms are the sizes of the matrix used. As the size of the matrix varies, a different number of wraps are required so that it spans the entire image. Wrapping is achieved by using the Numpy library which includes a "tile" function. The use of this function was proposed by Sankarasrinivasan, S and has been modified so that it fits in the algorithm. The tile function basically wraps a matrix by a multiplier (see Fig 22). For example, a matrix with a size of 2x2 and an image with dimensions 100x100 would require the matrix being tiled 50 times its current size to span the 100x100 image. Numpy matrices can be modified in its entirety for example, multiplication and division can be applied to the entire matrix at once rather than accessing individual elements. This was used when dividing the image matrix by 256 and the ordered matrix by the number of elements within (see Fig 23). Similarly, the entire matrix can have comparison statements against it meaning that the all elements of the matrix can be compared against a value. This was useful in the creation of the halftone image as each element of the image's matrix could be compared against the wrapped ordered matrix (see Fig 24).



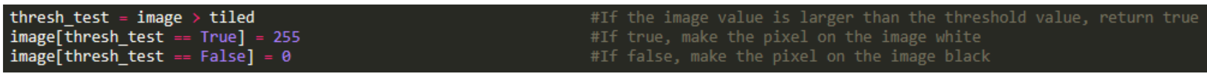
```
the_2x2 = np.array([[0,2],
                    [3,1]])

tiled = np.tile(the_2x2,(256,256))                    #So the matrix spans the entire image
```

Figure 22 – How wrapping is achieved in the ordered dithering algorithms.



```
image = np.array(image, 'float64')                    #Image to numpy array
image = np.divide(image, 256)                         #Divides image values by the range of pixel values. 256 for 8 bit images

the_2x2 = np.divide(the_2x2,4)                        #Divide the matrix by size
```

Figure 23 – Demonstration of how division can be applied to an entire Numpy matrix.

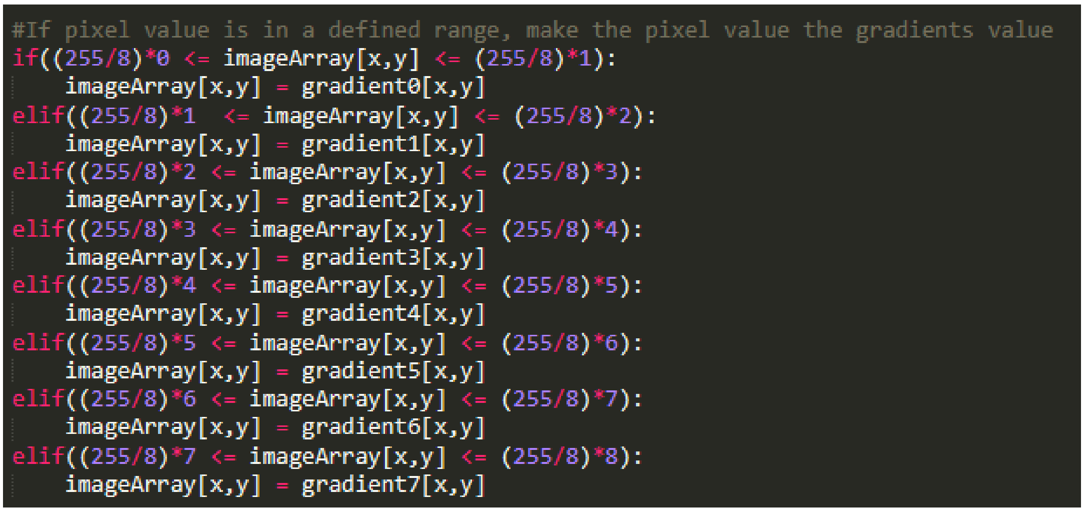


```
thresh_test = image > tiled                                     #If the image value is larger than the threshold value, return true
image[thresh_test == True] = 255                                #If true, make the pixel on the image white
image[thresh_test == False] = 0                                 #If false, make the pixel on the image black
```

Figure 24 – Demonstration of how comparison statements can be applied to two Numpy matrices.

4.4.3. Stippling

What changes between the stippling algorithms are the gradients which are used to create the stippled halftoned images. These range from 2 gradients to 8 gradients. Ranges are calculated by splitting the maximum possible greyscale pixel value (255) into equal ranges. If the greyscale value is within one of these ranges, the corresponding gradient value replaces the greyscale value. This is all shown in Fig 25.

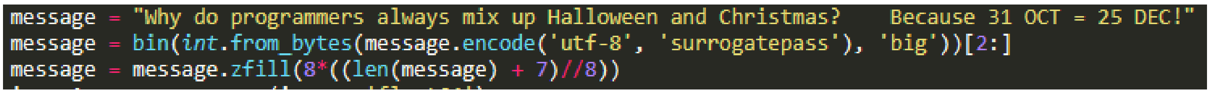


```
#If pixel value is in a defined range, make the pixel value the gradients value
if((255/8)*0 <= imageArray[x,y] <= (255/8)*1):
    imageArray[x,y] = gradient0[x,y]
elif((255/8)*1  <= imageArray[x,y] <= (255/8)*2):
    imageArray[x,y] = gradient1[x,y]
elif((255/8)*2 <= imageArray[x,y] <= (255/8)*3):
    imageArray[x,y] = gradient2[x,y]
elif((255/8)*3 <= imageArray[x,y] <= (255/8)*4):
    imageArray[x,y] = gradient3[x,y]
elif((255/8)*4 <= imageArray[x,y] <= (255/8)*5):
    imageArray[x,y] = gradient4[x,y]
elif((255/8)*5 <= imageArray[x,y] <= (255/8)*6):
    imageArray[x,y] = gradient5[x,y]
elif((255/8)*6 <= imageArray[x,y] <= (255/8)*7):
    imageArray[x,y] = gradient6[x,y]
elif((255/8)*7 <= imageArray[x,y] <= (255/8)*8):
    imageArray[x,y] = gradient7[x,y]
```

Figure 25 – How ranges are calculated for the creation of stippled images

4.5. Embedding

When embedding text information, this needs to be converted into binary form so that it may be embedded into binarised images. In Fig 26, this process is shown by taking the message string and converting each character into 8 bits. Character conversions sometimes generate a lower number of binary bits. This difference will result in the conversion from binary to ASCII being very difficult as the number of bits required to make each character can vary. This was solved by making the number of bits per character 8 bits long. Having these bits all the same length to create a character makes the decoding process much easier.



```
message = "Why do programmers always mix up Halloween and Christmas?    Because 31 OCT = 25 DEC!"
message = bin(int.from_bytes(message.encode('utf-8', 'surrogatepass'), 'big'))[2:]
message = message.zfill(8*((len(message) + 7)//8))
```

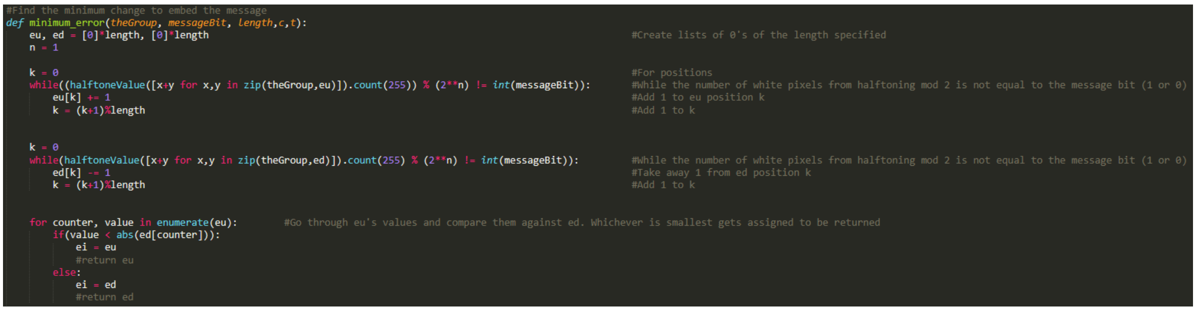
Figure 26 – How a message is converted to binary form

4.5.1. Pixel density transition (text information)

Part of the embedding procedure for this embedding algorithm was to include a feature called the “Pixel Mesh Markov Transition Matrix” which calculates the best possible pixel to change so that the density changes from one to another. This feature was not implemented due to the complexity of it. Instead, the first pixel that satisfies the density transition in each block is used to embed information. It still achieves the same result of density transition but not the best possible pixel to change to do this.

4.5.2. Greyscale pixel alteration (text information)

When calculating the minimum error to add to the selected group, two values are calculated to achieve the required result. This process is shown in Fig 27.



```
#Find the minimum change to embed the message
def minimum_error(theGroup, messageBit, length,c,t):
    eu, ed = [0]*length, [0]*length                                                                           #Create lists of 0's of the length specified
    n = 1

    k = 0                                                                                                     #For positions
    while((halftoneValue([x+y for x,y in zip(theGroup,eu)]).count(255)) % (2**n) != int(messageBit)):       #While the number of white pixels from halftoning mod 2 is not equal to the message bit (1 or 0)
        eu[k] += 1                                                                                            #Add 1 to eu position k
        k = (k+1)%length                                                                                      #Add 1 to k


    k = 0
    while(halftoneValue([x+y for x,y in zip(theGroup,ed)]).count(255) % (2**n) != int(messageBit)):         #While the number of white pixels from halftoning mod 2 is not equal to the message bit (1 or 0)
        ed[k] -= 1                                                                                            #Take away 1 from ed position k
        k = (k+1)%length                                                                                      #Add 1 to k


    for counter, value in enumerate(eu):       #Go through eu's values and compare them against ed. Whichever is smallest gets assigned to be returned
        if(value < abs(ed[counter])):
            ei = eu
            #return eu
        else:
            ei = ed
            #return ed
```

Figure 27 – How errors are calculated in selected groups

5. Results and evaluation

In this section, comparisons and analysis of results from the halftoning methods will be made. Comparisons and analysis of results from the embedding methods with each of the halftoning methods will be made.

5.1. Halftoning

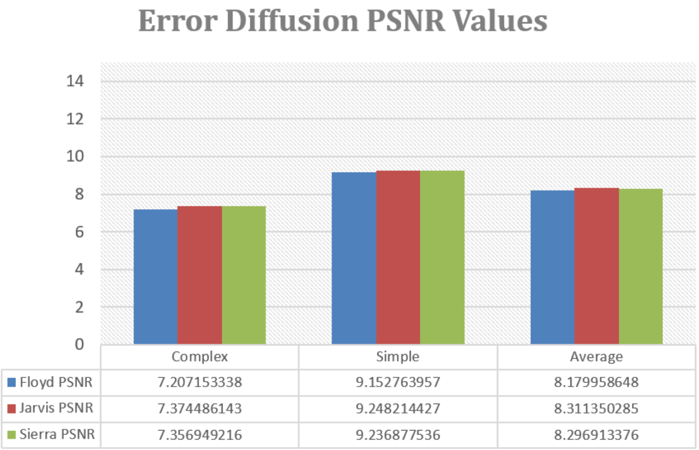
In this section, comparisons and analysis will be made to see if a simple or complex image determines the quality in different halftoning methods. The two-image metrics as described in section 2.2 will be used to determine these results against the original greyscale image. Each halftoning method has separate PSNR and SSIM graphs that show how simple and complex images determine the quality of halftoned images.

5.1.1. Error Diffusion

Simple images perform better in the PSNR measure (see Fig 28) due to less noise being added. This is because simple original images require less densely populated areas of black and white pixels to recreate the original image in halftoned form.

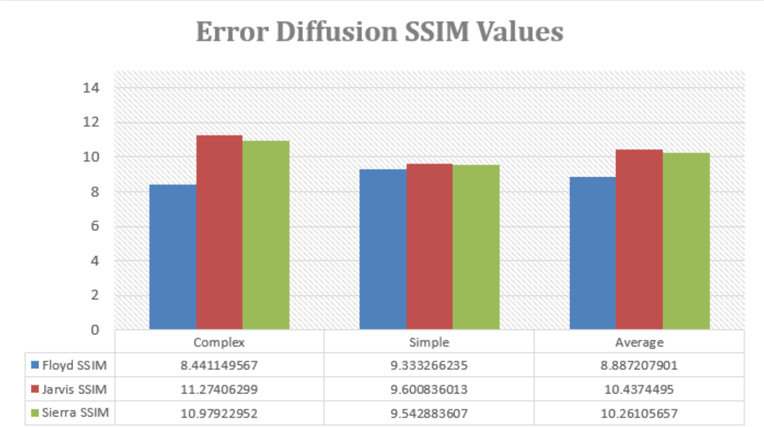
SSIM values appear to be determined by the combination of image complexity and error diffusion algorithm complexity. A complex image that is processed with a complex error diffusion algorithm result in a higher SSIM value than a complex image that is processed with a simple error diffusion algorithm. This can be seen in Fig 29 as the complex image processed with Floyds algorithm, a simpler error diffusion algorithm, performs worse than the simple image processed with Floyds algorithm.

On average, the results show that simple or complex images halftoned with a more complex error diffusion algorithm perform better than a simpler error diffusion algorithm. This is because there are more neighbouring pixels defined and in turn results in a higher number of diffused errors being added to each individual pixel thus creating a more accurate halftone image.



	Complex	Simple	Average
🟦 Floyd PSNR	7.207153338	9.152763957	8.179958648
🟥 Jarvis PSNR	7.374486143	9.248214427	8.311350285
🟩 Sierra PSNR	7.356949216	9.236877536	8.296913376

Figure 28 – Error Diffusion PSNR values in complex and simple image



	Complex	Simple	Average
🟦 Floyd SSIM	8.441149567	9.333266235	8.887207901
🟥 Jarvis SSIM	11.27406299	9.600836013	10.4374495
🟩 Sierra SSIM	10.97922952	9.542883607	10.26105657

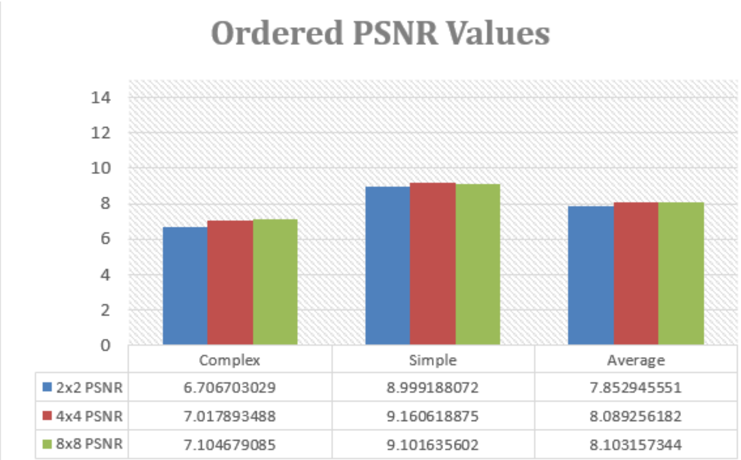
Figure 29 – Error Diffusion SSIM values in complex and simple images

5.1.2. Ordered dither

Again, simple images perform better in the PSNR measure (see Fig 30) due to less noise being added. This is because simple original images require less densely populated areas of black and white pixels to recreate the original image in halftoned form.

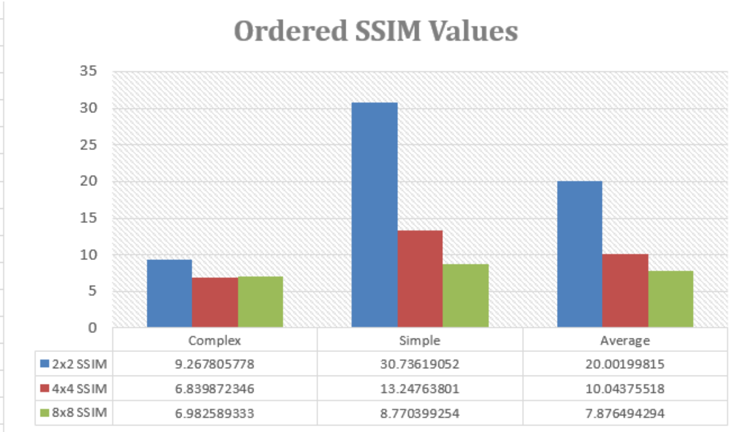
SSIM values appear to be determined by the combination of image complexity and matrix size (see Fig 31). Larger matrices used on both simple and complex images result in worse SSIM values. This can be due to one or more of the three factors used to calculate SSIM values. Larger matrices appear to affect at least one due to the larger number of patterns created in the halftoned image (see Fig. 6). The number of patterns (result of matrix size) used to create halftoned images could be a reason for the unexpected results.

On average, the results show that simple and complex images halftoned with a larger matrix size create better looking images if only PSNR values are only being used. This is due to the larger number of thresholds pixel values are compared against to determine a pixel's halftone value thus creating a more accurate halftone.



	Complex	Simple	Average
🟦 2x2 PSNR	6.706703029	8.999188072	7.852945551
🟥 4x4 PSNR	7.017893488	9.160618875	8.089256182
🟩 8x8 PSNR	7.104679085	9.101635602	8.103157344

Figure 30 – Order dithered PSNR values in complex and simple images



	Complex	Simple	Average
🟦 2x2 SSIM	9.267805778	30.73619052	20.00199815
🟥 4x4 SSIM	6.839872346	13.24763801	10.04375518
🟩 8x8 SSIM	6.982589333	8.770399254	7.876494294

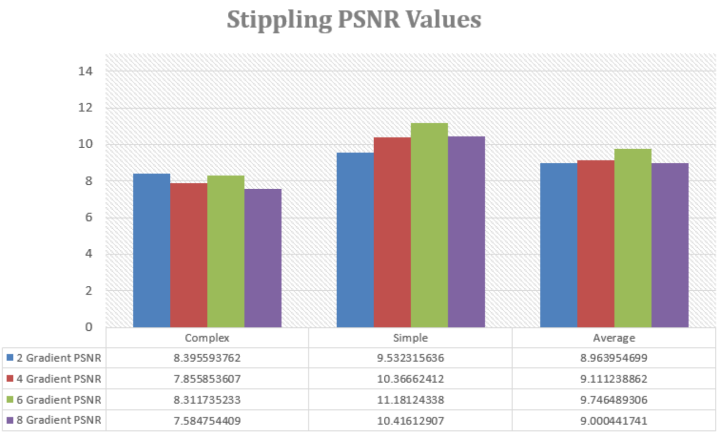
Figure 31 – Order dithered SSIM values in complex and simple images

5.1.3. Stippling

Again, simple images perform better in the PSNR measure (see Fig 32) due to less noise being added. This is because simple original images require less densely populated areas of black and white pixels to recreate the original image in halftoned form.

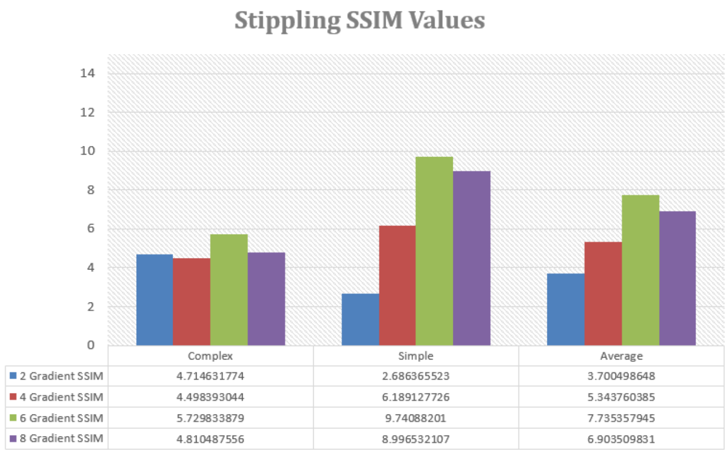
SSIM values appear to be determined by the combination of image complexity (see Fig 33) and the range of gradients used in simple images. In complex images, the quality is not directly affected by the number of gradients used. This is due to the uniformity of simple images. Simple images are uniform in pixel luminance and will result in large areas of the stippled image having large areas of one gradient. Complex images are not uniform in pixel luminance and will result in a smaller amount of large areas having one gradient thus preserving definition from original images.

On average, the results show that simple and complex images stippled with a larger number of gradients perform better than a smaller number of gradients. This is because the larger range in gradients allow a larger range of spaced gaps between stipples. A larger range of possible spacing between stipples results in a larger range of greyscale values perceived by the viewer. A reason for worse quality in eight gradient stipples could be due to the defined ranges being unsuitable for each gradient to replace greyscale values.



	Complex	Simple	Average
🟦 2 Gradient PSNR	8.395593762	9.532315636	8.963954699
🟥 4 Gradient PSNR	7.855853607	10.36662412	9.111238862
🟩 6 Gradient PSNR	8.311735233	11.18124338	9.746489306
🟪 8 Gradient PSNR	7.584754409	10.41612907	9.000441741

Figure 32 – Stippling PSNR values in complex and simple images



	Complex	Simple	Average
🟦 2 Gradient SSIM	4.714631774	2.686365523	3.700498648
🟥 4 Gradient SSIM	4.498393044	6.189127726	5.343760385
🟩 6 Gradient SSIM	5.729833879	9.74088201	7.735357945
🟪 8 Gradient SSIM	4.810487556	8.996532107	6.903509831

Figure 33 – Stippling SSIM values in complex and simple images

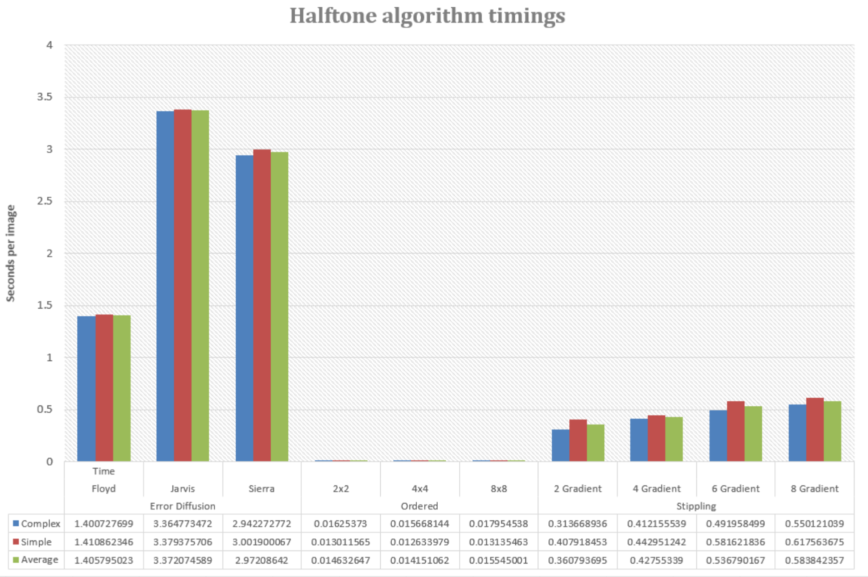
5.1.4. Execution time per image

Fig 34 demonstrates the execution time in seconds per image. It shows that the complexity of both error diffusion and stippling algorithms increase the execution time.

Error diffusion algorithms have the highest execution time due to the raster scanline method of the image and the required propagation of errors to neighbouring pixels. The more neighbouring pixels, the longer the execution time.

Ordered dithering algorithms have the lowest time due to the simplicity of the algorithm. It compares two matrices in one calculation which is far superior than requiring a raster scanline to process pixels individually.

Stippling algorithms have the second highest time due to the raster scanline method of the image. It does not require errors to be propagated to neighbouring pixels, so the execution time is lower than error diffusion algorithms. The more gradients used, the longer the execution time as more comparisons between pixel luminance and gradient threshold ranges will be made.



	Floyd	Jarvis	Sierra	2x2	4x4	8x8	2 Gradient	4 Gradient	6 Gradient	8 Gradient
		Error Diffusion			Ordered			Stippling		
🟦 Complex	1.400727699	3.364773472	2.942272772	0.01625373	0.015668144	0.017954538	0.313668936	0.412155539	0.491958499	0.550121039
🟥 Simple	1.410862346	3.379375706	3.001900067	0.013011565	0.012633979	0.013135463	0.407918453	0.442951242	0.581621836	0.617563675
🟩 Average	1.405795023	3.372074589	2.97208642	0.014632647	0.014151062	0.015545001	0.360793695	0.42755339	0.536790167	0.583842357

Figure 34 – Execution times, in seconds, per image for each halftoning method

5.1.5. Evaluation

Overall, the results show that error diffusion algorithms create the best quality halftoned images. From results, it shows that the more complex the error diffusion algorithm the higher the resulting quality of the halftoned image. The choice of complexity has an impact on the execution time per image. Simple and complex images appear to fare similarly in terms of PSNR and SSIM values when using this halftoning algorithm.

Order dithered algorithms create high quality halftoned images depending on the choice of matrix size. From results, it shows that the 8x8 matrix size give the best quality halftoned images. The choice of matrix size does not have an impact on the execution time. Simple images seem to fare better than complex images in terms of PSNR and SSIM results when using this halftoning algorithm.

Stippling algorithms create good quality halftoned images depending on the range of gradients used. From results, it shows that six gradients give the best quality halftoned images. The choice of gradient range also has an impact on execution time per image. Simple images appear to give better quality than complex ones when a high number of gradients is used.

5.2. Embedding

In this section, comparisons and analysis will be made to see if a simple or complex halftoned image affect embedded image quality. All halftoning methods will be tested against each of the embedding algorithms to see which combination give the best results. The two-image metrics as described in section 2.2 will be used to determine these results against the corresponding halftoned image. Each embedding method has separate PSNR and SSIM graphs that show how simple and complex images determine the quality of embedding methods.

5.2.1. Basic pixel embedding (text information)

As seen in appendices B, D, and F, this algorithm performs well across all embedding strengths in simple images. As the step is increased, the overall image quality is slightly affected. This can be due to the increase in area to embed information. An increased area of altered blocks can result in a higher chance of being perceived.

As seen in appendices C, E, and G, this algorithm performs well across all embedding strengths in complex images. Again, the overall image quality is slightly affected as the step size increases. The quality is not affected as much as seen in appendices B, D and F. This is due to the lack of uniformity in complex images. A lack of uniformity means that changes to pixel values whilst embedding does not affect areas of an image as much as a uniform area would. This is because changes to complex areas are less perceivable than changes made to simple areas. It also means that the likelihood of not requiring a change in pixel value to embedded information is quite high as black and white pixels are more evenly distributed in complex images.

5.2.1.1. Result

This algorithm performs best in complex halftoned images. Results show that error diffused halftone images perform best in terms of PSNR and SSIM values. Order dithered and stippled images both perform similarly as the halftoning process usually creates a more patterned effect in halftone images whereas error diffusion does not. Changes to patterns is generally more perceivable.

In all tests, 100% of embedded information was extracted.

The results show that a step of one yields the best resulting quality after embedding in both simple and complex images.

5.2.2. Pixel density transition (text information)

As seen in appendices H-M, this algorithm performs worse as the block size is increased. This is due to the algorithm requiring blocks to be one of two densities. As the block size increases, the number of these groups having one of these densities diminishes. The PSNR values in each increase of block size shows a reducing number of embedded blocks as no noise from embedding information is being added to the image. A result of the reduction of embedded blocks is a lower percentage of the full extracted message.

Complex images in each halftoning method all achieve better extraction results than simple images with the same halftoning method. This is due to again, uniformity in the halftoned image. Uniform images will likely have uniform blocks of mainly black or white pixels. These blocks will not be chosen as they do not meet the complexity criteria. This results in a lower number of blocks being complex and in turn a lower chance of blocks being one of the two embeddable ones. Non-uniform images will likely have more complex blocks and have a higher chance of being one of the two embeddable blocks.

This algorithm does have one disadvantage. This being with the selection of the two densities when embedding and extracting. When embedding, two densities are chosen and blocks with these densities are used to embed information. Embedding sometimes require these densities to change between one another. This can influence the selection of densities when extracting information. If a density is changed to the medium density too many times, that density total may become lower than the density on the other side of the medium density. This is an issue when extracting information as the other density is chosen due to it now being closer to the medium density. Information is still embedded in the original densities but will not be extracted due to the new selection of densities. This disadvantage causes some results of extraction to be incorrect even though the correct information is embedded in the image.

5.2.2.1. Result

This algorithm performs best in complex halftoned images. Results show that order dithered halftoned complex images give the best extraction results when compared across all halftoning methods.

It also shows that the PSNR and SSIM values are slightly better than error diffusion and stippling whilst maintaining the higher extraction results.

The results show that a block size of four yields the percentage of extracted information. Ordered dither algorithms tend to perform better in terms of both the percentage of extracted information and the quality of halftone image when compared to the original halftone image.

5.2.3. Greyscale pixel alteration (text information)

As seen in appendices N-Q, this algorithm performs best in complex images when combined with error diffusion and ordered dither algorithms as they have 100% extraction success. Simple images when combined with error diffusion and ordered dither algorithms achieve good results but not 100% extraction success. Simple images can have too few groups that satisfy the complexity criteria which can be caused by uniformity in pixel values. Fewer embeddable groups can cause part of a message being left out of the embedding process and irretrievable during the extraction process.

In appendices N and O, the embedding algorithm combined with an error diffusion algorithm yields poor PSNR and SSIM results when compared against the original halftone image. This is caused by the error diffusion algorithm propagating errors to nearby pixels. The embedding algorithm alters greyscale values to achieve the correct number of white pixels in a group when halftoned. These alterations are propagated to the nearby pixels which in turn affect the halftone values of those also. A shorter message to embed would result in higher PSNR and SSIM results due to this process occurring fewer times.

In appendices R and S, the embedding algorithm combined with stippling algorithms perform poorly in extraction percentages compared to the other two. When alteration values are being calculated, halftone values of the alterations may never achieve the required number of white pixels within the group. This is caused by using gradient images that replace greyscale pixel values. The greyscale pixel value may be altered enough so that another gradient is used to halftone the pixel but that new gradients value in the same location may be the same value as the previous gradient. The pixel may never be able to be changed into a white pixel. If this occurs enough times in a group, the group will never be able to embed the message bit.

In appendix S, an interesting result found is that the success rate of two gradient extraction in complex images went up as the group size was increased. This is due to the use of gradients when stippling an image. A low (mainly black) and high (mainly white) gradient is used when creating the two gradient stippled images. As the gradients used are mainly black and mainly white, alterations in the complex group are likely to achieve the required number of white pixels in a group. This result is not replicated in appendix R's 2 gradient extraction success as blocks in simple images are usually uniform (mainly black or white).

5.2.3.1. Result

This algorithm performs best in complex error diffused and order dithered images. Results show that order dithered images yield high PSNR and SSIM results when compared to the regular order dithered halftone image.

Overall image quality is best when the embedding algorithm is used with an error diffusion as alterations are blended with the propagation of the alterations.

5.2.4. Basic pixel embedding (image information)

As seen in appendices T-Y, changes to the step size directly affects how much of the image to embed is embedded within the halftone image. The lower the step, the higher percentage of the image to hide is embedded. The higher the step, the lower the percentage of the image to hide is embedded. There is a direct correlation between the step size and values of PSNR and SSIM. This is due to the added noise resulting from the embedding of the image to hide.

5.2.4.1. Result

This algorithm gives best results in complex error diffused images in terms of PSNR and SSIM values. This is due to the non-uniformity of error diffusion in complex images. The number of pixels required to be changed to embed the image is usually lower than that of a simple, uniform image.

The algorithm performs best in simple images in all halftoning methods. The results show better results in complex error diffused images but as the image is complex, the watermark is not as perceivable as it is in simple images. This is shown in Fig 35 where the same image is embedded at the same strength.

This algorithm is not suitable for embedding hidden image information within other images. This is because the image is too perceivable when embedded with a low step but undetectable when a high step is used.

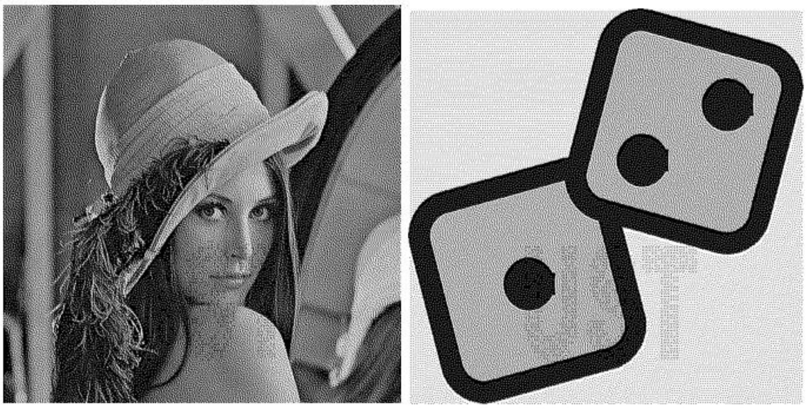


Figure 35 – Demonstration of how this algorithm is not suitable

5.2.5. Self-conjugate watermarking (image information)

As seen in appendices Z-AE, an increase in the threshold value results in a higher percentage of the extracted image made visible after the extraction process. As the threshold value increases, the PSNR and SSIM values decrease. This is due to an increase of the number of pixels changing to the opposing colour resulting in added noise to the image.

As seen in appendices Z and AA, when this embedding algorithm is used with an error diffusion algorithm on complex images, the percentage of the extracted embedded image is higher than that of simple images. This is due to the distribution of black and white pixels in the image. If both areas of the image (the current bottom half pixel and the corresponding pixel at a 180-degree rotation) are in complex areas and the change to the opposing colour is valid, the change is less perceivable. If both areas of the image are simple and the change to the opposing colour is valid, the change is more perceivable. The combination of these algorithms yields poor PSNR and SSIM results when compared against the original halftone image. This is caused by the error diffusion algorithm propagating errors to nearby pixels. The embedding algorithm alters greyscale values to achieve pixels changing to the opposite colour when halftoned. These alterations are propagated to the nearby pixels which in turn affect the halftone values of those also. A smaller image would result in higher PSNR and SSIM values as error diffusion would occur less.

In appendices AB and AC, when this algorithm is used with an ordered dither algorithm, an increase of threshold also results in the degradation of PSNR and SSIM values but results in a higher percentage of the extracted embedded image made visible. Both simple and complex images do not do particularly well in the percentage of the extracted hidden image.

In appendices AD and AE, when this algorithm is used with a stippling algorithm, better results are obtained with the use of complex images. This again, is a likely result of embedding in both areas that are complex (like error diffusion).

5.2.5.1. Result

This algorithm gives best results in complex error diffused images in terms of extraction. In all complex images, higher extraction results are achieved than in simple tests.

In general, there is a choice that can be made with the threshold value. If higher extraction success is preferred, a higher threshold can be used. If better halftone quality but a slightly worse extraction result is preferred, a lower threshold can be used.

6. Future work

There are a lot of things I would like to add to this project. The first being the total number of embedding algorithms. Five have been implemented but the original plan was to implement ten. More algorithms would allow better analysis between the different embedding methods and it would be interesting to see if certain types of embedding are affected by different half-toning algorithms. Future algorithms that I would have liked to implement would be at least three more watermarking algorithms as the self-conjugate algorithm was enjoyable to implement. This would allow a more detailed analysis could then be made between different watermarking algorithms.

The second would be to implement different steganalysis algorithms. This was the original plan for the project to measure the success in embedding but could not be implemented due to time constraints. Having these steganalysis algorithms would result in more uniform data that can be used to determine the best halftoning algorithm for each embedding algorithm. It would be more uniform as there would be no specific extraction algorithm used which can have varied results as shown in the pixel density transition embedding algorithm.

I would also like to implement the Pixel Mesh Markov Transition matrix. This could be used in the first embedding algorithm to improve upon the results obtained.

I would also like to explore block-based steganography in greater detail as I feel the implemented algorithm can be improved upon greatly. I would attempt to solve the disadvantage explained in section 5.2.2.

I am also keen to implement is the Voronoi diagrams that create stippled images. It would be interesting to see how each embedding algorithm can be altered to interact with the Voronoi diagram so that information is embedded within.

I also investigated edge-enhancement in the implemented half-toning algorithms but did not have sufficient time to implement them. It could be interesting to see if edge detection algorithms could be used to embed information in.

A GUI could be implemented for this project. Instead of running algorithms that process all images within a folder, the GUI could include a feature where an image is dragged in and results of halftoning and embedding can be displayed in real time without the need to save. This could in turn help in the testing phase for each algorithm as data can be tested instantly and be displayed in real time.

Steganography in colour images would open many possibilities for embedding. The use of halftone images limits the variation of embedding methods as only two colours are used and the number of bits in the images that can be altered are much smaller than that of colour images.

Overall, steganography is a very broad subject and there are many aspects I would like to further investigate.

7. Conclusion

Originally, as stated in my initial report, the project aims were as follows:

I aim for of all algorithms used to be written in the final report:

- Steganographic algorithms.
- Structural Similarity Index.
- Levenshtein Distance.
- Methods of attack.

I aim for my implementation and any other programs to give the best possible results whilst being efficient.

- The program needs to perform as well as the current available algorithms.

I aim to test hundreds of images to give better results

- The programs and algorithms created will need to be able to process images as efficiently as possible.

I aim for all analysis to have supporting graphs and or documentation

- The programs created will need to be able to automatically create results and graphs for all analysis to have supporting data.

All but Levenshtein Distance and Methods of attack have been implemented in this project. This is due to inappropriate results obtained using Levenshtein Distance and the constraints on the project timeline.

I have created a project that has different halftoning methods and different embedding methods implemented. Comparisons and analysis have been made between each embedding algorithm against each of the halftoning method to see which halftoning method created the best results in terms of PSNR values, SSIM values, and extraction success. There is a lot more work that I would like to continue in this area of image processing as there are many algorithms, I have taken an interest to during the research period of this project.

The implementation of halftoning algorithms in general went very well. This is due to more research papers being available that explain the area of halftoning techniques. The implementation of embedding algorithms did not go quite as well, this is due to a limited number of research papers on specific embedding algorithms. I did not understand some areas of the embedding algorithm as I did not fully understand the equations.

Overall, this project has been thoroughly enjoyable, and I would like to continue to research in steganography.

8. Reflection on learning

This project has had an overall impact on development of myself. I have learnt new soft transferable skills and hard technical skills.

The project is by far the largest project I have ever undertaken. I have had to work on my time management skills during this project. At the start of the project, my time management was poor and felt as though I could easily catch up on work not done. This thought process was quickly changed when I started research on the embedding algorithms. A lot of time has been spent looking at research papers of various algorithms and I now feel as though I understand quite a lot of the equations and can sometimes improve on small areas of algorithms in terms of efficiency.

At the start of the project, I foolishly thought that it could be an easy one to implement. Little did I know what was to come down the line. Embedding algorithms which are quite complex took a long time to implement and had I have my current work ethic towards the project sooner, the more embedding algorithms would have been implemented.

This is the largest report written and have slightly underestimated the mammoth task it has been in explaining all implementations of halftoning and embedding methods. Ideally, I would have focussed on fewer halftoning algorithms and a larger number of embedding algorithms. This would create a more cohesive report that would analyse one or two halftoning methods against a wider variety of embedding techniques. After writing this report, I feel as though my report writing skills have also improved.

I have learnt how to tackle problems when my implementations did not quite go to plan though the use of the Agile Methodology. The use of this helped me a great deal in troubleshooting any problems that occurred during the implementation.

I have learnt how to create complex graphs in Excel. This knowledge was not previously known and took a white to generate each graph you see in the appendix.

Overall, this project has taught me a lot in terms of approach to work. It has allowed me to work on my time management skills and self-management. It has interested me in pursuing a job in the field of steganography. I have found that I enjoyed the project a lot more the more I worked on it. This is due to me learning a great deal about the field of steganography and my ability to understand a wider range of research papers. It’s also due to me taking an interest in areas that challenge myself to learn new information which can be applied across many areas within computer science.

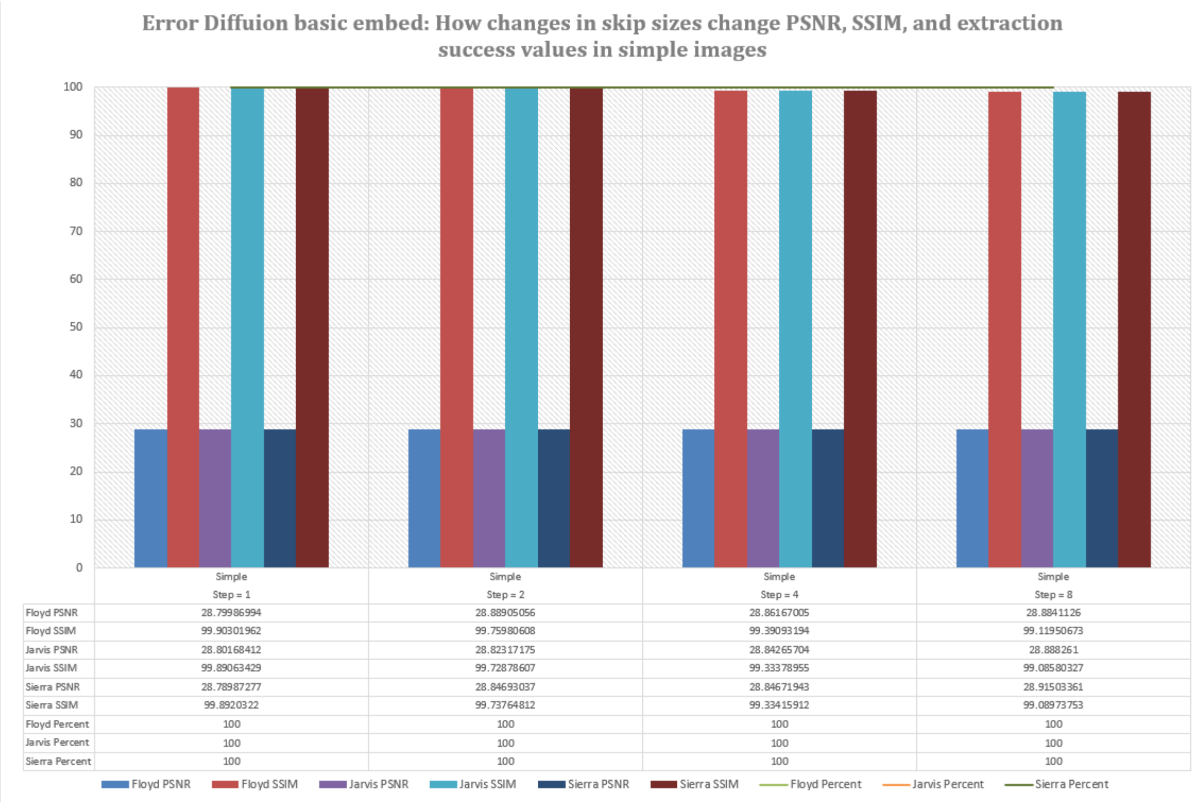
9. Appendices

- Analysis
    - PSNR+SSIM
        - __pycache__
        - PSNRSSIM.py
- Embedding
    - 1. Basic Text
        - Error Diffusion
            - Basic Floyd.py
            - Basic Jarvis.py
            - Basic Sierra.py
        - Ordered
            - Basic 2x2.py
            - Basic 4x4.py
            - Basic 8x8.py
        - Stippled
            - Basic 2 Gradient.py
            - Basic 4 Gradient.py
            - Basic 6 Gradient.py
            - Basic 8 Gradient.py
    - 2. Density Text
        - Error Diffusion
            - Density Floyd.py
            - Density Jarvis.py
            - Density Sierra.py
        - Ordered
            - Density 2x2.py
            - Density 4x4.py
            - Density 8x8.py
        - Stippled
            - Density 2 Gradient.py
            - Density 4 Gradient.py
            - Density 6 Gradient.py
            - Density 8 Gradient.py
    - 3. Greyscale Text
        - Error Diffusion
            - Greyscale Floyd.py
            - Greyscale Jarvis.py
            - Greyscale Sierra.py
        - Ordered
            - Greyscale 2x2.py
            - Greyscale 4x4.py
            - Greyscale 8x8.py
        - Stippled
            - Greyscale 2 Gradient.py
            - Greyscale 4 Gradient.py
            - Greyscale 6 Gradient.py
            - Greyscale 8 Gradient.py
    - 4. Basic Watermarking
        - Error Diffusion
            - Basic Floyd.py

            - Basic Jarvis.py
            - Basic Sierra.py
        - Ordered
            - Basic 2x2.py
            - Basic 4x4.py
            - Basic 8x8.py
        - Stippled
            - Basic 2 Gradient.py
            - Basic 4 Gradient.py
            - Basic 6 Gradient.py
            - Basic 8 Gradient.py
        - Image To Hide
            - toHide.png
    - 5. Watermarking
        - Error Diffusion
            - Watermark Floyd.py
            - Watermark Jarvis.py
            - Watermark Sierra.py
        - Ordered
            - Watermark 2x2.py
            - Watermark 4x4.py
            - Watermark 8x8.py
        - Stippled
            - Watermark 2 Gradient.py
            - Watermark 4 Gradient.py
            - Watermark 6 Gradient.py
            - Watermark 8 Gradient.py
        - Image To Hide
            - toHide.png
- Half-Toning
        - Error Diffusion
            - Floyd.py
            - Jarvis.py
            - Sierra.py
        - Ordered
            - 2x2.py
            - 4x4.py
            - 8x8.py
        - Stippled
            - 2 Gradient.py
            - 4 Gradient.py
            - 6 Gradient.py
            - 8 Gradient.py
- Images
    - Basic Halftone
        - ALL BASIC HALFTONE IMAGES. Under relevant folders.
    - Embedded
        - 1. Basic Text
            - Error Diffusion
                - ALL CORRESPONDING EMBEDDED IMAGES
            - Ordered

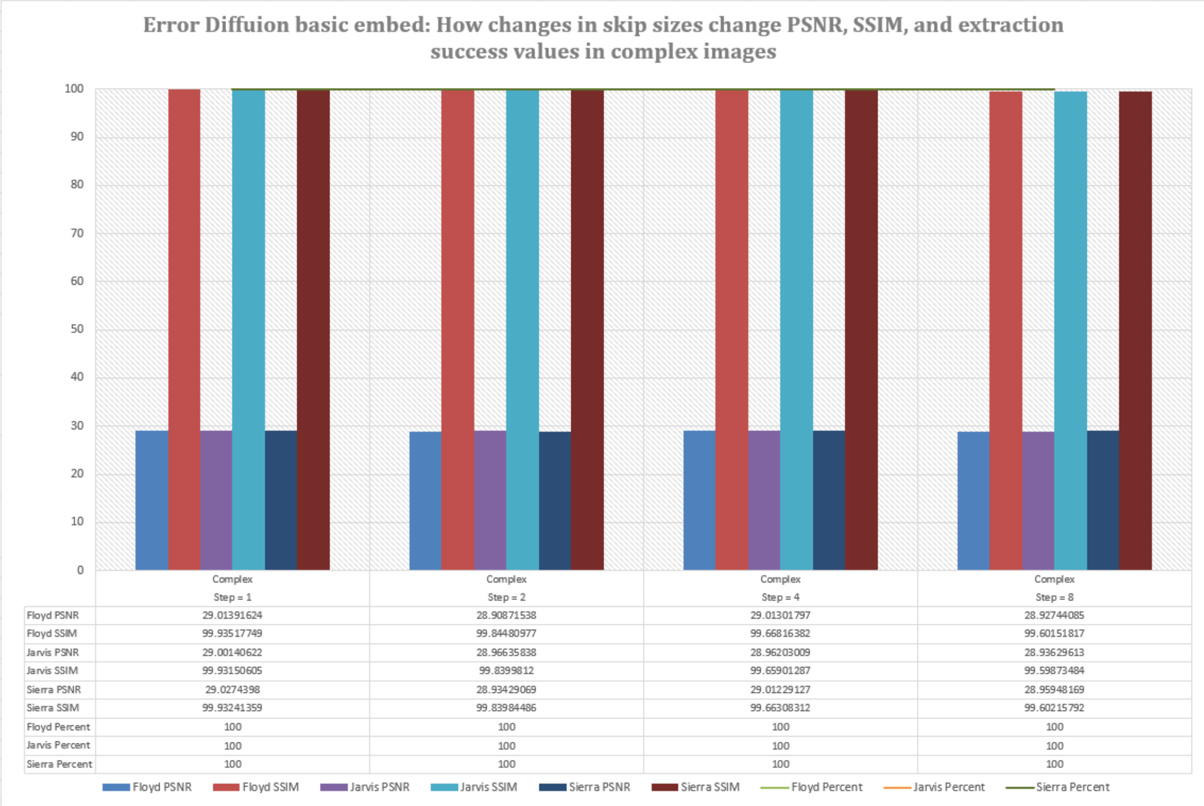
      - ALL CORRESPONDING EMBEDDED IMAGES
    - Stippled
      - ALL CORRESPONDING EMBEDDED IMAGES
  - 2. Density Text
    - Error Diffusion
      - ALL CORRESPONDING EMBEDDED IMAGES
    - Ordered
      - ALL CORRESPONDING EMBEDDED IMAGES
    - Stippled
      - ALL CORRESPONDING EMBEDDED IMAGES
  - 3. Greyscale Text
    - Error Diffusion
    - Ordered
    - Stippled
  - 4. Basic Watermark
    - Error Diffusion
      - ALL CORRESPONDING EMBEDDED IMAGES
    - Ordered
      - ALL CORRESPONDING EMBEDDED IMAGES
    - Stippled
      - ALL CORRESPONDING EMBEDDED IMAGES
  - 5. Watermark
    - Error Diffusion
      - ALL CORRESPONDING EMBEDDED AND XOR IMAGES
    - Ordered
      - ALL CORRESPONDING EMBEDDED AND XOR IMAGES
    - Stippled
      - ALL CORRESPONDING EMBEDDED AND XOR IMAGES
- Original
  - ALL ORIGINAL IMAGES

Appendix A – Structure to the projects algorithms



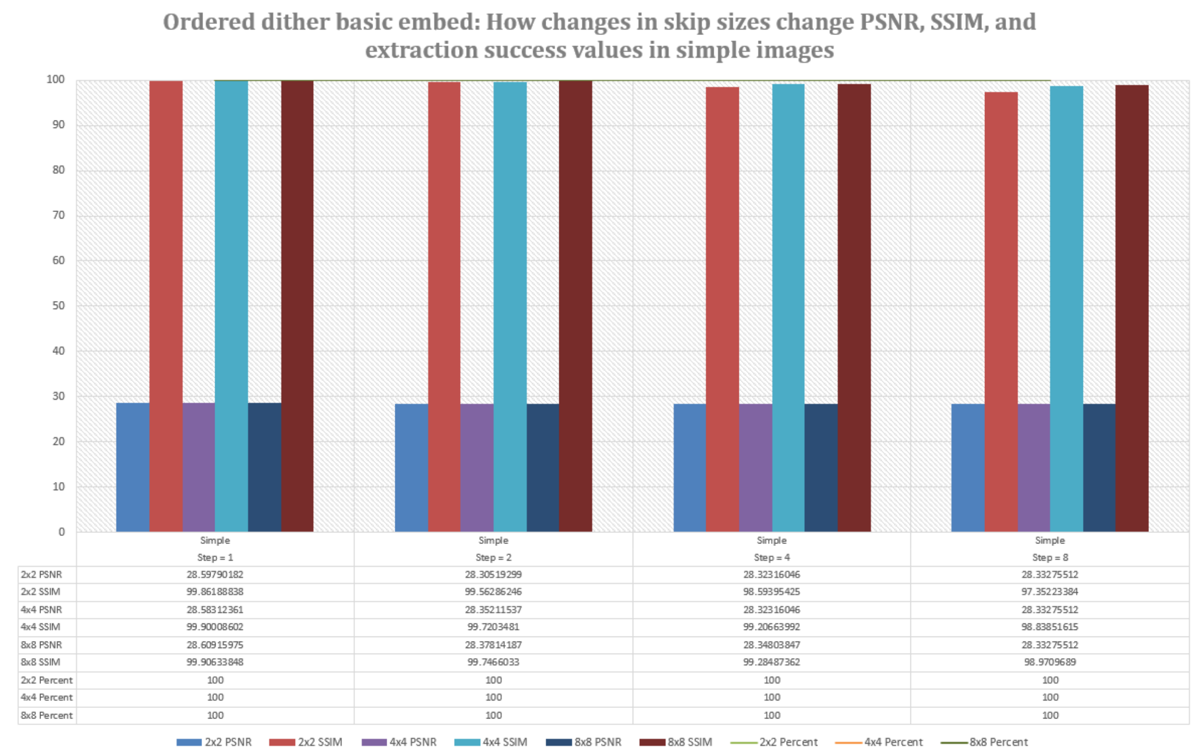
	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
Floyd PSNR	28.79986994	28.88905056	28.86167005	28.8841126
Floyd SSIM	99.90301962	99.75980608	99.39093194	99.11950673
Jarvis PSNR	28.80168412	28.82317175	28.84265704	28.888261
Jarvis SSIM	99.89063429	99.72878607	99.33378955	99.08580327
Sierra PSNR	28.78987277	28.84693037	28.84671943	28.91503361
Sierra SSIM	99.8920322	99.73764812	99.33415912	99.08973753
Floyd Percent	100	100	100	100
Jarvis Percent	100	100	100	100
Sierra Percent	100	100	100	100

Appendix B – Basic pixel embedding (text information) in simple error diffused images



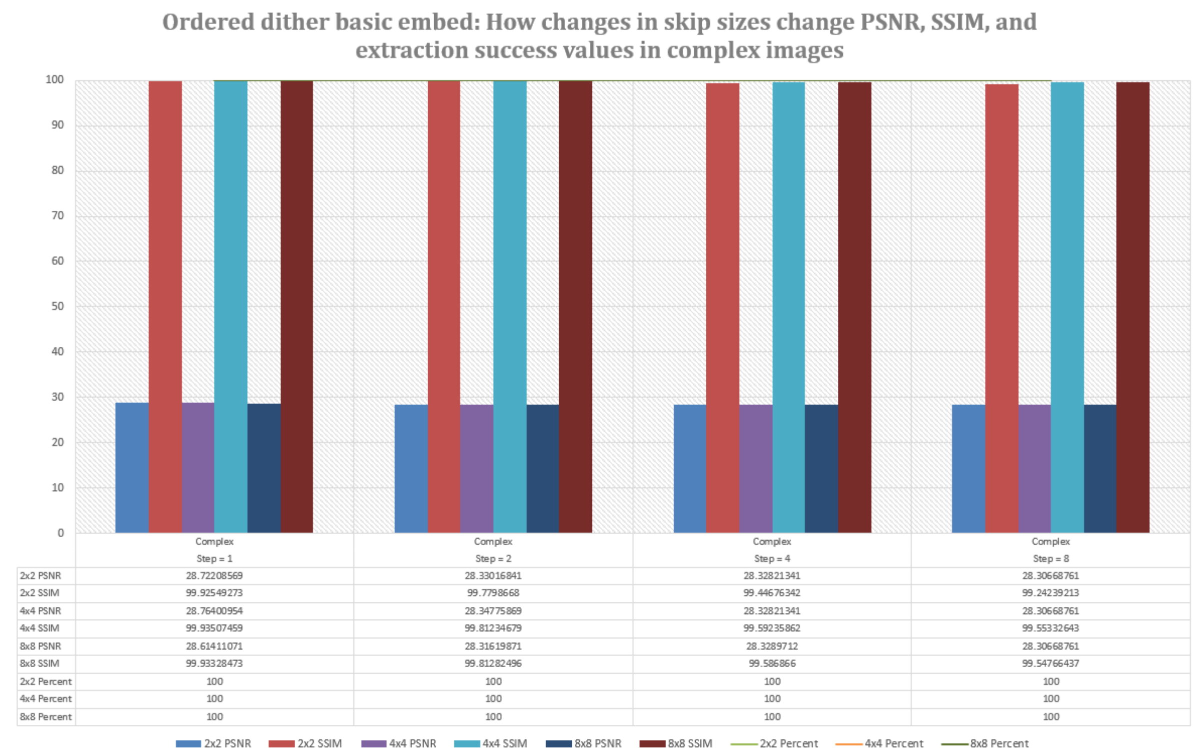
	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
Floyd PSNR	29.01391624	28.90871538	29.01301797	28.92744085
Floyd SSIM	99.93517749	99.84480977	99.66816382	99.60151817
Jarvis PSNR	29.00140622	28.96635838	28.96203009	28.93629613
Jarvis SSIM	99.93150605	99.8399812	99.65901287	99.59873484
Sierra PSNR	29.0274398	28.93429069	29.01229127	28.95948169
Sierra SSIM	99.93241359	99.83984486	99.66308312	99.60215792
Floyd Percent	100	100	100	100
Jarvis Percent	100	100	100	100
Sierra Percent	100	100	100	100

Appendix C – Basic pixel embedding (text information) in complex error diffused images



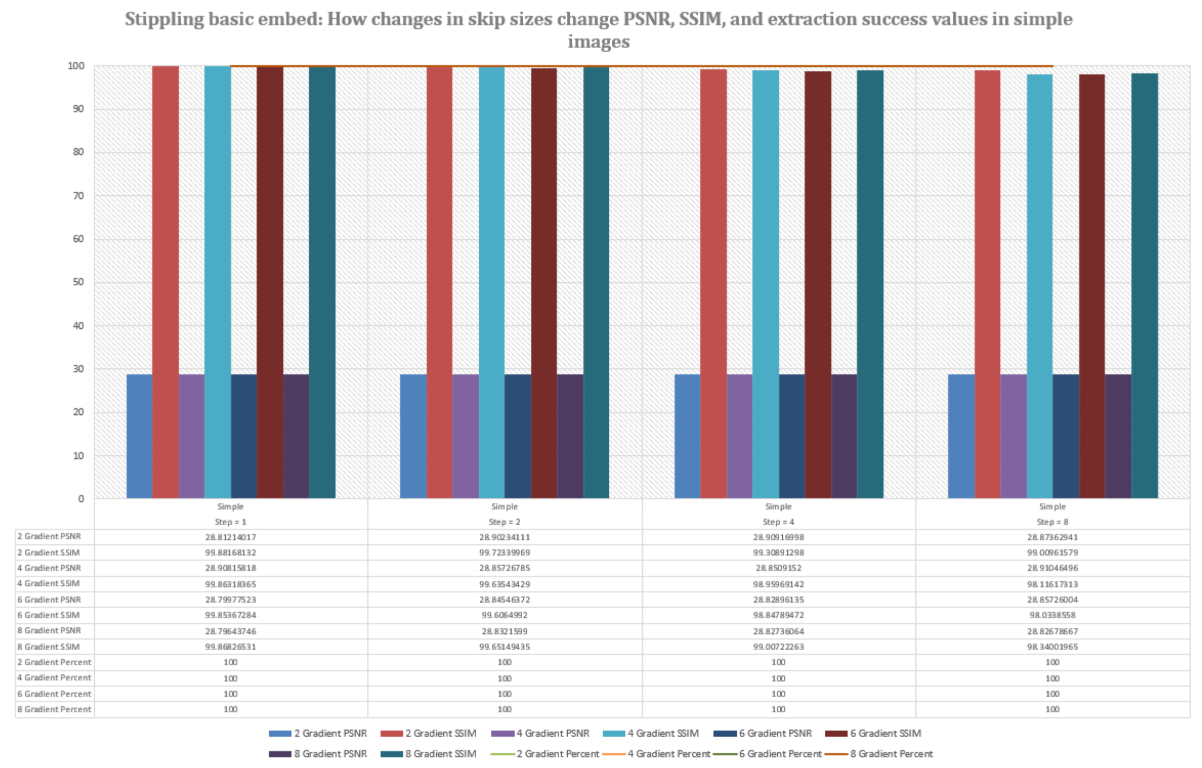
	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
2x2 PSNR	28.59790182	28.30519299	28.32316046	28.33275512
2x2 SSIM	99.86188838	99.56286246	98.59395425	97.35223384
4x4 PSNR	28.58312361	28.35211537	28.32316046	28.33275512
4x4 SSIM	99.90008602	99.7203481	99.20663992	98.83851615
8x8 PSNR	28.60915975	28.37814187	28.34803847	28.33275512
8x8 SSIM	99.90633848	99.7466033	99.28487362	98.9709689
2x2 Percent	100	100	100	100
4x4 Percent	100	100	100	100
8x8 Percent	100	100	100	100

Appendix D – Basic pixel embedding (text information) in simple order dithered images



	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
2x2 PSNR	28.72208569	28.33016841	28.32821341	28.30668761
2x2 SSIM	99.92549273	99.7798668	99.44676342	99.24239213
4x4 PSNR	28.76400954	28.34775869	28.32821341	28.30668761
4x4 SSIM	99.93507459	99.81234679	99.59235862	99.55332643
8x8 PSNR	28.61411071	28.31619871	28.3289712	28.30668761
8x8 SSIM	99.93328473	99.81282496	99.586866	99.54766437
2x2 Percent	100	100	100	100
4x4 Percent	100	100	100	100
8x8 Percent	100	100	100	100

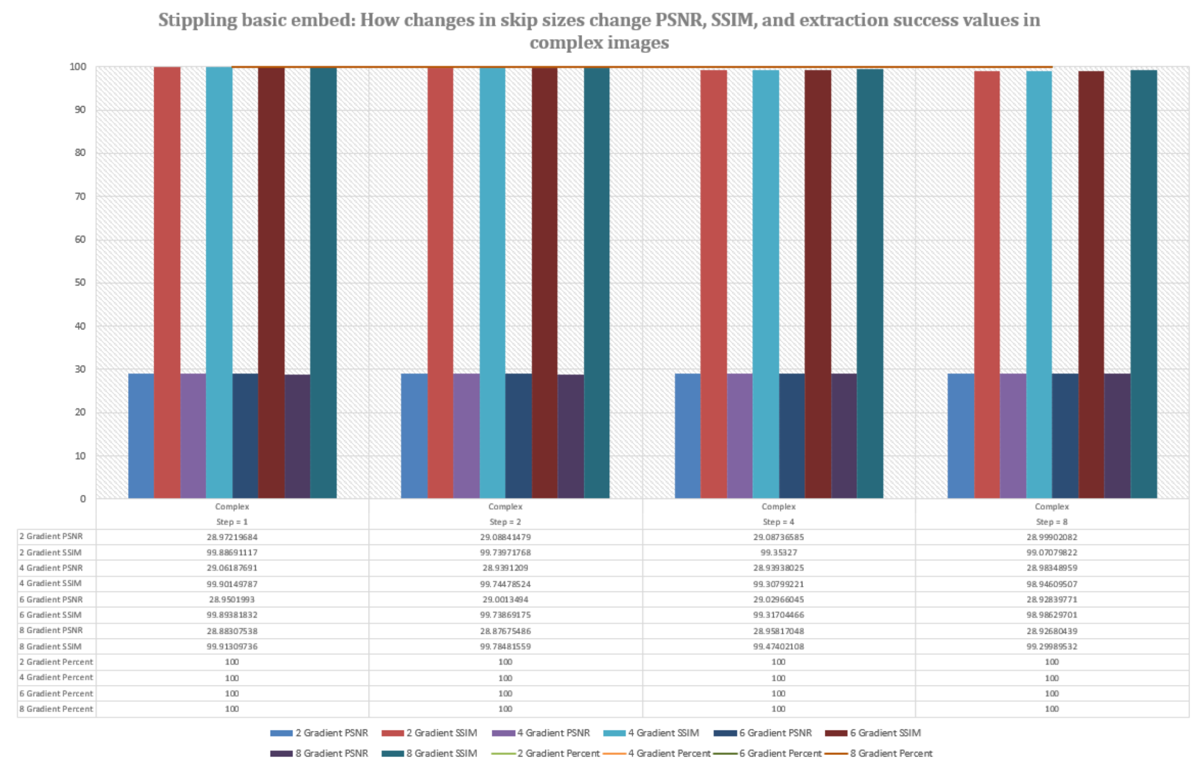
Appendix E – Basic pixel embedding (text information) in complex order dithered images



Stippling basic embed: How changes in skip sizes change PSNR, SSIM, and extraction success values in simple images

	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
2 Gradient PSNR	28.81214017	28.90234111	28.90916998	28.87362941
2 Gradient SSIM	99.88168132	99.72339969	99.30891298	99.00961579
4 Gradient PSNR	28.90815818	28.85726785	28.8509152	28.91046496
4 Gradient SSIM	99.86318365	99.63543429	98.95969142	98.11617313
6 Gradient PSNR	28.79977523	28.84546372	28.82896135	28.85726004
6 Gradient SSIM	99.85367284	99.6064992	98.84789472	98.0338558
8 Gradient PSNR	28.79643746	28.8321599	28.82736064	28.82678667
8 Gradient SSIM	99.86826531	99.65149435	99.00722263	98.34001965
2 Gradient Percent	100	100	100	100
4 Gradient Percent	100	100	100	100
6 Gradient Percent	100	100	100	100
8 Gradient Percent	100	100	100	100

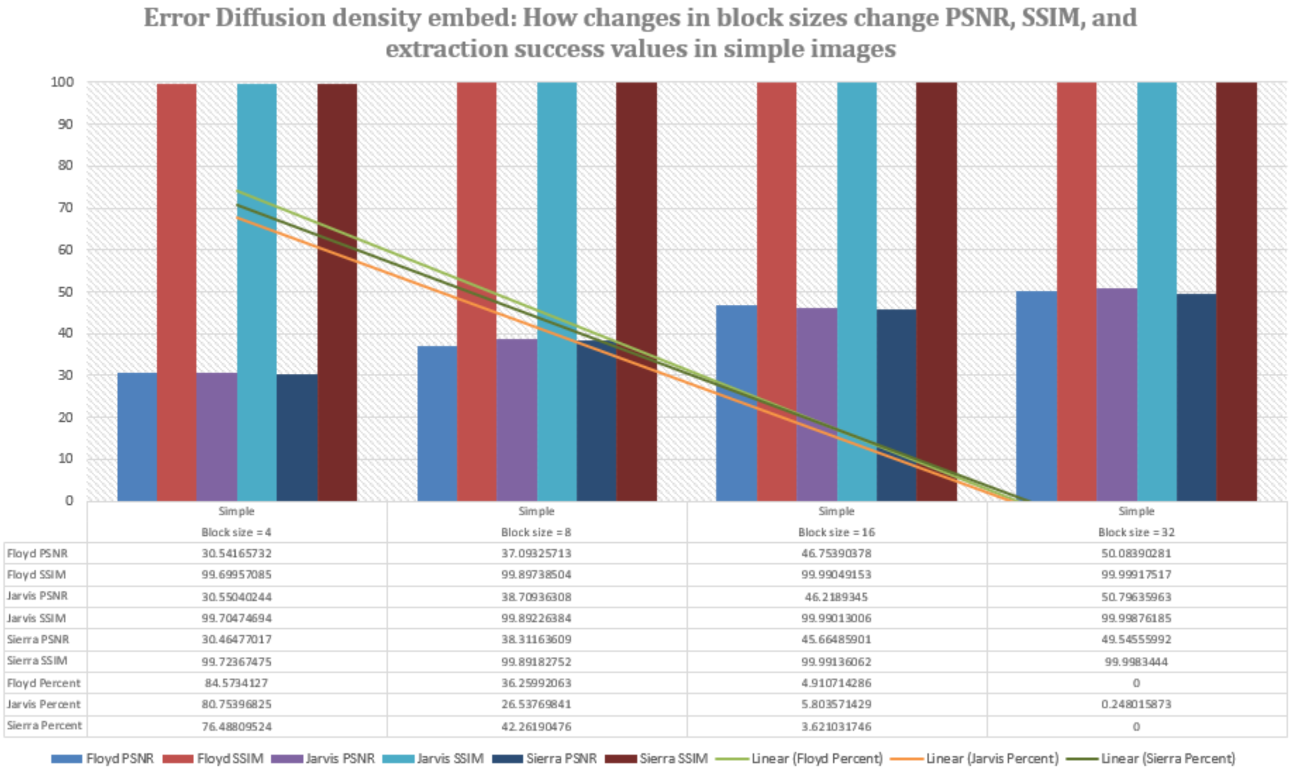
Appendix F – Basic pixel embedding (text information) in simple stippled images



Stippling basic embed: How changes in skip sizes change PSNR, SSIM, and extraction success values in complex images

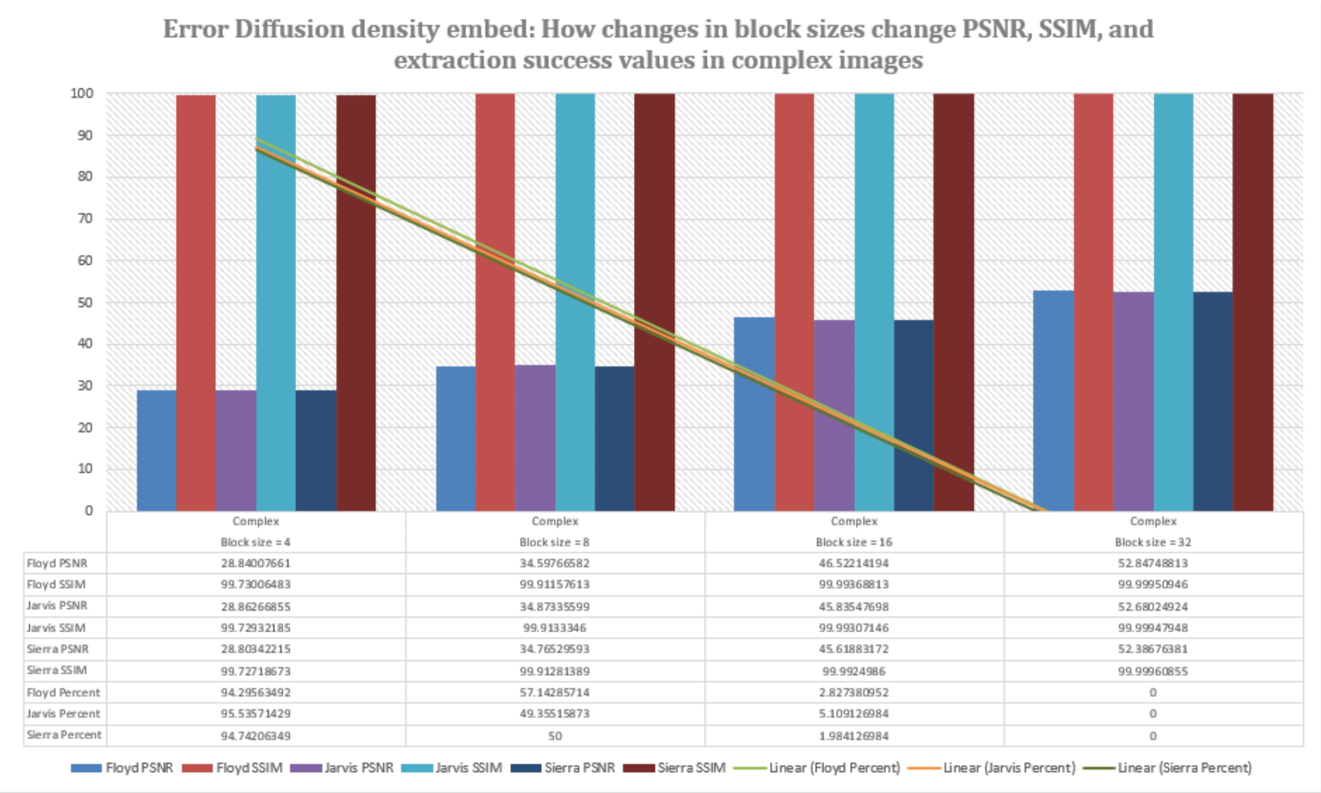
	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
2 Gradient PSNR	28.97219684	29.08841479	29.08736585	28.99902082
2 Gradient SSIM	99.88691117	99.73971768	99.35327	99.07079822
4 Gradient PSNR	29.06187691	28.9391209	28.93938025	28.98348959
4 Gradient SSIM	99.90149787	99.74478524	99.30799221	98.94609507
6 Gradient PSNR	28.9501993	29.0013494	29.02966045	28.92839771
6 Gradient SSIM	99.89381832	99.73869175	99.31704466	98.98629701
8 Gradient PSNR	28.88307538	28.87675486	28.95817048	28.92680439
8 Gradient SSIM	99.91309736	99.78481559	99.47402108	99.29989532
2 Gradient Percent	100	100	100	100
4 Gradient Percent	100	100	100	100
6 Gradient Percent	100	100	100	100
8 Gradient Percent	100	100	100	100

Appendix G – Basic pixel embedding (text information) in complex stippled images



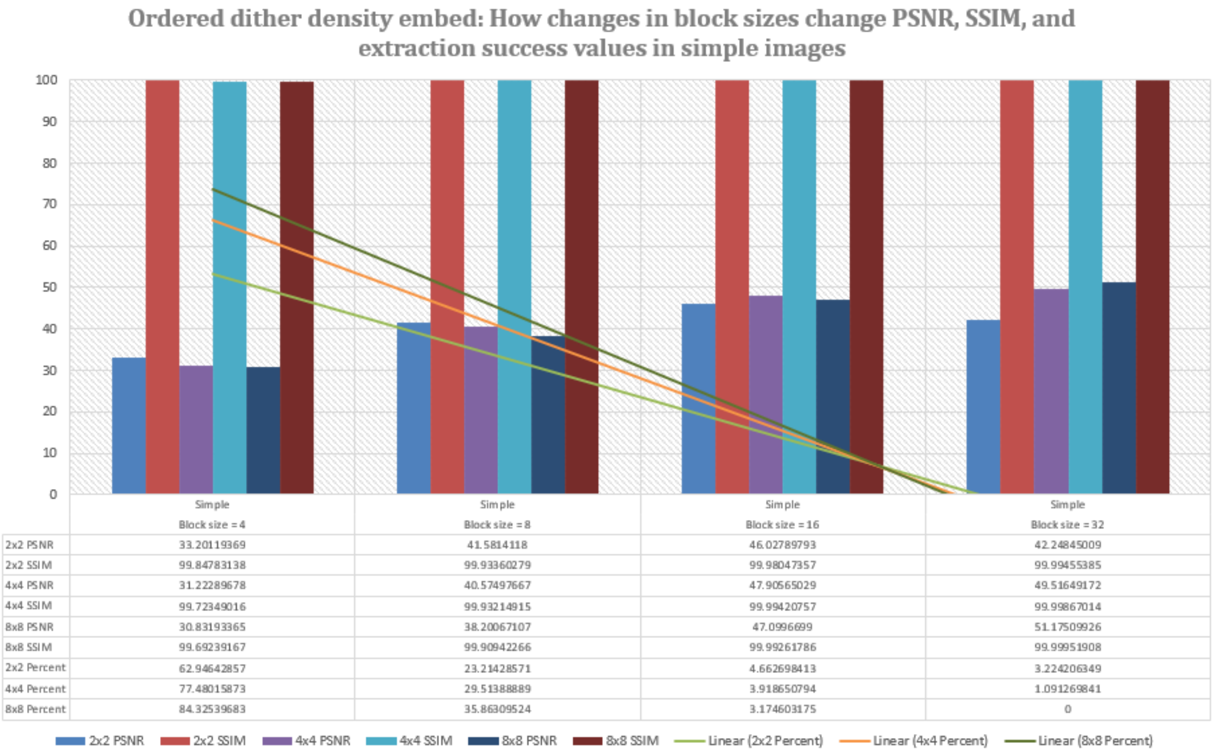
	Simple Block size = 4	Simple Block size = 8	Simple Block size = 16	Simple Block size = 32
Floyd PSNR	30.54165732	37.09325713	46.75390378	50.08390281
Floyd SSIM	99.69957085	99.89738504	99.99049153	99.99917517
Jarvis PSNR	30.55040244	38.70936308	46.2189345	50.79635963
Jarvis SSIM	99.70474694	99.89226384	99.99013006	99.99876185
Sierra PSNR	30.46477017	38.31163609	45.66485901	49.54555992
Sierra SSIM	99.72367475	99.89182752	99.99136062	99.9983444
Floyd Percent	84.5734127	36.25992063	4.910714286	0
Jarvis Percent	80.75396825	26.53769841	5.803571429	0.248015873
Sierra Percent	76.48809524	42.26190476	3.621031746	0

Appendix H – Pixel density transition (text information) in simple error diffused images



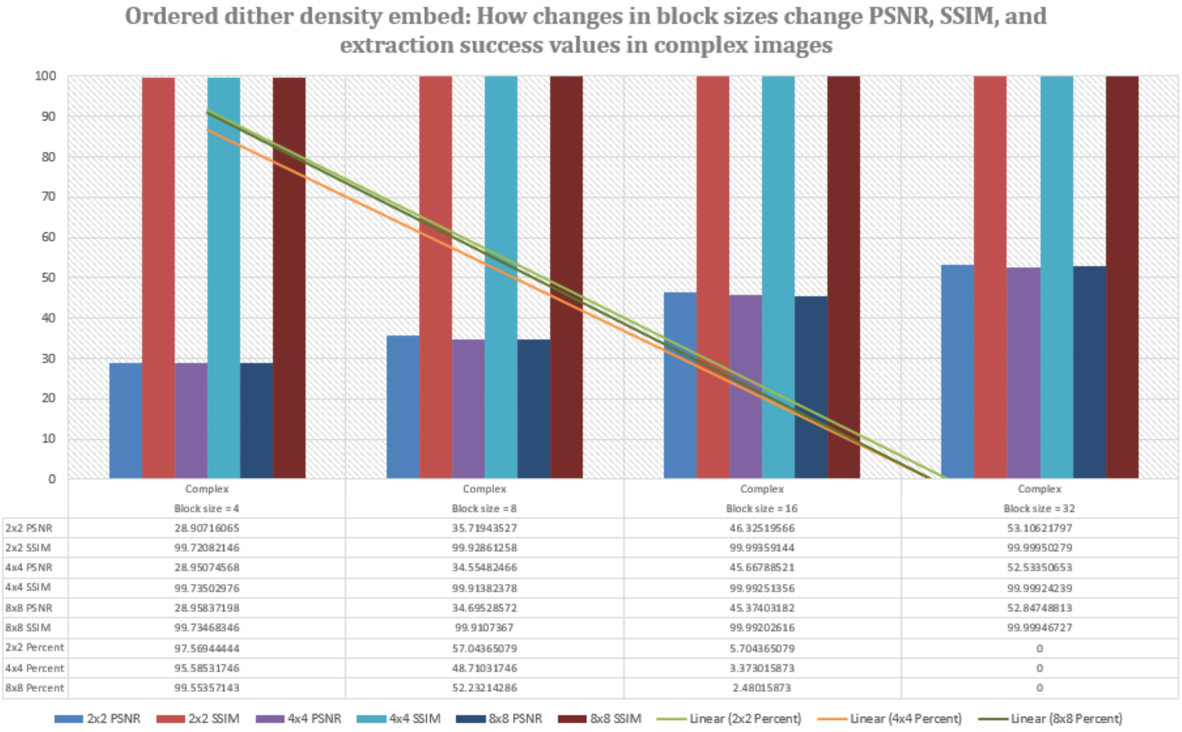
	Complex Block size = 4	Complex Block size = 8	Complex Block size = 16	Complex Block size = 32
Floyd PSNR	28.84007661	34.59766582	46.52214194	52.84748813
Floyd SSIM	99.73006483	99.91157613	99.99368813	99.99950946
Jarvis PSNR	28.86266855	34.87335599	45.83547698	52.68024924
Jarvis SSIM	99.72932185	99.9133346	99.99307146	99.99947948
Sierra PSNR	28.80342215	34.76529593	45.61883172	52.38676381
Sierra SSIM	99.72718673	99.91281389	99.9924986	99.99960855
Floyd Percent	94.29563492	57.14285714	2.827380952	0
Jarvis Percent	95.53571429	49.35515873	5.109126984	0
Sierra Percent	94.74206349	50	1.984126984	0

Appendix I – Pixel density transition (text information) in complex error diffused images



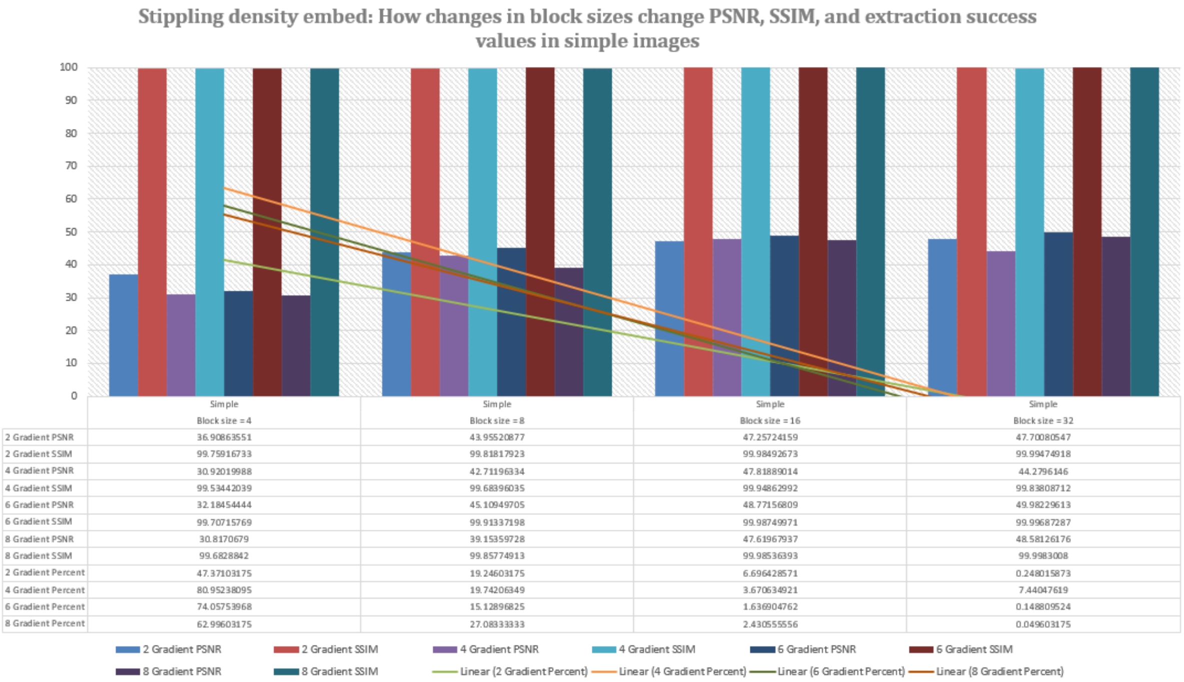
	Simple Block size = 4	Simple Block size = 8	Simple Block size = 16	Simple Block size = 32
2x2 PSNR	33.20119369	41.5814118	46.02789793	42.24845009
2x2 SSIM	99.84783138	99.93360279	99.98047357	99.99455385
4x4 PSNR	31.22289678	40.57497667	47.90565029	49.51649172
4x4 SSIM	99.72349016	99.93214915	99.99420757	99.99867014
8x8 PSNR	30.83193365	38.20067107	47.0996699	51.17509926
8x8 SSIM	99.69239167	99.90942266	99.99261786	99.99951908
2x2 Percent	62.94642857	23.21428571	4.662698413	3.224206349
4x4 Percent	77.48015873	29.51388889	3.918650794	1.091269841
8x8 Percent	84.32539683	35.86309524	3.174603175	0

Appendix J – Pixel density transition (text information) in simple order dithered images



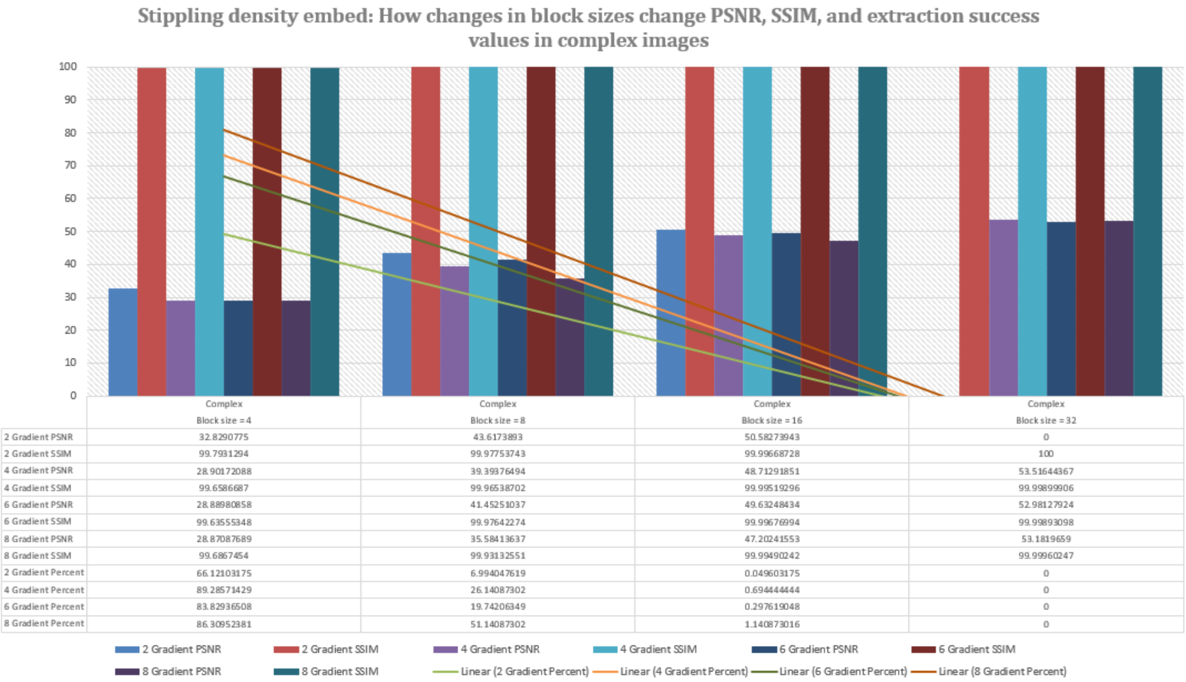
	Complex Block size = 4	Complex Block size = 8	Complex Block size = 16	Complex Block size = 32
2x2 PSNR	28.90716065	35.71943527	46.32519566	53.10621797
2x2 SSIM	99.72082146	99.92861258	99.99359144	99.99950279
4x4 PSNR	28.95074568	34.55482466	45.66788521	52.53350653
4x4 SSIM	99.73502976	99.91382378	99.99251356	99.99924239
8x8 PSNR	28.95837198	34.69528572	45.37403182	52.84748813
8x8 SSIM	99.73468346	99.9107367	99.99202616	99.99946727
2x2 Percent	97.56944444	57.04365079	5.704365079	0
4x4 Percent	95.58531746	48.71031746	3.373015873	0
8x8 Percent	99.55357143	52.23214286	2.48015873	0

Appendix K - Pixel density transition (text information) in complex order dithered images



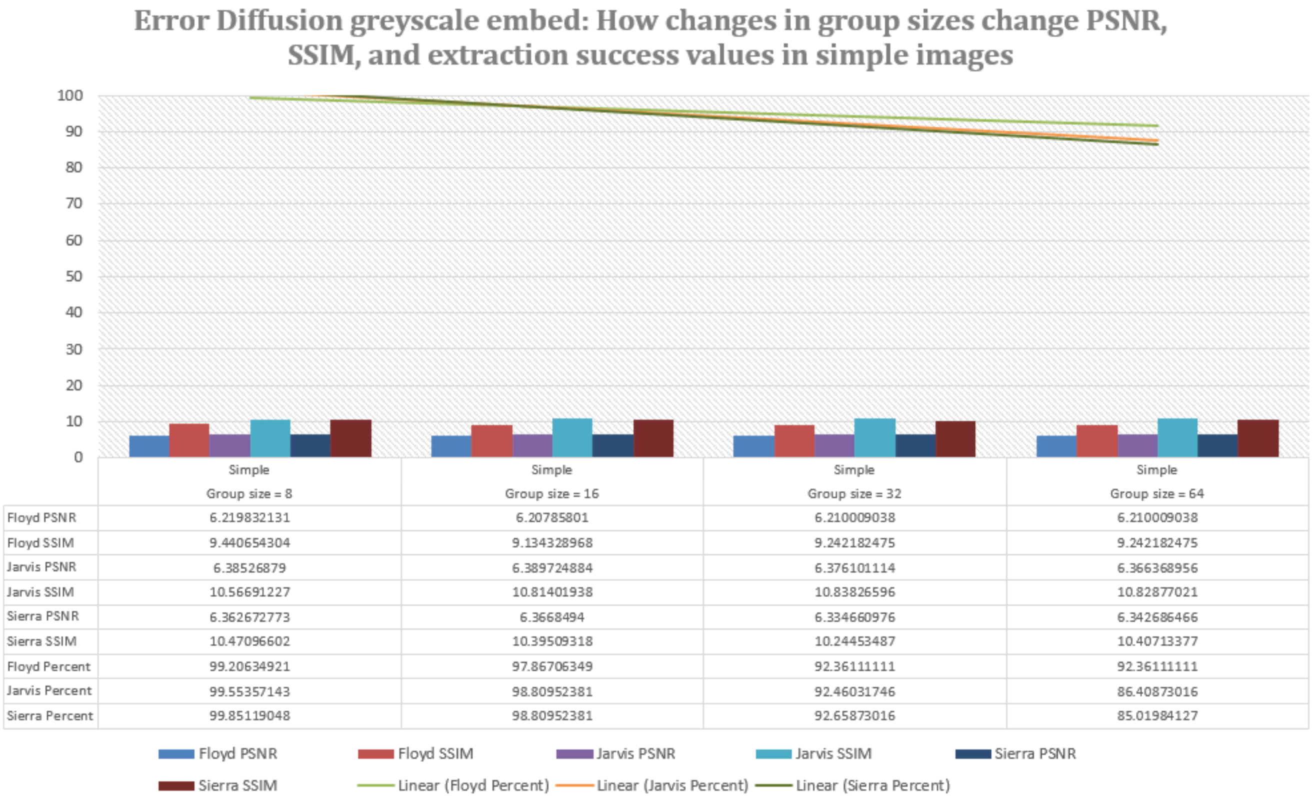
	Simple Block size = 4	Simple Block size = 8	Simple Block size = 16	Simple Block size = 32
2 Gradient PSNR	36.90863551	43.95520877	47.25724159	47.70080547
2 Gradient SSIM	99.75916733	99.81817923	99.98492673	99.99474918
4 Gradient PSNR	30.92019988	42.71196334	47.81889014	44.2796146
4 Gradient SSIM	99.53442039	99.68396035	99.94862992	99.83808712
6 Gradient PSNR	32.18454444	45.10949705	48.77156809	49.98229613
6 Gradient SSIM	99.70715769	99.91337198	99.98749971	99.99687287
8 Gradient PSNR	30.8170679	39.15359728	47.61967937	48.58126176
8 Gradient SSIM	99.6828842	99.85774913	99.98536393	99.9983008
2 Gradient Percent	47.37103175	19.24603175	6.696428571	0.248015873
4 Gradient Percent	80.95238095	19.74206349	3.670634921	7.44047619
6 Gradient Percent	74.05753968	15.12896825	1.636904762	0.148809524
8 Gradient Percent	62.99603175	27.08333333	2.430555556	0.049603175

Appendix L – Pixel density transition (text information) in simple stippled images



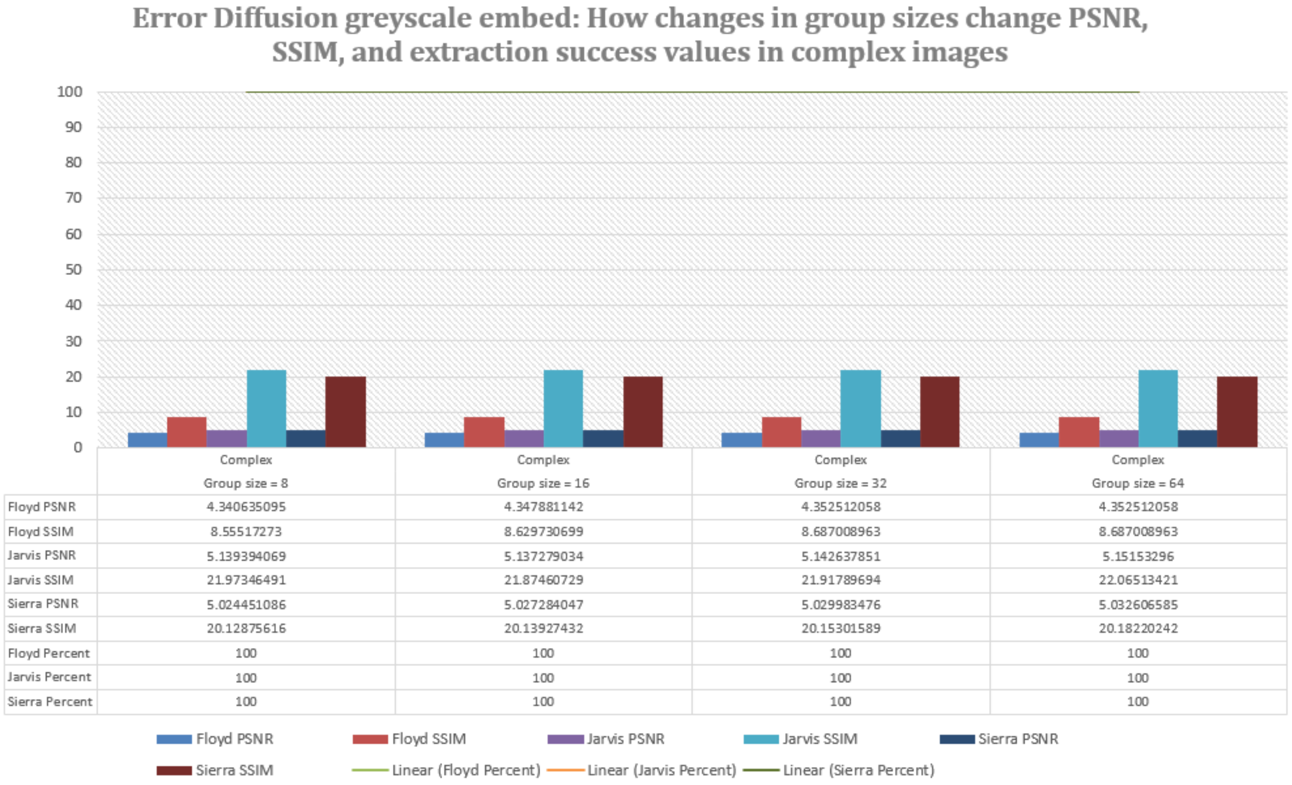
	Complex Block size = 4	Complex Block size = 8	Complex Block size = 16	Complex Block size = 32
2 Gradient PSNR	32.8290775	43.6173893	50.58273943	0
2 Gradient SSIM	99.7931294	99.97753743	99.99668728	100
4 Gradient PSNR	28.90172088	39.39376494	48.71291851	53.51644367
4 Gradient SSIM	99.6586687	99.96538702	99.99519296	99.99899906
6 Gradient PSNR	28.88980858	41.45251037	49.63248434	52.98127924
6 Gradient SSIM	99.63555348	99.97642274	99.99676994	99.99893098
8 Gradient PSNR	28.87087689	35.58413637	47.20241553	53.1819659
8 Gradient SSIM	99.6867454	99.93132551	99.99490242	99.99960247
2 Gradient Percent	66.12103175	6.994047619	0.049603175	0
4 Gradient Percent	89.28571429	26.14087302	0.694444444	0
6 Gradient Percent	83.82936508	19.74206349	0.297619048	0
8 Gradient Percent	86.30952381	51.14087302	1.140873016	0

Appendix M – Pixel density transition (text information) in complex stippled images



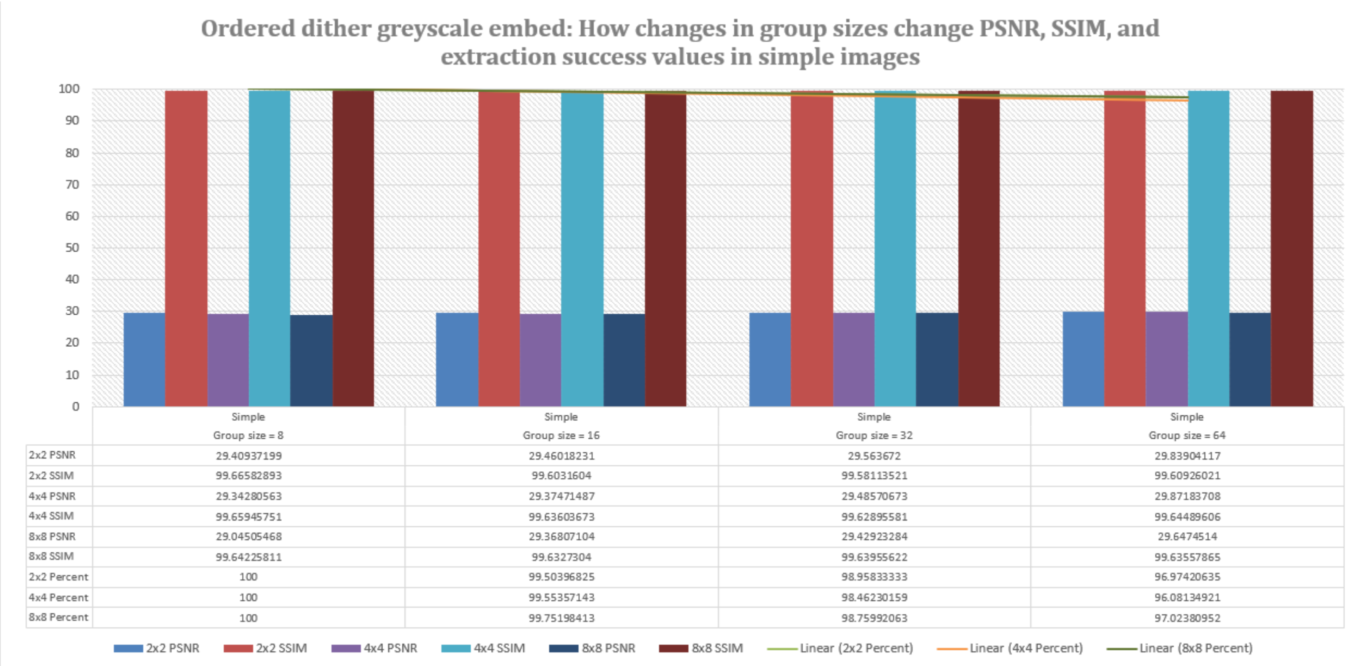
	Simple Group size = 8	Simple Group size = 16	Simple Group size = 32	Simple Group size = 64
Floyd PSNR	6.219832131	6.20785801	6.210009038	6.210009038
Floyd SSIM	9.440654304	9.134328968	9.242182475	9.242182475
Jarvis PSNR	6.38526879	6.389724884	6.376101114	6.366368956
Jarvis SSIM	10.56691227	10.81401938	10.83826596	10.82877021
Sierra PSNR	6.362672773	6.3668494	6.334660976	6.342686466
Sierra SSIM	10.47096602	10.39509318	10.24453487	10.40713377
Floyd Percent	99.20634921	97.86706349	92.36111111	92.36111111
Jarvis Percent	99.55357143	98.80952381	92.46031746	86.40873016
Sierra Percent	99.85119048	98.80952381	92.65873016	85.01984127

Appendix N – Greyscale pixel alteration (text information) in simple error diffused images



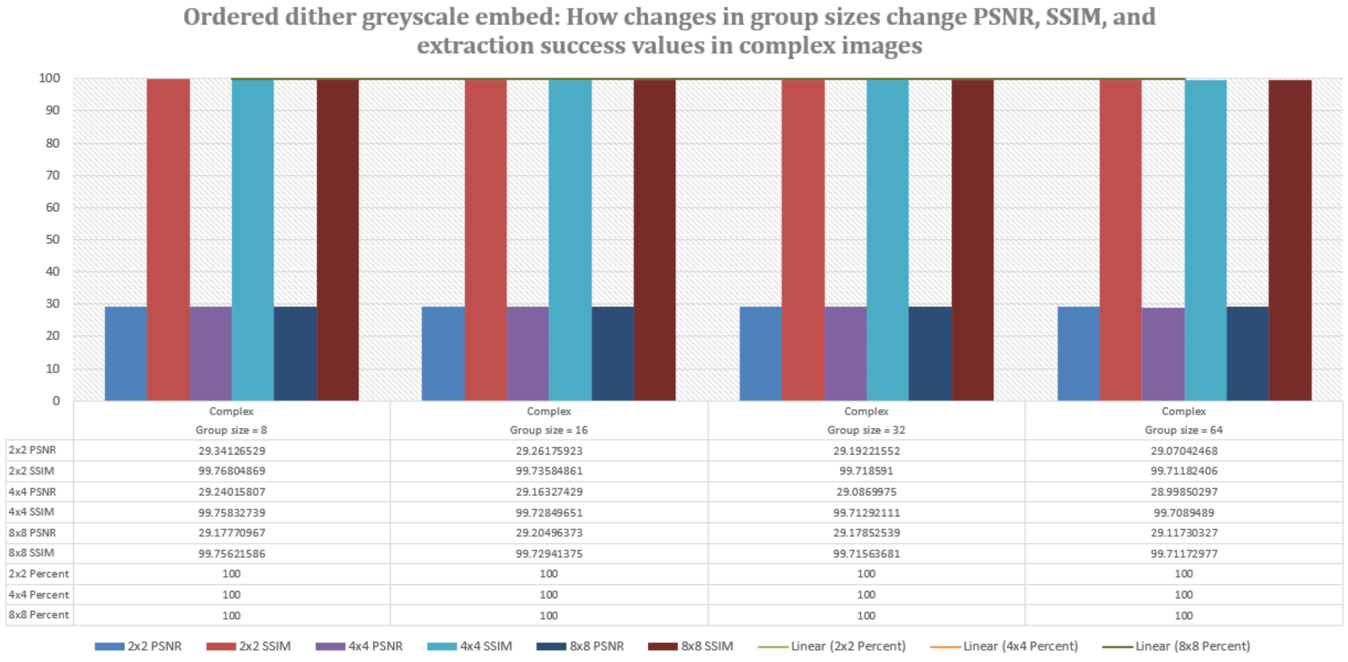
	Complex Group size = 8	Complex Group size = 16	Complex Group size = 32	Complex Group size = 64
Floyd PSNR	4.340635095	4.347881142	4.352512058	4.352512058
Floyd SSIM	8.55517273	8.629730699	8.687008963	8.687008963
Jarvis PSNR	5.139394069	5.137279034	5.142637851	5.15153296
Jarvis SSIM	21.97346491	21.87460729	21.91789694	22.06513421
Sierra PSNR	5.024451086	5.027284047	5.029983476	5.032606585
Sierra SSIM	20.12875616	20.13927432	20.15301589	20.18220242
Floyd Percent	100	100	100	100
Jarvis Percent	100	100	100	100
Sierra Percent	100	100	100	100

Appendix O – Greyscale pixel alteration (text information) in complex error diffused images



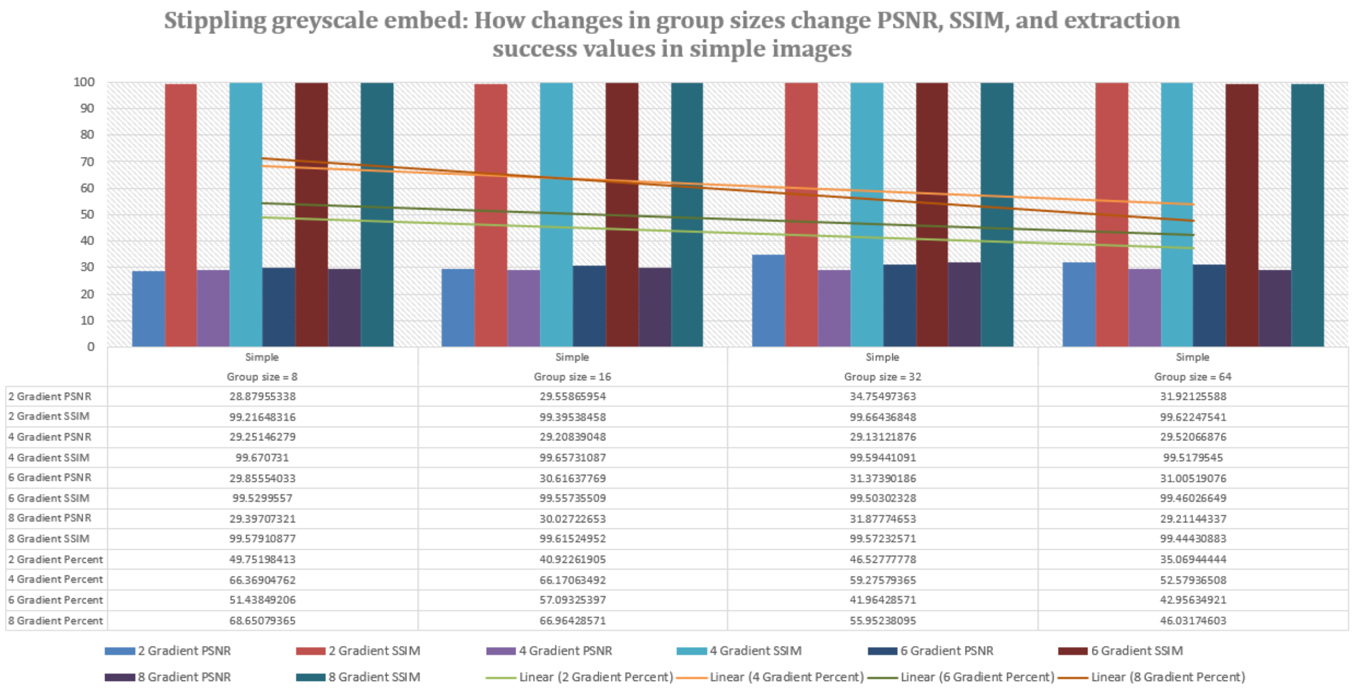
	Simple Group size = 8	Simple Group size = 16	Simple Group size = 32	Simple Group size = 64
2x2 PSNR	29.40937199	29.46018231	29.563672	29.83904117
2x2 SSIM	99.66582893	99.6031604	99.58113521	99.60926021
4x4 PSNR	29.34280563	29.37471487	29.48570673	29.87183708
4x4 SSIM	99.65945751	99.63603673	99.62895581	99.64489606
8x8 PSNR	29.04505468	29.36807104	29.42923284	29.6474514
8x8 SSIM	99.64225811	99.6327304	99.63955622	99.63557865
2x2 Percent	100	99.50396825	98.95833333	96.97420635
4x4 Percent	100	99.55357143	98.46230159	96.08134921
8x8 Percent	100	99.75198413	98.75992063	97.02380952

Appendix P – Greyscale pixel alteration (text information) in simple order dithered images



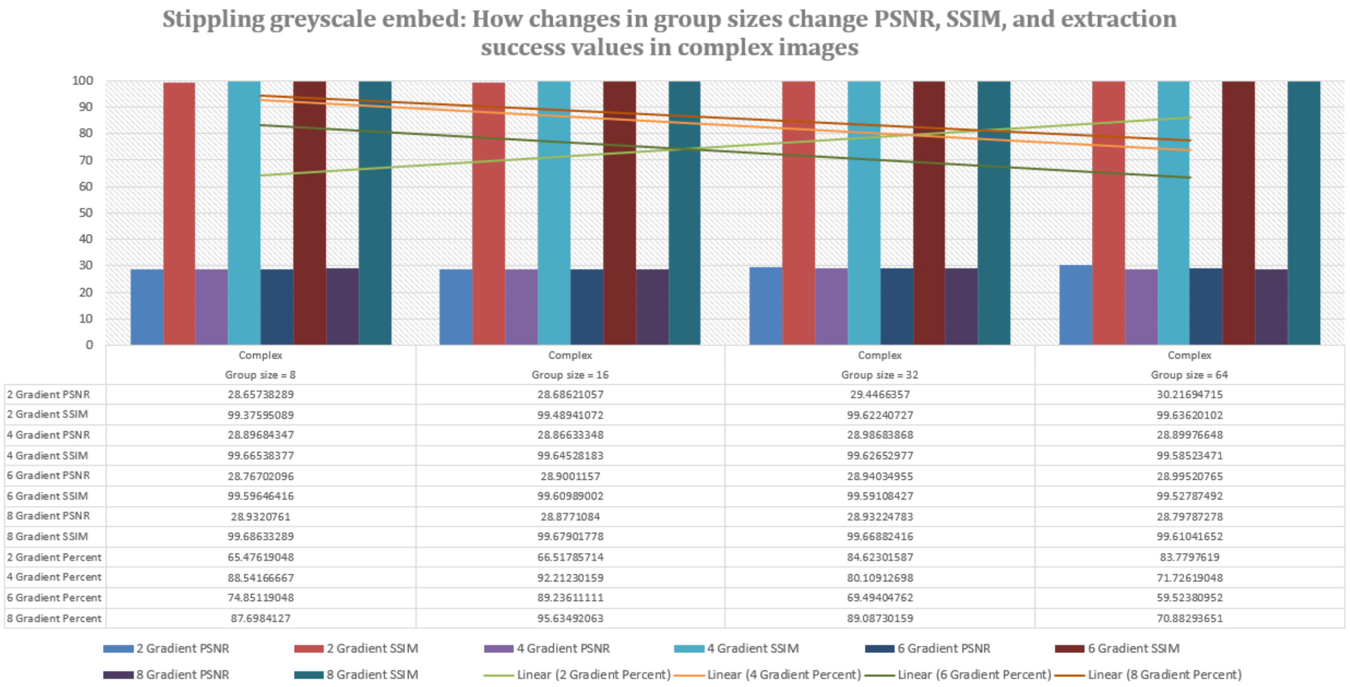
	Complex Group size = 8	Complex Group size = 16	Complex Group size = 32	Complex Group size = 64
2x2 PSNR	29.34126529	29.26175923	29.19221552	29.07042468
2x2 SSIM	99.76804869	99.73584861	99.718591	99.71182406
4x4 PSNR	29.24015807	29.16327429	29.0869975	28.99850297
4x4 SSIM	99.75832739	99.72849651	99.71292111	99.7089489
8x8 PSNR	29.17770967	29.20496373	29.17852539	29.11730327
8x8 SSIM	99.75621586	99.72941375	99.71563681	99.71172977
2x2 Percent	100	100	100	100
4x4 Percent	100	100	100	100
8x8 Percent	100	100	100	100

Appendix Q – Greyscale pixel alteration (text information) in complex order dithered images



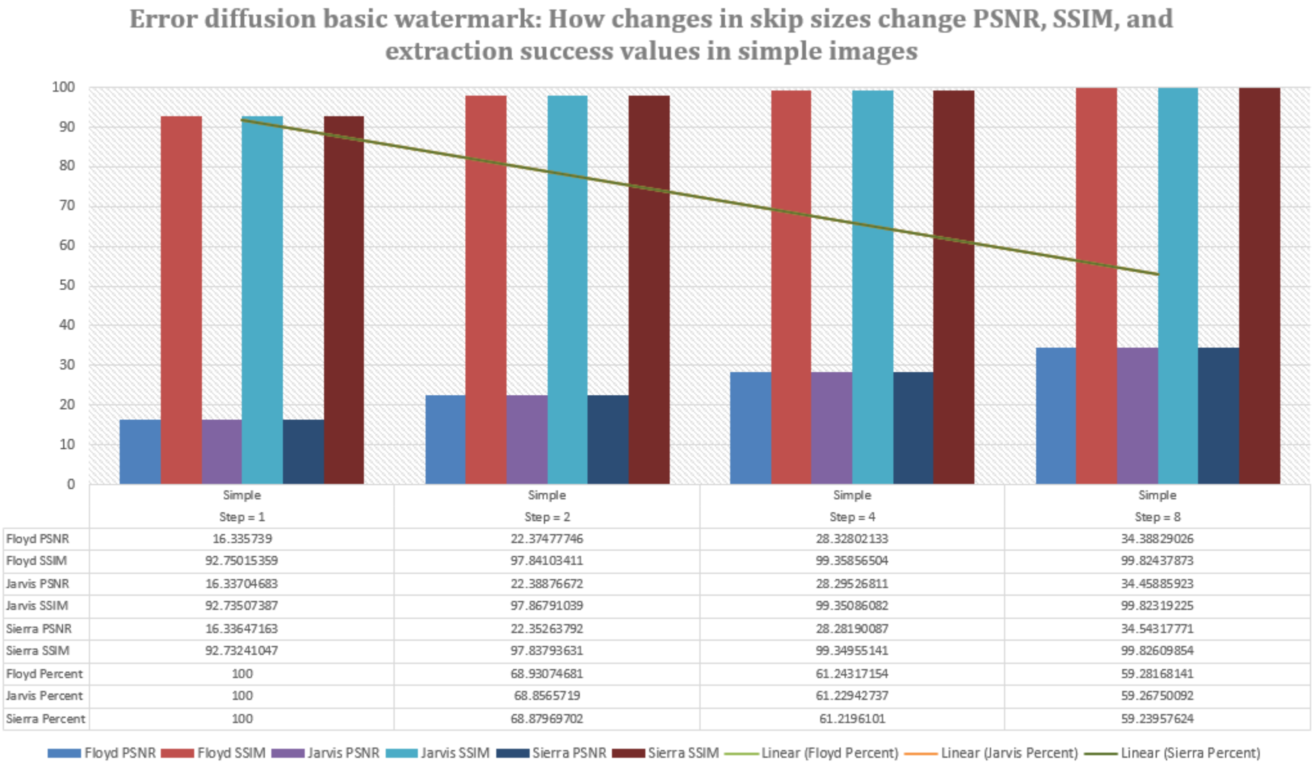
	Simple Group size = 8	Simple Group size = 16	Simple Group size = 32	Simple Group size = 64
2 Gradient PSNR	28.87955338	29.55865954	34.75497363	31.92125588
2 Gradient SSIM	99.21648316	99.39538458	99.66436848	99.62247541
4 Gradient PSNR	29.25146279	29.20839048	29.13121876	29.52066876
4 Gradient SSIM	99.670731	99.65731087	99.59441091	99.5179545
6 Gradient PSNR	29.85554033	30.61637769	31.37390186	31.00519076
6 Gradient SSIM	99.5299557	99.55735509	99.50302328	99.46026649
8 Gradient PSNR	29.39707321	30.02722653	31.87774653	29.21144337
8 Gradient SSIM	99.57910877	99.61524952	99.57232571	99.44430883
2 Gradient Percent	49.75198413	40.92261905	46.52777778	35.06944444
4 Gradient Percent	66.36904762	66.17063492	59.27579365	52.57936508
6 Gradient Percent	51.43849206	57.09325397	41.96428571	42.95634921
8 Gradient Percent	68.65079365	66.96428571	55.95238095	46.03174603

Appendix R – Greyscale pixel alteration (text information) in simple stippled images



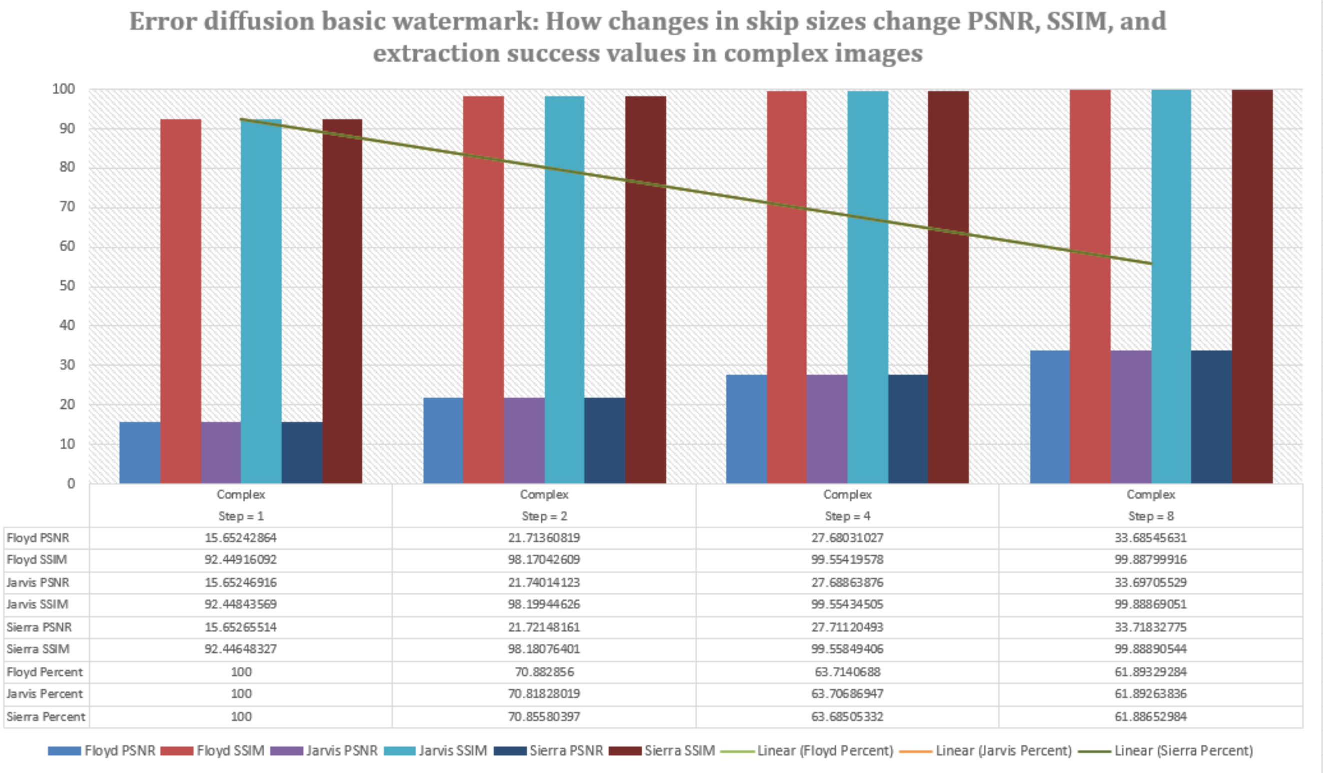
	Complex Group size = 8	Complex Group size = 16	Complex Group size = 32	Complex Group size = 64
2 Gradient PSNR	28.65738289	28.68621057	29.4466357	30.21694715
2 Gradient SSIM	99.37595089	99.48941072	99.62240727	99.63620102
4 Gradient PSNR	28.89684347	28.86633348	28.98683868	28.89976648
4 Gradient SSIM	99.66538377	99.64528183	99.62652977	99.58523471
6 Gradient PSNR	28.76702096	28.9001157	28.94034955	28.99520765
6 Gradient SSIM	99.59646416	99.60989002	99.59108427	99.52787492
8 Gradient PSNR	28.9320761	28.8771084	28.93224783	28.79787278
8 Gradient SSIM	99.68633289	99.67901778	99.66882416	99.61041652
2 Gradient Percent	65.47619048	66.51785714	84.62301587	83.7797619
4 Gradient Percent	88.54166667	92.21230159	80.10912698	71.72619048
6 Gradient Percent	74.85119048	89.23611111	69.49404762	59.52380952
8 Gradient Percent	87.6984127	95.63492063	89.08730159	70.88293651

Appendix S – Greyscale pixel alteration (text information) in complex stippled images



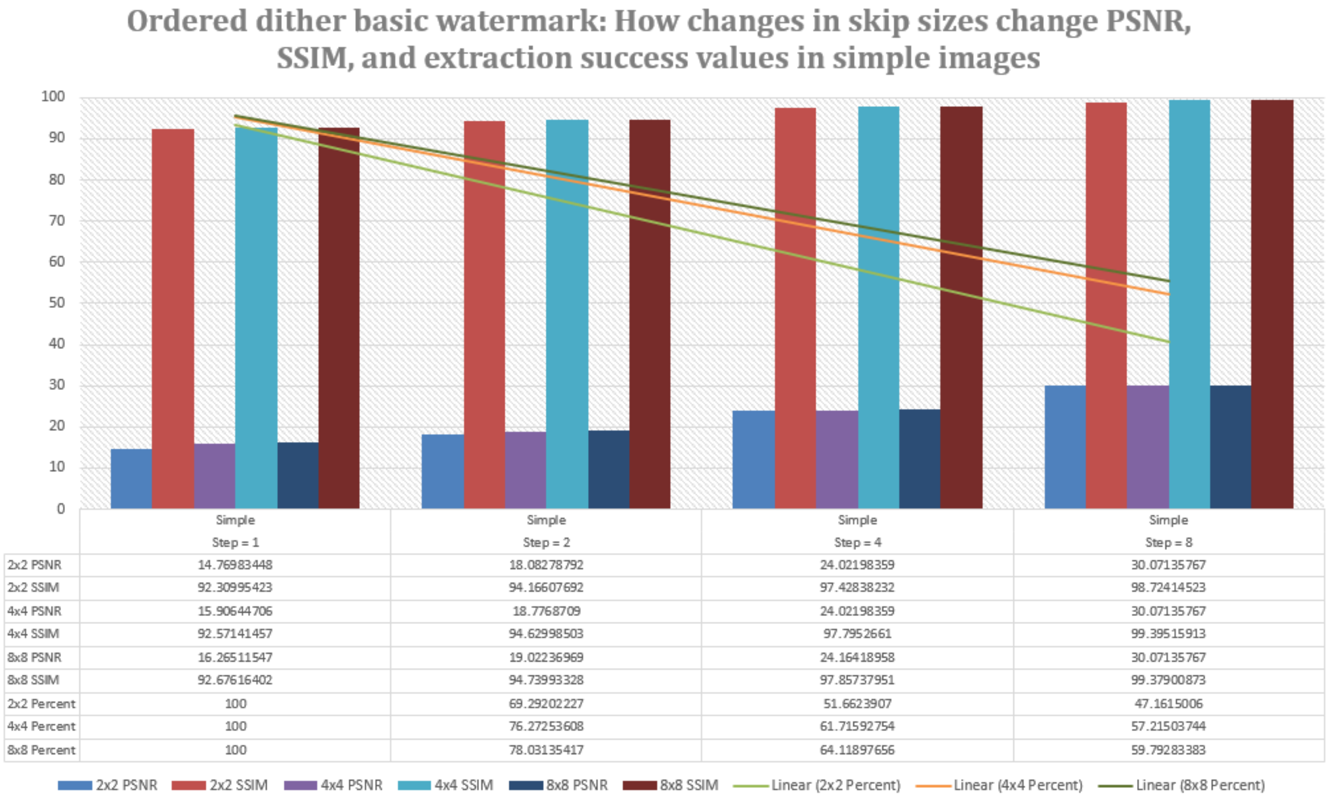
	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
Floyd PSNR	16.335739	22.37477746	28.32802133	34.38829026
Floyd SSIM	92.75015359	97.84103411	99.35856504	99.82437873
Jarvis PSNR	16.33704683	22.38876672	28.29526811	34.45885923
Jarvis SSIM	92.73507387	97.86791039	99.35086082	99.82319225
Sierra PSNR	16.33647163	22.35263792	28.28190087	34.54317771
Sierra SSIM	92.73241047	97.83793631	99.34955141	99.82609854
Floyd Percent	100	68.93074681	61.24317154	59.28168141
Jarvis Percent	100	68.8565719	61.22942737	59.26750092
Sierra Percent	100	68.87969702	61.2196101	59.23957624

Appendix T – Basic pixel embedding (image information) in simple error diffused images



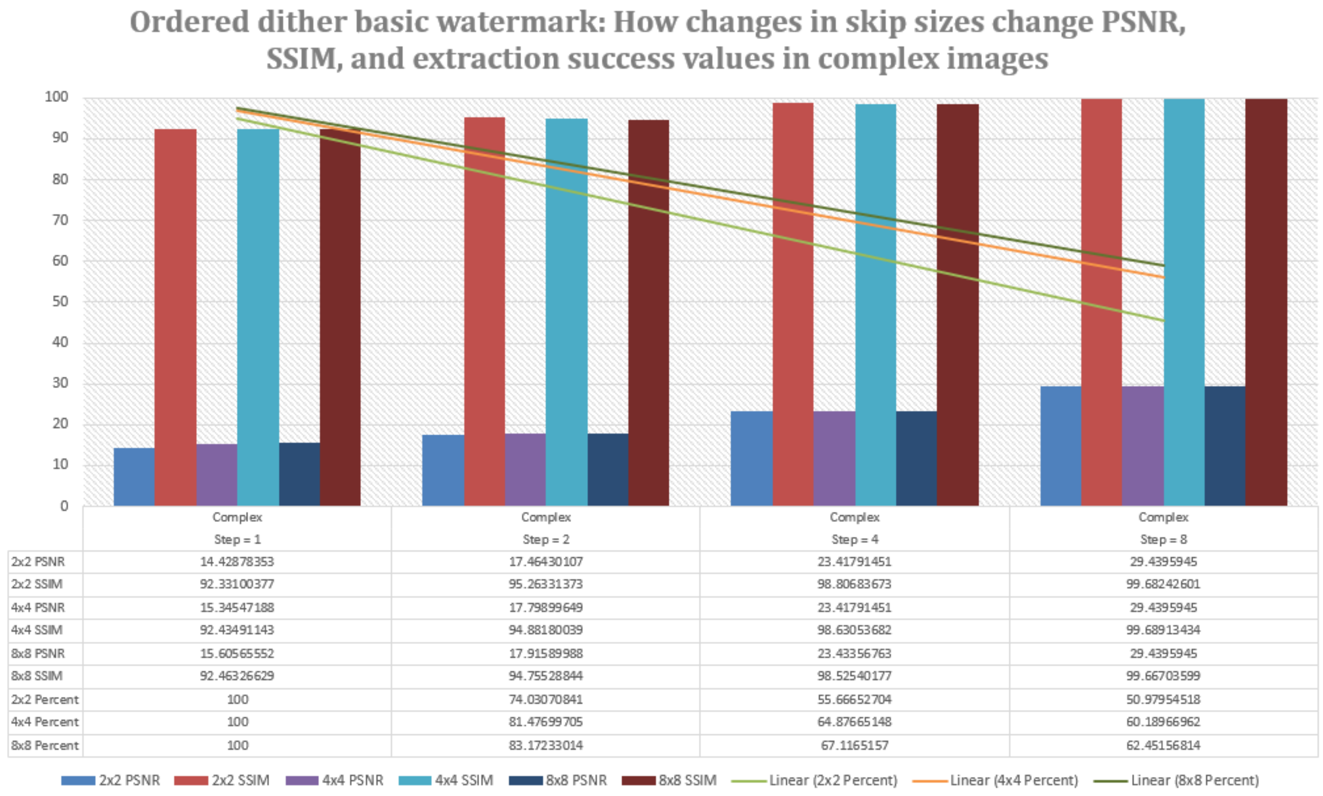
	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
Floyd PSNR	15.65242864	21.71360819	27.68031027	33.68545631
Floyd SSIM	92.44916092	98.17042609	99.55419578	99.88799916
Jarvis PSNR	15.65246916	21.74014123	27.68863876	33.69705529
Jarvis SSIM	92.44843569	98.19944626	99.55434505	99.88869051
Sierra PSNR	15.65265514	21.72148161	27.71120493	33.71832775
Sierra SSIM	92.44648327	98.18076401	99.55849406	99.88890544
Floyd Percent	100	70.882856	63.7140688	61.89329284
Jarvis Percent	100	70.81828019	63.70686947	61.89263836
Sierra Percent	100	70.85580397	63.68505332	61.88652984

Appendix U – Basic pixel embedding (image information) in complex error diffused images



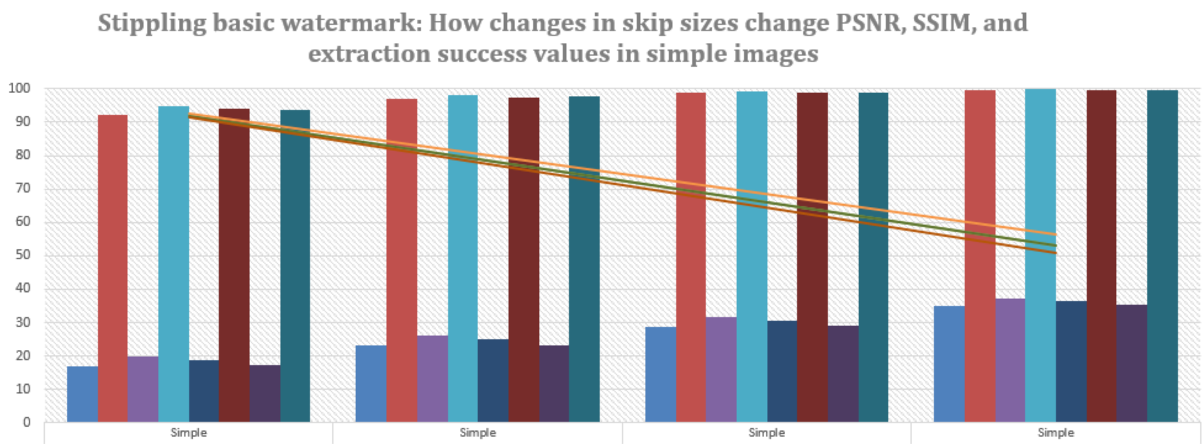
	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
2x2 PSNR	14.76983448	18.08278792	24.02198359	30.07135767
2x2 SSIM	92.30995423	94.16607692	97.42838232	98.72414523
4x4 PSNR	15.90644706	18.7768709	24.02198359	30.07135767
4x4 SSIM	92.57141457	94.62998503	97.7952661	99.39515913
8x8 PSNR	16.26511547	19.02236969	24.16418958	30.07135767
8x8 SSIM	92.67616402	94.73993328	97.85737951	99.37900873
2x2 Percent	100	69.29202227	51.6623907	47.1615006
4x4 Percent	100	76.27253608	61.71592754	57.21503744
8x8 Percent	100	78.03135417	64.11897656	59.79283383

Appendix V – Basic pixel embedding (image information) in simple order dithered images



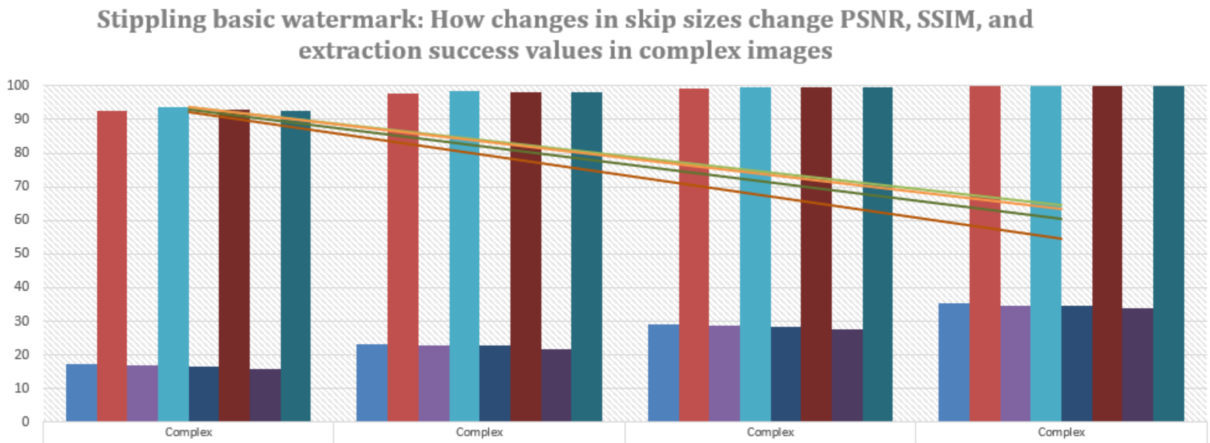
	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
2x2 PSNR	14.42878353	17.46430107	23.41791451	29.4395945
2x2 SSIM	92.33100377	95.26331373	98.80683673	99.68242601
4x4 PSNR	15.34547188	17.79899649	23.41791451	29.4395945
4x4 SSIM	92.43491143	94.88180039	98.63053682	99.68913434
8x8 PSNR	15.60565552	17.91589988	23.43356763	29.4395945
8x8 SSIM	92.46326629	94.75528844	98.52540177	99.66703599
2x2 Percent	100	74.03070841	55.66652704	50.97954518
4x4 Percent	100	81.47699705	64.87665148	60.18966962
8x8 Percent	100	83.17233014	67.1165157	62.45156814

Appendix W – Basic pixel embedding (image information) in complex order dithered images



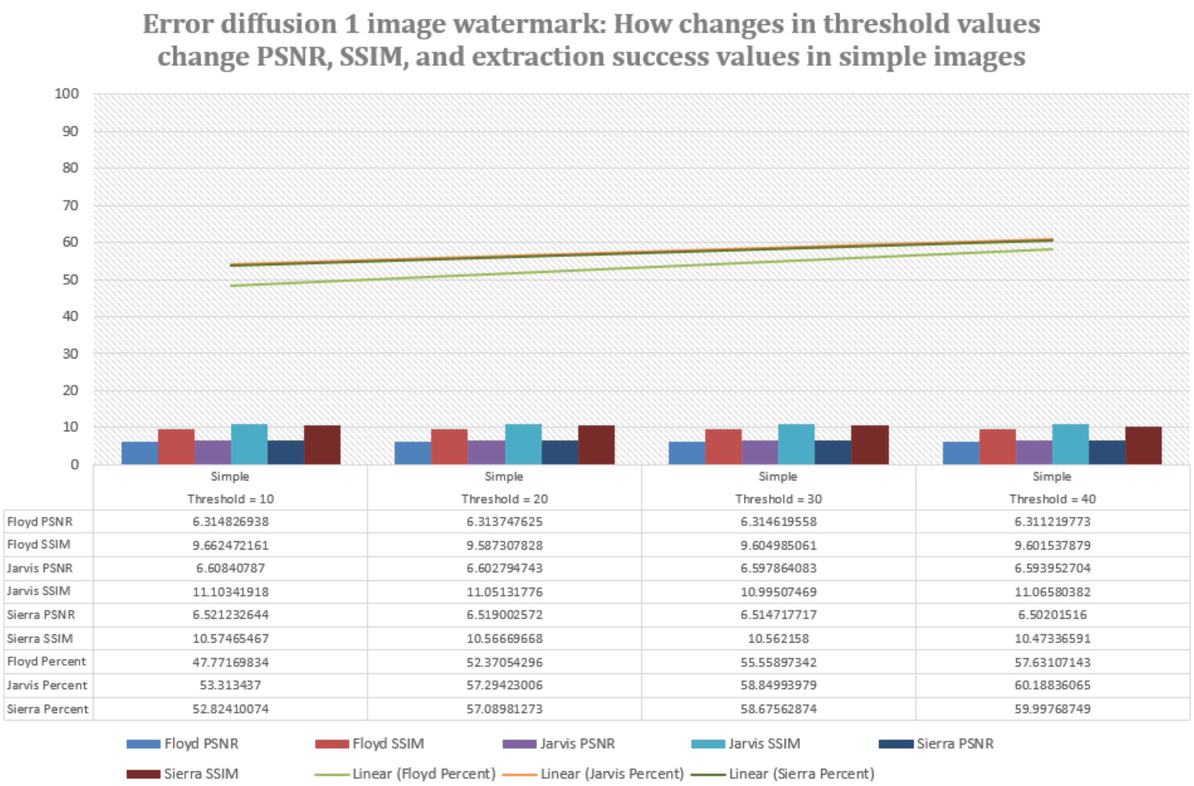
	Simple Step = 1	Simple Step = 2	Simple Step = 4	Simple Step = 8
2 Gradient PSNR	16.94030457	23.08230678	28.89388358	34.97805961
2 Gradient SSIM	92.31129919	97.1510982	98.93123805	99.67836728
4 Gradient PSNR	20.02987796	26.23066671	31.63202145	37.13372381
4 Gradient SSIM	94.74240484	98.08673554	99.31325613	99.79240023
6 Gradient PSNR	18.79034829	24.95220819	30.55275622	36.36466243
6 Gradient SSIM	93.91632607	97.43199182	98.83894937	99.57104362
8 Gradient PSNR	17.23806121	23.33233383	29.17668223	35.26666626
8 Gradient SSIM	93.68619417	97.55937145	98.95083588	99.61894801
2 Gradient Percent	100	68.99728607	61.46111489	59.52035011
4 Gradient Percent	100	71.17999197	64.1102501	62.28707437
6 Gradient Percent	100	68.87380666	61.27916819	59.32138681
8 Gradient Percent	100	67.57836361	59.65539208	57.60750999

Appendix X – Basic pixel embedding (image information) in simple stippled images



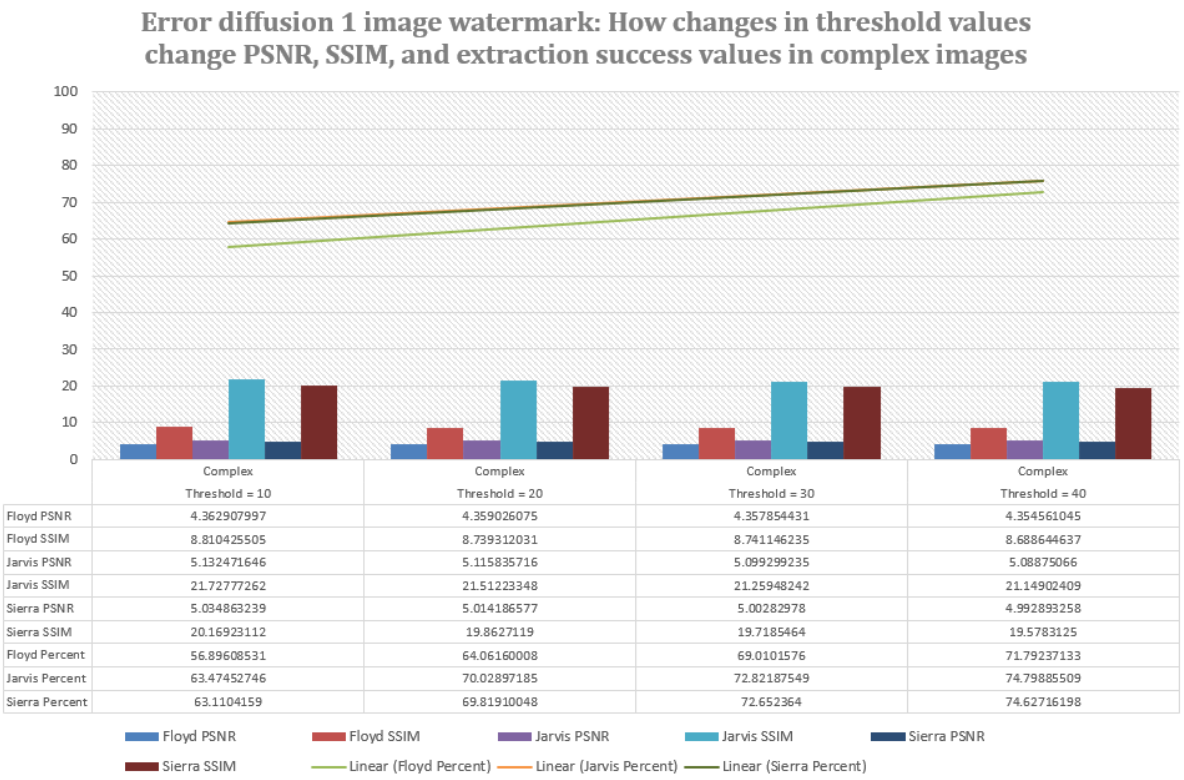
	Complex Step = 1	Complex Step = 2	Complex Step = 4	Complex Step = 8
2 Gradient PSNR	17.23571007	23.36856641	29.16578497	35.24754528
2 Gradient SSIM	92.48294177	97.76296371	99.26740463	99.79533966
4 Gradient PSNR	16.82719196	22.87094399	28.77592958	34.7063787
4 Gradient SSIM	93.81348214	98.42137892	99.57605111	99.8868953
6 Gradient PSNR	16.55841592	22.66100737	28.51695043	34.56025986
6 Gradient SSIM	93.03413784	98.02001635	99.39282134	99.8332508
8 Gradient PSNR	15.73285705	21.8198382	27.72051095	33.78840133
8 Gradient SSIM	92.71122461	98.05110427	99.45755988	99.85947027
2 Gradient Percent	100	76.49549715	70.84511405	69.36139763
4 Gradient Percent	100	75.77578233	69.80404733	68.26644501
6 Gradient Percent	100	73.72135539	67.35562071	65.6938845
8 Gradient Percent	100	70.00606489	62.70550814	60.80772117

Appendix Y – Basic pixel embedding (image information) in complex stippled images



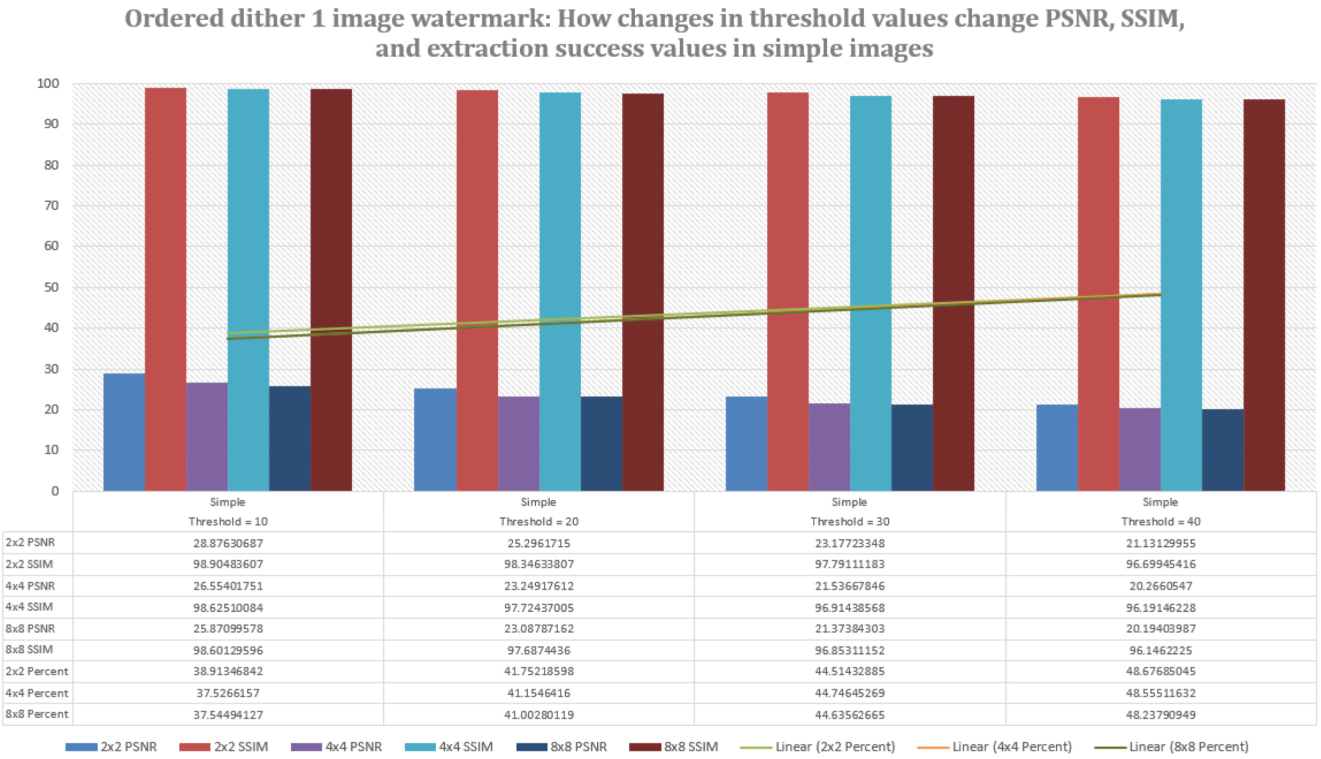
	Simple Threshold = 10	Simple Threshold = 20	Simple Threshold = 30	Simple Threshold = 40
Floyd PSNR	6.314826938	6.313747625	6.314619558	6.311219773
Floyd SSIM	9.662472161	9.587307828	9.604985061	9.601537879
Jarvis PSNR	6.60840787	6.602794743	6.597864083	6.593952704
Jarvis SSIM	11.10341918	11.05131776	10.99507469	11.06580382
Sierra PSNR	6.521232644	6.519002572	6.514717717	6.50201516
Sierra SSIM	10.57465467	10.56669668	10.562158	10.47336591
Floyd Percent	47.77169834	52.37054296	55.55897342	57.63107143
Jarvis Percent	53.313437	57.29423006	58.84993979	60.18836065
Sierra Percent	52.82410074	57.08981273	58.67562874	59.99768749

Appendix Z – Self-conjugate watermarking (image information) in simple error diffused images



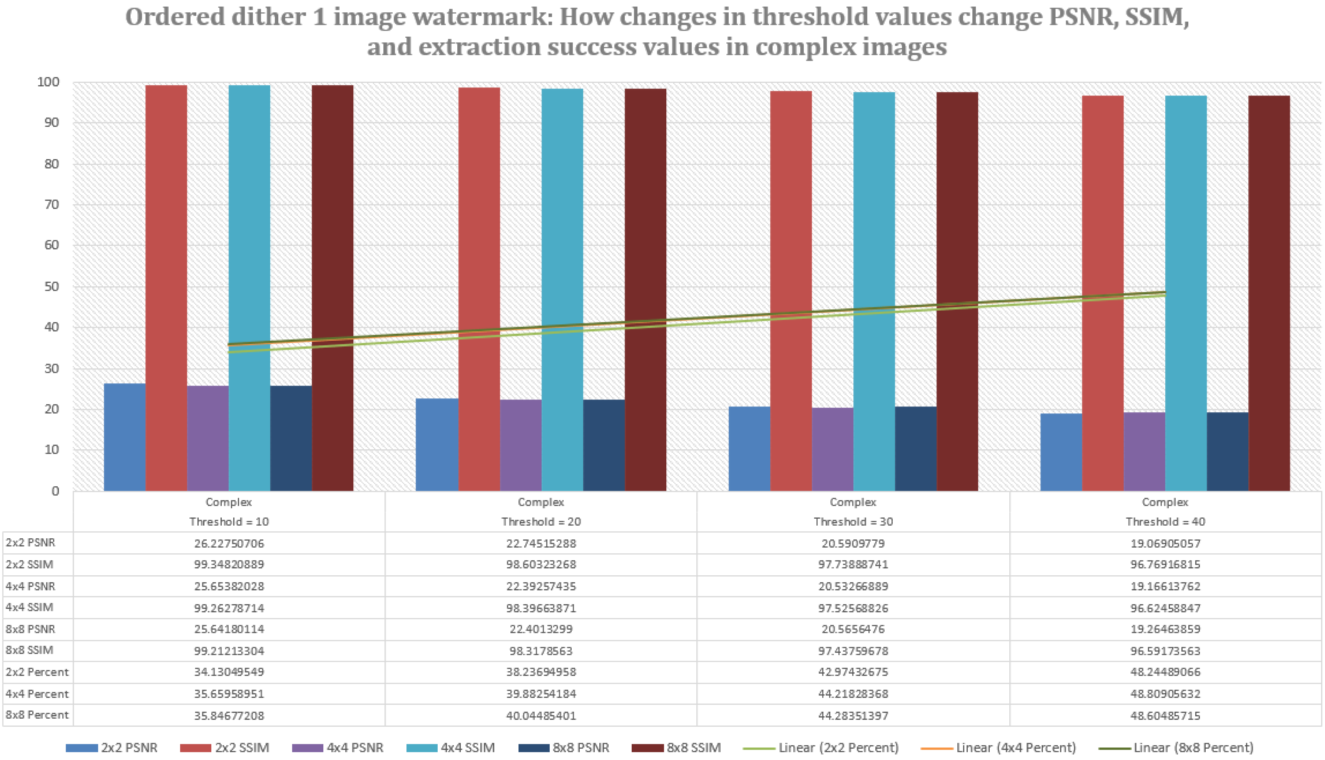
	Complex Threshold = 10	Complex Threshold = 20	Complex Threshold = 30	Complex Threshold = 40
Floyd PSNR	4.362907997	4.359026075	4.357854431	4.354561045
Floyd SSIM	8.810425505	8.739312031	8.741146235	8.688644637
Jarvis PSNR	5.132471646	5.115835716	5.099299235	5.08875066
Jarvis SSIM	21.72777262	21.51223348	21.25948242	21.14902409
Sierra PSNR	5.034863239	5.014186577	5.00282978	4.992893258
Sierra SSIM	20.16923112	19.8627119	19.7185464	19.5783125
Floyd Percent	56.89608531	64.06160008	69.0101576	71.79237133
Jarvis Percent	63.47452746	70.02897185	72.82187549	74.79885509
Sierra Percent	63.1104159	69.81910048	72.652364	74.62716198

Appendix AA – Self-conjugate watermarking (image information) in complex error diffused images



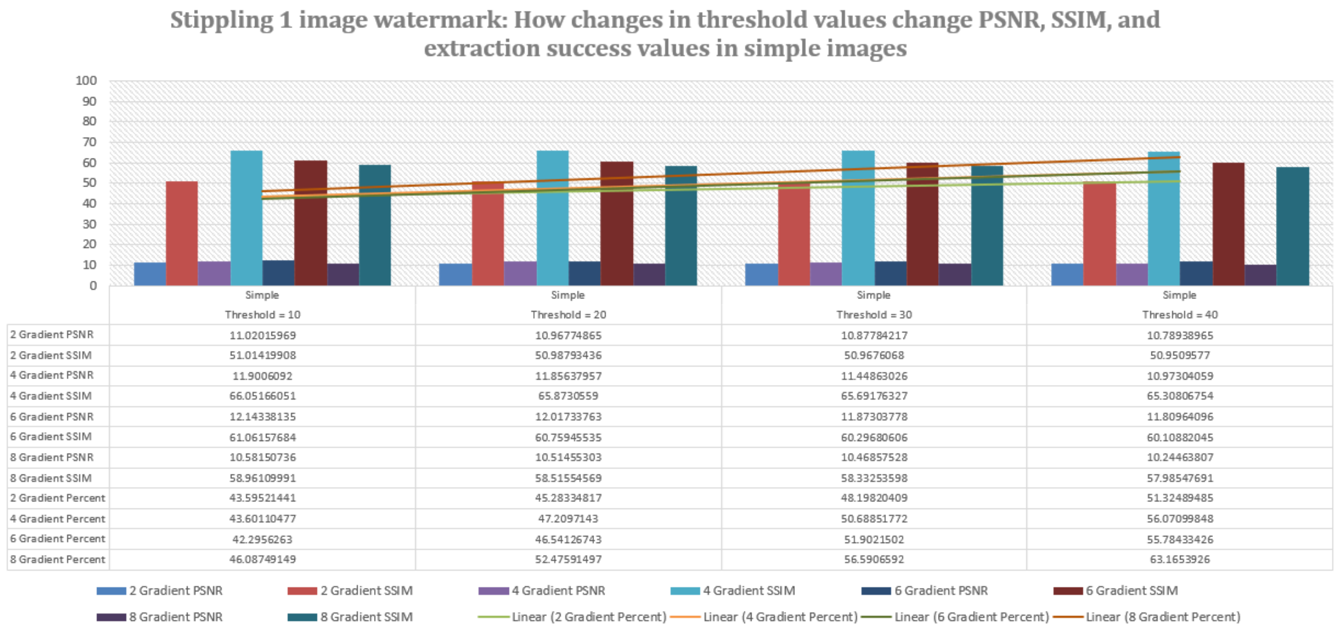
	Simple Threshold = 10	Simple Threshold = 20	Simple Threshold = 30	Simple Threshold = 40
2x2 PSNR	28.87630687	25.2961715	23.17723348	21.13129955
2x2 SSIM	98.90483607	98.34633807	97.79111183	96.69945416
4x4 PSNR	26.55401751	23.24917612	21.53667846	20.2660547
4x4 SSIM	98.62510084	97.72437005	96.91438568	96.19146228
8x8 PSNR	25.87099578	23.08787162	21.37384303	20.19403987
8x8 SSIM	98.60129596	97.6874436	96.85311152	96.1462225
2x2 Percent	38.91346842	41.75218598	44.51432885	48.67685045
4x4 Percent	37.5266157	41.1546416	44.74645269	48.55511632
8x8 Percent	37.54494127	41.00280119	44.63562665	48.23790949

Appendix AB – Self-conjugate watermarking (image information) in simple order dithered images



	Complex Threshold = 10	Complex Threshold = 20	Complex Threshold = 30	Complex Threshold = 40
2x2 PSNR	26.22750706	22.74515288	20.5909779	19.06905057
2x2 SSIM	99.34820889	98.60323268	97.73888741	96.76916815
4x4 PSNR	25.65382028	22.39257435	20.53266889	19.16613762
4x4 SSIM	99.26278714	98.39663871	97.52568826	96.62458847
8x8 PSNR	25.64180114	22.4013299	20.5656476	19.26463859
8x8 SSIM	99.21213304	98.3178563	97.43759678	96.59173563
2x2 Percent	34.13049549	38.23694958	42.97432675	48.24489066
4x4 Percent	35.65958951	39.88254184	44.21828368	48.80905632
8x8 Percent	35.84677208	40.04485401	44.28351397	48.60485715

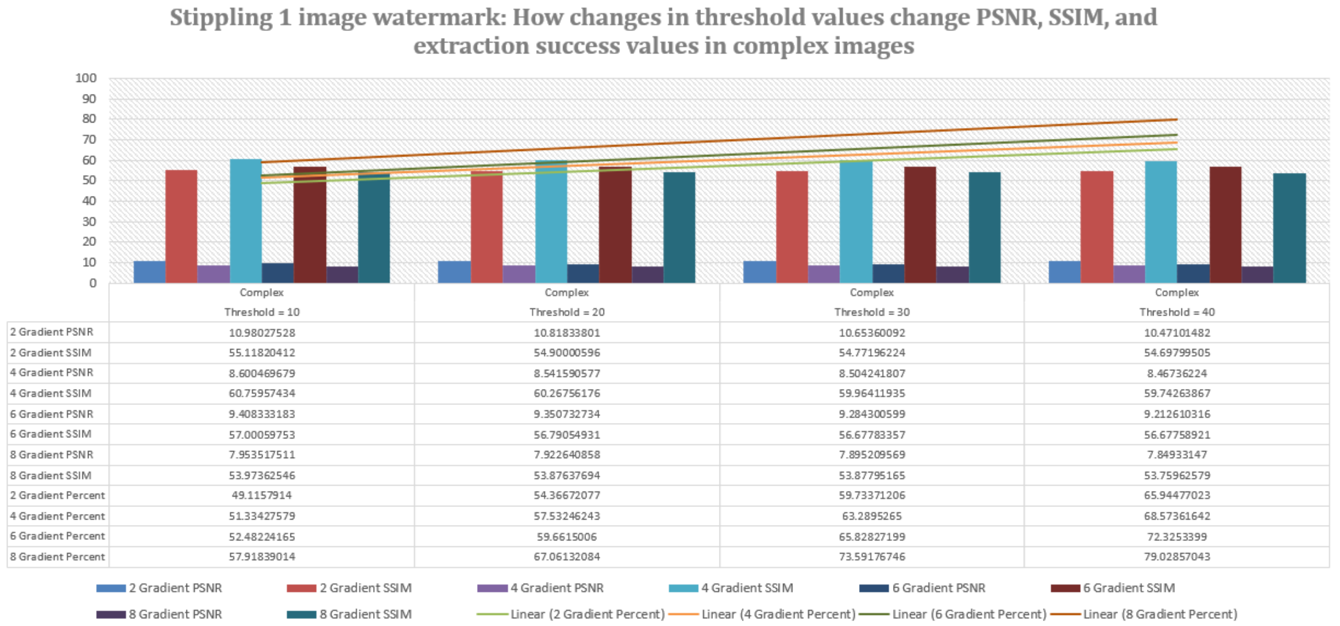
Appendix AC – Self-conjugate watermarking (image information) in complex order dithered images



Stippling 1 image watermark: How changes in threshold values change PSNR, SSIM, and extraction success values in simple images

	Simple Threshold = 10	Simple Threshold = 20	Simple Threshold = 30	Simple Threshold = 40
2 Gradient PSNR	11.02015969	10.96774865	10.87784217	10.78938965
2 Gradient SSIM	51.01419908	50.98793436	50.9676068	50.9509577
4 Gradient PSNR	11.9006092	11.85637957	11.44863026	10.97304059
4 Gradient SSIM	66.05166051	65.8730559	65.69176327	65.30806754
6 Gradient PSNR	12.14338135	12.01733763	11.87303778	11.80964096
6 Gradient SSIM	61.06157684	60.75945535	60.29680606	60.10882045
8 Gradient PSNR	10.58150736	10.51455303	10.46857528	10.24463807
8 Gradient SSIM	58.96109991	58.51554569	58.33253598	57.98547691
2 Gradient Percent	43.59521441	45.28334817	48.19820409	51.32489485
4 Gradient Percent	43.60110477	47.2097143	50.68851772	56.07099848
6 Gradient Percent	42.2956263	46.54126743	51.9021502	55.78433426
8 Gradient Percent	46.08749149	52.47591497	56.5906592	63.1653926

Appendix AD – Self-conjugate watermarking (image information) in simple stippled images



Stippling 1 image watermark: How changes in threshold values change PSNR, SSIM, and extraction success values in complex images

	Complex Threshold = 10	Complex Threshold = 20	Complex Threshold = 30	Complex Threshold = 40
2 Gradient PSNR	10.98027528	10.81833801	10.65360092	10.47101482
2 Gradient SSIM	55.11820412	54.90000596	54.77196224	54.69799505
4 Gradient PSNR	8.600469679	8.541590577	8.504241807	8.46736224
4 Gradient SSIM	60.75957434	60.26756176	59.96411935	59.74263867
6 Gradient PSNR	9.408333183	9.350732734	9.284300599	9.212610316
6 Gradient SSIM	57.00059753	56.79054931	56.67783357	56.67758921
8 Gradient PSNR	7.953517511	7.922640858	7.895209569	7.84933147
8 Gradient SSIM	53.97362546	53.87637694	53.87795165	53.75962579
2 Gradient Percent	49.1157914	54.36672077	59.73371206	65.94477023
4 Gradient Percent	51.33427579	57.53246243	63.2895265	68.57361642
6 Gradient Percent	52.48224165	59.6615006	65.82827199	72.3253399
8 Gradient Percent	57.91839014	67.06132084	73.59176746	79.02857043

Appendix AE – Self-conjugate watermarking (image information) in complex stippled images

10. References

Binascii. *A Python library to convert between binary and ascii character formats*. [Online]

Available at: https://docs.python.org/3/library/binascii.html

Blidh, H., 2016. *Error diffusion using maps*. [Online]

Available at: https://github.com/hbldh/hitherdither/tree/master/hitherdither

Bloisi, D., Iocchi, L., 2007. *Image based steganography and cryptography*. [Online]

Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.1140&rep=rep1&type=pdf

Chung, K-L., Pei, S-C., Pan, Y-L., Hsu, W-L., Huang, Y-H., Yang, W-N., Chen, C-H., 2011. *A gradient-based adaptive error diffusion method with edge enhancement*. *Figure 1*. [Digital Image]

Available at: https://www.sciencedirect.com/science/article/pii/S0957417410007074?via%3Dihub

Cox, I., Miller, M., Bloom, J., Fridrich, J., Kalker, T., 2007. *Digital Watermarking and Steganography.* [Online]

Available at: https://books.google.co.uk/books?id=JZQLpzihtecC&dq=watermarking+&lr=&source=gbs_navlinks_s

Deussen, O., Isenberg, T. *Chapter 3: Halftoning and Stippling*. [Book]

Available at: https://link.springer.com/chapter/10.1007/978-1-4471-4519-6_3

Fridrich, J., 2009. *Steganography in digital media*. Cambridge: University Press. [Book]

Guo, J.M., Liu, Y.F., 2011. *Halftone-image security improving overall minimal-error searching.* [Online]

Available at: https://ieeexplore.ieee.org/abstract/document/5738334

Hellend, T., 2012. *Image dithering*. [Online]

Available at: https://tannerhelland.com/2012/12/28/dithering-eleven-algorithms-source-code.html

Horé, A., Ziou, D., 2010. *Image quality metrics: PSNR vs. SSIM.* [Online]

Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5596999

Huttler, F., Kuttler, M., 1999. *Multimedia Watermarking Techniques*. [Online]

Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=771066

Johnson, N.F., Duric, Z., Jajodia, S., 2000. *Information Hiding: Steganography and Watermarking – Attacks and Countermeasures*. London: Kluwer Academic Publishers. [Book]

Johnson, N.F., Jaiodia, S., 1998. *Exploring steganography: Seeing the unseen*. [Online]

Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4655281

Katzenbeisser, S., Petitcolas, F.A.P., 2000. *Information Hiding techniques for steganography and digital watermarking.* London: Artech House Publishers [Book]

Khan, D., 1996. *The Codebreakers,* second edition. New York: Macmillan. [Book]

Kim, D., Son, M., Lee, Y., Kang, H., Lee, S., 2009. *Feature-guided Image Stippling*. [Online]

Available at: https://onlinelibrary.wiley.com/doi/epdf/10.1111/j.1467-8659.2008.01259.x

Kim, S.Y., Maciejewki, R., Isenberg, T., Andrews, W.M., Chen, W., Sousa, M.C., Ebert, D.S., 2009. *Stippling by example*. [Online]

Available at: https://dl.acm.org/doi/abs/10.1145/1572614.1572622

Kise, K., Sato, A., Iwata, M., 1998. *Segmentation of Page Images Using the Area Voronoi Diagram.* [Online]

Available at: https://www.sciencedirect.com/science/article/pii/S1077314298906841

Knuth, D.E.,1987. *Digital Halftones by Dot Diffusion.* [Online]

Available at: https://dl.acm.org/doi/abs/10.1145/35039.35040

Krenn, J.R., 2004. *Steganography and Steganalysis.* [Online]

Available at: http://www.retawprojects.com/uploads/steganalysis.pdf

Lien, B.K., Pei, W.D., 2009. *Reversible data hiding for ordered dithered halftone images*. [Online]

Available at: https://ieeexplore.ieee.org/abstract/document/5413515

Lu, W., Xue, Y., Yeung, Y., Huang, J., Shi, Y-Q., Fellow, L., 2019. *Secure Halftone Image Steganography Based on Pixel Density Transition*. [Online]

Available at: https://ieeexplore.ieee.org/abstract/document/8789545

Mahdavi, M., Samavi, S., 2015. *Steganography of Halftone Images by Group Alteration of Grayscale Pixels*. [Online]

Available at: http://jcomsec.ui.ac.ir/article_21872.html

Meşe, M., Vaidyanathan, P.P., 2002. *Recent Advances in Digital Halftoning and Inverse Halftoning Methods*. [Online]

Available at: https://ieeexplore.ieee.org/abstract/document/1010034

Nielsen, F., 2005. *Visual computing: Geometry, Graphics, and Vision.* Jennifer Niles [Book]

OpenCV. *A Python Library that includes computer vision algorithms.* [Online]

Available at: https://docs.opencv.org/master/d1/dfb/intro.html

Openpyxl, 2020. *A Python Library to read/write Excel 2010 xlsx/xlsm files*. [Online]

Available at: https://openpyxl.readthedocs.io/en/stable/

Pei, S.C., Guo, J.M., 2006. *High-capacity data hiding in halftone images using minimal-error bit searching and least-mean square filter*. [Online]

Available at: https://ieeexplore.ieee.org/abstract/document/1632219

PIL, Pillow. *A Python Library to read/save images*. [Online]

Available at: https://pillow.readthedocs.io/en/stable/reference/Image.html

Pitas, I., 2000. *Digital Image Processing Algorithms and Applications.* New York: Wiley

Ramani, K., Prasad, E.V., Varadarajan, S., 2007. *Steganography using BPCS to the integer wavelet transformed image.* [Online]

Available at: http://kresttechnology.com/krest-academic-projects/krest-major-projects/ECE/B-Tech%20Papers/50.pdf

Russ, J.C., 1990. *Computer-Assisted Microscopy.* Chapter: *Image Processing*. Boston: Springer [Book]

Sajedi, H., Jamzad, M., 2010. *BSS: Boosted steganography scheme with cover image preprocessing.* [Online]

Available at: https://www.sciencedirect.com/science/article/pii/S0957417410003714

Sankarasrinivasan, S., 2019. *Ordered dithering using Numpy's tile feature for different sized matrices*. [Online]

Available at: https://github.com/SankarSrin/Digital-Halftoning/blob/master/Digital%20Halftoning/Python/halftoning.py

Sara, U., Akter, M., Uddin, M., 2019. *Image Quality Assessment through FSIM, SSIM, MSE and PSNR – A Comparative Study.* [Online]

Available at: https://www.scirp.org/journal/paperinformation.aspx?paperid=90911

Secord, A., 2002. *Weighted Voronoi stippling*. [Online]

Available at: https://dl.acm.org/doi/abs/10.1145/508530.508537

Shell, S.M., 2019. *An introduction to Numpy and Scipy*. [Online]

Available at: https://sites.engineering.ucsb.edu/~shell/che210d/numpy.pdf

Spaulding, K.E., Miller, R.L., Schildkraut, J., 1997. *Methods for generating blue-noise dither matrices for digital halftoning*. [Online]

Available at: https://www.spiedigitallibrary.org/journals/Journal-of-Electronic-Imaging/volume-6/issue-2/0000/Methods-for-generating-blue-noise-dither-matrices-for-digital-halftoning/10.1117/12.266861.full?SSO=1

Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P., 2004. *Image Quality Assessment: From Error Visibility to Structural Similarity.* [Online]

Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1284395

Zhang, Y., 1997. *Adaptive Ordered Dither*. [Online]

Available at: https://www.sciencedirect.com/science/article/pii/S1077316996904141

Zim, H.S, 1948. *Codes and Secret Writing*. New York: William Marrow and Company. [Book]