The Application of Steganography to Fractal Image Compression

Draft Thesis by: PAUL DAVERN. BSc

Supervisor: DR. MICHAEL SCOTT PHD

Submitted to

DUBLIN CITY UNIVERSITY

COMPUTER APPLICATIONS

for the degree of

Doctor of Philosophy

Sept 1997

Declaration: I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Doctor of Philosophy is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible] ID No.: 86701355

Date: 12 September, 1997

Contents

THE APPLICATION OF STEGANOGRAPHY
TO FRACTAL IMAGE COMPRESSION

Abstract

Author: Paul Davern

Steganography is the science of hiding data in otherwise plain text or images.

Fractal image compression is a lossy method for compressing a graphical image. The compression involves finding a representation of an image in terms of a set of fractals. A fractal is a mathematical formula for describing an image or sub-image.

The new and novel steganographic method detailed in this thesis involves producing a new image that is visually and statistically similar to the original image. The produced new image will have steganographic data hidden in it. The method exploits the fact that image data isn't sensitive to noise. The algorithm selects regions of the image that are visually least sensitive to modification. Fractal image compression theory is used to construct fractal transforms which when applied to other regions of the image produce slight modifications in these least sensitive areas. A modified area is visually similar to the original region. The modified area is also statistically similar to surrounding areas. Hiding the steganographic data involves producing a new region in the manner described for each bit of steganographic data.

The security of the method was tested under various conditions. The results show that the method is a very secure means of hiding and retrieving steganographic information. An Investigation was made in a series of techniques which allow steganographic information to be hidden in and retrieved from multimedia documents in a robust manner.

Table of Figures

Chapter

1.

Introduction.

1.1 Introduction

Since the arrival of computers there has been a vast dissemination of information, some of which needs to be kept private, some of which does not. **Plaintext** is the name given to information that is to be put into secret form. The information may be hidden in two basic ways. The methods of **steganography** conceal the very existence of the secret information. The methods of **cryptography** on the other hand, do not conceal the presence of a secret information, but render it unintelligible to outsiders by various transformations of the plaintext. The **cryptologist**[1] is the entity that performs cryptography. The **stegator** is the entity that performs steganography. The **cryptanalyst** is the entity that tries to break the cryptographic code, or find the steganographic information.

Steganography is the science of hiding data in otherwise plain text or images. When steganography is applied to electrical communications it is called **transmission security**. Steganography is a method by which a message can be disguised, by making it appear to be something else. It derives from two Greek words *Stego-* meaning *roof* or *cover,* and *graphy* meaning writing. Stego is the same word used in *stegosaur* or *roof lizard,* because of the large bony plates that decorate its back. *Steganography* means covert writing.

Steganography is an old science but its application in computer based methods is new. The objective of this thesis is the development of a steganographic method, that is based on the fractal image compression of images. Because computer based steganography is a new area; there is little published material in the field. Also as far as we are aware, this is the first time an attempt has been made, to apply fractal image compression to steganography. Therefore to learn basic steganographic strategies and techniques, an investigation was made into the history of steganography. The documentation of this investigation is in section 1.3.

Computer based steganography[2] is usually based on randomness. There is plenty of examples of random data in computer based information. Steganographic data can be hidden into this random information. The merits of a steganographic method are judged on whether or not, the addition of the steganographic data changes the perceived randomness[3].

Image files are good examples of random data. The data in image files represent a set of pixels, which when displayed on a computer system, give an approximation to a particular image. Image files are made up of portions that are visually constant. For example, in a scanned picture of a person, parts of the person's hair may be visually constant. However the pixel values representing these parts, and the boundaries between the parts are not usually constant. These variations can be due to lighting effects in the original picture.

The variations in the visually constant parts of a given image, leads to a random spread of noise over the image. Therefore an image file contains random data. An investigation into currently available methods which exploit this randomness was made as part of the thesis. Some of these steganographic methods exploit the random spread of noise over an image as a means to data hiding. There is very little research into finding visually constant parts of an image and using these for data hiding. The documentation of this investigation is in Chapter 2.

Fractal image compression was chosen as a basis for the new steganographic technique, because it is a lossy method of image compression. Fractal image compression takes as input an image file, and produces an approximation to this image. The objective in fractal image compression is to find parts of an image that match one another. The matching process involves finding a transformation, which maps a portion of the image to a smaller portion. The output of the fractal image compression process is a set of these transformations. A fractal is a set of transformations which when applied to any image always yields the same output image. The output image is known as the attractor of the fractal transformations. Fractal image decompression yields an image which is an approximation to the original. If steganography could be applied during the fractal image compression process, then it would be difficult for a cryptanalyst to decide if the resulting image degradation was due to the compression process or due to data hiding. There are numerous different algorithms and strategies for achieving fractal image compression. The main effort in this thesis was an investigation to find the best steganographic method that could be devised using any one of a multitude of fractal image compression techniques. Fractal image compression is discussed in Chapter 3.

Least Squares Polynomial Approximation is discussed in chapter 4. This is the theory required for the retrieval of the steganographic data in our algorithm.

Chapter 5 discusses the DCT (Discrete Cosine Transform). This is the theory for various techniques that were tried to enhance our algorithm of chapter 6.

The new and novel steganographic method detailed in this thesis, involves producing a new image that is visually and statistically identical to the original image. The produced new image will have steganographic data hidden in it. The method selects parts of the image that are visually least sensitive to modification. Fractal image compression theory is used to identify a series of sub-images (selected range blocks) and a corresponding series of matching sub-images (domain blocks). The matching process produces a domain block and a fractal transform. When this fractal transform is applied once to the domain block it produces a new sub-image that is visually similar to the original range block. The domain blocks are collected into a domain library, and the library is then split into two halves. If the current bit of steganographic data is a 0, we search for a match in the first half of the domain library, otherwise we search the second half. The algorithm selects a range block, and then for each domain block in the selected half of the domain library, determines the components of the fractal transform. When the transform is applied to the domain block, it produces a new range block that is visually like the original range block. The algorithm selects the domain block (if any), that through this process produces a new range block, which is most visually like the original range block. Hiding the steganographic data involves producing a new range block, in the manner described, for each bit of steganographic data. The method is discussed in great detail in chapter 6.

Chapter 7 contains the results. The results section shows the theory of chapter 6 in action and the results show that the theory works. The results presented in this section are drawn from a proof of concept implementation.

Chapter 8 discusses various enhancements to the basic algorithm to increase the security of the method. Various techniques are presented some of which are mutually exclusive.

Chapter 9 discusses an enhancement to the method, which allows it to be used in multimedia document labelling. In multimedia environments the multimedia documents would be subjected to normal image processing operations which do not seriously reduce the quality of the document. Therefore the steganographic data could also be subjected to for example lossy image compression, image format conversions.

Chapter 10 discusses enhancements to the basic algorithm if the priority of the user is for steganographic ability. In this chapter we assume that no lossy imaging techniques, have been applied to the steganographic image.

Chapter 11 presents a series of conclusions drawn from the research.

1.2 Introduction to our algorithm

- The method takes as input an image file and the steganographic data and produces a new image file that contains the steganographic data. The output image file is called the steganographic image file and it is similar to the input image file.

- Fractal image compression[4] techniques were chosen as a basis for the method.
- Fractal image compression operates in the spatial domain.
- The frequency domain has been shown to be most suited for multimedia document labelling. However we have attempted to adapt our method so that it can be used as a multimedia document labelling[5] method.
- Fractal image compression techniques naturally identify parts of the image that are most suited for data hiding.
- The Fractal image compression process splits the image into blocks. Blocks that are visually similar are identified.
- One bit of steganographic data is hidden by
    1. Transforming one similar block into an approximation for another.
    2. Storing the approximation in position of the original block in the steganographic image file.
- The transforming block is random and so the transformed block is also random. The pixels themselves have not changed in randomness. The relationship between pixels in the steganographic image file have changed.
- The idea of a visual key naturally evolved from the method. The image is split into two regions. Blocks in one region are modified as described above. Blocks from the other region are used in the comparison process. The user selects the position of each region in the image by moving bounding boxes over the image. The bounding boxes are positioned on a pixel boundary, thus the user has to remember the co-ordinates of the bounding boxes. When retrieving the steganographic label the user would have to specify the correct two regions.
- Multiple steganographic labels may be hidden in the same image. Giving the ability to store bogus messages.
- The steganographic label can be encrypted before being hidden. The encryption key could be stored in another part of the image.
- As far as we are aware this is the first time an attempt has been made to apply fractal image compression to steganography.
- Because fractal image compression techniques have been applied to moving pictures our steganographic method can also exploit this medium.

1.3 History of Steganography

Steganography is a science which dates back to ancient times. It has been used by ordinary people, spies, rulers, governments, armies etc. down through the ages.

It is the original method of information concealment. Information has been hidden in drawings, paintings, books, newspapers, in speech, in written word, even in postage stamps.

The information in this section is mainly taken from the excellent works by Kahn[6] [7] which documents the history of cryptography from its beginnings.

1.3.1 Ancient times

The Greeks

In a story about **Bellerophon** from **Homer**'s Iliad we have the first reference to secret writing. Bellerophon was a famous warrior and loved by all the women in his land. Queen Anteia told her husband **Proetus** that Bellerophon tried to ravish her. She did this because Bellerophon rejected her advances. King Proetus wanted to put Bellerophon to death but it was a thing that he dared not do. So Proetus sent Bellerophon carrying a message of introduction to Proetus's Father-in-law the Lycian King. The message was written on stone and it contained some allusion to the manner in which the King had treated Glaucus, naming someone whom the king had assassinated. The message was such that Bellerophon did not know its real meaning even though he could read it. The King forced Bellerophon to perform numerous dangerous tasks in the hope that he would be killed. Bellerophon successfully completed all the tasks given him so eventually the King relented. The King realising that Bellerophon stood under the divine protection of the gods and gave him his daughter and half his kingdom.

Several stories in the Histories of **Herodotus** deal specifically with methods of steganography. One story relates that a message from Harpagus was placed in the belly of a hare. The hare was carried past the unsuspecting road guards of the king of Medes to Cyrus, king of Persia. The message helped Cyrus defeat the king of Medes in battle.

According to Herodotus one of the most important messages in the history of Western civilisation was transmitted secretly. It gave the Greeks the crucial information that Persia was planning to conquer them.

The Greek **Aeneas the Tactician** wrote a entire chapter on communications security in one of his works on military science. Therein he listed several steganographic systems. One example is that holes representing the letters of the Greek alphabet were bored through a disk and the disk used to convey the secret message. Another which is still in use in the 20th century suggested pricking holes in a book above or below the letters of the secret message.

The Egyptians

The Egyptians used illustrations to conceal messages. The idea being that one party could send the illustration to the other in reasonable confidence that if the messenger was questioned then the illustration would not arouse any interest from his enemies.

Some of the hieroglyphics on the monuments of the Egyptians contain secret writing. The artist would describe the life of his master but would put in some hidden meaning. It became a kind of a contest where a person was recognised as being more important if his life story contained more of this secret information. In order to put in the secret information the writing became so complex that nobody but the writer could understand the story. Therefore the practice began to died out.

The Chinese

The Chinese would often write on thin silk or paper, which they rolled into a ball and covered in wax. A messenger hid the ball somewhere on his person, sometimes by swallowing it. This is a form of steganography.

1.3.2 Middle Ages

Cryptography became very common place in the middle ages. Secret writing was employed by the Catholic Church in its various struggles down the ages and by the major governments of the time. Steganography was normally used in conjunction with cryptography to further hide secret information.

The monk Trithemius

In the late 1400s a monk named Trithemius wrote many excellent books on religion, on the lives of famous people including princes and monks, on the life of saint Maximum etc. In 1499 he wrote a set of books which he called Steganographia meaning *covered writing*. He detailed telepathical methods of message passing. For example one way to send a secret message was to say it over an image of a planetary angel at a moment determined by complicated astrological calculations, wrap the image up with the image of the recipient and say the proper incantations. He told of using a network of angles for thought transference and for gaining a knowledge of all things happening in the world. He got into a lot of trouble with the Church because of these books. But later he wrote six books (in a average of ten days each) on what he called *Polygraphia* because of the multiplicity of ways of writing that it included. It mainly dealt with methods of cryptography.

Venice in the 1500s

In the 1500s cryptanalysis was very important to the Council of Ten who ruled Venice. When cipher dispatches fell into the hands of the Venetians their translation was ordered at once. The Council of Ten held a contest in ciphers in which a Marco Rafael produced a new method for invisible writing.

Girolamo Cardano

Girolamo Cardano was born in 1501 in Milan. He published 131 books and left behind 111 in manuscript. He discussed mathematics, astronomy, physics, chess, gambling, death etc. He also wrote two books on cryptography and he produced the Cardano grille.

The Cardano grille is a sheet of stiff material into which rectangular holes are cut at irregular intervals. The encipherer lays the grille over a sheet of writing paper and writes the secret message through the perforations. The perforations can take a whole word, a letter or a syllable. He then removes the grille and fills in the spaces with an innocuous-sounding cover message. The decipherer simply places his grille on the message he receives and reads the hidden information through the perforations. The problem with this method is that awkwardness in phrasing may give away the fact that a message is being sent. Many countries made use of the Cadano grille in their diplomatic correspondence in the 1500s and 1600s.

Pirrho Musefili

In Florence from 1540 to 1557 Pirrho Musefili solved ciphers for clients. Among his clients was the King of England who sent him a cryptogram that had been found in a sole of a pair of golden shoes.

Giovanni Porta

Giovanni Porta was born in Naples in 1535. He wrote numerous books on human physiology, meteorology, refraction of light, the design of villas and 14 books of prose. He also wrote a collection of 20 books which he called *Magia Naturalis* in which he recorded natural science oddities. One of these books gave numerous recipes for secret ink and for such tricks as writing invisibly on an egg and on human skin. He wrote *so that messengers may be sent, who shall neither know that they carry letters nor can they be found about them.* He also described hiding messages in living creatures, for example by feeding a letter in meat to a dog and then killing him to retrieve it.

Mary Queen of Scots

In the 1570s Mary Queen of Scots was the heir apparent to the throne of England. But Elizabeth was the Queen. Mary was held incommunicado under house arrest in Chartley. But she was able to communicated with the French by enciphering messages and hiding them in beer kegs. But unfortunately for Mary a double agent was giving over the messages to Elizabeth's secret service. Mary was convicted of high treason and was executed along with six young men who were to assassinate Elizabeth.

John Wilkins

In 1641 John Wilkins who was the Bishop of Chester wrote the first book in English on cryptology. He described a system where a message is represented as dots, lines and triangles. Letters are represented as dots. These dots are then connected to form lines and triangles. The encrypted message looks like a drawing or some kind of graph.

The Culpers

In 1779 two American agents Samuel Woodhull and Robert Townsend supplied George Washington with information about the Redcoat occupation of New York. They signed themselves the Culpers and used page and line numbers from books. They used invisible ink extensively. Washington supplied them with the ink getting it from Sir James Jay. Jay was a physician in London and his brother was an American statesman. James Jay had experimented in producing this secret ink for several months before sending the completed product to his brother.

The Culpers customarily wrote their message on a blank sheet of paper, inserting it in a predetermined point in a whole package of the same paper. This method of secret writing was very successful and the Culpers give details of troop movements, what provisions were entering the town, etc. without ever being caught.

Benjamin Thompson

The Redcoats used invisible ink even earlier than the Americans. For example Benjamin Thompson detailed some military plans of the New England patriot forces. He described how the Americans were going to attack Boston. The English were able to prevent this attack. He used gallotannic acid for his ink in a method described by Porta in his *Natural Magick.*

The black chambers

In the late 1700s almost all European countries had a secret service and had what was known as a black chamber. The black chamber consisted of a group of people who open letters of their countrymen as well as foreigners. The correspondence of the newly created American nation was also scrutinised. Most of the American's secret writing was in invisible ink. In 1777 English chemists discovered a number of letters in secret ink which were written by Benjamin Franklin.

1.3.3 Modern times

A spy's success depends on his not being seen or heard. Messages have to be sent in a steganographic manner using open codes, microdots, hollow heels, invisible inks etc. Spies had to be very clever to evade the censorship bureau which played a big part in the life of ordinary people during the World Wars because all mail was scrutinised.

Secret inks

Spies in World War I and II used invisible inks extensively. One system they used was suggested by Aeneas where the letters of newspapers and books were dotted with invisible ink. But with newspapers being carried as third-class mail this was not the fastest method of getting information to where it was going.

In 1940 British censorship in Bermuda stopped a letter because the language seemed forced. There were many more letters from the same author who signed that letters as *Joe K*. After much analysis by chemists and scientists secret-ink writing was found on the back of the typed sheets. This secret writing gave details of troop movements etc.

Other such secret-ink writings were discovered by the same station in Bermuda. A man was executed in Cuba for secretly sending information on the ships that were being loaded in Havana harbour. He used Spanish-language letters that contained the secret writings, but the Spanish had a rather Germanic cast and so was suspect.

Nazi spies employed many measures to frustrate the secret writing tests of the censorship bureau, one was to split a piece of paper, write a secret-ink message on the inner surface then rejoin the halves. With the ink on the inside no reagent applied to the outside could develop it. The technique came to light when one spy used too much ink and the excess soaked through.

Jargon code

In Jargon code an apparently innocuous word stands for the real in a text that has been made look as innocent as possible. Jargon code was used extensively during the World Wars.

In 1917 Captain William Hall was head of the cryptanalytic group of the British navy. This group was known as Room 40 their room number in the Admiralty. Their job was to analyse any German cryptographic information captured by the British.

Trebitsch Lincoln was an embittered former member of Parliament. He sent a lot of military information to the Germans in jargon code. In one of his jargon systems, family names meant ships or ports; in another, various petroleum products stood for them. A message that read *Cable prices five consignments Vaseline, eight paraffin* really meant [At] *Dover* [there are] *five first-class cruisers, eight sea-going destroyers*. Lincoln evaded the British authorities and escaped to New York.

In World War II a series of letters describing the condition and availability of dolls escaped detection as they appeared innocent even though the letters contained jargon code. Suspicion was drawn to them when one was returned from Buenos Aires, marked *Unknown address*, to the lady marked as sender. Not having sent it, the lady brought it to the FBI. *I have just secured a lovely Siamese Temple Dancer* the letter said *and it had been damaged. But now it is repaired and I like it very much.* Other letters contained text like *The broken English dolls will be in the doll hospital for a few months before repairs can be completed. The doll hospital is working day and night.* The FBI determined that the dolls represented warships in jargon code, each kind a specific type of ship. The English dolls referred to above now reads:- *The damaged English warships will be in the shipyard for several months before repairs can be completed. The shipyard is working day and night.* The trail to the spy lead to a Mrs. Dickinson who ran an exclusive doll shop in New York. She was found to have received thousands of dollars from Japanese officials for her illegal activities. Mrs. Dickinson was executed for being spy.

Null cipher and geometrical code

In the null cipher only certain words and letters in text that contains the null are significant for example every fifth word. Letters written that contain the null cipher can sound very strained because of the difficulty in getting the correct order to the words of the secret message. Because of this the null ciphers were not used by spies but by loyal Americans who were trying to beat the censor.

In the geometrical open code the significant words can be placed at locations decided before hand by the sender and receiver for example at the intersections of the lines, near punctuation marks, breaking the flow of handwriting near significant letters. These codes were used extensively throughout the World Wars. The Cardano grille allows the hiding of a message by using geometrical code.

Microdots

The chief difficulty with secret inks was their inability to handle the great volume of information that spies had to transmit. This was solved with what J. Edgar Hoover called the *enemy's masterpiece of espionage* the microdot. The microdot was a circular photograph of an espionage message which was 0.05 inches in diameter. The spy inserted the microdot over a period in a letter and cemented it in place. The Mexican microdot ring microphotographed trade and technical publications that were barred from international channels and sent them to addresses in Europe with as many as twenty microdots in a single letter.

Microdots are very shiny so to conceal them many techniques were used such as gumming them onto the surfaces of envelopes whose shininess camouflaged them. They were hidden in love letters, business communications, under stamps etc.

1.4 Steganography on Computer

Nowadays there is much communication between peoples and organisations through the use of the phone, the fax, computer communications, radio etc. All of this communication should be secure. The person for whom a mail message was intended should be the only person who can, and should read that message.

To achieve this privacy one has to employ the techniques of cryptography and steganography. With cryptography there is always the possibility of cracking the code. With steganography there is the possibility that the existence of secret information in the writing or image could be discovered. By encrypting the secret information first and then concealing its very presence we combine the advantages of both methods.

On computer systems there are many different realms where secrecy is necessary. For example electronic mail, communication across WANs and LANs (eg to database servers from client applications), data storage of sensitive information etc.

If there are vast quantities of information to be kept secret there is no point in trying to hide this information using steganography. The main use of steganography on computer systems is in the sending of short messages that need to be secure.

On computer systems there are a lot of examples of random looking data. For example data on unformated disks, executable files, graphical image files, encrypted data, compressed data, sound and audio files etc. It is this randomness that can be exploited to conceal the presence of a message. If a cryptanalyst is monitoring communications from an organisation he could very well disregard random looking data on the communications line.

If data appears random and adding information into this data does not change the apparent randomness, then we have achieved steganography.

Chapter

2.

Review of Computer based Steganography.

2.1 Available Steganographic Software

In this section we review some software[8] that hides information in graphical images and by other computer based data files. Most of the image based software reviewed uses the same basic technique, that of replacing the least significant bit of pixel colour information with the steganographic data. There are other methods reviewed that use the same basic idea. We are particularly interested in the image based methods.

The data in an image is very random. A given byte of data in an image describes the colour of a pixel or several pixels. This byte can contain any value. That implies randomness.

In a given image there may be sections of it that are of a similar colour. A human viewer of this image would not notice a very slight change in colour from one pixel to the next, for example a very slightly different shade of blue. This different shade of blue is produced by manipulating the least significant bit. This slight variation in colour can only be determined by computer analysis.

2.1.1 MandelSteg by Henry Hastur

MandelSteg stands for Mandelbrot Steganography. It generates a Mandelbrot fractal, injects secret information into it and generates a GIF file for the resulting image.

The package consists of two executables *Mandelsteg* to create the fractal and hide the information. *Gifextract* to extract the secret information from the fractal.

For the lowest security level, the data will simply be stored in the specified bit of each pixel, and a 128-colour palette created such that the pixels look the same regardless of whether there is data stored in them or not. This will be sufficient to survive a cursory examination, but will be obvious to anyone versed in the arts of steganography. In particular replacing the supplied palette for the image with another will show up the data bits hidden in areas of solid colour.

To avoid this problem, one should specify the -ns flag, which will only store data in areas of non-solid colour (note that this can greatly decrease the amount of data that you can store in the image).

If there is not enough space for the data in the image, one can simply increase the size of the image by using the -sz flag, followed by the width and height in pixels. Alternatively, one can select a different area of the mandelbrot set by using -md followed by the start x, start y, width and height, specified with floating-point values.

GIFExtract is the program which is used to extract the specified bitplane from an image and sends the data to stdout. The program defaults to extracting bit 7 of each pixel, but the bit can be specified with the -b command line option, with -ns it will only extract data from non-solid areas, -bp can be used to ignore the first specified number of bytes extracted, and -a to analyse the distribution of zero and one bits in the image.

2.1.2 Steg by the JPEG Group

Steg compresses image files using the JFIF format of the JPEG standard and in the process hides steganographic information into the resulting compressed file. This information can be retrieved through the decompression process.

The JPEG Group have produced two executables *cjpeg* which compresses image files using the JPEG standard and *djpeg* which decompresses a JPEG file. These two executables are used when hiding and revealing the secret information.

JPEG encoding is split into lossy and non-lossy stages. The lossy stages use a discrete cosine transform and a quantisation step to compress the image data; the non-lossy stage then uses Huffmann coding to further compress the image data.

The JPEG encoding procedure divides an image into 8x8 blocks of pixels. These blocks are run through a discrete cosine transform (DCT), and the resulting frequency coefficients are scaled to remove the ones, which a human viewer would not detect under normal conditions. If steganographic data is being loaded into the JPEG image, the loading occurs after this step. The lowest-order bits of all non-zero frequency coefficients are replaced with successive bits from the steganographic source file and these modified coefficients are sent to the Huffmann coder.

The resulting format is designed to make the steganographic data as innocuous as possible. There is no magic cookie at the start giving the format. There is however a length field at the beginning of the data. An EOF tag is infeasible because the EOF tag that one used could be part of the supplied data.

2.1.3 S-tools by Andrew Brown

S-Tools (Steganography Tools) gives one the capability of hiding information within Windows sound files (.WAV).

Sound samples are, by their very nature, inaccurate estimates of the correct value of the sound wave at a particular moment in time. The sound samples in Windows WAV files are stored as either 8 or 16 bit values that eventually get passed to the sound board.

S-Tools spreads the bit-pattern that corresponds to the file that you want to hide across the least significant bits of the sound sample. The values of the sound samples are changed by, at most, one value either way. This will be inaudible to the human ear.

2.1.4 Gzip by Andrew Brown.

Gzip is a lossless file compression utility. Two executables are provided *gzip* and *gunzip* for the compression and decompression of files. Steganography can be obtained by selecting options on the command line of the two executables.

Gzip uses LZ77 which compresses data by storing length/offset pairs that refer back in the uncompressed data stream to previous occurrences of the information being compressed. Gzip considers a length of 3 to be the shortest acceptable length.

If the length is at least 5 then 1 is subtracted, and bit 0 is set to the value of the bit that we need to hide. We have now hidden information in the length without pushing it beyond a valid value.

2.1.5 Stego by Romana Machado

Stego is a tool that hides data in, and retrieves data from, Macintosh PICT format files, without changing the appearance or size of the PICT file.

Stego hides data into the least significant bit (or LSB) of each of the RGB colour values. Stego inserts the data to be hidden in sequential order into the PICT file.

The length of the data file is hidden in the LSB's of the first 32 useable bytes. To disguise this value the second to least significant bits, of the second 32 useable bytes are XORed with the 32 bit file length. Then the XORed file length is hidden into the LSB's of the first 32 useable bytes.

2.1.6 HideSeek by Colin Maroney

The HideSeek package consists of two executables. They allow data hiding and retrieval from GIF files.

There is some required header information that is stored into the gif file. This header information is encrypted and stored into the gif.

Hideseek uses the Least Significant Bit of each pixel to encode characters. It uses dispersion in a random manner to spread the data throughout the GIF.

2.2 Steganography for Identification of Computer based Information

In this section we describe some research areas that are current in Steganography[9]. Most research at the moment in is the area of document marking for identification purposes. The goals of this research is to discourage illicit distribution of documents and information by embedding a identification mark with the document or information. We will discuss some published material in this area that is of interest to this thesis.

2.2.1 Hidden Image Copyright Labelling

These papers[10] [11] [12] describe a method for identifying the ownership and distribution of multimedia information (image or video data) in digital networked environments. The method involves inserting a copyright label into the multimedia information. Video steganography is also addressed in the papers[13] [14].

We will describe the objectives of the authors and an introduction to the method itself. The label as described is steganographic data, and we will refer to it as such in our discussion.

The authors want the copyright label to be a reliable property identification tool so they impose the following requirements on the system:-

- The image must contain the copyright mark.
- The image data is labelled in a manner which allows its distribution to be tracked.
- The code must be visually and statistically camouflaged.

This method assumes that the main purpose of any attack would be to make the embedded label unverifiable. A forger should not be able to produce a valid copyright code. A forger should not be able to identify the label in an image.

The system must also be robust enough to withstand normal image processing activities which do not significantly alter the images appearance or affect its value. For example image compression, transforming to a different format etc.

The method described is called SysCoP (System for Copyright Protection). It is based on the fact that natural images are tolerant of noise. This means that controlled loss of information in an image is hardly noticeable, even if the altered image is compared with the original image. Given these facts the method described is based on the lossy image compression method JPEG[15].

The method involves two phases

- The first phase takes as input the copyright code and produces a random sequence of locations for embedding the code in the image.

- The second phase actually embeds the steganographic data into the image at the specified locations. It does this by

1. Using the location information to determine a block in the image.
2. It compresses the block using JPEG techniques. It then determines where in the compressed block information loss is acceptable[16]. This step produces a quantised signal which represents the image block. The method modifies some of the low order frequency components of the DCT representation of an image block. These modifications are such that they are not noticed by the viewer.
3. It then embeds the steganographic data into the JPEG signal at the determined locations.
4. The quantised data is then decoded; and inversely transformed to produce the labelled image data.

This method uses JPEG to give a representation of an image file, it then embeds the steganographic data into this representation, and then decodes the representation back to a new image file.

JPEG was chosen because:-

- It allows direct access to frequency bands in the image. This allows the label data to be inserted in a manner in which it is visually camouflaged.

- JPEG also identifies high frequency components in the image and allows them to be more coarsely quantised. Hiding the steganographic in these components makes the label robust to normal image processing activities.

2.2.2 Identification Techniques to Discourage Document Copying

In these papers[17] [18] the authors describe a set of novel steganographic methods to secretly embed robust labels into documents. The purpose of the method is for the identification of the copyright holder and original distributor.

We will describe the objectives of the authors and an introduction to the method itself. The authors propose techniques that discourage illicit distribution of documents by embedding each document with a unique codeword.

The method has the objectives of discouraging illicit dissemination of documents distributed by computer network and to make paper copies of documents traceable. The coding methods developed involve altering features of the text in documents. The major goal described in the papers is to design coding methods, that are reliably decodeable (even in the presence of noise) yet are largely indiscernible to the reader.

The authors propose three coding methods and describe one in detail, and present experimental results which show that the identification techniques are highly reliable.

The document marking is achieved by altering the text formatting, or by altering certain characteristics of textual elements (e.g. characters).

2.2.2.1 Line-Shift Coding

This is a method of altering a document by vertically shifting locations of text lines to encode the document uniquely. Each codeword specifies a set of text lines to be moved in the document. This encoding may be applied either to a format file (e.g. PostScript file) or to a bitmap of a page image. Decoding does not require the original document.

2.2.2.2 Word-Shift Coding

This is a coding method that is applied either to a format file or to a bitmap image of a document. This is a method of altering a document by horizontally shifting the locations of words within text lines. The method is only applicable to documents with variable spacing between adjacent words. Because of the variable spacing, decoding requires the original image. This is to give the spacing between words in the original document.

2.2.2.3 Feature Coding

This is a coding method that is applied either to a format file or to a bitmap image of a document. The image is examined for chosen text features, and those features are altered, or not altered, depending on the codeword. Decoding requires the original image, or more specifically, a specification of the change in pixels at a feature. There are many possible choices of text features. For example alter upward vertical endlines. That is the tops of the letters, *b*, *d*, *h* etc. The endlines are altered by extending or shortening their lengths by one (or more) pixels.

2.3 Alternative Methods of Steganography on Computer

There are many other possible alternatives for applying steganography to computer related information. For example data has been hidden into the following:-

- Between the various layers of the OSI network model[19].
- ISDN speech data[20].
- Sound media for example, compact disks etc[21].

We describe one technique we investigated in this section.

A PC disk can contain bad sectors. These sectors are marked as unusable in the FAT (File Allocation Table). The data in an unformated disk is very random and is different from disk to disk. A steganographic application could be developed that would describe some sectors that are readable as bad in the FAT and use these to hold the sensitive information.

The information could be encrypted before insertion. Encrypted data is by its nature random so this would give us two means of security. First steganography is being applied as we have hidden random information in a location that should have arbitrary data. Secondly if the steganography was discovered the encryption process gives another layer of protection.

There are a numerous other ways to implement steganography using floppy disks as a basis. For example another way would be to use the space left on the disk when a file gets deleted. We could use the blocks of a deleted executable file and hide our information in these. Then send the disk to the receiver who reads the secret information out of the deleted file.

Chapter

3.

Introduction to Fractal Image Compression.

In this chapter we give an introduction to fractal image compression. This chapter is based on the work of Yuval Fisher[22]. We will describe the quadtree fractal image compression method which is used as the basis of the data hiding algorithm.

3.1 What is a fractal?

A fractal is an image with infinite detail. A fractal is described by a set of transformations on a set of real numbers.

3.2 What is Fractal Image Compression?

Figure 1 shows an initial image being reproduced to 1/3 its size and translated. Three of these reproduced images are combined to produce a new image. This process is repeated 6 times.

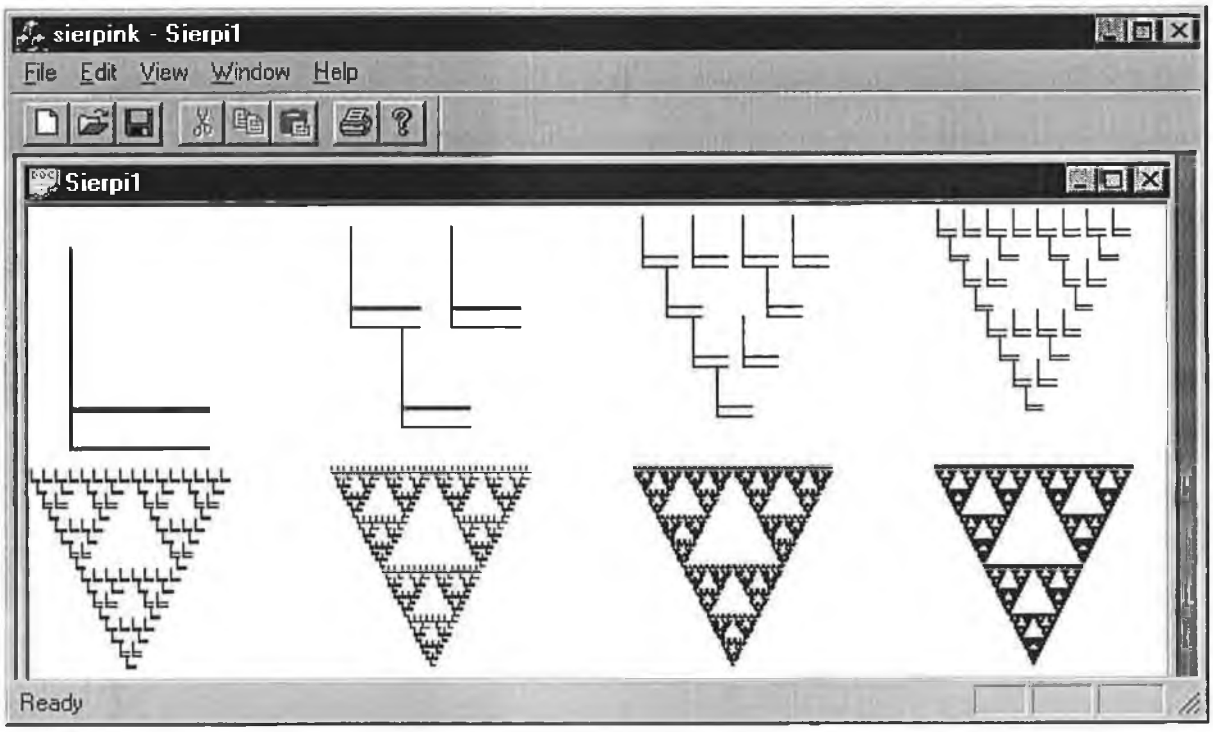


Figure 1. Sierpinki triangle from |_

Figure 2 shows the same process applied to a different start image. The final image in both figures is identical we call this image the *attractor*.

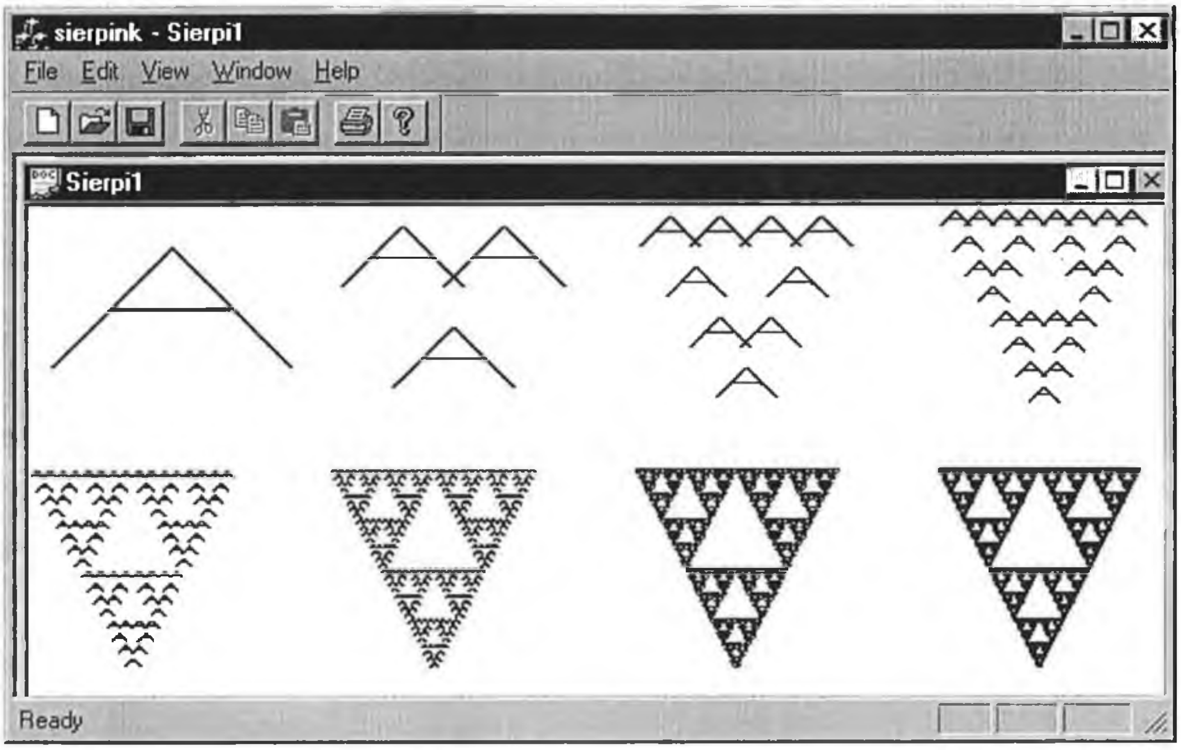


Figure 2. Sierpinki triangle from A

Thus, the initial image does not effect the final attractor. In fact, it is only the position and the orientation of the copies that determines what the final image will look like.

Since it is the way the input image is transformed that determines the final result of this process, we only describe these transformations. Different transformations lead to different attractors, with the technical limitation that the transformations must be contractive. That is, a given transformation when applied to any two points in the input image must bring them closer together in the copy. Except for this condition, the transformations can have any form. In practice, choosing transformations of the form

$$w_i\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}a_i & b_i\\c_i & d_i\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}+\begin{bmatrix}e_i\\f_i\end{bmatrix}$$

is sufficient. Such transformations are called *affine transformations* of a plane. An affine transformation can skew, stretch, rotate, scale and translate an input image.

For example the following set of affine transformations generate the Sierpinki triangle. There are three affine transformations required to generate the triangle[23]. The Sierpinki triangle is the final attractor in figures 1 and 2. The transformations are listed in table 1 as 1,2,3 and the values for a,b,c,d,e,f are given.

W	**a**	**b**	**c**	**d**	**e**	**f**
1	0.5	0	0	0.5	0	0
2	0.5	0	0	0.5	50	0
3	0.5	0	0	0.5	25	50

Table 1. Transformations for Sierpinki triangle

The images in Figure 1 and Figure 2 are generated using the affine transformations in Table 1. A common feature of these and all attractors formed this way is that in the position of each of the images of the original square on the left there is a transformed copy of the whole image. Thus, each image is formed from transformed (and reduced) copies of itself, and hence it must have detail at every scale. That is, the images are fractals. This method of generating fractals is due to John Hutchinson[24], and more information about many ways to generate such fractals can be found in books by Barnsley[25] and Peitgen, Saupe, and Jurgens[26] .

Barnsley[27] suggested that perhaps storing images as collections of transformations could lead to image compression. His argument went as follows: the triangle in figure 2, is generated from only three affine transformation. Each affine transformation is defined by 6 numbers, which would take 768 bits to store on computer. Storing the image of the triangle as a collection of pixels, however, requires much more memory . So if we wish to store a picture of a triangle, then we can do it by storing the numbers that define the affine transformations and simply generate the triangle when ever we want to see it.

Now suppose that we were given any arbitrary image, say a face. If a small number of affine transformations could generate that face, then it too could be stored compactly. Fractal image compression is a method for finding these transformations.

3.2.1 Iterated Function Systems

The process that built figures 1 and 2 is described mathematically by an iterated function system (IFS). An iterated function system consists of a collection of contractive transformations

$$\{w_i : R^2 \to R^2 \mid i = 1,.....,n\}$$

which maps the plane R^2 to itself. The collection of transformations defines a map

$$W(.) = \bigcup_{i=1}^{n} w_i(.)$$

The transformations w_i are in the same form as in table 1. That is they contain position, scaling, stretching skewing and rotation factors. Given an input set *S*, we can compute $w_i(S)$ for each *i*, take the union of these sets, and get a new set *W(S)*. *W* is a map on the space of subsets of the plane. We will call a subset of the plane an image.

When the w_i are contractive in the plane, then *W* is contractive in a space of subsets of the plane. These subsets have to be closed and bounded; that is an image is bound by its dimensions[28].

In order to determine if *W* is contractive we need a means of determining distance between two sets. There is such a means, called the Hausdorff metric, which measures the difference between two closed and bounded subsets of the plane, and in this metric *W* is contractive on the space of closed and bounded subsets of the plane.

The contractive mapping fixed point theorem[29], which tells us that a map W will have a unique fixed point in the space of all images. That is, whatever image (or set) we start with, we can repeatedly apply W to it and we will converge to a fixed image. In other words, given an input image f_0 we can apply *W* once get f_1 = $W(f_0)$, twice to get $f_2 = f(S_1) = f(f(S_0)) \equiv f^{o2}(S_0)$ and so on. The superscript "o" indicates that we are talking about iterations, not exponents. The attractor which is the result of applying this a number of times, is the limit set

$$|W| \equiv S_\infty = \lim_{n\to\infty} W^{on}(S_0)$$

which is not dependent on the choice of S_0.

3.3 Self- Similarity in Images

Given an image, the objective of fractal image compression is to find an IFS which when applied to any starting image will produce the given image. In this section we will show how an IFS is found for a given input image.

3.3.1 An Image as a Mathematical Function

When we wish to refer to an image, we refer to the function *f(x,y)* which gives the grey level at each point *(x,y)*. This is our model of an image. The function *f* gives us a *z* value for each *x,y* position and so defines a surface in 3D space. For simplicity, we assume we are dealing with square images of size 1. That is,

$$(x,y) \in \{(u,v): 0 \le u,v \le 1\} \equiv I^2, \text{ and } f(x,y) \in I \equiv [0,1].$$

I means the interval [0,1] and I^2 is the unit square.

3.3.2 A Metric on Images.

A metric is a function that measures distance. We want to find a map W which takes an input image and yields an output image. We want to know when W is contractive so we have to define a distance between two images. That is the distance between f_0 and f_1

where $f_1 = W(f_0)$. There are many metrics to choose from, but we will use the RMS (root mean square) metric

$$d_{rms}(f,g) = \sqrt{\sum_{i=1}^{n}\left(f(x_i,y_i) - g(x_i,y_i)\right)^2} . \qquad (1)$$

This metric gives us the distances between two images *f(x,y)* and *g(x,y)*, and in the process allows us calculate the components of a transformation for transforming *g(x,y)* into *f(x,y)*.

3.3.3 Self-Similarity in Natural images

A typical image of a face, does not contain the type of self-similarity that can be found in the fractals of figure 2. A face does contain a different sort of self-similarity. For example the left side of a persons jaw is similar (after transformation) to the right side.

The distinction from the kind of self-similarity we saw in figure 2, is that rather than having the image formed of copies of its whole self, here the image will be formed of copies of transformed parts of itself. These transformed parts do not fit together, to form an exact copy of the original image.

3.4 Partitioned Iterated Function Systems

In this section we extend the notion of the IFS to a PIFS which is used to copy grey scale images.

As before, the definition of a PIFS is not dependent on the type of transformations that are used. Again they are affine transformations. The grey level adds another dimension, so the transformations w_i are of the form,

$$w_i\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} \qquad (2)$$

where s_i controls the scaling and o_i the translation of the transformation. The s_i and o_i factors are applied to the z component of the pixel.

We can also write

$$v_i(x,y) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

The v_i has the same form as in table 1 and so is an IFS. Since an image is modelled as a function *f(x,y)*, we can apply w_i to an image *f* by

$$w_i(f) \equiv w_i(x, y, f(x,y)).$$

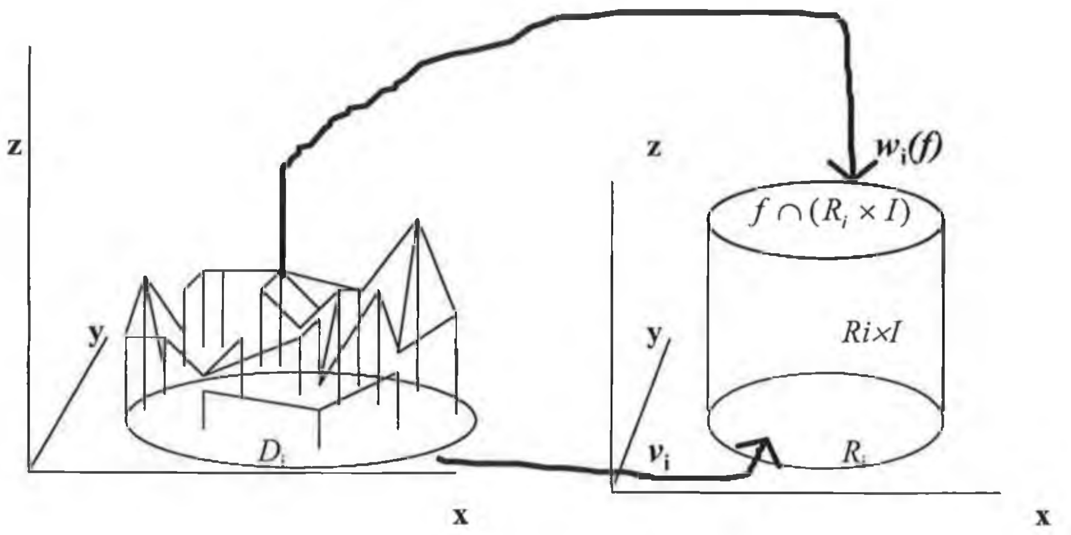
A PIFS selects a portion of the original, which we denote by D_i and copies that part to a part of the produced copy which is denoted R_i. We denote this transformation by w_i. The PIFS is applied in feedback loop; its own output is fed back as its new input again and again.

Now v_i determines how the partitioned pieces of an original are mapped to the copy, while s_i and o_i determine the scaling and offset of the transformation. This means that $v_i(D_i)=R_i$. Each w_i applies only to the part of the image that is above the domain D_i. Mathematically each w_i is restricted to $D_i \times I$, which is the volume with D_i as its base and height as $f(x_1,y_1),\ldots,f(x_n, y_n)$ where $(x_1,y_1),\ldots, (x_n, y_n) \in D_i$.

Figure 3 shows a D_i and a R_i. A representation of D_i is shown as a circle; this in general would not be the case. A representation for the surface over D_i is shown as the connected lines in the left hand diagram. These represent the pixel levels at some points in the D_i.

The representation of R_i is again shown as a circle and the surface over R_i is shown as a circle. Therefore the volume of 3D space with R_i as a circle and the surface as a circle is the cylinder shown. The surface of the image over R_i would in general be a connected set of line segments. The full surface of the image is f and $R_i \times I$ is a volume with R_i as its base and [0,1] as its height. So mathematically the piece of the image over R_i is $f \cap (R_i \times I)$.

The transformation $w_i(f)$ is shown as mapping the pixel values in D_i to the pixel values in R_i. The mapping v_i is a mapping of D_i to R_i in the x-y plane. The factors s_i and o_i in w_i act on the z axis.



Volume above D_i. Showing the z co-ordinates of some points in D_i.

Volume above R_i. Showing the z co-ordinates of some points in R_i. This graph indicates that f applied to all points in (R_i) result in the same grey level. Then $R_i \times I$ is the column shown. Also the circle at the top of the cylinder is the intersection of $R_i \times I$ with f.

Figure 3. Domain to Range

3.4.1 Fixed points for PIFS

Since the metric we chose in equation 1 is only sensitive to what happens in the z direction, it is not necessary to impose contractivity conditions in the x or y directions. The transformation W will be contractive when each $s_i < 1$. That is, when z distances are shrunk by a factor less than 1.

3.5 Encoding Images

In this section we will use the above theory and develop it into a method for encoding and decoding images.

Let the collection of domains D_i be referred to as the domain library $\boldsymbol{D}$.

Suppose we are given an image f that we wish to encode. That is, we want f be the fixed point of applying the following

$$f = W(f) = w_i(f) \cup w_2(f) \cup \ldots w_N(f)$$

That is, we seek a transformation W whose fixed point $f' = |W|$ is close to, or looks like f.. In that case,

$$f \approx f' = W(f') \approx W(f) = w_i(f) \cup w_2(f) \cup \ldots w_N(f)$$

We can determine that f' is close to f when $d_{rms}(f, f')$ is small. It is sufficient to approximate the parts of the image with transformed pieces. We do this by minimising the following quantities

$$d_{rms}(f \cap (R_i \times I), w_i(f)) \quad i = 1, \ldots, N \tag{3}$$

Here f is the surface which represents the whole image, $(R_i \times I)$ is the vertical space above R_i. Therefore $f \cap (R_i \times I)$ is the part of the image over R_i. Since each w_i is restricted to the image over D_i, and w_i maps the image above D_i to the image above R_i. Therefore $w_i(f)$ is the transformed part of the image over D_i.

That is we find pieces D_i and maps w_i, so that when we apply a w_i to the part of the image over D_i, we get something that looks like the image over R_i.

Given two squares containing n pixel intensities, $a_1,...,a_n$ (from D_i) and $b_1,...,b_n$ (from R_i), and given an R_i we wish to find a D_i for which the RMS metric is a minimum:-

$$d_{rms}(f \cap (R_i \times I), w_i(f))$$

The RMS metric seeks to find a D_i (we search all domains in D) whose pixel gray scale values most match the R_i. In equation 2, the z values are only affected by the scaling factor s_i and offset factor o_i. Therefore given two squares containing n pixel intensities a_1 ,....,a_n from D_i and b_1,....,b_n from R_i we multiply the a_1 ,....,a_n by s_i and add o_i to the result.

The s_i and o_i are constant for a D_i so we will refer to them as s and o. We will first of all evaluate a mean square distance metric. The root mean squared metric is the square root of this[30]. We therefore seek s and o to minimise the quantity

$$d_{ms} = \sum_{i=1}^{n} (s \cdot a_i + o - b_i)^2$$

This will give us scaling and offset factors that make the transformed a_i values have the least squared distance from the b_i values. The minimum of d_{ms} occurs when the partial derivatives with respect to s and o are zero, this occurs when

$$s = \frac{\left[n\sum_{i=1}^{n} a_i b_i - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i\right]}{\left[n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)^2\right]}$$

and

$$o = \frac{1}{n}\left[\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i\right]$$

In that case,

$$d_{ms} = \frac{1}{n}\left[\sum_{i=1}^{n} b_i^2 + s\left(s\sum_{i=1}^{n} a_i^2 - 2\sum_{i=1}^{n} a_i b_i + 2o\sum_{i=1}^{n} a_i\right) + o\left(no - 2\sum_{i=1}^{n} b_i\right)\right]$$

Then

$$d_{rms} = \sqrt{d_{ms}}$$

3.6 Quadtree Partitioning

This is the fractal image compression method that is used in this thesis. A quadtree partition is a representation of an image as a tree data structure in which each node, corresponds to a square portion of the image. The range blocks in a Quadtree partition are obtained by splitting the image into 4 equal sized sub-squares. These sub-squares are each broken into 4 other sub-squares see figure 4. The process continues until each square is of minimum size.

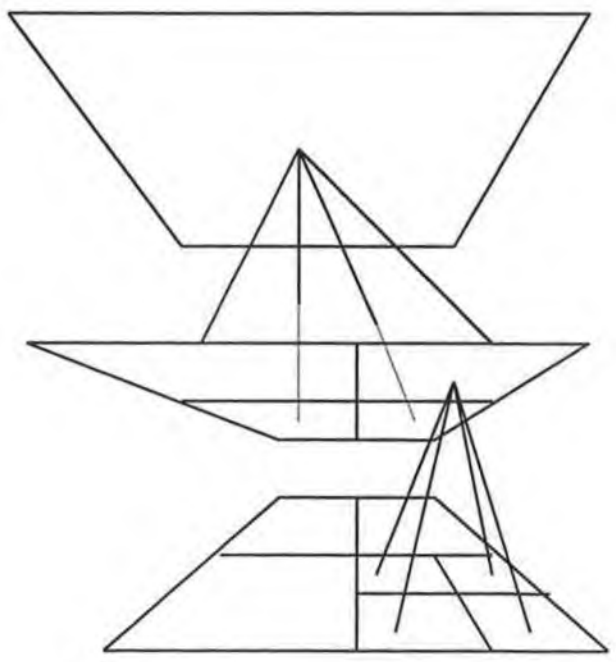


Figure 4. Quadtree partition

Then a range block is compared with all domain blocks that are twice the size of the range blocks. The comparison here involves applying the RMS metric to the range block and an averaged domain block. The averaged domain block is calculated by grouping 4 pixels together and getting their average. The domain block is now the same size as the range block. If the range block cannot be covered by a domain block it is split into 4 sub-squares. The process is then repeated for each of the 4 sub-squares.

Instead of comparing a range block with all domain blocks a classification[31] can be used to decide which domain blocks are most likely to be similar to the range block.

The Quadtree method of image compression is in two parts. First the image is compressed to a transformation file. The transformation file can then be decompressed to reproduce an approximation to the original image file.

The following algorithm for encoding images based on this idea is used in this thesis. The encoding algorithm proceeds as follows[32]:-

- Choose for the collection D of permissible domains all the sub-squares in the image of size 16, 32, 48. These domains are overlapping.

- Build up the ranges which are non-overlapping. Each range is a square sub-image. We do this by partitioning the image recursively until each range is of a minimum size for example 32.

```
• Loop
    • For each uncovered square in the quadtree partition $R_i$ , attempt to cover it by
      a domain that is larger; this makes the $v_i$ contractive.
    • If a predetermined tolerance RMS value $E$ is met, then
          call the square $R_i$ and the covering domain $D_i$ . Mark the $R_i$ as covered.
          Write out the transformation.
      Else
          subdivide the square and repeat.
• End Loop
```

The following algorithm for decoding images based on this idea is implemented in this thesis The decoding algorithm proceeds as follows:-

```
• Starting with any initial image.
• Read in all the $w_i$'s.
• Loop
    • For each $w_i$ find the domain $D_i$
        • apply $v_i$ to transform the domain to its range.
        • multiply each pixel value by $s_i$ and add $o_i$ to the result.
        • Put the resulting pixels in the position of the $R_i$.
• Until attractor is built. This typically takes 10 iterations.
```

Chapter

4.

Least Squares Polynomial Approximation.

We know that the relationship between the domain block and a modified range block is a linear one. That is in building the new range block

- We have matched a range block to a domain block using the RMS metric to a specified tolerance level.
- We have obtained a scaling factor *s* and offset factor *o* from the RMS metric.
- Multiply each pixel in the domain block by the scaling factor and adding the offset factor to produce a pixel in the new range block.

That is if $y_1, y_2, \ldots, y_n$ are pixels in the new range block and $x_1, x_2, \ldots, x_n$ are pixels of the domain block then each y_i is formed by

$$y_i = x_i * s + o$$

On retrieval of the steganographic data we need to determine if a match occurs between a given range and a domain block. This match is the discovery that the linear relationship as described above exists between the two blocks. However we do not have an exact linear relationship because we have to store an integer approximation to the real result of the above equation. So we only store an approximation to the variable y_i in the steganographic image file.

However in a multimedia environment the relationship may be even more non-linear. For example if the steganographic image is subjected to JPEG compression followed by decompression then the pixels of both the domain and range blocks would have been modified. This modification is such that it should still be possible to determine that a linear relationship exists between the two blocks to some probability. The method we choose is the Least Squares Method of line fitting.

We choose Least Squares because it allows the calculation of a equation of the form

$$y = M*x+C$$

for any two given sets of data. The two sets of data in this case are pixels from the domain block (x in the equation) and pixels from the range block (y in the equation). We use the correlation coefficient statistic to determine how good this equation fits the data. If we have a good fit then we have found the linear relationship and hence a match.

4.1 The Method of Least Squares

Very often in practice a relationship is found to exist between two (or more) variables, and one wishes to express this relationship in mathematical form by determining an equation connecting the variables.

- The first step is the collection of data showing corresponding values of the variables x and y.
- The next step is to plot the points $(x_1,y_1),(x_2,y_2), \ldots,(x_n,y_n)$. The resulting set of points is sometimes called a scatter diagram[33].

From the scatter diagram it is often possible to visualise a smooth curve approximating the data. Such a curve is called an approximating curve. In Fig 5, for example, the data appears to be approximated well by a straight line, and we say that a linear relationship exists between the variables. We may also get non linear relationships and no relationships at all.

The general problem of finding equations of approximating curves which fit given sets of data is called curve fitting. In practice the type of equation is often suggested from the scatter diagram. The basic approach in practice is usually the same: You choose a mathematical model for the data with a particular choice of parameters.

Data in general is not exact. The data is subject to errors (or noise). Thus, typical data never exactly fits the model that is being used, even when the model is correct. We need a means of assessing how good the fit of the model is to the data. We need to test the *goodness-of-fit*[34] against some useful statistical standard.

4.2 Least Squares in Retrieval of steganographic information

In our case we need to fit a curve (straight line) to linear data. That is

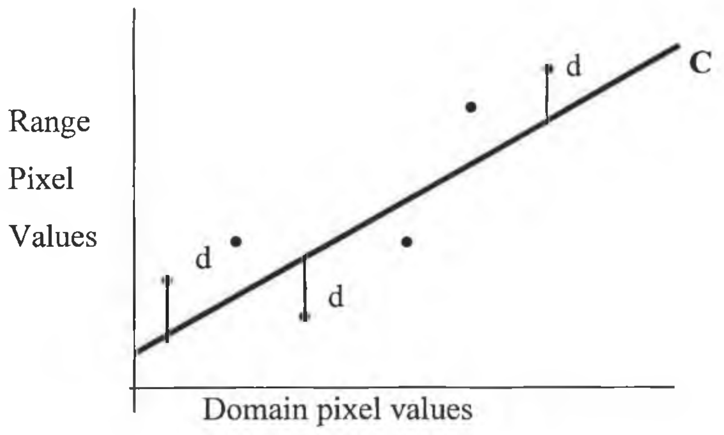


Figure5. Linear Regression Curve

For a given value of x, say x_1 there is an error (or noise) in the signal we represent this error as d_1. This difference is referred to as a deviation error or residual and may be positive, negative or zero. Similarly, corresponding to the values $x_2,...,x_n$ we obtain deviations $d_2,...,d_n$.

A measure of the goodness of fit of the curve C to the set of data is provided by the quantity

$$d_1^2+d_2^2+ \ldots +d_n^2$$

if this quantity is small the fit is good, if it is large the fit is bad.

The reason we square the deviations is because if we added them they would cancel each other out. On the other hand the modulus[35] is very complex to deal with.

Of all curves approximating a given set of data points, the curve having the property that

$$d_1^2+d_2^2+ \ldots +d_n^2 = a\ minimum$$

is the *best-fitting curve*.

Thus a line having this property is called a least-squares line.

Here we are saying that x is an independent variable and y is the dependent variable. However in a multimedia environment both x and y will be subject to error of approximately equal amounts. Therefore we may consider minimising the sum of the squares of the perpendicular distances.

The least-squares line[36] approximating the set of points $(x_1,y_1),(x_2,y_2), ...,(x_n,y_n)$ has the equation

$$y = Mx+B$$

Therefore

$$d_1^2 = (y_1 - Mx_1 - B)^2$$

So, the sum of the squares is

$$S = \sum_{i=1}^{n}(y_i - Mx_i - B)^2$$

So we need to find *M* and *B* for which *S* is a minimum this will give us the least-squares line. That is we find the minimum with respect to *M* and *B*.

$$\frac{dS}{dB} = -2\sum_{i=1}^{n} 1 \bullet (y_i - Mx_i - B)$$

and

$$\frac{dS}{dM} = -2\sum_{i=1}^{n} x_i \bullet (y_i - Mx_i - B)$$

We follow the standard minimum-finding course and set derivatives to zero.

Rewriting we have

$$N \bullet B + \left(\sum_{i=1}^{n} x_i\right) \bullet M = \sum_{i=1}^{n} y_i$$

and

$$\left(\sum_{i=1}^{n} x_i\right) \bullet B + \left(\sum_{i=1}^{n} x_i^2\right) \bullet M = \sum_{i=1}^{n} x_i \bullet y_i$$

Introducing the symbols

$$s1 = \left(\sum_{i=1}^{n} x_i\right)$$

and

$$s2 = \left(\sum_{i=1}^{n} x_i^2\right)$$

and

$$t0 = \left(\sum_{i=1}^{n} y_i\right)$$

and

$$t1 = \left(\sum_{i=1}^{n} x_i \bullet y_i\right)$$

Gives

$$M = \frac{s_0 \bullet t_1 - s_1 \bullet t_0}{s_0 \bullet s_2 - s_1^2}$$

and

$$B = \frac{s_2 \bullet t_0 - s_1 \bullet t_1}{s_0 \bullet s_2 - s_1^2}$$

4.3 Determining a Goodness of fit

In order to determine if the fit of the least-squares line to the data is good or not. We look at statistics theory.

4.3.1 Sample Variances Covariance and Correlation Coefficient

If $X_1, X_2, \ldots, X_n$ denote the random variables of a sample of size n, then the random variable giving the variance of the sample or the *sample variance*[37] is defined as

$$S^2 = \frac{\left(X_1 - \overline{X}\right)^2 + \left(X_2 - \overline{X}\right)^2 + \ldots.. + \left(X_n - \overline{X}\right)^2}{n}$$

where $\overline{X}$ is the mean of the sample or the *sample mean*

$$\overline{X} = \frac{X_1 + X_2 + \ldots.. + X_n}{n}$$

The *covariance*[38] of two random variables X and Y is defined as

$$S_{XY} = \frac{\sum_{i=1}^{n} \left(X_i - \overline{X}\right) \bullet \left(Y_i - \overline{Y}\right)}{n}$$

The sample variances and covariance of x and y are given by:-

the variance of x is

$$S_x^2 = \frac{\sum_{i=1}^{n} \left(x_i - \overline{x}\right)^2}{n}$$

and variance of y is

$$S_y^2 = \frac{\sum_{i=1}^{n} (y_i - \bar{y})^2}{n}$$

and the covariance of x and y is

$$S_{xy} = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) \bullet (y_i - \bar{y})}{n}$$

Then the *sample correlation coefficient*[39] is given by

$$r = \frac{S_{xy}}{S_x S_y}$$

4.3.2 Standard Error of Estimate

If we let y_{est} denote the estimated value of y for a given value of x, as obtained from the regression curve of y on x, then a measure of the scatter about the regression curve is given by

$$S_{x.y} = \sqrt{\frac{\sum_{i=1}^{n} (y_i - y_{est})^2}{n}}$$

which is called the *standard error of estimate*[40] *of y on x*.

Since

$$\sum_{i=1}^{n}(y_i - y_{est}) = \sum_{i=1}^{n} d_i{}^2$$

and

$$y_{est} = Mx + B$$

Then

$$S_{y.x}^2 = \frac{\sum_{i=1}^{n} y_i^2 - B\sum_{i=1}^{n} y_i - M\sum_{i=1}^{n} x_i \bullet y_i}{n}$$

We can now express $S_{y.x}^2$ for the least-squares line in terms of the variance and correlation coefficient as

$$S_{y.x}^2 = S_y^2\left(1 - r^2\right)$$

4.3.3 The Linear Correlation Coefficient

From the definitions of $S_{y.x}$ and S_y we have

$$r^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y_{est})^2}{\sum_{i=1}^{n}\left(y_i - \bar{y}\right)^2}$$

This can be expressed as

$$r^2 = \frac{\sum_{i=1}^{n} \left(y_{est} - \bar{y}\right)^2}{\sum_{i=1}^{n} \left(y_i - \bar{y}\right)^2}$$

Thus r^2 can be interpreted as the fraction of the total variation which is explained by the least-squares regression line. That is, r measures how well the least-squares regression line fits the sample data. If the total variation is all explained by the regression line, that is when $r^2 = 1$, we say that there is a perfect linear correlation. On the other hand if the total variation is all unexplained then $r = 0$.

That is

r = (Explained Variation)/(Total Variation).

In practice the quantity r, lies between 0 and 1.

For a linear least-square line we will use the following formula

$$r = \frac{n \bullet \sum_{i=1}^{n} (x_i \bullet y_i) - \left(\sum_{i=1}^{n} x_i\right) \bullet \left(\sum_{i=1}^{n} y_i\right)}{\sqrt{\left(\left[n \bullet \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right] \bullet \left[n \bullet \sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right]\right)}}$$

This equation will give rounding errors due to the small valued floating point data. However in practice we found that these rounding errors are insignificant to our algorithm.

4.4 The Retrieval algorithm with least squares

On retrieval of the steganographic data we need to determine if match exists between a modified range block and an associated domain block. On data hiding we have modified a range block by using the pixels of an associated domain block. On retrieval we basically need to find a correlation between the two blocks. The correlation is a linear relationship. If we can find this linear relationship between a range block and a domain block we say a match has occurred. This match is confirmed as follows:-

- For a given range block we select a block from the domain library.
- We calculate the *linear correlation coefficient* r between the two blocks.
- If r is greater than some minimum level then we say a match has occurred. If the steganographic image file is not subject to modification then r should be very close to 1.

Chapter

5.

The DCT Transform

5.1 The Discrete Cosine Transform (DCT)

The DCT[41] is in a class of mathematical operations that includes the Fast Fourier Transform (FFT). The basic operation performed by these transforms is to take a signal and transform it from one type of representation to another.

The FFT is used when analysing digital audio samples. In this case we collect a set of sample points form an incoming audio signal. We end up with a collection of points which show the voltage level at each point in time. That is the x-axis is the time and the y axis is the amplitude. The FFT transforms the set of sample points into a set of frequency values that describe exactly the same signal. After processing the x-axis is the frequency and the Y axis is the magnitude of that frequency.

The DCT takes a set of points from the spatial domain and transforms them into an identical representation in the frequency domain. The DCT operates on a three dimensional signal X,Y and Z. In this case the signal is a graphical image. The X and Y axes are the two dimensions of the image. The amplitude of the signal in this case is simply the value of a pixel in the image. This is the spatial representation of the signal.

The DCT is used to convert spatial information into *frequency* or *spectral* information. The X and Y axis represents the frequencies of the signal in two different dimensions.

There is an Inverse DCT (IDCT) function that can convert the spectral representation of the signal back to the spatial one.

The DCT is performed on an N x N square matrix of pixels values, and it yields an N x N matrix of frequency coefficients. The formula for the DCT is

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j)\sum_{x=1}^{N}\sum_{y=1}^{N} pixel(x,y) COS\left(\frac{(2x+1)i\Pi}{2N}\right) COS\left(\frac{(2y+1)j\Pi}{2N}\right)$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if } x \text{ is } 0, \text{ else } 1 \text{ if } x > 0$$

The formula for the inverse DCT transform (IDCT) is

$$Pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=1}^{N}\sum_{j=1}^{N} C(i)C(j)DCT(i,j) COS\left(\frac{(2x+1)i\Pi}{2N}\right) COS\left(\frac{(2y+1)j\Pi}{2N}\right)$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if } x \text{ is } 0, \text{ else } 1 \text{ if } x > 0$$

In the DCT N x N matrix, all elements in row 0 have a frequency component of zero in one direction of the signal. All elements in column 0 have a frequency component of zero in the other direction. As rows and columns move away from the origin, the coefficients in the transformed DCT matrix begin to represent higher frequencies.

Most graphical images are composed of low-frequency information. It turns out that the components found in row and column 0 carry more useful information about the image than the higher-frequency components.

So the DCT transformation identifies pieces of information in the signal that can be thrown away without seriously compromising the quality of the image.

Chapter

6.

Our Steganographic Algorithm.

6.1 Introduction

For this thesis we designed an algorithm which allows a user to insert steganographic data into image files.

In this section we discuss our algorithm which is specific to the quadtree fractal image compression method. However almost any of fractal image compression method[42] could have been used.

The data hiding algorithm exploits the noise[43] in an image. An objective of the method is that the modified parts of the steganographic image file will contain the same level of noise as in the surrounding areas.

If a noise filtering algorithm[44] has been applied to an image then this image is not suitable for data hiding. This is because introduced noise by our algorithm would be clearly visible.

6.2 The hiding steganographic data algorithm

- The image file is copied to produce a new image file which we refer to as the steganographic image file.

- The user graphically selects two non-overlapping areas of the image. One is known as the range region the other is known as the domain region. These areas of the image form the visual key. We refer to sub-images within the range region as range blocks. We refer to sub-images within the domain region as domain blocks.
- Blocks in the range region are modified. Blocks in the domain region are used in modifying the range blocks.
- We refer to the set of domain blocks as the domain library.
- A domain library[45] is built from a set of blocks in the domain region. The domain library is split into two halves.
- Fractal image compression theory is used to identify a series of range blocks, and a corresponding series of matching domain blocks[46]. The matching process produces for each range block, the best matching domain block and a corresponding fractal transform. We may be unable to find a good match for some range blocks.
- If a good match is found the algorithm produces a new range block which is most visually like the original range block. This new range block is stored in the position of the original block in the steganographic image file.
- The algorithm selects a suitable range block, and then for each domain block in the selected half of the domain library determines a fractal transform.
- If the current bit of steganographic data is a 0, we search for a match in the first half of the domain library, otherwise we search the second half.
- The fractal transform basically involves multiplying a pixe,l in the selected domain block, by an average scaling factor and adding an offset factor to the result. This calculation produces a corresponding pixel in the new range block. The pixel value stored is an integer approximation.
- The fractal transform is applied to the domain block, to produce a new range block that is visually like the original range block. This new block is written to the steganographic image, in place of the old range block.
- Hiding the steganographic data involves producing a new range block, in the manner described for each bit of steganographic data.

6.3 The retrieving steganographic data algorithm

- The retrieving steganographic data algorithm is the reverse of the hiding.
- In retrieving the user specifies the position of the range and the domain regions. These form the visual key.
- The domain library is built from a set of blocks in the domain region. The domain library is split into two halves.
- To find a bit of steganographic data in the steganographic image file we select a range block from the range region in the same order as in the hiding algorithm and find a corresponding domain block that matches it by a linear relationship.
- The linear relationship is determined by calculating the least squares coefficient between the domain and range blocks.
- If the match occurred with a domain block in the first half of the library then a zero bit is retrieved otherwise a one bit is retrieved.

6.4 Theory

In this section we will describe the theory and algorithm that allows our algorithm to hide steganographic data into images. Our algorithm is based on the quadtree fractal image compression algorithm as described in chapter 3.

The algorithm is described based on the following assumptions, these are only to simplify the discussion.

- We have a square grey scale image of size 256x256.
- That each byte in the image data represents a pixel whose level of grey is from 0 to 255.
- We use square shaped regions and blocks.
- The domain blocks are twice as big as the range blocks.

The hiding algorithm takes as input and image *I*, and a set of steganographic data *steg*. The hiding algorithm outputs a visually identically image I_{new}, that has the steganographic data hidden in it.

The revealing algorithm takes as input an image *I*, and outputs a set of steganographic data *steg*.

We select two square (square is a simplification) parts of the image. These parts can be at any location in the image. One part, is used as a domain region, the other is used as the range region. Let the range region be I_1 and the domain region be I_2.

As described in chapter 3 the domain library[47] is built, from a set of blocks in the domain region. These blocks are bigger than the range blocks. For example choose for the collection *D* of permissible domains all the sub-squares in I_2 of size 4x4, 5x5, 6x6. These domain blocks can overlap one another. The blocks are overlapping to allow more blocks in the domain library. All we really need for a domain block, is a collection of pixels taken from the domain region.

Let the domain library *D* be represented by the blocks $\{d_1,d_2,....,d_t\}$. Where *t* is the number of blocks in the domain library. The domain library *D* is split into two halves $\{d_1,d_2,...,d_{t/2}\}$ and $\{d_{(t/2)+1},d_{(t/2)+2},...,d_t\}$. Let us call the first half D_0 and the second half D_1.

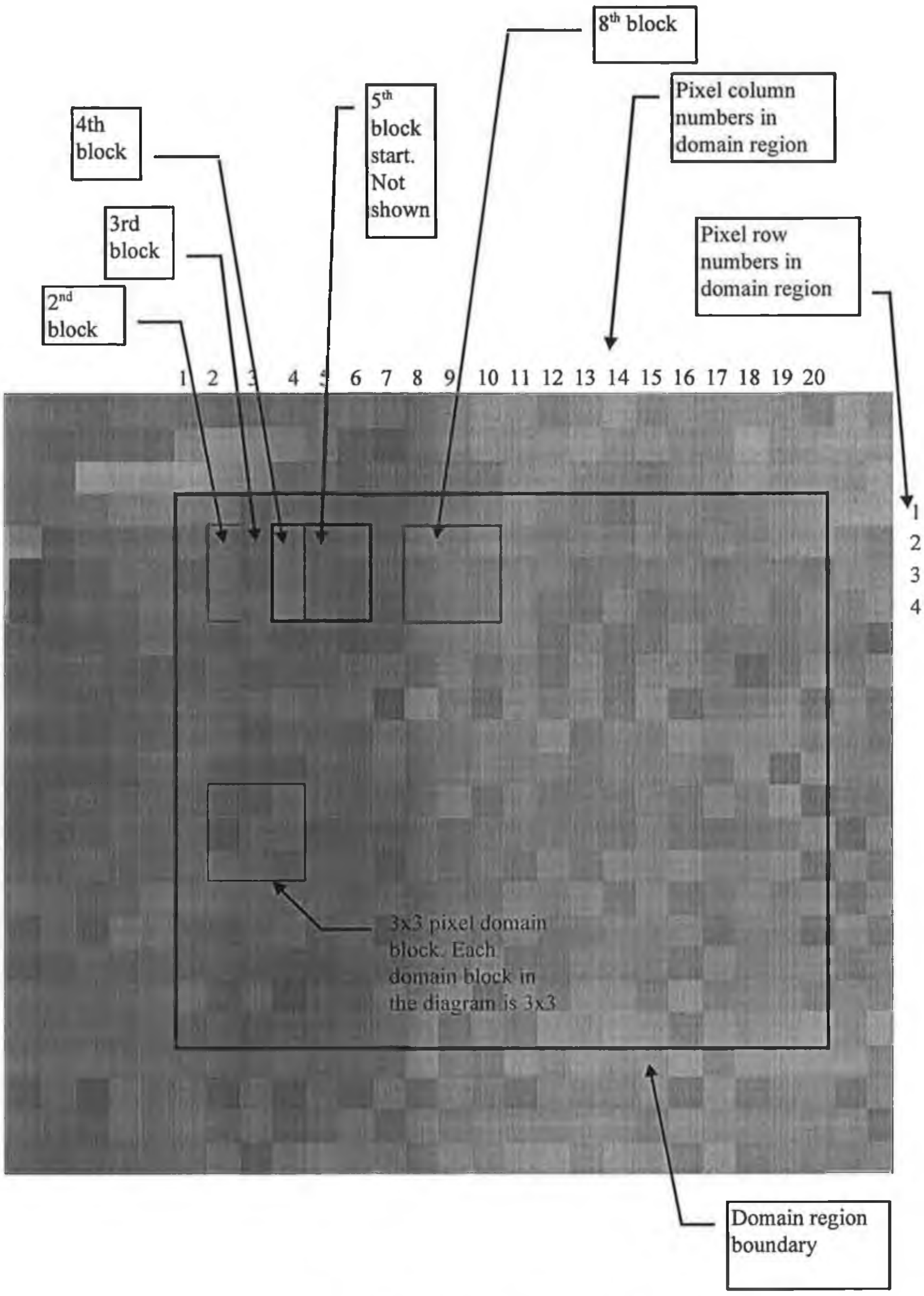

8th block
Pixel column numbers in domain region
5th block start. Not shown
4th block
3rd block
2nd block
Pixel row numbers in domain region
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1
2
3
4
3x3 pixel domain block. Each domain block in the diagram is 3x3
Domain region boundary

Figure 6. Part of the Domain library

Let $R = \{r_1, r_2, ..., r_u\}$.be the collection of range block. The range blocks are non-overlapping. The range blocks cannot overlap, because these blocks are modified by the algorithm, and we only want to modify one block at a time. Each range block is a square sub-image. We determine R by partitioning I_1 recursively until each range block is of a minimum size, for example 4x4. We now continue to partition I_1 until all range blocks are of a maximum size for example 2x2. During this process we add all range blocks of size 4x4 to size 2x2 to R.

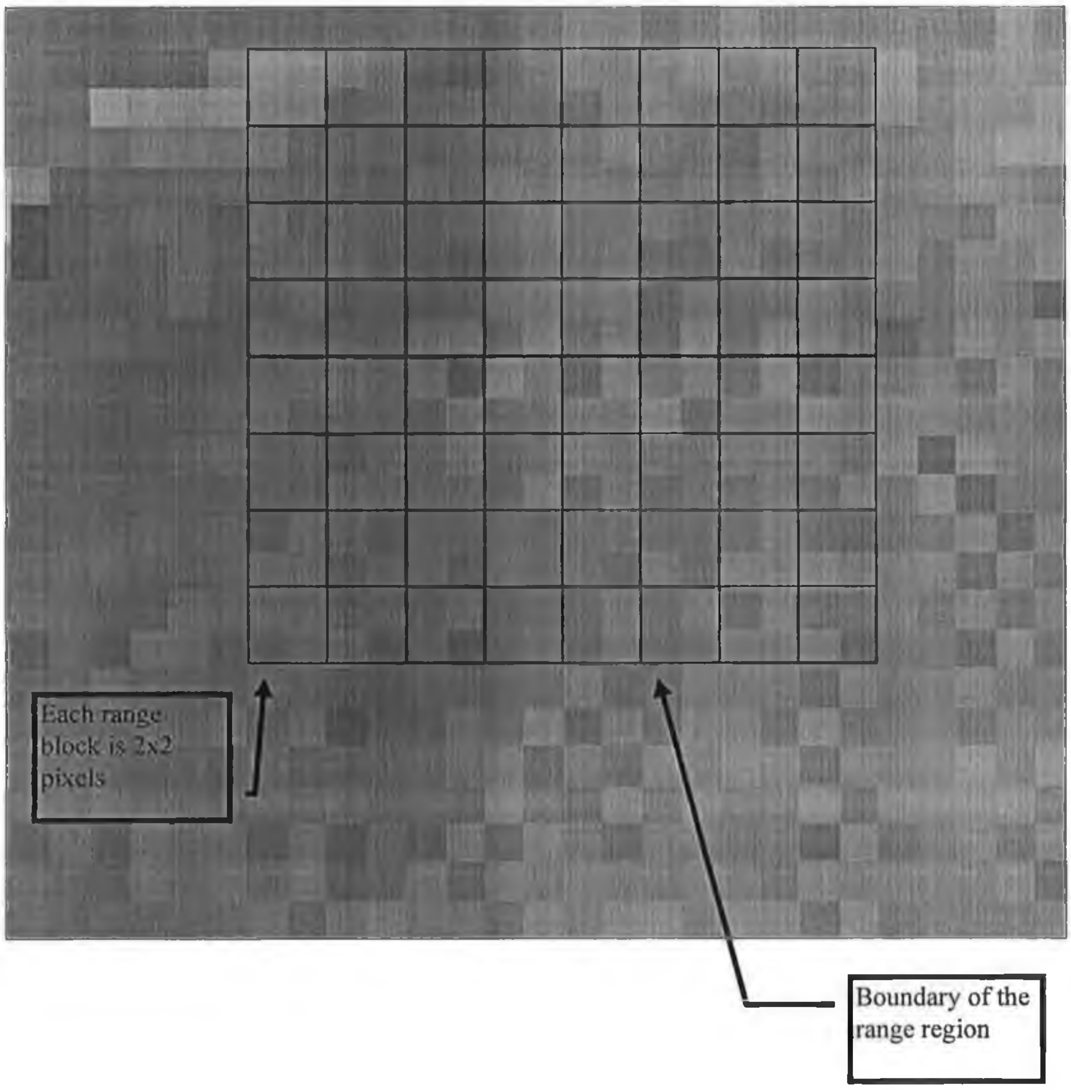


Figure 7. The Range blocks in the range region.

When hiding we use the Root Mean Square metric (RMS) to determine if the range block matches a domain block. While calculating the RMS metric we also calculate average scaling s and offset o factors. When these factors are applied to the pixels of the domain block they yield a new block r_{new} which looks like the original range block. We replace the original range block with r_{new}. Each replacing of a range block by its equivalent new block allows us to hide 1 bit of the steganographic data.

When retrieving we need to determine if a selected range block r_i has been replaced by a new block as described above. To do this we search through D and calculate the best correlation coefficient for the r_i with each domain block d_i. A match indicates that a 1 or a 0 bit has been hidden for this range block.

Let $\{y_1, y_2, \ldots, y_n\}$ represent the pixels in a range block. Let $\{x_1, x_2, \ldots, x_n\}$ represent the **averaged** pixel values in the corresponding domain block. The length of any side in a domain block is twice that of a range block therefore there are four times as many pixels in a domain block than there are in a range block. Therefore in matching we will take an average of four pixels in a domain block for each range block (as described in chapter 3). We do the averaging so that in the domain block x_n is the last pixel value. Therefore the domain block and the range block have been made the same size.

All of the above apply to both the hiding and revealing algorithms. We will now describe the specifics of both algorithms.

6.4.1 The add steganographic data algorithm

We will use an example throughout this description to illustrate the workings of the algorithm.

Let the steganographic data *steg* be represented as a binary bit stream $\{b_1, b_2, \ldots, b_n\}$. Let n be the number of bits in the bit stream. Let *CurrentBit* be the current bit in the stream. *CurrentBit* has values of 0 or 1.

Let the predetermined tolerance RMS value be E; the value for E is chosen to be ≈ 5.0. This value was chosen from the results of Fisher[48].

We use a set of selected range blocks represented by *SelectedRanges* which is initially the empty set { }. The hiding algorithm then selects a range block r_i from R where r_i and any of its sub-blocks do not belong to *SelectedRanges*. We then add r_i and all its sub-blocks to *SelectedRanges*. For example if r_i is of size 8x8 it will be made up of 4 sub-blocks of size 4x4.

We now look at an example range block r_i,. This is a 4x4 pixel block and we show the pixel values in this block.:-

	Row 1	Row 2	row 3	row 4
Column 1	**49**	**58**	**35**	**52**
Column 2	**50**	**57**	**56**	**55**
Column 3	**50**	**62**	**54**	**55**
Column 4	**51**	**58**	**55**	**58**

Table 4 The pixel values of the range block r_i

We now search for a match in one of the domain Libraries D_0 or D_1. If CurrentBit is 0 we search D_0 otherwise we search D_1. During this search we apply the Root Mean Squared(RMS) metric between r_i and a d_j. We chose the block d_j for which the RMS value is a minimum and is less than E (as described in chapter 3).

If this search is successful then

Let d_{Found} represent the domain block that matches the range block r_i. We will use the example to illustrate the calculation of the RMS value.

	Row 1	Row 2	row 3	row 4
Column 1	**66**	**79**	**47**	**71**
Column 2	**67**	**76**	**77**	**77**
Column 3	**70**	**84**	**73**	**76**
Column 4	**68**	**77**	**76**	**81**

Table 5 The pixel values of the domain block d_{Found}

As described in chapter 3 to determine the RMS metric between the range and domain blocks we first determine *s*, *o* (the scaling and offset factors).

$$s = \frac{\left[n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i \right]}{\left[n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i \right)^2 \right]}$$

In this case *n* is 16 (the number of pixels in the block) and each x_i is a pixel from the domain block and each y_i is a pixel from the range block.

In this case:-

$$s = 0.702929587$$

and

$$o = \frac{1}{n}\left[\sum_{i=1}^{n} y_i - s\sum_{i=1}^{n} x_i \right]$$

In this case:-

$$o = 2.255439438$$

Now the *RMS* metric is calculated with:-

$$RMS = \sqrt{\frac{1}{n}\left[\sum_{i=1}^{n} y_i^2 + s\left(s\sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i y_i + 2o\sum_{i=1}^{n} x_i \right) + o\left(no - 2\sum_{i=1}^{n} y_i \right) \right]}$$

In this case:-

$$RMS = 0.907089508$$

This RMS value is less than *E*. Let us assume that this *RMS* value is the minimum calculated during our search.

We build a new block r_{new} by applying the transformations to the pixels of the domain block d_{Found}. Let $\{z_0, z_1, \ldots, z_n\}$ represent the pixels of r_{new}.

where $z_i = s * x_i + o$

In our example we build the block r_{new} :-

	Row 1	Row 2	row 3	row 4
Column 1	**48.649**	**57.787**	**35.293**	**52.163**
Column 2	**49.352**	**55.678**	**56.381**	**56.381**
Column 3	**51.461**	**61.302**	**53.569**	**55.678**
Column 4	**50.055**	**56.381**	**55.678**	**59.193**

Table 6 Values in r_{new} before being rounded

Because *s* and *o* are real numbers when we apply them to a pixel of the d_{Found} they produce a real result. But we have to store an integer approximation to this real number in r_{new}. This integer approximation is based on rounding the real number to the nearest whole number.

Applying rounding to the pixels of r_{new} gives:-

	Row 1	Row 2	row 3	row 4
Column 1	**49**	**58**	**35**	**52**
Column 2	**49**	**56**	**56**	**56**
Column 3	**51**	**61**	**54**	**56**
Column 4	**50**	**56**	**56**	**59**

Table 7 Pixels stored in r_{new}.

We now place r_{new} in the appropriate position in the steganographic image file I_{new}.

We now show a comparison of the pixels of the original range block r_i and the block replacing it r_{new}.

Pixel number (row, column)	Pixels of r_i	Pixels of r_{new}
1,1	**49**	**49**
2,1	**50**	**49**
3,1	**50**	**51**
4,1	**51**	**50**
1,2	**58**	**58**
2,2	**57**	**56**
3,2	**62**	**61**
4,2	**58**	**56**
1,3	**35**	**35**
2,3	**56**	**56**
3,3	**54**	**54**
4,3	**55**	**56**
1,4	**52**	**52**
2,4	**55**	**56**
3,4	**55**	**56**
4,4	**58**	**59**

Table 8. Comparison of pixels in r_{new} and r_i.

Now we show the relevant portions of the original image and the steganographic image file. The block r_i is marked in the original image file and the block r_{new} is marked in the steganographic image file.

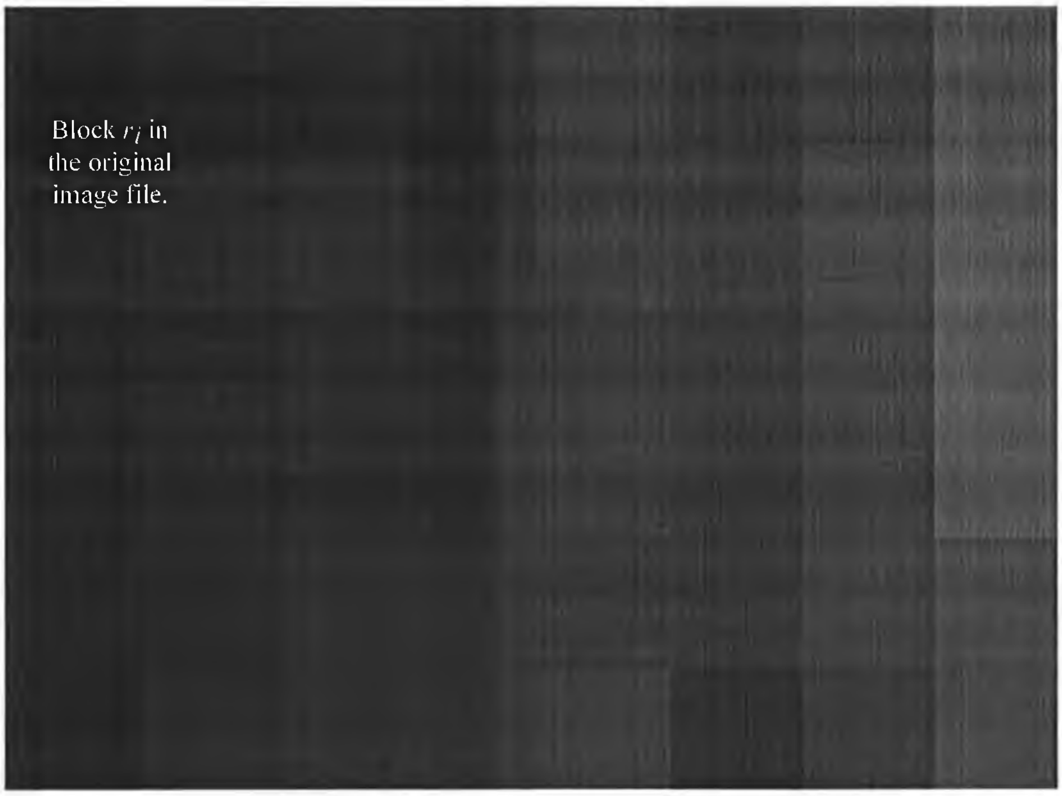


Figure 8. Original range block and its replacement

6.5 Algorithm pseudo code

Define the function *Bound* as

```
Function Bound(val) Integer
Begin
    if ( val < 0 ) then
        return 0
    else
        if ( val > 255 ) then
            return 255
        else
            return val
        endif
    endif
End
```

Hide is a function which finds a matching domain block for the given range block. *Hide* will also modify the appropriate part of I_{new}.:-

```
Hide($r_i$)
Begin
    If CurrentBit equals 0 then
        For each domain in $D_0$ find the domain that has the least Root Mean
        Squared Difference to $r_i$. Let this domain be $d_{RMSmin}$ and let this RMS
        value be $\mathrm{RMS}_{\min}$.
        If $\mathrm{RMS}_{\min} < E$ then
            Determine $s_i, o_i$ for transforming $d_{RMSmin}$ to $r_i$.
            Create $r_{new}$ which is the same size as $r_i$ initially set to { }.
            Place $r_{new}$ in the position of $r_i$ in $I_{new}$.
        Endif
        Else
                For each domain in $D_1$ find the domain that has the least Root Mean
                Squared Difference to $r_i$. Let this domain be $d_{RMSmin}$ and let this RMS
                value be $\text{RMS}_{\text{min}}$.
                If $\text{RMS}_{\text{min}} < E$ then
                        Determine $s_i, o_i$ for transforming $d_{RMSmin}$ to $r_i$.
                        Create $r_{new}$ which is the same size as $r_i$ initially set to { }.
                        Place $r_{new}$ in the position of $r_i$ in $I_{new}$.
                Endif
        Endif
End
```

The hiding algorithm proceeds as follows:-

```
Choose for the domain library $D$ all the sub-squares in $I_2$ of size 8,10, and 16 . Average
the pixels in each domain block so that they now have sizes of 4,5 and 8. That is we
average 4 pixels in $I_2$ to make 1 pixel in a domain block.
Split $D$ into $D_0$ and $D_1$.
Partition $I_1$ recursively until each range block is of size 16.
Continue to partition $I_1$ recursively until each range block is of size 4 and in the process
add each range block to $R$.
Set SelectedRanges to { }.
Set Index = 0
Set CurrentBit to steg[Index]
Set $E$ to 5.0
While ( Index <N and SelectedRanges <> R )
        Select a range block $r_i$ from $R$.
        Set RangeAndSubBlocks to {}.
        Add $r_i$ to RangeAndSubBlocks.
        Add all sub-blocks of $r_i$ to RangeAndSubBlocks.
        If ( $RangeAndSubBlocks \cap SelectedRanges = \in$ ) then
```

```
            Add RangeAndSubBlocks to SelectedRanges.
            Hide($r_i$).
            Increment Index.
            Set CurrentBit to steg[Index]
        Endif
End While
```

6.5.1 The Retrieve Steganographic Data Algorithm

Let the steganographic data *steg* be represented as a binary bit stream which is initially set to { }.

We set $corr_{min}$ the minimum match correlation coefficient to the experimental determined value.

We use a set of selected range blocks represented by *SelectedRanges* which is initially the empty set { }. The retrieving algorithm then selects a range block r_i from R where r_i and any of its sub-blocks do not belong to *SelectedRanges*. We then add r_i and all its sub-blocks to *SelectedRanges*. For example if r_i is of size 8 it will be made up of 4 sub-blocks of size 4.

We now search for a match in the domain libraries D_0 and D_1. If a match occurs we set *FoundBit* to TRUE. If a match occurs in D_0 we set *CurrentBit* to 0 otherwise if a match occurs in D_1 we set *CurrentBit* to 1. This search involves finding a corresponding domain block d_{Found} (if any) that has the correlation coefficient $corr > \mathrm{corr}_{\mathrm{min}}$. The correlation coefficient is calculated for each block in the domain library. The calculation of the correlation coefficient for a given domain and range blocks is as follows:-

$$corr = \frac{n \bullet \sum_{i=1}^{n}(x_i \bullet y_i) - \left(\sum_{i=1}^{n} x_i\right) \bullet \left(\sum_{i=1}^{n} y_i\right)}{\sqrt{\left[n \bullet \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right] \bullet \left[n \bullet \sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right]}}$$

where

$$\{x_0, x_1, x_2, \ldots, x_n\} \in r_i$$

$$\{y_0, y_1, y_2, \ldots, y_n\} \in d_{found}$$

corr is the correlation coefficient

We will use the following example to illustrate the algorithm. We use the same domain block as in the hiding algorithm example.

	Row 1	Row 2	row 3	row 4
Column 1	**66**	**79**	**47**	**71**
Column 2	**67**	**76**	**77**	**77**
Column 3	**70**	**84**	**73**	**76**
Column 4	**68**	**77**	**76**	**81**

Table 9 Domain block

We use the range block generated in the hiding algorithm example:-

	Row 1	Row 2	row 3	row 4
Column 1	**49**	**58**	**35**	**52**
Column 2	**49**	**56**	**56**	**56**
Column 3	**51**	**61**	**54**	**56**
Column 4	**50**	**56**	**56**	**59**

Table 10 Range block

We calculate the correlation coefficient between the domain and range blocks as

$$corr = 0.997071$$

which is greater that $corr_{min}$.

We will now detail the algorithm:-

For each domain block in the domain library we calculate the sum of the pixels *Dsumpixels* and the sum of the square of each pixel call it *Dsumsquarepixels*. We store these values in a data structure for easy retrieval afterwards.

```
Function $IsRangeTransformed(r_i, d_{Found})$ BOOLEAN
Begin

    Let $\{x_0, x_1, ......, x_n\} \in r_i$
    Let $\{y_0, y_1, ......, y_n\} \in d_{Found}$
    Set $i$ to 0.
    Set $n$ to number of pixels in range block
    Set $Rsumpixels$ to 0
    Set $Rsumsquarepixels$ to 0
    Set $sumRD$ to 0
    Loop
        Set $sumRD$ to $sumRD + x_i*y_i$
        Set $Rsumpixels$ to $Rsumpixels + x_i$
        Set $Rsumsquarepixels$ to $Rsumsquarepixels + x_i*x_i$
        Increment $i$.
    Until $i == n$
```

```
set
$corr = \frac{n * sumrd - Rsumpixels * Dsumpixels}{\sqrt{\left[\left(n * Rsumsquarepixels - Rsumpixels^2\right) * \left(n * Dsumsquarepixels - Dsumpixels^2\right)\right]}}$

if ( $corr > corr_{min}$)
        return TRUE
else
        return FALSE
End



$Reveal(r_i)$
Begin
        Set $i$ to 0
        Loop
                Set $d_{Found}$ to $d_i$ from $D$.
                Set $FoundBit$ to FALSE
                If ($IsRangeTransformed(r_i,d_{Found})$ = TRUE )
                        Set $FoundBit$ to TRUE
                        If $d_{Found} \in D_0$ then
                                Set $CurrentBit$ to 0
                                Else
                                Set $CurrentBit$ to 1
                        Endif
                        return
                Endif
        End Loop
End
```

The retrieving algorithm proceeds as follows:-

```
Choose for the domain library $D$ all the sub-squares in $I_2$ of size 8, 16, 32.
Average the pixels in each domain block so that they now have sizes of 4,8 and 16. That
is we average 4 pixels in $I_2$ to make 1 pixel in a domain block.
Split $D$ into $D_0$ and $D_1$.
Partition $I_1$ recursively until each range block is of size 16.
Continue to partition $I_1$ recursively until each range block is of size 4 and in the process
add each range block to $R$.
Set $SelectedRanges$ to { }.
Set $Index = 0$
Set $FoundBit$ to FALSE
While ( $SelectedRanges <> R$ )
    Select a range block $r_i$ from $R$.
    Set $RangeAndSubBlocks$ to {}.
    Add $r_i$ to $RangeAndSubBlocks$.
    Add all sub-blocks of $r_i$ to $RangeAndSubBlocks$.
    If ( $RangeAndSubBlocks \cap SelectedRanges = \{ \}$ ) then
        Add $RangeAndSubBlocks$ to $SelectedRanges$.
        $UnHide(r_i)$.
        If $FoundBit$ = TRUE then
            Set $steg[Index]$ to $CurrentBit$
            Increment $Index$.
        Endif
    Endif
End While
```

See Appendix B for implementation details of our algorithm.

Chapter

7.

Results

This chapter describes the results obtained from testing the theory of chapter 6. The results obtained were as the result of running frasteg. Frasteg is proof of concept application, see Appendix A for details. Given a input bitmap file and steganographic data Frasteg generates a resulting bitmap file which is visually identical to the original. Frasteg successfully retrieves the hidden data.

We describe a series of visual and statistical tests on the steganographic image file, we next, present tests of the method for *goodness* of data hiding. Statistical methods are applied to the steganographic image file to determine how good the method is at hiding information. Also tests of the robustness of the method are presented.

7.1 Steganographic Image Quality

7.1.1 Visual Tests

Figure 9 shows an original bitmap girl.bmp and a new bitmap test.bmp. With the steganographic data *Hello world* hidden in test.bmp.

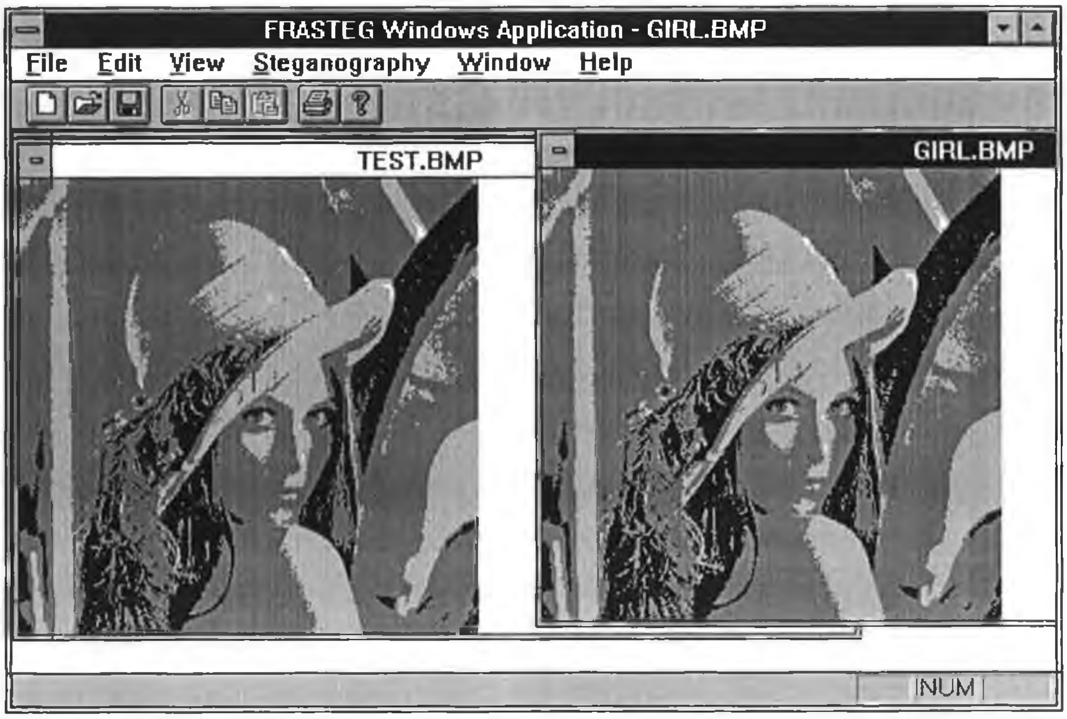

FRASTEG Windows Application - GIRL.BMP
File Edit View Steganography Window Help
TEST.BMP
GIRL.BMP
NUM

Figure 9. Girl with hidden data 1

The steganographic data has been hidden in the bottom left corner of the test.bmp. But as you can see the two bitmaps are virtually identical.

Figure 10 shows the range region (zoomed by a factor of 600%) of the original bitmap girl.bmp and the corresponding region in the new bitmap test.bmp. But again the modifications are not visible to the human eye.

However if the original and modified images are placed side by side at a zoom factor of 5000% it may be possible to see the modifications (see figure 8). But it is very difficult to do so. With a zoom factor of 5000% it is only possible to see a 10x10 pixel region on a 21 inch monitor. Several people were asked if they noticed differences in the two zoomed images. Some said they did but admitted that they were not sure.

A test took place where several people were shown the original and modified images side by side zoomed by a factor of 3000%. The participants in the test were told the position of the range region. Nobody could see the modified blocks. Several guesses were made but all wrong.

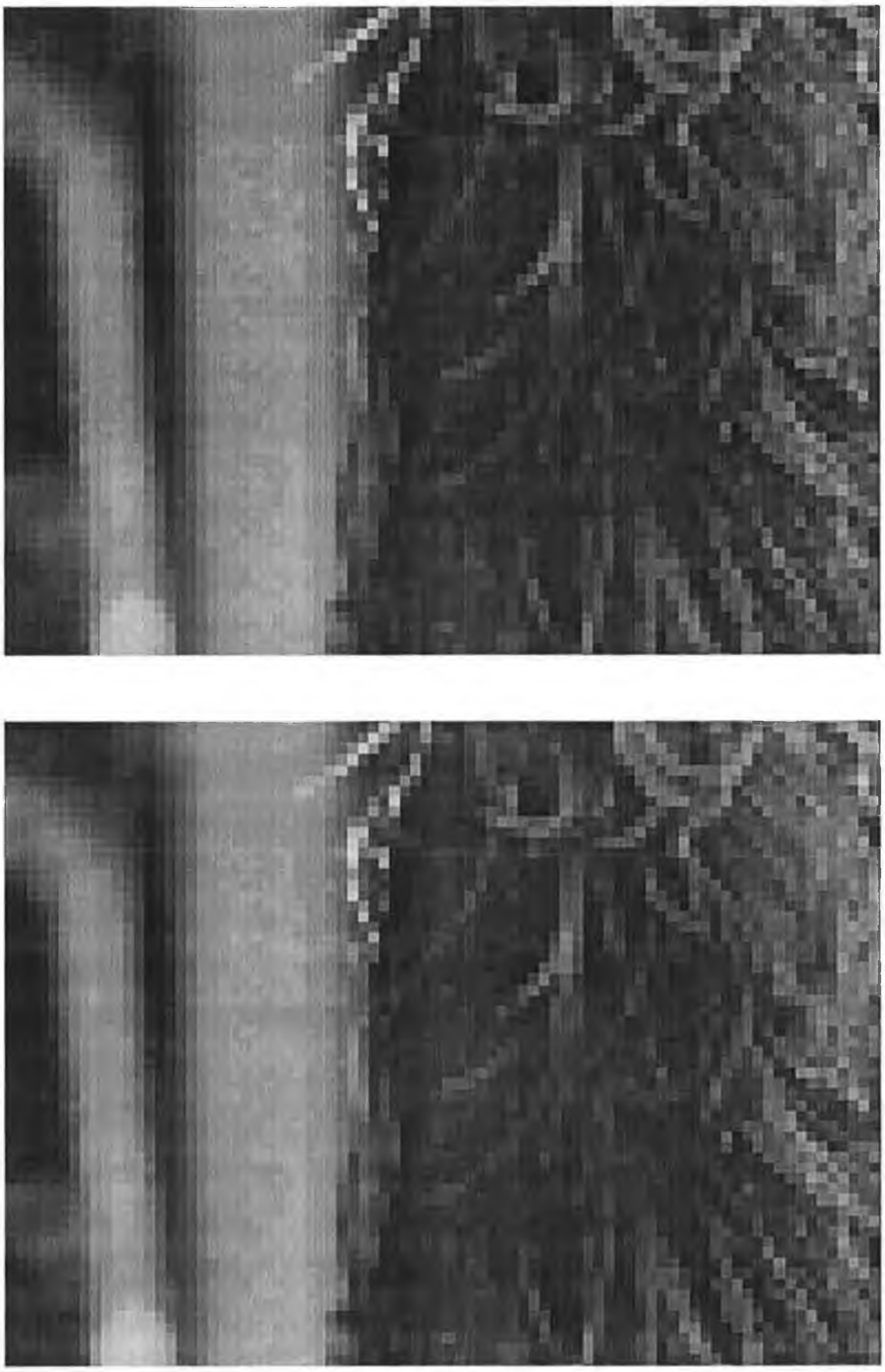


Figure 11. Range Region before and after data hiding

Figure 12 shows the selected range blocks with the modified pixel values. The selected locations are in the least sensitive parts of the image. That is the hair and shoulder which is in shadow.

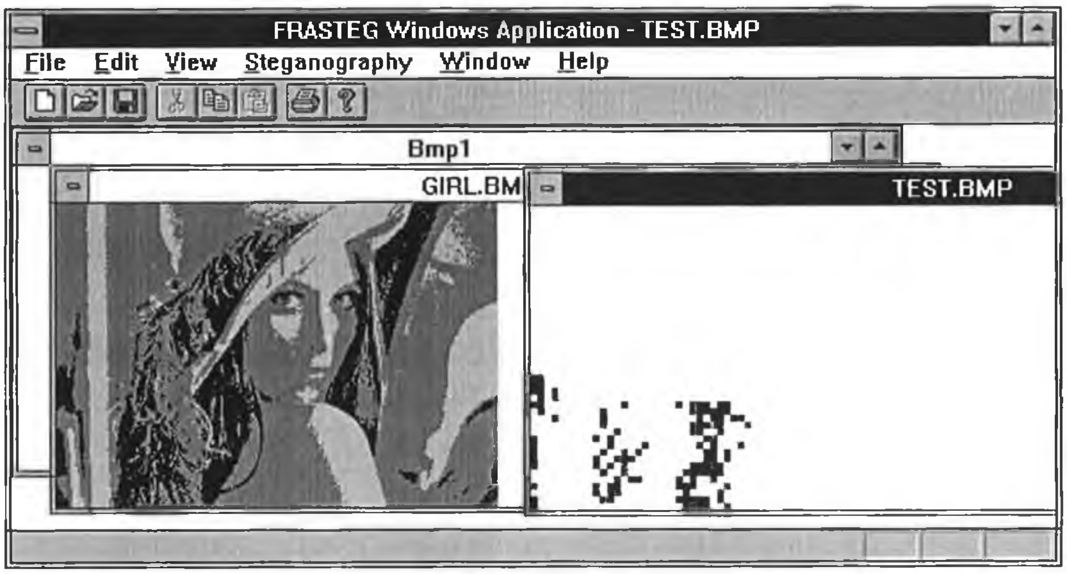

FRASTEG Windows Application - TEST.BMP
File Edit View Steganography Window Help
Bmp1
GIRL.BM
TEST.BMP

Figure 12. Ranges for Hiding

Table 11 is a list of the pixel values in a portion of the girl.bmp image. The pixel values in the selected range when hiding *Hello world* are shown in bold.

48	71	31	2f	4d	43	3f	8a	5c	31	7d	5d	2d	31	31	30	45	4f	43	46	27	34	49	**60**	29	5e	49	79	39	**4a**	**50**	**51**	**5e**
51	75	2a	4b	4a	4c	7a	65	29	39	83	7b	38	26	27	3c	59	4a	35	2c	29	37	5f	4a	57	82	7f	31	36	**36**	**48**	**4f**	**52**
50	69	25	5e	52	85	7f	31	25	38	7a	7f	33	48	37	3a	5b	40	58	27	4c	51	7c	53	76	76	2f	27	25	**2a**	**37**	**3e**	**4d**
54	70	38	56	68	92	53	29	29	31	64	4b	3b	3d	27	50	42	6d	3a	32	48	57	69	5f	6f	2a	25	1e	24	**2a**	**32**	**3c**	**47**
3e	67	4a	40	8e	85	3d	2a	25	51	74	2c	41	21	30	50	6c	65	24	3a	43	68	6e	4d	35	**31**	**2c**	**2b**	**24**	**30**	**32**	**30**	**38**
39	6a	42	53	82	65	37	28	51	63	77	4c	21	22	43	4a	86	2b	29	56	5e	78	41	2d	3d	**27**	**33**	**28**	**26**	**33**	**2d**	**31**	**31**
43	6f	3d	76	7b	35	27	24	64	64	6d	2d	1e	29	62	74	47	30	48	71	76	73	54	43	46	**27**	**25**	**25**	**2e**	**28**	**2e**	**3b**	**2e**
49	70	4a	7c	68	24	24	32	5b	75	4d	1f	3b	58	39	79	31	4d	64	62	7d	38	6f	61	35	**2d**	**2b**	**2d**	**29**	**28**	**20**	**25**	**2c**
5d	69	75	64	4c	2c	21	5b	53	7d	20	1e	**3d**	34	72	38	4e	34	80	6a	64	65	6e	66	28	29	2d	2e	2a	**2c**	**29**	**23**	**2b**
64	69	77	74	49	3b	36	2e	69	41	45	2c	39	75	4d	57	32	53	78	70	84	94	65	3a	2c	35	21	2e	2b	**26**	**34**	**33**	**2b**
56	6c	4f	5d	2f	2e	2b	34	66	26	2b	41	44	89	2e	2c	38	47	66	73	a4	7b	29	43	34	2e	23	2c	2d	**28**	**2d**	**28**	**2c**
4d	64	54	3e	2a	2e	2d	5c	47	2c	31	2f	7a	3e	3e	42	59	55	47	8b	82	4e	28	49	53	27	26	28	30	**2d**	**25**	**24**	**2d**
45	55	53	2a	2a	2e	2a	5b	47	47	3a	32	61	2e	45	33	4f	45	5c	9b	6b	58	24	4c	46	**34**	**28**	**2e**	**2c**	**34**	**28**	**27**	**2a**
50	56	31	2e	31	26	34	57	58	33	32	51	2c	3b	39	6c	35	44	76	8b	37	53	5c	67	3b	**2e**	**2c**	**2f**	**2f**	**2a**	**31**	**29**	**26**
45	51	35	32	30	24	42	54	37	2d	37	57	33	2f	46	5b	2f	54	46	77	48	77	8e	6b	2a	**2e**	**34**	**2b**	**30**	**2f**	**2b**	**2a**	**2b**
3c	3b	2b	2f	31	2c	41	37	23	3c	3a	40	3b	39	40	38	47	3e	49	7f	8e	80	62	57	41	**2f**	**34**	**2b**	**29**	**27**	**2c**	**2c**	**31**
3c	**2e**	**31**	**2e**	29	34	36	24	36	47	3b	3a	32	2a	31	43	43	81	83	aa	84	5a	54	73	8d	**35**	**2e**	**2d**	**29**	2b	2a	2e	2a
3e	**31**	**2e**	**29**	29	32	27	2b	3b	2d	5b	34	33	30	23	4d	83	76	8d	86	67	5f	65	5c	77	**2b**	**2b**	**33**	**2c**	33	31	2e	2c
3f	**31**	**41**	**2f**	28	23	26	41	39	29	39	2e	2e	25	5c	67	9d	90	71	8c	49	61	79	7b	54	**28**	**26**	**33**	**37**	29	24	28	2a
37	**3d**	**34**	**32**	2d	28	37	3d	2a	31	2e	24	2b	63	6c	86	92	64	91	6a	71	58	39	6e	56	**29**	**28**	**30**	**36**	31	2a	2f	32
32	**33**	**29**	**36**	2a	34	35	2f	23	22	33	29	61	5a	7a	71	80	90	68	71	57	56	7d	5c	30	23	2a	2e	36	**3b**	**2b**	**28**	**2d**
3b	**2a**	**26**	**2e**	31	34	2f	31	55	1f	26	65	49	63	80	8e	**98**	70	2b	37	67	6d	73	3f	5a	2d	30	2f	36	**30**	**31**	**2c**	**25**
2d	**2f**	**2f**	**35**	33	31	2c	2d	75	1f	48	75	60	7a	67	9e	55	31	48	48	42	45	4b	42	96	3e	3c	55	32	**2b**	**33**	**32**	**24**
2d	**29**	**27**	**37**	29	28	39	38	3a	41	61	5e	56	85	90	53	30	40	49	28	41	3a	30	37	52	51	58	65	52	**25**	**2e**	**32**	**2c**
29	2e	37	2c	28	26	50	2f	31	41	2f	3c	79	7a	8a	31	4c	2b	20	2f	3c	5b	47	29	5b	7e	5b	52	7e	23	33	3b	34
30	40	37	2b	27	28	46	4f	33	31	2b	47	7b	76	43	5e	2a	3b	29	23	5c	96	32	34	56	44	40	80	7f	25	31	30	35
3e	32	43	40	2f	24	3d	6f	35	2a	2b	33	6a	61	43	2a	30	2d	38	62	95	6d	39	30	3f	47	7b	b3	b4	69	2c	32	3a
32	2b	46	31	3d	2e	2d	60	3c	2f	35	38	65	59	22	2b	63	54	87	6e	83	53	3b	3e	5d	80	9b	7c	73	a9	25	35	30
2e	38	67	2a	31	35	3a	35	58	3d	39	4a	71	27	24	85	93	7f	8e	59	7e	6b	59	42	72	6a	86	6c	3a	81	2a	2a	3c
2f	3b	37	28	2c	2b	45	2b	4f	56	3b	61	50	31	5d	6a	46	59	5d	8d	72	61	3c	63	81	93	50	58	35	33	21	27	44
37	49	2b	28	2e	28	4b	28	38	5d	3e	5d	25	55	56	4d	72	6b	74	7b	6a	5f	74	79	8f	6b	2b	69	29	30	28	26	3a
37	46	2b	27	27	23	3c	27	2d	5a	63	24	2a	55	41	52	60	79	61	81	83	64	7b	7e	8a	29	2b	78	25	25	46	20	24

Table11. Girl pixel values

Table 12 shows the selected range blocks with the modified pixel values in hello.bmp. This is after Hello world is hidden in hello.bmp.

48	71	31	2f	4d	43	3f	8a	5c	31	7d	5d	2d	31	31	30	45	4f	43	46	27	34	49	60	29	5e	49	79	39	**4f**	**56**	**53**	**60**
51	75	2a	4b	4a	4c	7a	65	29	39	83	7b	38	26	27	3c	59	4a	35	2c	29	37	5f	4a	57	82	7f	31	36	**3e**	**47**	**53**	**55**
50	69	25	5e	52	85	7f	31	25	38	7a	7f	33	48	37	3a	5b	40	58	27	4c	51	7c	53	76	76	2f	27	25	**2c**	**32**	**3d**	**4b**
54	70	38	56	68	92	53	29	29	31	64	4b	3b	3d	27	50	42	6d	3a	32	48	57	69	5f	6f	2a	25	1e	24	**2d**	**36**	**39**	**44**
3e	67	4a	40	8e	85	3d	2a	25	51	74	2c	41	21	30	50	6c	65	24	3a	43	68	6e	4d	35	**2b**	**29**	**29**	**29**	**31**	**31**	**34**	**33**
39	6a	42	53	82	65	37	28	51	63	77	4c	21	22	43	4a	86	2b	29	56	5e	78	41	2d	3d	**2a**	**2c**	**2a**	**29**	**30**	**31**	**31**	**30**
43	6f	3d	76	7b	35	27	24	64	64	6d	2d	1e	29	62	74	47	30	48	71	76	73	54	43	46	**2a**	**2b**	**27**	**28**	**2e**	**30**	**30**	**2c**
49	70	4a	7c	68	24	24	32	5b	75	4d	1f	3b	58	39	79	31	4d	64	62	7d	38	6f	61	35	**2b**	**2a**	**2a**	**29**	**26**	**22**	**23**	**25**
5d	69	75	64	4c	2c	21	5b	53	7d	20	1e	3d	34	72	38	4e	34	80	6a	64	65	6e	66	28	29	2d	2e	2a	**2b**	**2d**	**2e**	**2a**
64	69	77	74	49	3b	36	2e	69	41	45	2c	39	75	4d	57	32	53	78	70	84	94	65	3a	2c	35	21	2e	2b	**28**	**29**	**2d**	**2b**
56	6c	4f	5d	2f	2e	2b	34	66	26	2b	41	44	89	2e	2c	38	47	66	73	a4	7b	29	43	34	2e	23	2c	2d	**2c**	**2e**	**28**	**2c**
4d	64	54	3e	2a	2e	2d	5c	47	2c	31	2f	7a	3e	3e	42	59	55	47	8b	82	4e	28	49	53	27	26	28	30	**28**	**28**	**28**	**2d**
45	55	53	2a	2a	2e	2a	5b	47	47	3a	32	61	2e	45	33	4f	45	5c	9b	6b	58	24	4c	46	**30**	**2d**	**2e**	**2c**	**2f**	**2c**	**24**	**2c**
50	56	31	2e	31	26	34	57	58	33	32	51	2c	3b	39	6c	35	44	76	8b	37	53	5c	67	3b	**30**	**2f**	**2d**	**2e**	**2e**	**2e**	**2b**	**26**
45	51	35	32	30	24	42	54	37	2d	37	57	33	2f	46	5b	2f	54	46	77	48	77	8e	6b	2a	**30**	**30**	**2d**	**2d**	**2f**	**2a**	**2b**	**2a**
3c	3b	2b	2f	31	2c	41	37	23	3c	3a	40	3b	39	40	38	47	3e	49	7f	8e	80	62	57	41	**31**	**31**	**2d**	**2b**	**28**	**2a**	**2f**	**2d**
35	**32**	**2d**	**2b**	29	34	36	24	36	47	3b	3a	32	2a	31	43	43	81	83	aa	84	5a	54	73	8d	**34**	**28**	**2a**	**2c**	2b	2a	2e	2a
38	**35**	**31**	**2f**	29	32	27	2b	3b	2d	5b	34	33	30	23	4d	83	76	8d	86	67	5f	65	5c	77	**2b**	**2d**	**2f**	**2f**	33	31	2e	2c
32	**37**	**35**	**30**	28	23	26	41	39	29	39	2e	2e	25	5c	67	9d	90	71	8c	49	61	79	7b	54	**2a**	**2e**	**35**	**34**	29	24	28	2a
41	**44**	**39**	**33**	2d	28	37	3d	2a	31	2e	24	2b	63	6c	86	92	64	91	6a	71	58	39	6e	56	**2c**	**30**	**35**	**32**	31	2a	2f	32
35	**2c**	**2b**	**3a**	2a	34	35	2f	23	22	33	29	61	5a	7a	71	80	90	68	71	57	56	7d	5c	30	23	2a	2e	36	**33**	**2e**	**27**	**2c**
36	**2b**	**2d**	**39**	31	34	2f	31	55	1f	26	65	49	63	80	8e	98	70	2b	37	67	6d	73	3f	5a	2d	30	2f	36	**31**	**33**	**2b**	**21**
28	**2b**	**2e**	**39**	33	31	2c	2d	75	1f	48	75	60	7a	67	9e	55	31	48	48	42	45	4b	42	96	3e	3c	55	32	**2e**	**31**	**32**	**28**
28	**2b**	**2e**	**3a**	29	28	39	38	3a	41	61	5e	56	85	90	53	30	40	49	28	41	3a	30	37	52	51	58	65	52	**28**	**31**	**31**	**30**
29	2e	37	2c	28	26	50	2f	31	41	2f	3c	79	7a	8a	31	4c	2b	20	2f	3c	5b	47	29	5b	7e	5b	52	7e	23	33	3b	34
30	40	37	2b	27	28	46	4f	33	31	2b	47	7b	76	43	5e	2a	3b	29	23	5c	96	32	34	56	44	40	80	7f	25	31	30	35
3e	32	43	40	2f	24	3d	6f	35	2a	2b	33	6a	61	43	2a	30	2d	38	62	95	6d	39	30	3f	47	7b	b3	b4	69	2c	32	3a
32	2b	46	31	3d	2e	2d	60	3c	2f	35	38	65	59	22	2b	63	54	87	6e	83	53	3b	3e	5d	80	9b	7c	73	a9	25	35	30
2e	38	67	2a	31	35	3a	35	58	3d	39	4a	71	27	24	85	93	7f	8e	59	7e	6b	59	42	72	6a	86	6c	3a	81	2a	2a	3c
2f	3b	37	28	2c	2b	45	2b	4f	56	3b	61	50	31	5d	6a	46	59	5d	8d	72	61	3c	63	81	93	50	58	35	33	21	27	44
37	49	2b	28	2e	28	4b	28	38	5d	3e	5d	25	55	56	4d	72	6b	74	7b	6a	5f	74	79	8f	6b	2b	69	29	30	28	26	3a
37	46	2b	27	27	23	3c	27	2d	5a	63	24	2a	55	41	52	60	79	61	81	83	64	7b	7e	8a	29	2b	78	25	25	46	20	24

Table 12 Girl with hidden data pixel values

In table 12 (steganographic image), pixel values shown *in bold* are virtually identical to those in table 11 (original image). Also the resulting pixel values are reasonable, in that there is no obvious pattern between them, and the surrounding unmodified pixels.

7.1.2 Statistical Tests

The quality of the steganographic image file is all important. The image file with the hidden information must be as close as possible to the original. If the steganographic image file is not modified very much then our modifications can be confused with noise.

With our method only parts of the image will be changed. That is we modify small sub-regions of the image. We replace one block with a similar block. In this section we present details on the relative modification.

The first test that we did is the peak signal-to-noise ratio (PSNR) which we used to measure the difference between the original block and the modified block. It is defined as

$$PSNR = 20 \log_{10}\left(\frac{b}{rms}\right)$$

where *b* is the largest value of the signal (255 in this case) and *rms* is the rms difference between the two images. The PSNR is given in decibel units (dB). An increase of 20 dB corresponds to a ten-fold decrease in the rms difference between the two images.

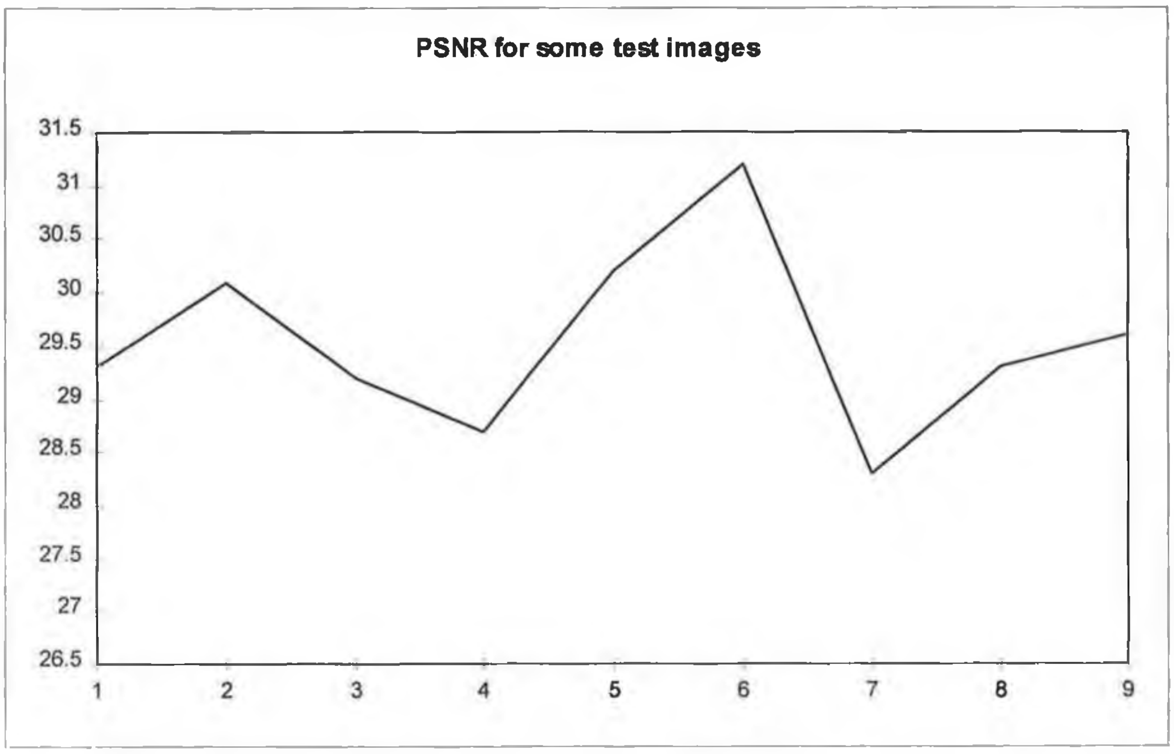


Figure 13. PNSR for modified range blocks

The graph shows that the PSNR is approximately 29.5 for the test images. This compares favourably with JPEG. In these tests we kept the maximum tolerance level (E) between the two blocks at 5.0.

We also tested the standard deviation between an original pixel and the one that is replacing it. A different stream of steganographic data was used in each test, and this was inserted into the same image.

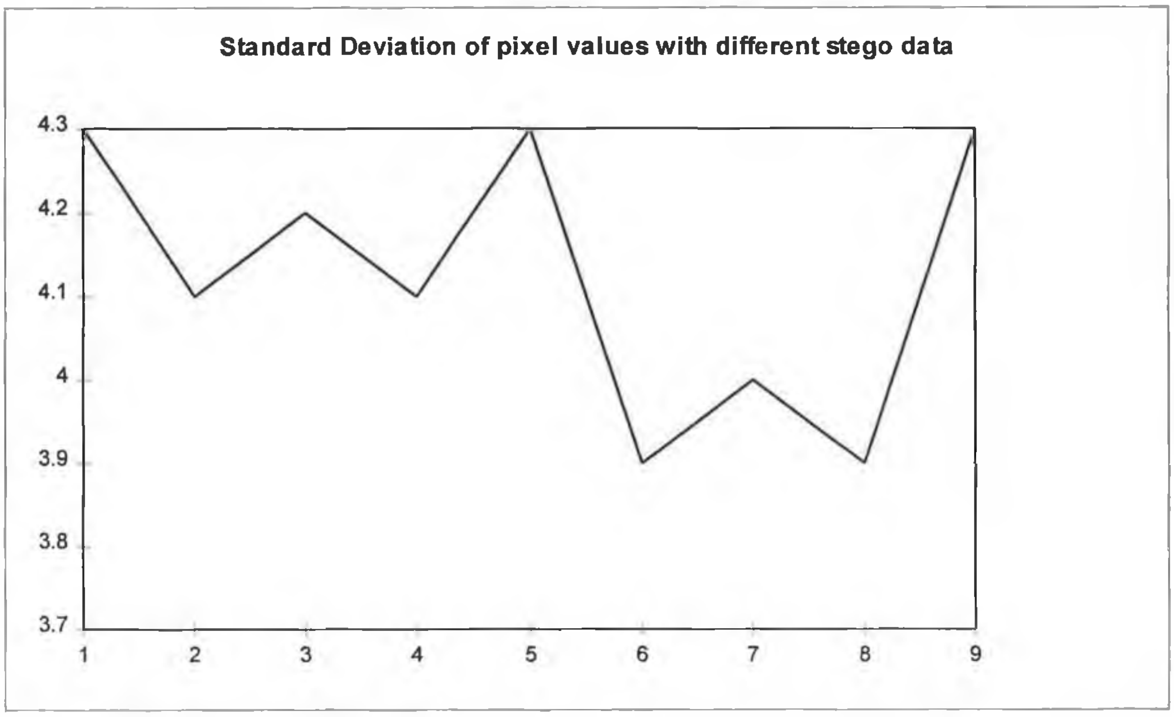


Figure 14. Standard Deviation of pixel values with different steganographic data

The average standard deviation was found to be 4.1. When different steganographic streams are to be hidden in the same image then different blocks will be selected for modification. The selected tolerance levels would be different. The higher the tolerance level the more the standard deviation for that particular block. This explains the graph.

We also tested the standard deviation between an original pixel and the one that is replacing it. We did this across a series of images with the same stream of steganographic data.

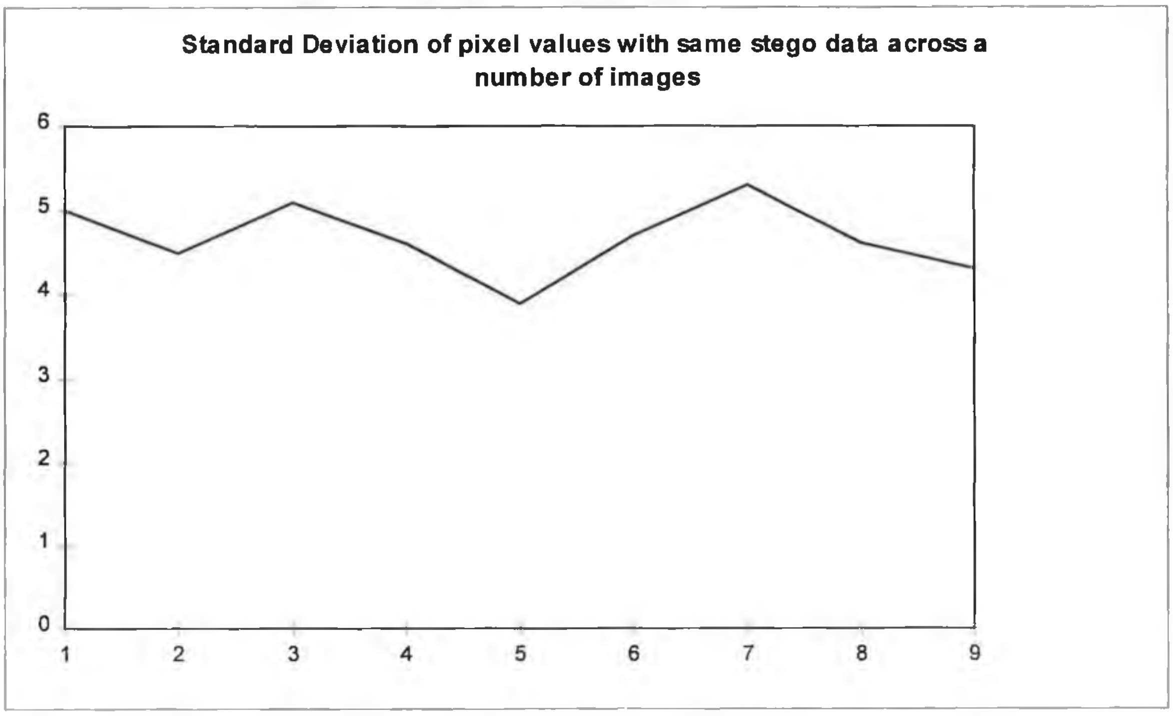


Figure 15. Standard Deviation across a number of images

The average standard deviation was found to be 4.6 in this case. This means that, if a pixel is modified by our algorithm, then on average ±5 levels of grey will be added to the pixel. This corresponds with adding noise into the image; noise normally effects the least significant 3 bits.

7.2 Data Hiding Ability

We show here the results of applying some standard statistical tests[49] to the resulting steganographic image files.

7.2.1 Run Length Encoding Tests

The first series of results show some run length encoding tests. The first series of tests applied to the data hiding was the number of consecutive occurrences of a 0 or a 1. On the

Y-axis is the number of occurrences, on the X-axis is the length of the run. For example we check how many times a run-length of 6 ones occurs in a row.

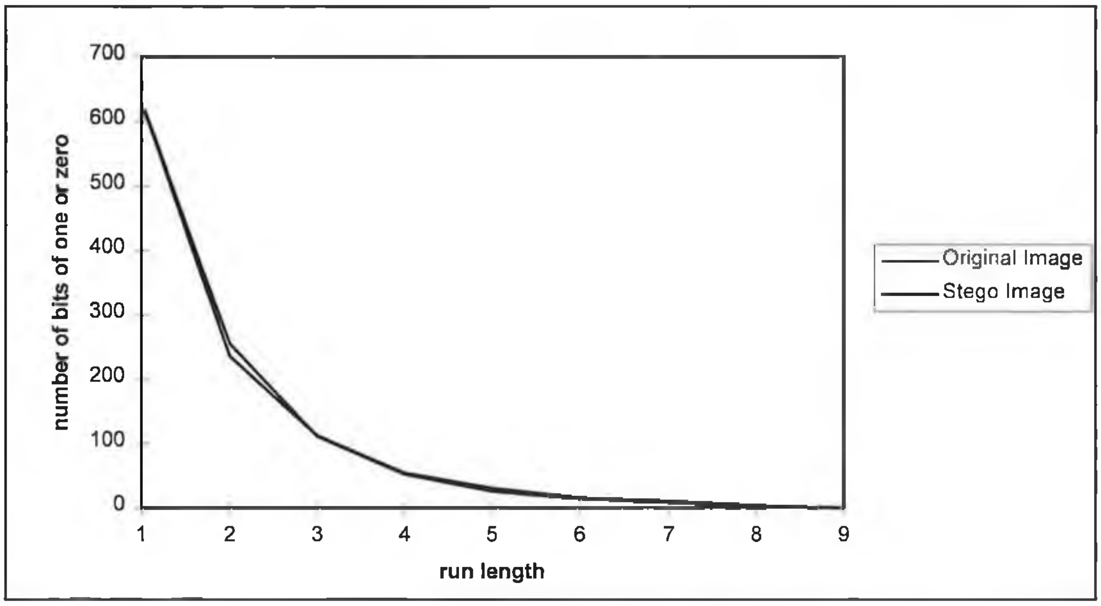


Figure 16A. Comparison of run lengths between original and stego images

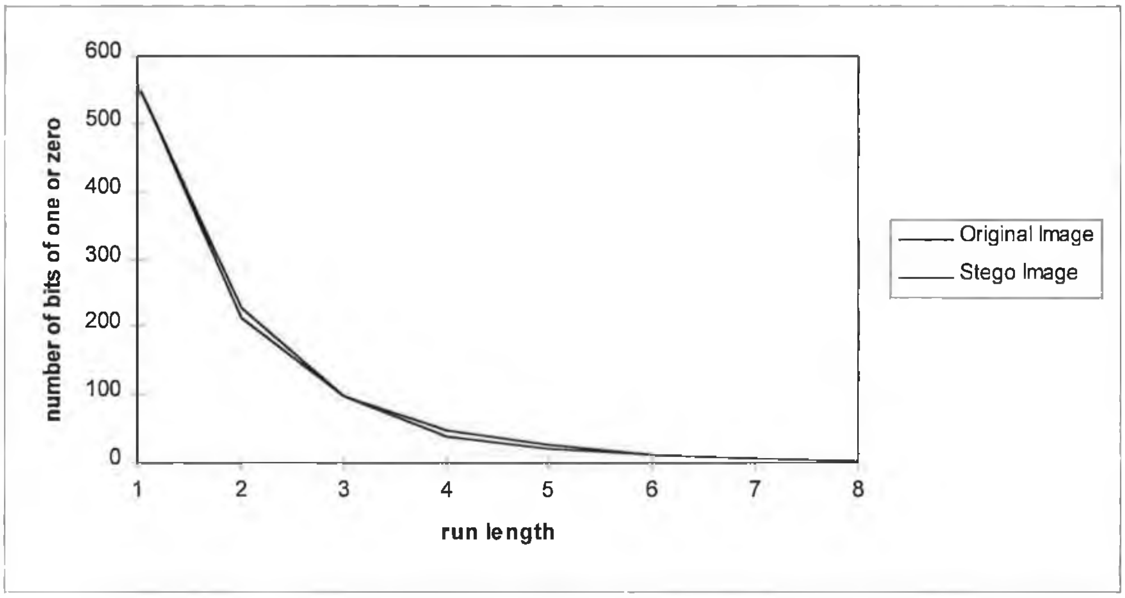


Figure 16B. Comparison of run lengths between original and stego images

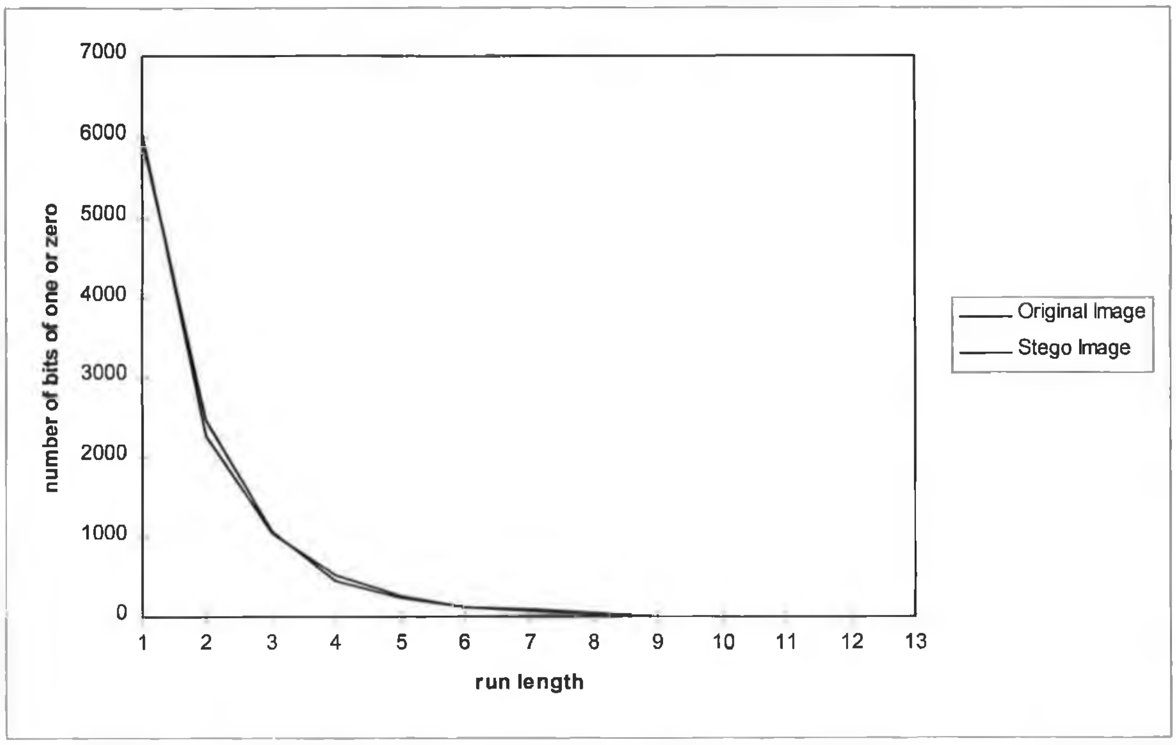


Figure 16C. Comparison of run lengths between original and stego images

In the above graphs the results for the original image and the steganographic image can be seen to be almost identical. So a cryptanalyst could not reliably use this test as a means of determining modification. Also this statistical test has been applied to the modified blocks. A cryptanalyst would not have this information. So his/her analysis would be restricted to the whole image or to portions of it.

The following tests graph the number of occurrences of values of 1-7 that occur in the original image file against the steganographic image file.

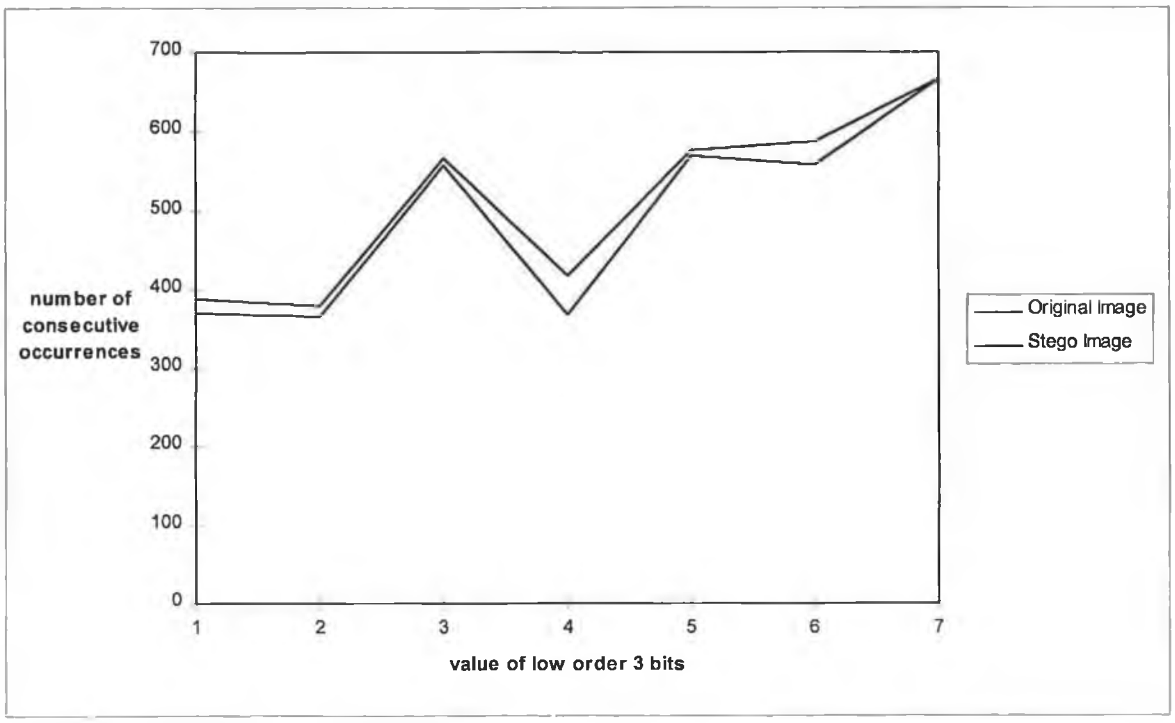


Figure 17A. Consecutive occurrences of values in modified pixels

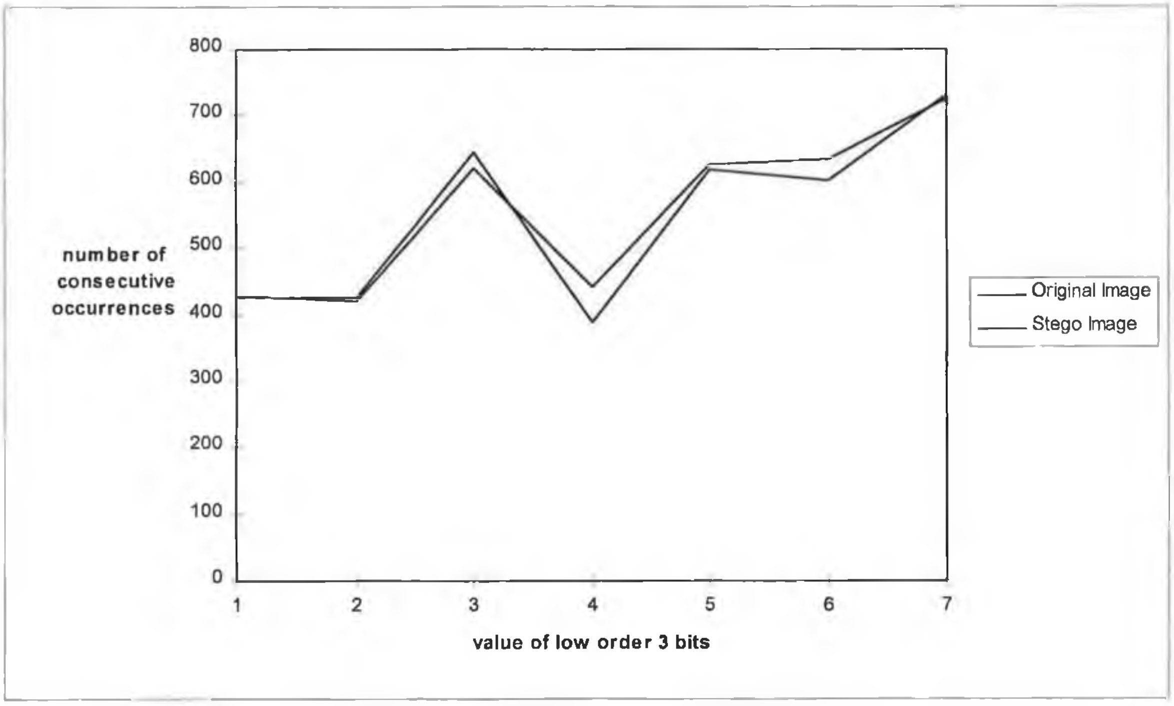


Figure 17B. Consecutive occurrences of values in modified pixels

In the above graphs the results for the original image and the steganographic image can be seen to be almost identical. So a cryptanalyst could not reliably use this test as a means of determining modification. Also this statistical test has been applied to the modified blocks. A cryptanalyst would not have this information. So his/her analysis would be restricted to the whole image or to portions of it.

7.2.2 Modified Pixel Variations

In the following tests we look at how much the pixels are modified. Here we are trying to determine how far away from the original pixel is a modified one. We graph the original image against the steganographic image file. The graph shows the pixel values in the modified blocks plotted against their original values.

These tests will help to determine if there is major changes in pixel values from the original values. We don't want major changes as this would be noticeable by a number of means. For example a cryptanalyst could graph the pixel values across a line of pixels in the image. These lines can be diagonal, vertical etc. The cryptanalyst could then watch for major deviations in say one pixel value in the line. This may then point the cryptanalyst to a particular portion of the image.

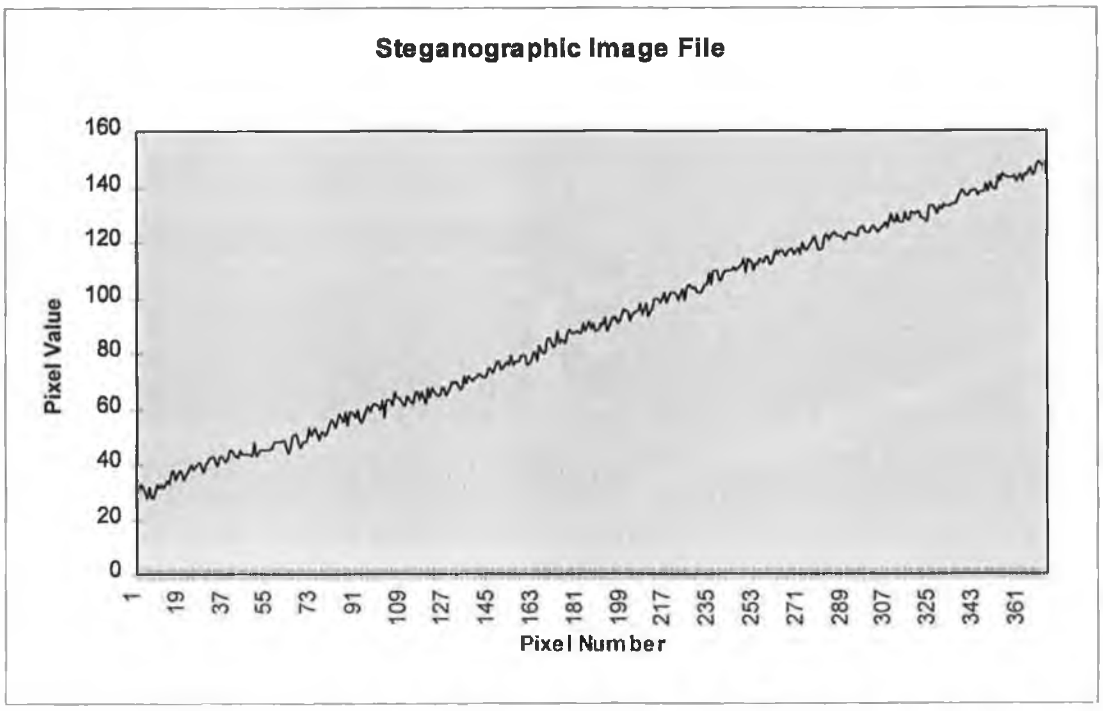


Figure 18A. Modification of Pixels in Stego Image

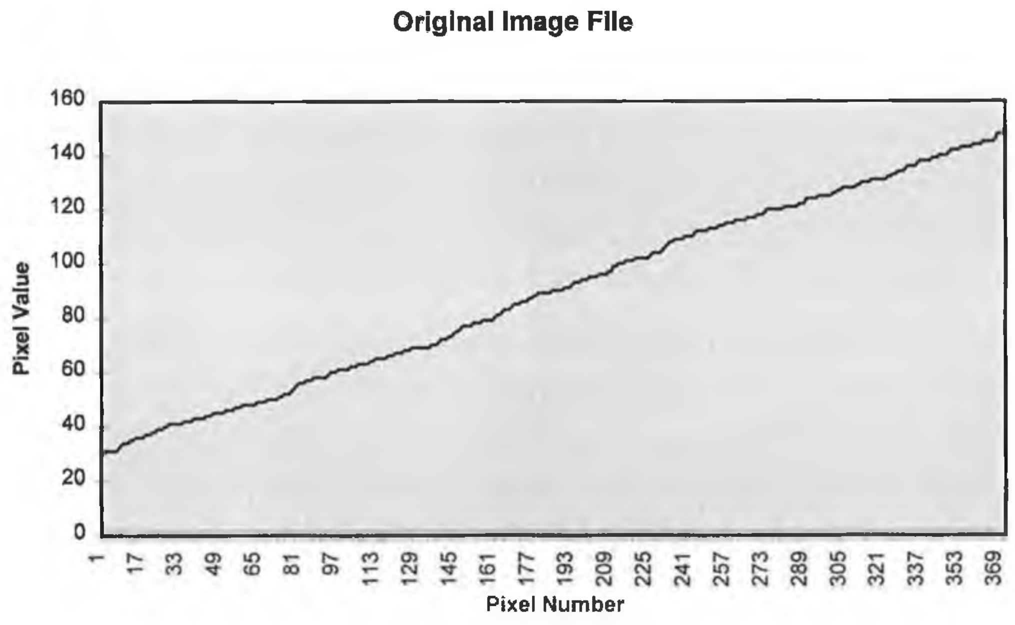


Figure 18B. The Original Image file

There is no pattern of modification, that is the modifications appear random, and as such will be confused with noise. We found that there were no outliners, that is pixels whose value varies widely from surrounding pixels. We are using the modified blocks in this test, a cryptanalyst would not have this information available.

7.3 Variation in Pixel Values in Lines in the Range Region

In this section we take lines of pixels at random across a modified range block, and check for major discrepancies between pixel values. These tests are again, a standard means of determining data hiding ability.

We will take scan lines for simplicity. We plot the pixel values for a portion of a scan line.

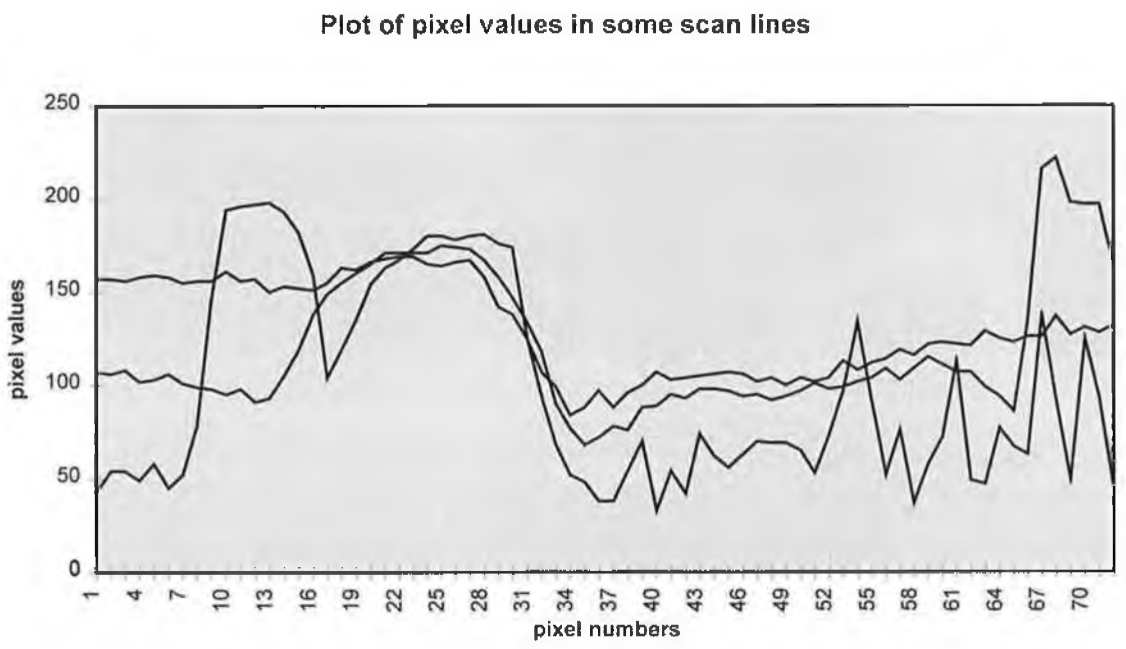


Figure 19A. Plot of pixel values in some scan lines of stego image file

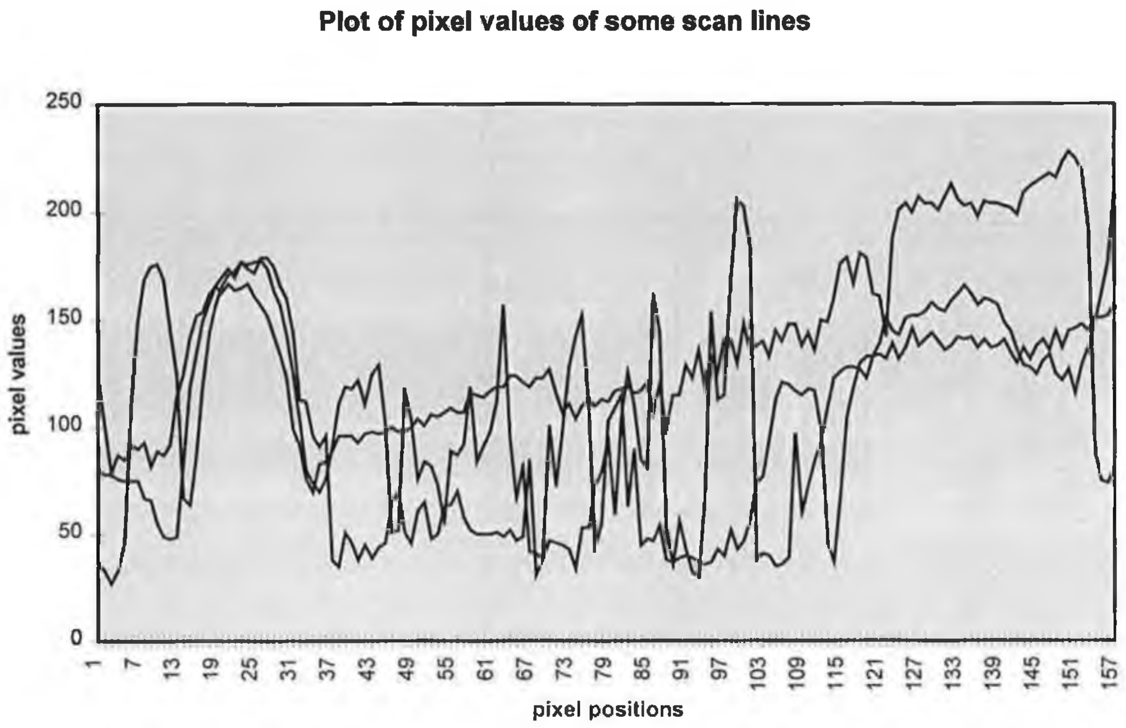


Figure 19B. Plot of pixel values of some scan lines of stego image file

We found no major discrepancies between pixel values in our tests.

7.4 Robustness

In this section we present the results for the robustness of our algorithm. We test the ability for the steganographic data to survive corruption.

7.4.1 Minimum value for linear correlation coefficient

On retrieval we will be testing for a linear relationship between a given range block and a domain block. Each domain block in the domain library has to be checked. For each match we determine the linear correlation coefficient *r*. If *r* is greater than some minimum value we say a match has occurred. In any given image when comparing blocks of the image in the manner required by our algorithm values for *r* can be high for any two such arbitrary blocks.

We want to set the minimum value for *r* for which a match is said to exist. We would like this minimum to be as much less than 1 as possible. We want this because in a multimedia environment the liner relationship will be skewed due to compression etc in a multimedia environment. We determine the minimum value for *r* (we call it $corr_{min}$) by taking a set of images and matching blocks at random and so determine values for *r* which could occur during information retrieval.

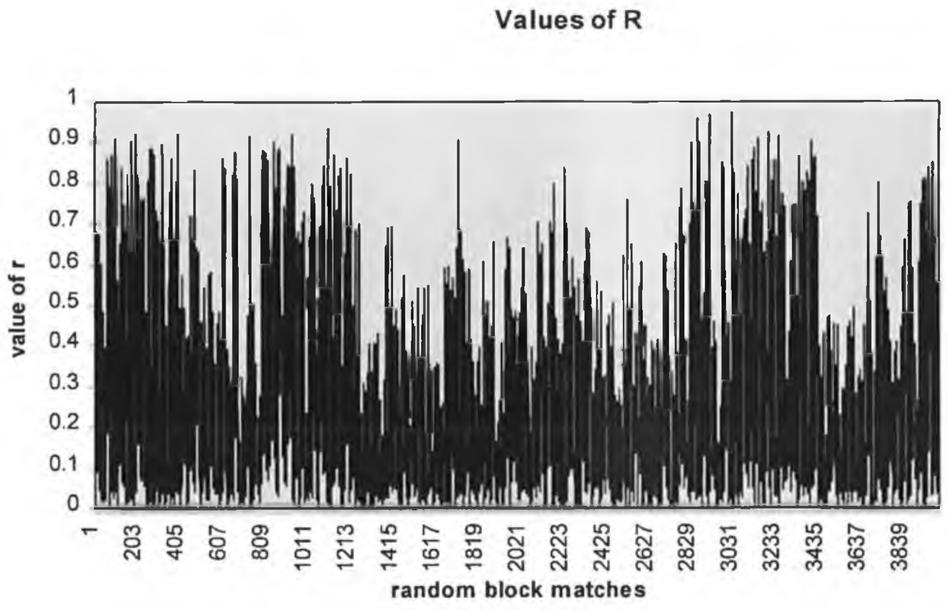


Figure 20. Calculation of minimum match correlation coefficient

For the experiments the minimum value for $corr_{min}$ was determined to be 0.96. However it is still possible to match two blocks at random and get a value for $corr_{min}$ greater than 0.96. We will deal with these blocks in Chapter 10.

7.4.2 Adding Noise to the Steganographic Image File

In these tests we add noise into the steganographic image file. In the following results noise was added to the whole image. The noise added was plus or minus one to a random pixel value. A particular pixel value could be modified several times. The values for *r* were determined for the modified range blocks and their associated domain blocks.

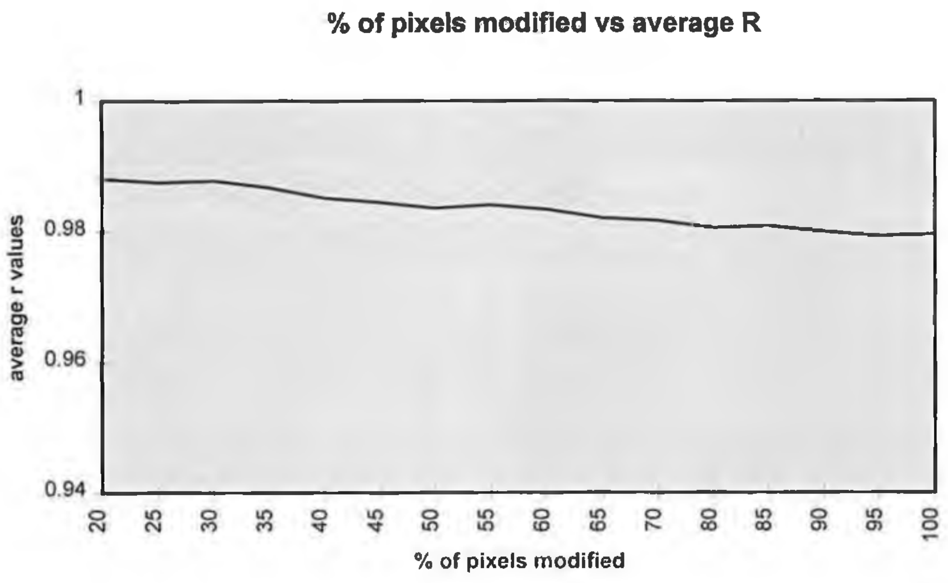


Figure 21A. Plot of average r against % modified pixels

In this test we found that the *r* values will remain greater than the minimum correlation coefficient.

In the next test random pixels were modified by plus or minus 3. The percentage of modified pixels by this amount was plotted against the average *r* of the modified range blocks and their associated domain blocks.

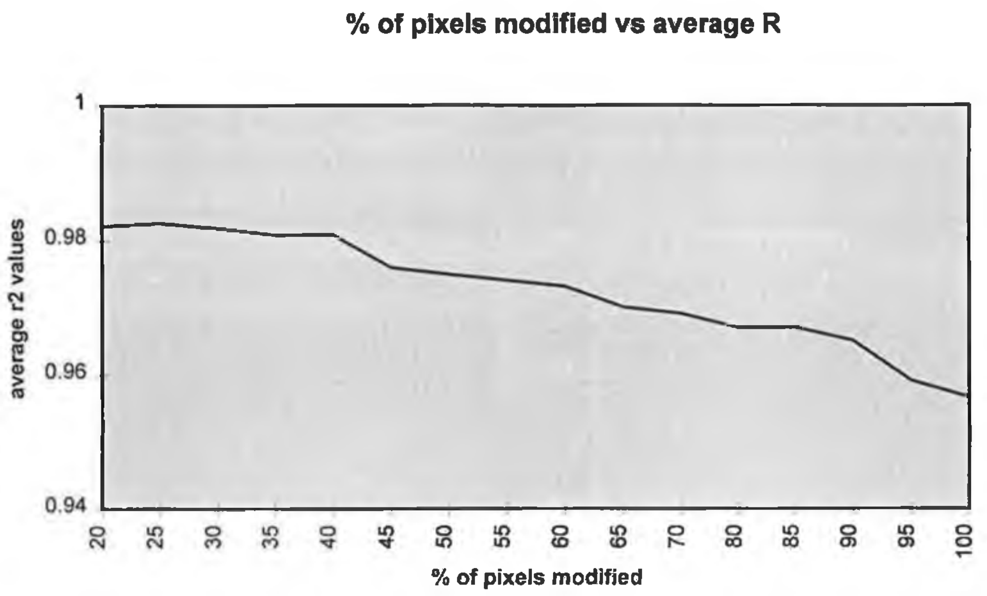


Figure 21B. Plot of average r against % modified pixels

In this test it was found that the average correlation coefficient for the modified range blocks will slip below $corr_{min}$ when 95% of the pixels were modified.

7.4.3 Adding Noise to the Steganographic information

In these tests noise was added to the modified range and their associated domain regions. The steganographic data was retrieved. The x-axis in the following graphs indicates the number of bits modified in the domain and range regions. The y axis indicates the value of r. If the value for r remains greater than $corr_{min}$ then we will be able to successfully retrieve the hidden information.

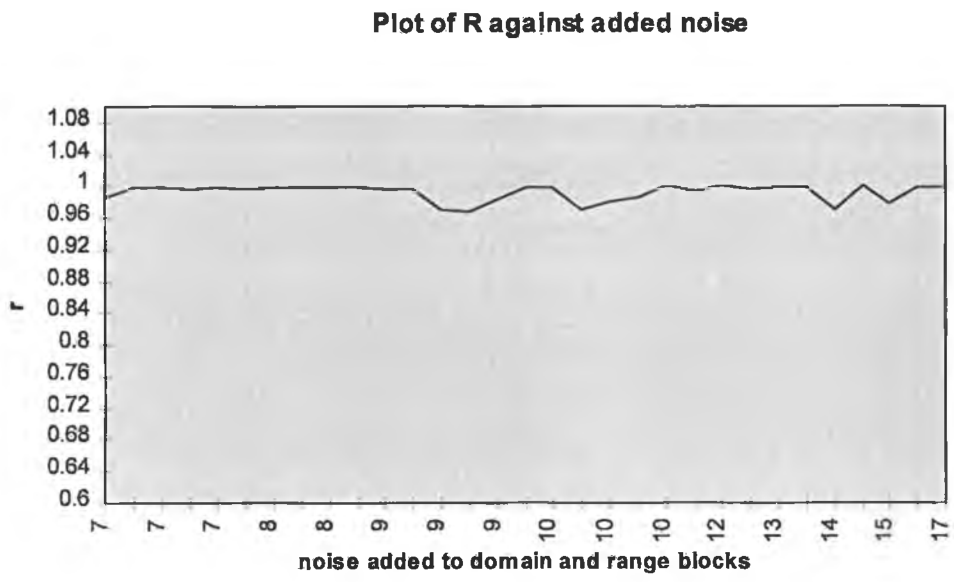


Figure 22A. Plot of r values against noise added to pixels of modified range blocks

In figure 22A the steganographic data was successfully retrieved.

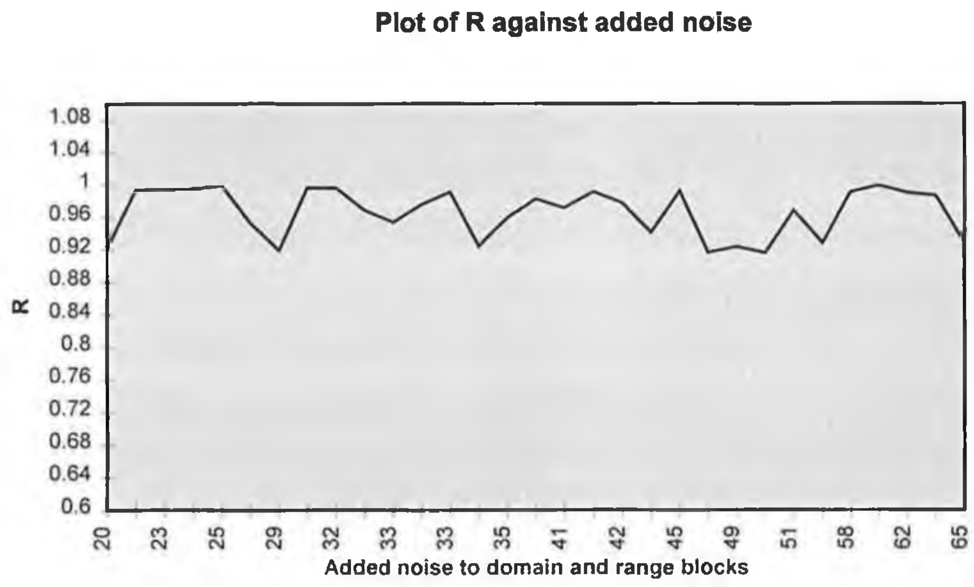


Figure 22B. Plot of r values against noise added to pixels of modified range blocks

In figure 22B the steganographic data was not successfully retrieved when the minimum match correlation coefficient was kept at 0.96. If the coefficient was reduced to 9.1 the modified ranges were successfully found, however this caused 5% spurious matches to occur. That is we retrieved 5% extra bits in error.

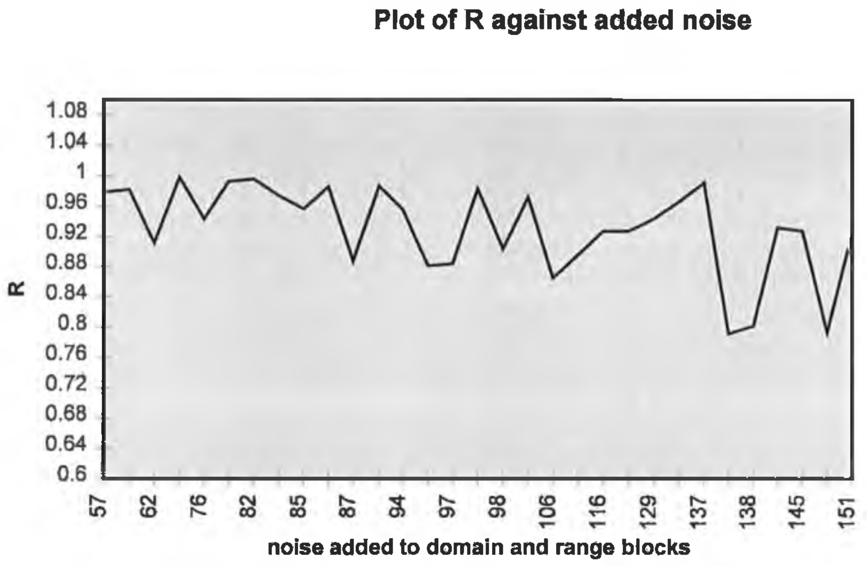


Figure 22C. Plot of r values against noise added to pixels of modified range blocks

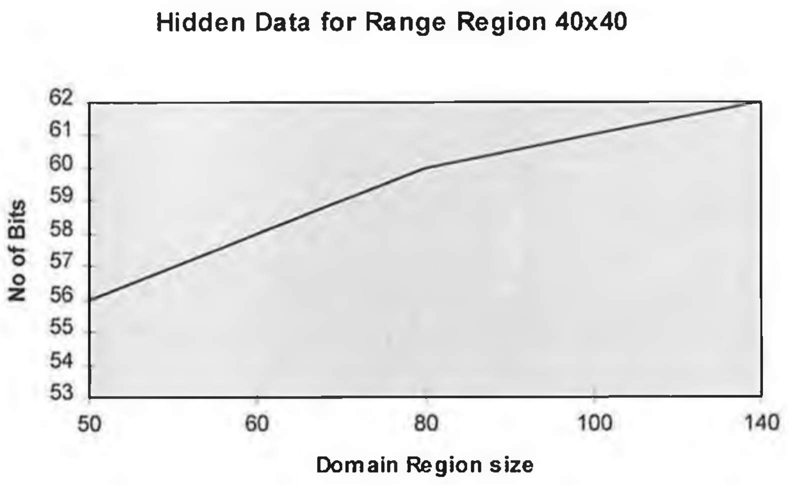
In figure 22C the steganographic data was not successfully retrieved. Reducing the minimum match value of r to 0.9 yielded a 75% success rate of bit retrieval. Reducing the minimum match value ($corr_{min}$) to 0.7 caused the steganographic data to be retrieved successfully. However there was a 25% extra bits retrieved due to spurious matches of range with domain blocks.

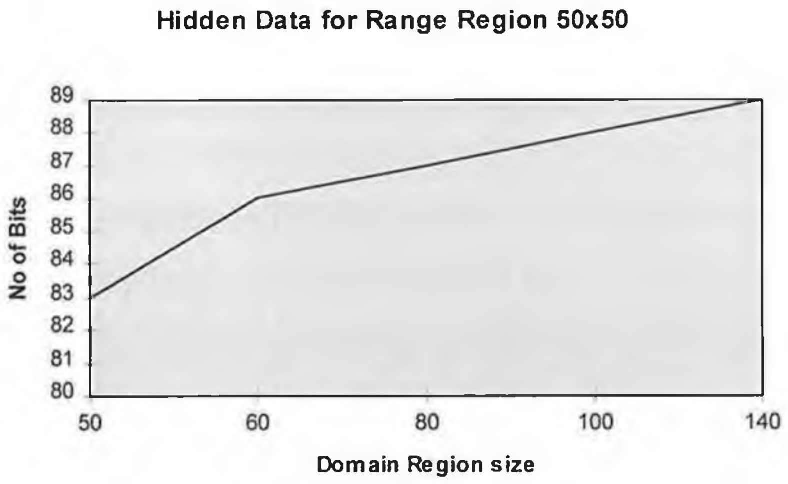
7.5 Data Hiding Bandwidth

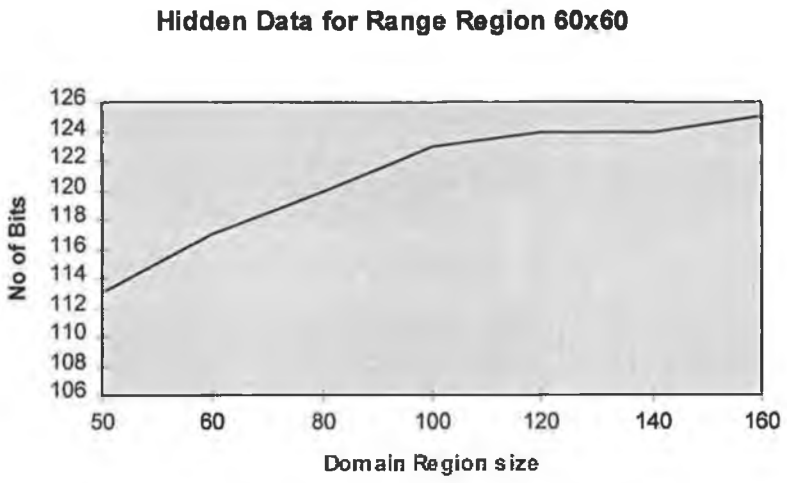
In these tests we measure the limits of data hiding bandwidth. In these tests we attempt to measure the amount of information that can be hidden in a typical range using a typical domain region.

The amount of information that can be hidden depends on the size of the domain and range regions.

The following graphs show the relationship between bandwidth of data hiding and the domain and range region sizes. In each of these graphs the domain and range regions are square shaped.







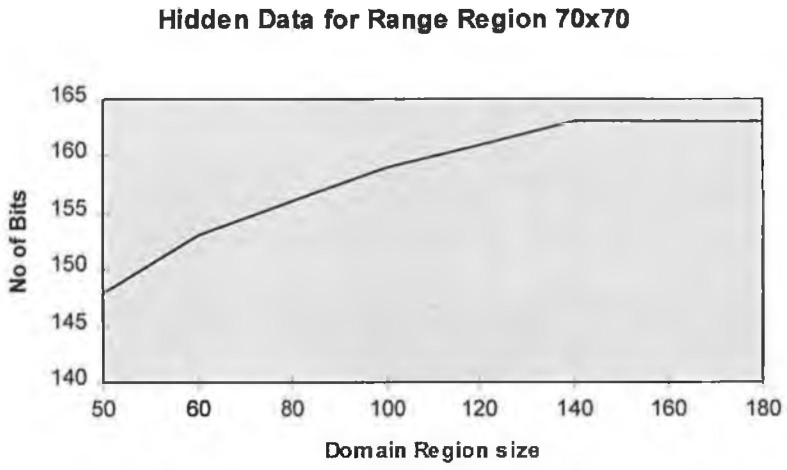


Figure 23 Plot of Range Region Bandwidth against Domain Region size.

The results show that:-

- for a range region size there is a point at which the region gets saturated for data hiding. This is because there are a limited number of range blocks in a given range region that we can use.
- there is a steady increase of bandwidth against range region size.
- we can hide approximately 0.035 bits of hidden data per pixel in the range region.

We tried to apply Shannon's information theory, to determine the theoretical limits on the data hiding bandwidth (See Appendix B for details). However we found that this theory is only applicable to a data hiding method that operates in the frequency domain.

7.5.1 Visual Effect of Increasing Bandwidth

The next test is to show how the steganographic image is visually effect by increasing the bandwidth of hidden information. The range region is 70x70 pixels and the range blocks are 3x3 pixels. The following table lists the various figures and the amount of hidden information is in each.

	# of Bits of Hidden Information	% of range blocks used
Figure 25	**56**	**9**
Figure 26	**131**	**21**
Figure 27	**199**	**32**
Figure 28	**245**	**39**

Table 13 Amount of hidden information in the figures.

We first show the original range region and then the same region after inserting various amounts of hidden information.

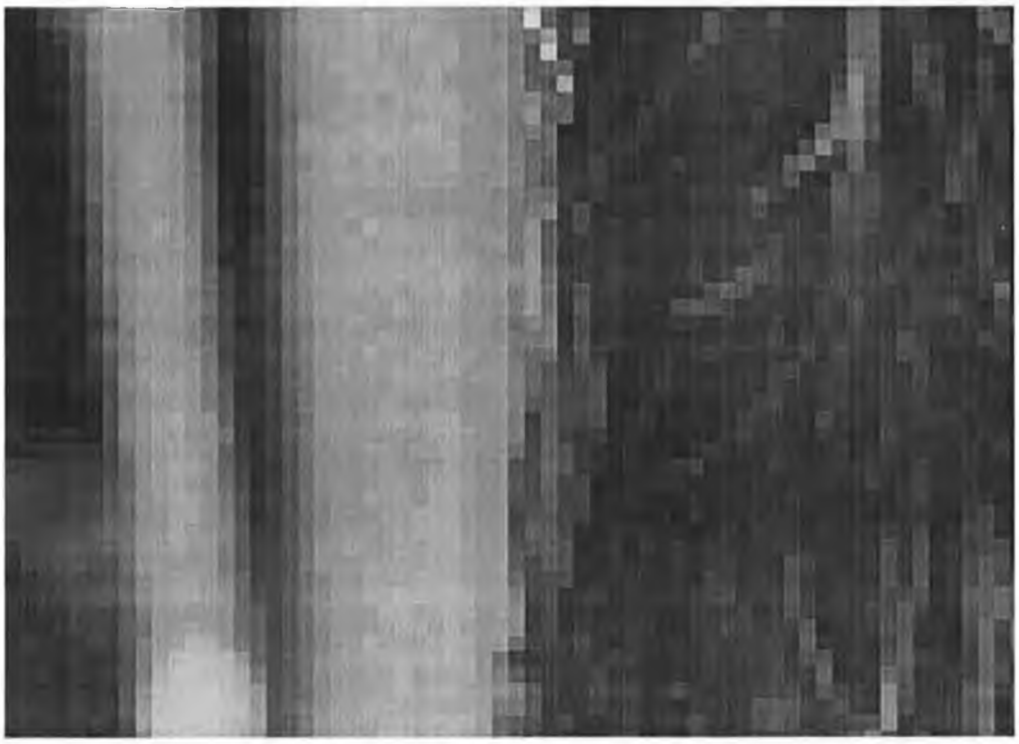


Figure 24 Original Range region before data hiding

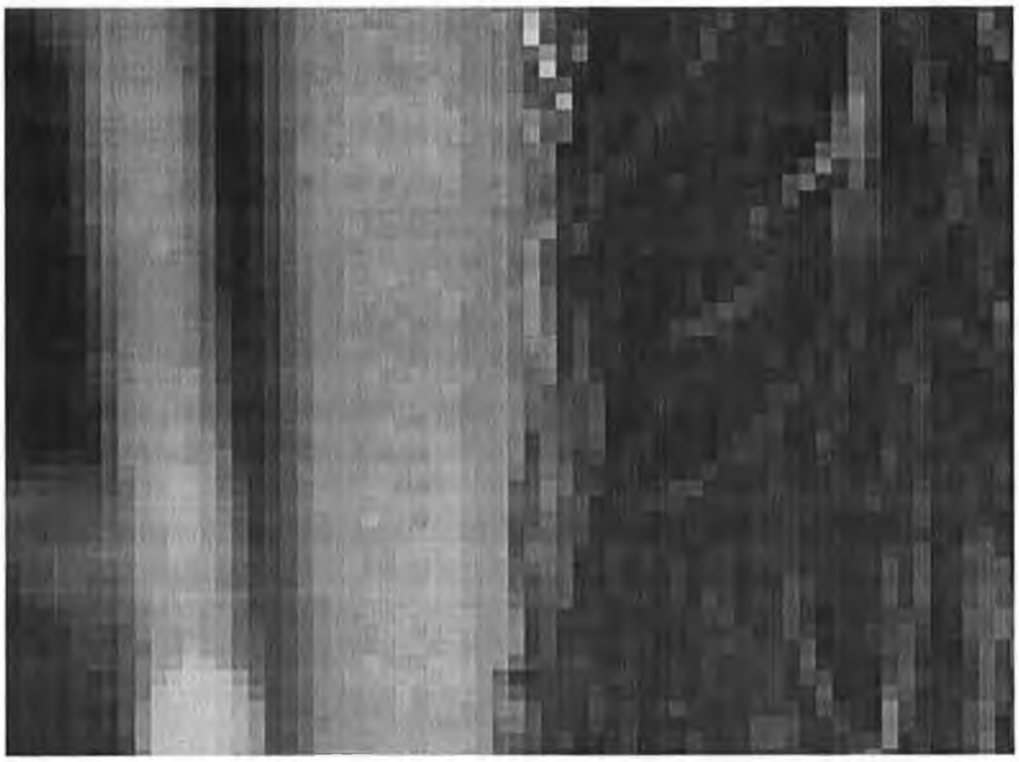


Figure 25 Range Region after hiding 56 bits of hidden information.

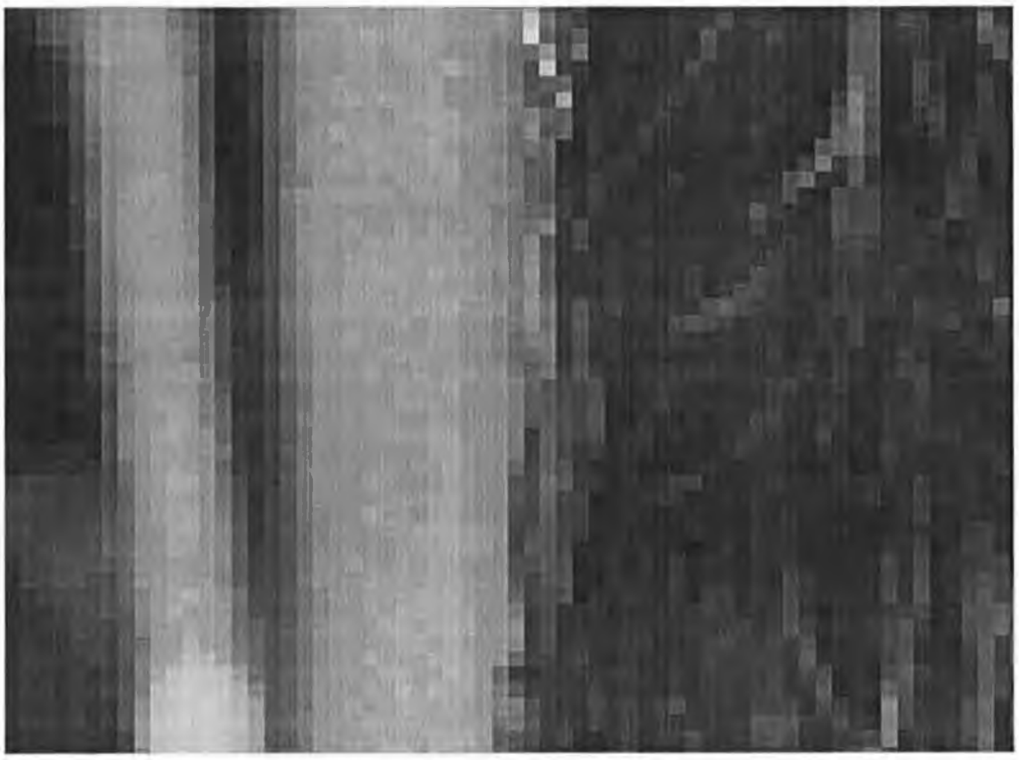


Figure 26 Range Region after hiding 131 bits of hidden information.

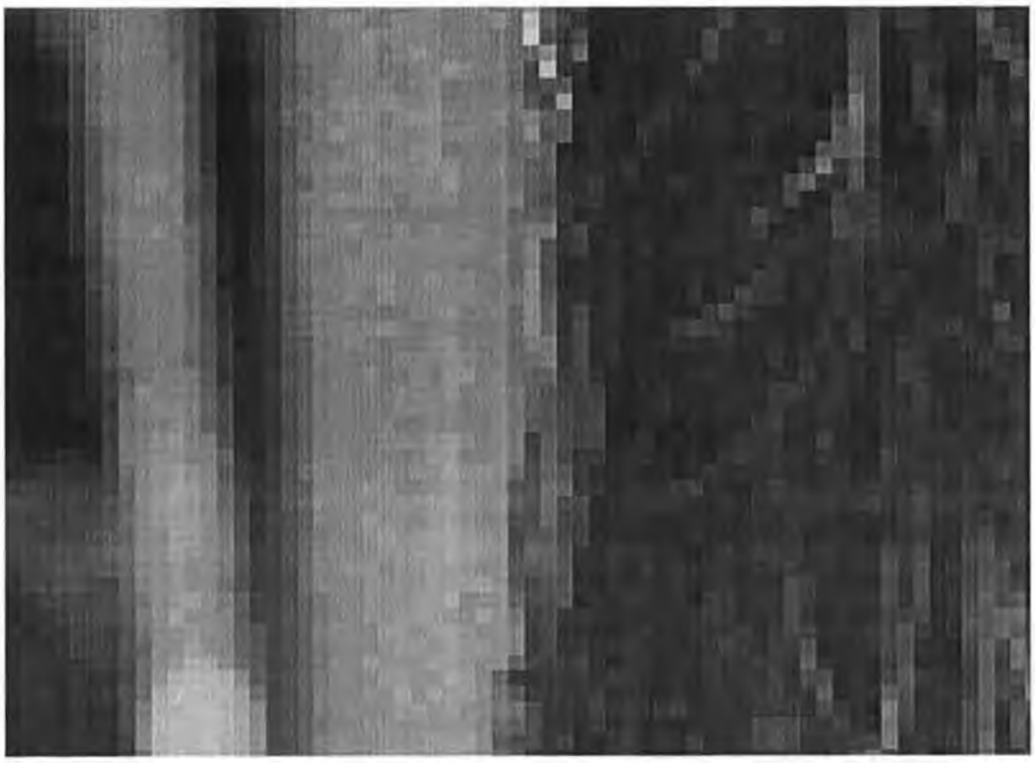


Figure 27 Range Region after hiding 199 bits of hidden information.

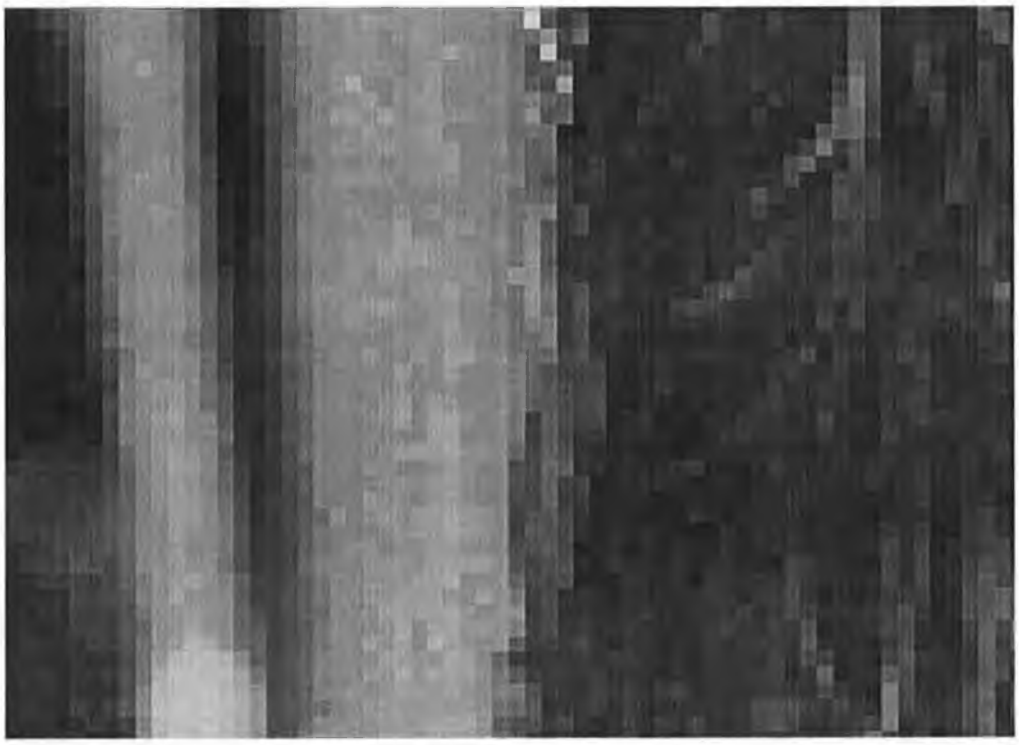


Figure 28 Range Region after hiding 245 bits of hidden information.

The results in this test show that visually there is little difference between the range regions. However the more modifications that are made to a region the more likely that the hidden information will be discovered.

Chapter 8. Security Enhancements

In this chapter we discuss the security of the method. We discuss the ability of the method to stand up to cryptanalysis, and keep the hidden message secure. We look at various enhancements to the basic algorithm to increase the security of the method.

In order to retrieve the hidden information a cryptanalyst has to discover:-

- The correct positions of the range and domain blocks in the image.
- The correct shape of the range and domain blocks within the region.
- The correct order of blocks selected during hiding.

We make the following assumptions:-

- The cryptanalyst knows that our method has been applied to the image.
- In trying the range and domain regions that the cryptanalyst does not incorrectly identify a hidden message. This in fact can very easily occur especially if we relax our $corr_{min}$ value. The cryptanalyst can legitimately discover matches between blocks in chosen range and domain regions.

  We could further confuse the cryptanalyst, by deliberately placing bogus messages into the unused portions of the image. We discuss this ploy later in this chapter; these bogus messages could be very short. If the steganographic information is encrypted, before being hidden, then these spurious messages will be a major problem for the cryptanalyst; Because each message that is found, has to be validated before being tossed away. We can place many of these bogus messages into a typical image. These bogus messages usually cause no reduction in bandwidth; because the algorithm will chose approximatly, the same set of range blocks to modify, in any selected range region.

- We have not taken into account the correct adjustment of the $corr_{min}$ value by the cryptanalyst; this is very important. As if too low a value is chosen extra bits of hidden information will be retrieved, too high a value and bits will be lost. We assume that by luck the cryptanalyst chooses the correct value for $corr_{min}$.

8.1 Selecting Domain and Range Regions

The method allows part of an image to be chosen as a range region and part to be chosen as the domain region. A user could then implement further protection on the steganographic data by selecting known parts of the image for range and domain regions. These regions would form a kind of key.

For example the user marks a square part of the image as the range region I_1 and another square part of the image as the domain region I_2 . The data hiding algorithm would then proceed as described in chapter 6.

When retrieving the steganographic data, the user specifies the correct initial squares for the range and domain regions. There are many possible combinations here. For example in a 256×256 image with the range regions being 32×32 and the domain region being 32×32. There are 225*225 = 50625 possible range regions of this size. The domain regions cannot overlap the range regions. Therefore in this case there are 193*193 = 37249 possible domain regions for each of the 50625 range regions. This yields a total of 1,885,730,625 possibilities.

Some regions need not be searched because they could be background. But in practice the background of a typical image may not consist of constant values. If a noise filtering algorithm (such as the median filtering algorithm[50]) has been applied to the image then this image is not suitable for data hiding. Noise in an image causes slight variations in pixel values, in a region that should have constant values. A noise filtering algorithm typically causes these regions to become constant in terms of pixel value.

Also the numbers given are for a grey scale image of 256x256. If we had an image of 1024x1024 this would yield a lot more possible regions.

However the domain region does not have to be the same size as the range region. Also the regions themselves don't have to be square. For example we could have a range region of size 21×43 and a domain region of 31×46. Therefore a cryptanalyst would have to try all domain and range regions of size say 32,33,34, and so on.

When we try polygonal regions the number of combinations increases. For example we could have 3,4,5 sided regions.

The major advantage of this pseudo key is that the key is visual. A user does not have to remember funny numbers. He/she simply remembers the range and domain region sizes and their positions.

8.2 Other Range and Domain block Partitioning methods

This section describes a simple update to the algorithm in chapter 6. The data hiding algorithm would work in the exact same way. It is just the underlying fractal method that changes.

The use of a more sophisticated algorithm than quadtree could be used as the basis for the method. For example the H-V[51] partitioning of the image. In this algorithm rectangles are used as the basis for the range and domain blocks. A range block is recursively partitioned either horizontally or vertically to form two new rectangles. The decision to split horizontally is based on the context. Figure 24 shows possible steps in building the range blocks with a H-V partitioning algorithm.

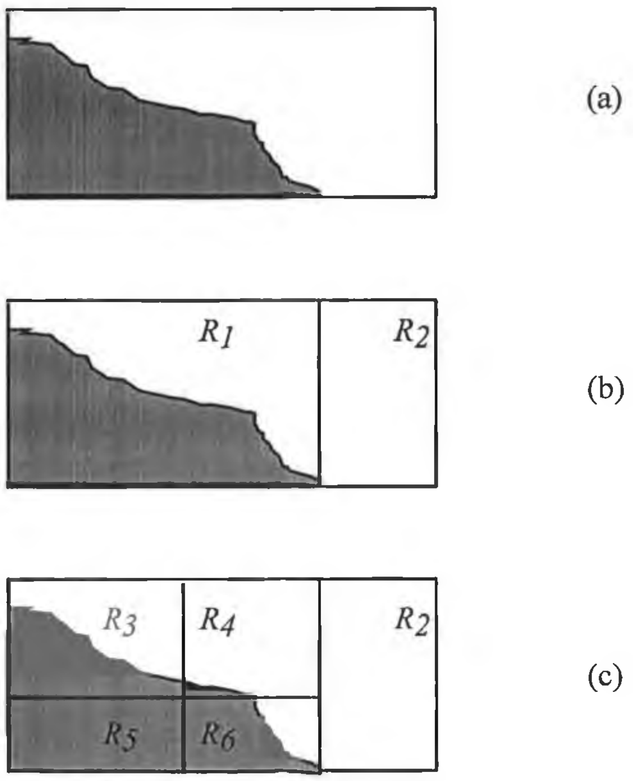


Figure 29. H-V Partition

The range and domain blocks are rectangular shaped. The use of this algorithm as a basis for the data hiding method would make it even more difficult for a cryptanalyst to find the hidden data. This is because user input could be used as a parameter in the decision as to the size of some of the rectangular blocks.

The triangular partition scheme would yield even better results if used as the basis for the data hiding method. In this method the ranges are built by splitting the image into two triangles. Each of these triangles is recursively subdivided into four triangles. This process continues until a range can be covered by a domain. The domain blocks in the domain library are also triangular.

8.3 Range and Domain block sizes

Within a range region of say size 64 we can choose to use ranges of size 3,4,5 or any combination of those sizes. The quadtree algorithm has domain blocks of twice the size as the domain region. The hiding algorithm would work for domain blocks smaller or bigger by various factors. For example domain blocks that are 1.5 times as big as the range blocks. All that is required is a consistent averaging function which brings the domain block size to that of the range region. It is even possible to have domain blocks that are say 0.5, 0.75,1.0,1.25,1.5 etc. times the size of the range blocks.

All of these factors could be inputted by the user to the hiding algorithm yielding a vast amount of combinations that would have to be searched by a cryptanalyst.

8.4 Distributed Blocks

It is possible to distribute a block throughout a region. This means that a block is made up a pseudo random set of pixels taken from different parts of the region. The only limitation required is that a pixel only belongs to one block. A range and domain blocks may be distributed. If this scheme of things was implemented then it would yield a vast amount of searches.

A pseudo random pixel selector would have to choose pixels for a block. This is so that the resulting pixels output by the algorithm would be close to the originals.

For example if we had a 32x32 region with block sizes of 4x4 with all blocks being used then we would have

$$1024*1023*1022 \ldots$$

possible choices for the pixels. Not all blocks could be used and not all pixel combinations could go together. For example, we can eliminate domain blocks that have no variation in pixel value; because if a domain block consists of mainly the same pixel value, then the algorithm will produce a new range block with constant values, these constant values will be in the same locations as those of the domain block. In general, this produced block will not be a good approximation to the original range block. So, to make our algorithm more efficient we should chose pixel combinations in an intelligent manner.

However this formula assumes that we know were the region is and the shape of the blocks in it. As well as that we have not included the domain region in our calculations. In the domain region the same pixels could be used over and over again meaning we get even more choices. However the more complex we make the hiding algorithm the slower it becomes.

Chapter

9.
Robustness in a Multimedia Environment

In general in a multimedia environment noise is added to the pixels in an image. This noise is spread in a constant manner over the image. In theory there is a trade off between robustness and security. To make the method more robust we must basically do more modification of the original image. The more modification we do the more the cryptanalyst will known that something is going on. So there is a trade off between robustness and security.

There is no problem with inserting the hidden information. But a problem arises during retrieval if the steganographic image is corrupted. In the first section we clarify what problems there are with our algorithm in a multimedia environment.

The conclusions of chapter 7, show that our method, is not robust enough to allow hidden information, to survive through multimedia processing. In fact any spatial domain information hiding method will have the problem. This is because a spatial domain method usually hides information into the low order bits of the pixels. The modification of these bits does not effect image quality. Image compression techniques such as JPEG, also exploit this fact by storing information about the high order bits of pixels, but ignoring the low order bits; this leads to compression.

A frequency domain information hiding method, has been proved to be more robust for allowing hidden information to survive through multimedia processing[52]. We therefore consider enhansing our method to include some frequency information to allow the hiding data to survive through multimedia processing. Several schemes were tried in line with the fractal method. Each will be described and results presented, if the scheme proved successful enough to bring to a proof of concept implementation.

9.1 Problems with the retrieval

In a multimedia environment the steganographic image file may be highly corrupted. This leads to a few problems which are related to this corruption. These problems must be solved in order to make our algorithm more robust.

- The linear relationship between the range block and the domain block will be corrupted. That is the steganographic information will be corrupted. In this case the calculation of the *linear correlation coefficient r* may yield a value less than $corr_{min}$. This would lead to us to miss a relationship between a range and a domain block. Therefore we would loose one bit of the steganographic data.

- On retrieval there may just happen to be a linear relationships between a given range block and a domain block. We don't want to assume that this relationship was placed in the steganographic image file by us. In this case the calculation of *linear correlation coefficient r* may give a value greater than $corr_{min}$. We would incorrectly assume that a match exits between the range and the domain blocks. Therefore we would retrieve an extra erroneous bit of steganographic data.

We will now describe a set of the major attempts that were made so as to assist in making our steganographic algorithm more robust. We have identified the following techniques to assist in the robustness of the algorithm. The results of each of these techniques are presented.

9.2 Destroying spurious relationships

During hiding we do a *linear correlation coefficient* test for unused range blocks. If this test yields a result close to our minimum match probability, we do a DCT and quantization phase of JPEG on the range block. Then we reverse the quantization and do an IDCT to create a new range block similar to the original. This new range block is stored in place of the original. We do this do as to destroy the spurious linear relationship between the original range and domain blocks. However we have to do this again in a loop because we may now have introduced another spurious relationship with a different domain block.

This method does not stop spurious relationships from occurring in a multimedia environment. When we do a linear regression on a block in the range region and a particular block in the domain region we may generate correlation coefficient of close to $corr_{min}$. This correlation coefficient would be close to $corr_{min}$ due to corruption of the pixels in the steganographic image.

9.2.1 Security

The security here would be much the same as the previous chapter. However we need to statistically analyse those blocks for which an approximation has been stored. The following graph shows some original pixel values taken from various modified range blocks and their modified equivalents.

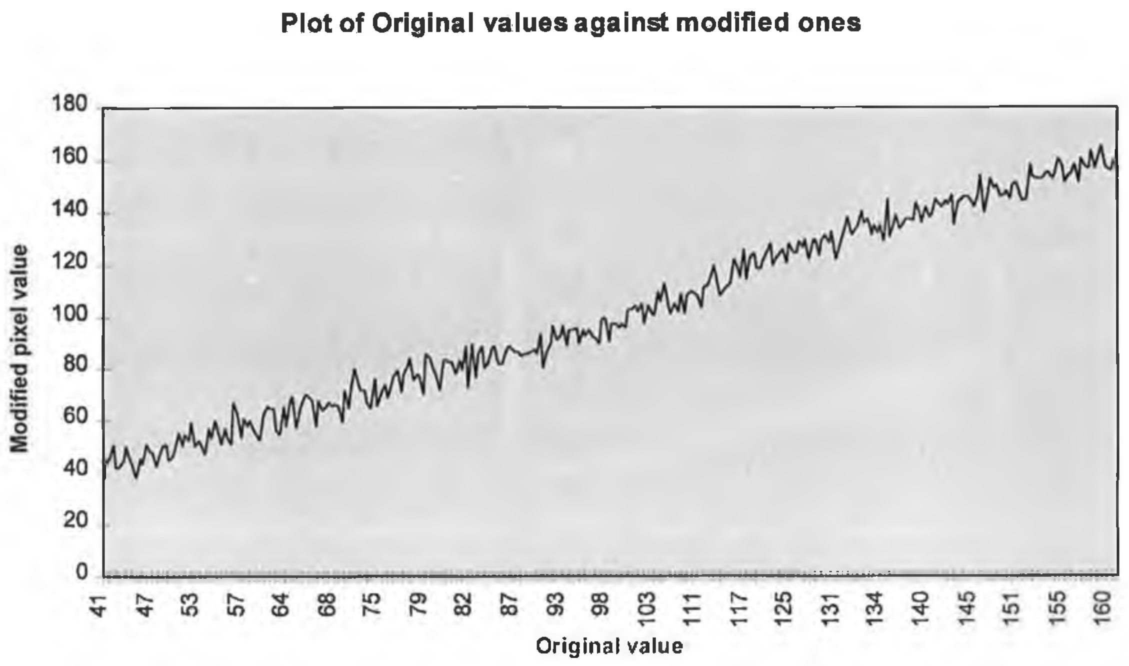


Figure 30. DCT method: Modified pixels vs originals

Our tests show that there is very little discrepancy between the original and the modified one. This discrepancy produces little by way of a signature that can be seen by a cryptanalyst. These modifications may guide the cryptanalyst away from the real hidden data blocks.

9.2.2 Robustness

Our tests show that this technique is more robust allowing us to reduce our minimum correlation coefficient to 8.9. However there is no guarantee in general that this minimum will be surpassed by a spurious relationship between two blocks.

In the following test which is similar to Figure 19C above.

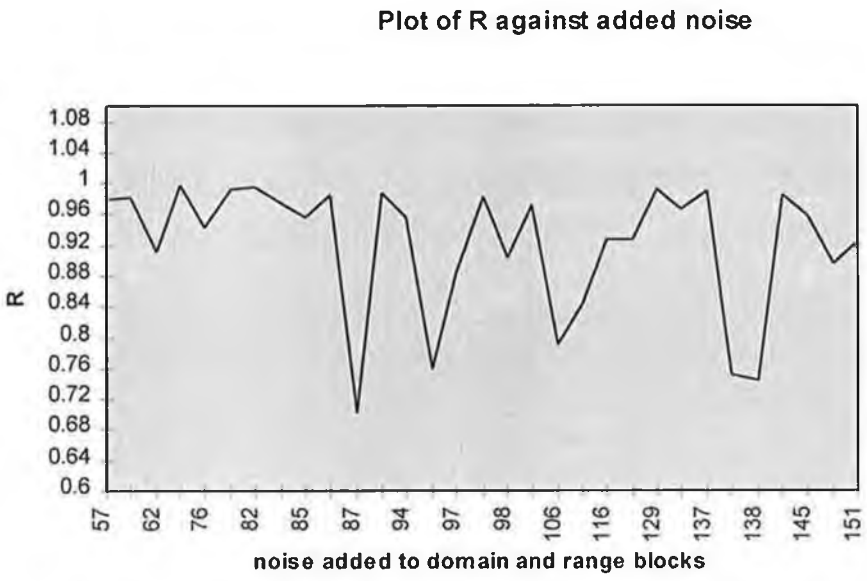


Figure 31. DCT method: Plot of r values against noise added to pixels of modified range blocks

The steganographic data was successfully retrieved. Even reducing the minimum match value of r to 0.9 yielded a 75% success rate of bit retrieval. Reducing the minimum match value ($corr_{min}$) to 0.7 caused the steganographic data to be retrieved successfully. However there was a 5% extra bits retrieved due to spurious matches of range with domain blocks. Without the use of this enhancement there were 25% extra spurious bits retrieved. Therefore the robustness of the method has been enhanced.

9.2.3 Conclusions

- This technique increases the robustness of the algorithm. It eliminates a lot of spurious relationships even in a multimedia environment. However it cannot guarantee that the hidden information will be successfully retrieved.

- The method proved not to be robust enough to support a corruption of more than ±4 in the pixel values. This would not be good enough to survive in a multimedia environment where pixel values could be modified by much more than that.

- This technique could be used along with others described to give a probability that an image is watermarked with a particular set of hidden information. That is given a set of possible watermarks[53] we could say that the image is marked with this particular watermark.

	Destroying Spurious Relationships
Robustness	It eliminates a lot of spurious relationships even in a multimedia environment. However it cannot guarantee that the hidden information will be successfully retrieved.
Security	• Extra JPEGed blocks could be noticed by cryptanalyst. • The cryptanalyst how has to pick the value for corr_{min}, which is a real number. So extra complexity for cryptanalyst.
Image Quality	Affects image quality because we are doing extra things to image.
Bandwidth	No effect.

Table 14. Destroying Spurious Relationship Results

9.3 Use larger range and domain blocks.

The hope here is that the use of larger range blocks will increase the robustness of the algorithm.

9.3.1 Security

- During hiding to use larger blocks we need to increase the tolerance. The larger blocks the more the standard deviation will increase between the steganographic image file and the original.

- It will be easier to spot the modified range block. When we produce an approximation to a large block there is more of an approximation taken and so more noise is introduced.

We will be able to store less steganographic information into a typical image. We could for example be modifying say 64 pixels to hide one bit of the steganographic data.

9.3.2 Robustness

- This will give more data points in the linear relationship.
- The more data points we have the more the error will be spread over a larger number of pixels and so give a value for *r* will be closer to 1.
- The more data points we have the less likely that a spurious relationship who's correlation coefficient is greater than $corr_{min}$ will exist between a range and a domain block.

The tests we tried were to insert noise into the range and domain regions in the same manner as in chapter 7.

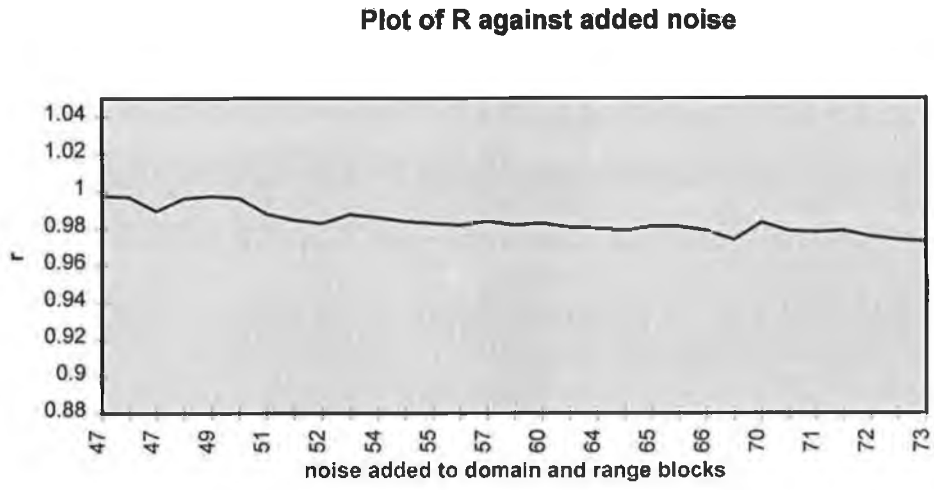


Figure 32. Large Range Blocks: Plot of r values against noise added to pixels of modified range blocks

The steganographic data was successfully retrieved. Here we show the results of using larger range blocks in our algorithm.

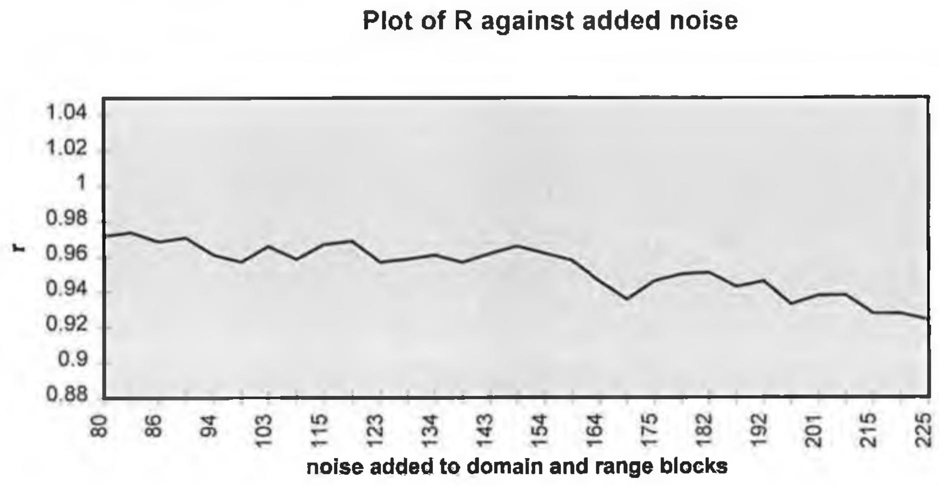


Figure 33. Large Range Blocks: Plot of r values against noise added to pixels of modified range blocks

The steganographic data was not successfully retrieved when the minimum match correlation coefficient was kept at 0.96. However if the coefficient was reduced to 9.2 the data was successfully retrieved.

9.3.3 Conclusions

- A very small amount of steganographic data could be hidden. In a typical image only a few bytes. In order to store more data it would visually become apparent that something was going on.

- The method proved not to be robust enough to support a corruption of more than ±3 in the pixel values. This would not be good enough to survive in a multimedia environment where pixel values could be modified by much more than that.

	Larger Range and Domain Blocks
Robustness	Easier to recover hidden data in multimedia environment.
Security	Larger modified range blocks could be noticed by cryptanalyst.
Image Quality	Effects image quality because we making bigger modifications to the range blocks.
Bandwidth	Bandwidth greatly reduced.

Table 15. Larger Range and Domain Blocks

9.4 Fractal based DCT

The algorithm of this section is as follows:-

- We transform the original image file into the frequency domain.
- We find matching domain and range blocks in a similar manner to chapter 3 except we are using the frequency coefficients instead of the pixel values.

- We quantizise the resulting frequency coefficients. That is we maintain the usefull frequency components and throw away the others.
- For a matching domain and range block we determine scaling and offset factors using the RMS metric. We don't use the full range and domain block for this match only the high order frequency coefficients.
- We build a new range block using the determined scaling and offset factors.
- We store this new range block in place of the original in the steganographic frequency coefficient file.
- We reverse quantizise the frequency coefficients.
- We use the reverse DCT transform to build a resulting steganographic image file.
- The retrieval algorithm does the reverse of this process.

This is entirely new work. As far as we are aware no one has tried to hide information in this manner. See the work of[54] [55] for techniques on which this idea is based.

We built an implementation of this algorithm to test its merits the results are presented in the next section.

9.4.1 Security

The security of this method was found to be excellent. We made the tolerance level less than 3.0 to produce blocks that were very similar to the original.

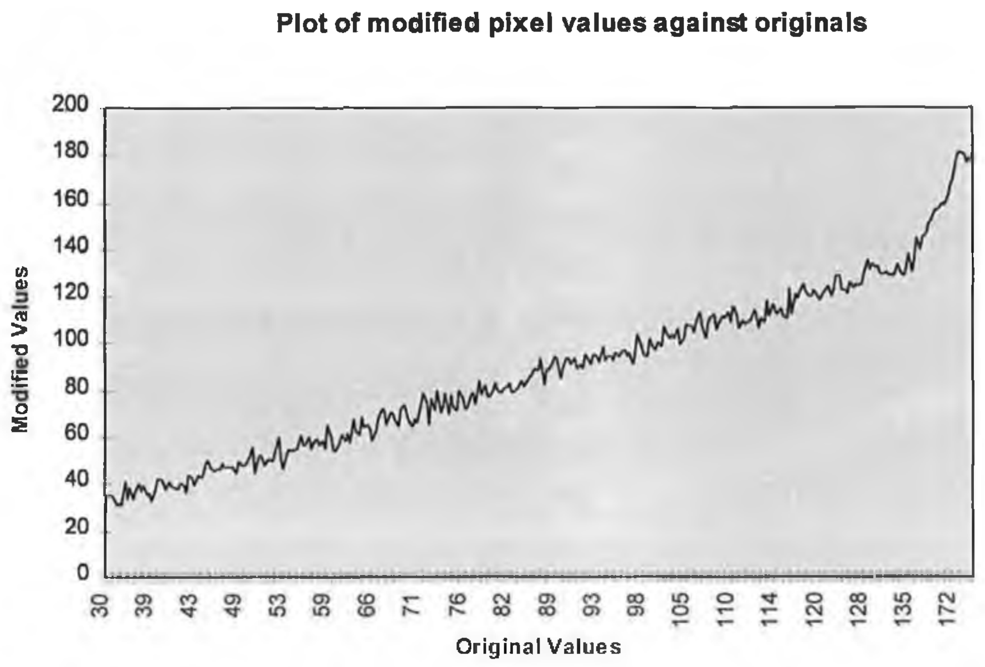


Figure 34. Fractal Based DCT: Modified pixels vs original values

The modified pixel values are very close to the original. The pixel run test were applied to the resulting image as well and there is no statistical reason to point a cryptanalyst at a particular modified block.

9.4.2 Robustness

We first of determined the minimum value for $corr_{min}$ for this technique by taking a set of images and matching blocks at random and so determine values for r which could occur during information retrieval.

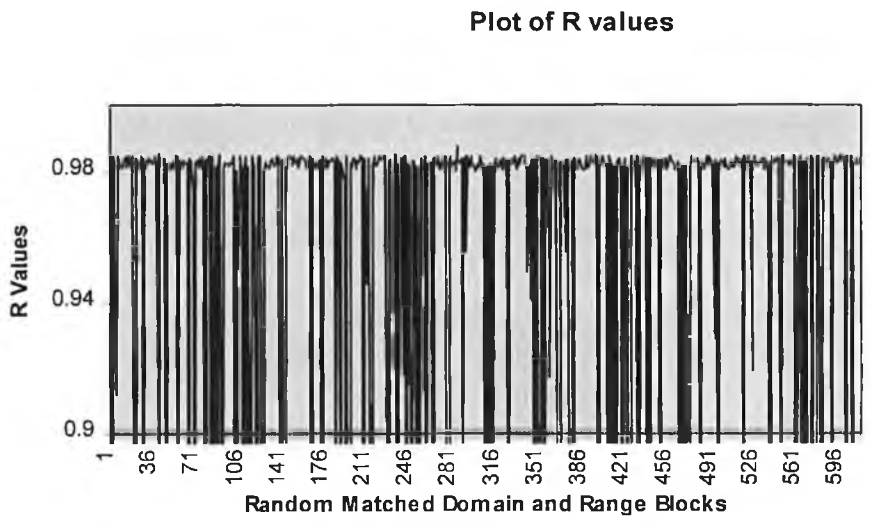


Figure 35. Fractal based DCT: Calculating value for $corr_{min}$

For the experiments the minimum value for $corr_{min}$ was determined to be 0.991. However it is still possible to match two blocks at random and get a value for $corr_{min}$ greater than 0.991. The minimum correlation coefficient in this case proved to be very close to the maximum possible which is 1.

Thus this value for $corr_{min}$ gives very little room for error. In practice many spurious block matches occurred during retrieval.

9.4.3 Conclusions

This technique proved to be of little assistance in retrieving the hidden information in a multimedia environment.

9.5 Marking Used Range Blocks.

Another use of the DCT and quantization processes in our algorithm would be to mark the used and unused range blocks by modifying components of the high order frequencies in a predefined manner.

The algorithm of this section is as follows:-

- We partition the image into 8x8 blocks using the quadtree method for use by the DCT transform.
- We select a range block from one of these 8x8 blocks and we find a matching domain block as described in chapter 6.
- We build a new range block as described in chapter 6.
- We now DCT transform the 8x8 block which contains the modified range block and mark it by modifying components of the high order frequencies in a predefined manner[56]. In the process of marking we modify the high frequency components so that we can use them to store bits of the steganographic data.
- We reverse the DCT process for the block.
- We realign the linear relationship between the domain and the range block. This will not effect the marking of the block. That is the realignment of the linear relationship has little effect on the high order coefficients.
- We mark all unused range blocks in a manner which indicates that they are unused.

On retrieval we look for used range blocks. When one is found

- We retrieve the marked bits.
- We then search for a domain block in the usual manner.

9.5.1 Security

The security concerns here were on two levels:

1. The marking of the domain blocks could be discovered.
2. The use of our algorithm in hiding extra bits of steganographic data.

The sequrity issues here were insurmountable, that is:- the marking of the range blocks could be discovered with knowledge of the algorithm by DCT transforming the entire image and observing published mark details in the resulting DCT blocks.

9.5.2 Robustness

On an initial investigation the algorithm proved to be robust allowing the value for $corr_{min}$ to fall to 8.5. Allowing an average error of ±6. However because of the security issues further tests were not performed to determine the reliability of these results.

9.5.3 Conclusions

Because of the lack of reliable security this technique proved to be unusable as a steganographic method.

Chapter

10.

Enhancing the Steganographic Ability of the Method

In this chapter we view our algorithm as a purely steganographic method. Enhancement to our algorithm are investigated which have the following properties :-

- We emphases the steganographic properties of the algorithm.

  In this chapter we make the assumption that the algorithm is only going to be used as a steganographic method. Therefore we can adjust the $corr_{min}$ value to suit the steganographic data, the position and composition of the range and domain regions.

- We relax the requirement of the steganographic data to survive in multimedia environments.

  In chapter 9 we looked at enhancements to our algorithm that help the steganographic data to survive being corrupted. The technique looked at in this chapter allows some corruption to the steganographic image file. But we assume that the image file will remain uncorrupted.

10.1 Steganographic Enhancement

In this section we look at the parts of our algorithm as described in chapter 6 that are to be enhanced. We present each enhancement by describing the part of our original algorithm that is to be enhanced. We then look at the modifications to be made to our original algorithm.

The objective of the enhancements is to add further complexity into the cryptanalysis process. The enhancements are as follows:-

- Reduce the value for $corr_{min}$ for the range and domain region.

  After inserting our steganographic data into a range region. We introduce noise[57] into the whole image. The correlation coefficient between a modified range and a domain block will have decreased. Therefore we must do a recalculation of $corr_{min}$ for the modified range blocks so that we will find all modified range blocks during the retrieval process.

  By reducing the value for $corr_{min}$ it causes lots more potential spurious hidden messages to occur during cryptanalysis. As stated in Chapter 8 we assume that a cryptanalyst knows that our algorithm has been applied to a particular image. Therefore in order for the cryptanalyst to find all hidden bits he has to use a low value of $corr_{min}$.

  By using a lower value for $corr_{min}$ than 9.6 (see chapter 7) the cryptanalyst would find spurious block matches all over the steganographic image file. The cryptanalyst would even discover spurious block matches in regions of our bogus messages. There are various techniques for inserting noise into images[58]. However in our case the noise we insert does not contain any data and so the discovery of the noise itself does not cause us a problem.

  These bogus messages and spurious block matches will not effect our user because he knows the range and domain regions. We can even have our $corr_{min}$ as a user input. This would allow us to decrease the value for $corr_{min}$ even lower. Making cryptanalysis even more difficult. However the $corr_{min}$ value may not be suitable as a user input because it is a real number. However that being said the user would just have to remember a number like 8.7.

- Place lots of bogus messages into the steganographic image file.

  The cryptanalyst will have to show that each discovered message is bogus before continuing. This becomes an exceptionally long process if each bogus message is encrypted.

- Match unused blocks with domain blocks that are not in the domain library.

  When hiding a one bit of steganographic data we search either D_0 or D_1 for a matching domain block. If we don't find such a matching block, the range block is unused. We call this unused range block R_{unused}.

  We now match R_{unused} with a domain block D_{found} from a separate domain library say D_2. D_2 is made up of domain blocks that are not in the domain region. We replace the R_{unused} with a similar block $R_{UnusedNew}$. Pixels in $R_{UnusedNew}$ are built by applying the scaling and offset factors to the corresponding pixels of D_{found} as described in Chapter 6.

  This procedure causes confusion to the cryptanalyst because if he finds the correct range region. Then he will think that he has lots of potential domain regions. Each of these potential domain regions has to be verified.

  We cannot use this procedure when the range blocks are distributed as described in 8.4 above.

10.2 Security analysis

The above enhancements were added to our algorithm and the results are presented in this section.

Several bogus messages were inserted into a test image. We then inserted our steganographic data and random noise. We found that the $corr_{min}$ value was 8.9. We then selected a series of range and corresponding domain regions in the image. We then tried to match 40 random range blocks against corresponding random domain blocks from the regions. For the matching process we set the $corr_{min}$ value to 9.4. We noted how many spurious matches occurred. The results are presented in the next graph.

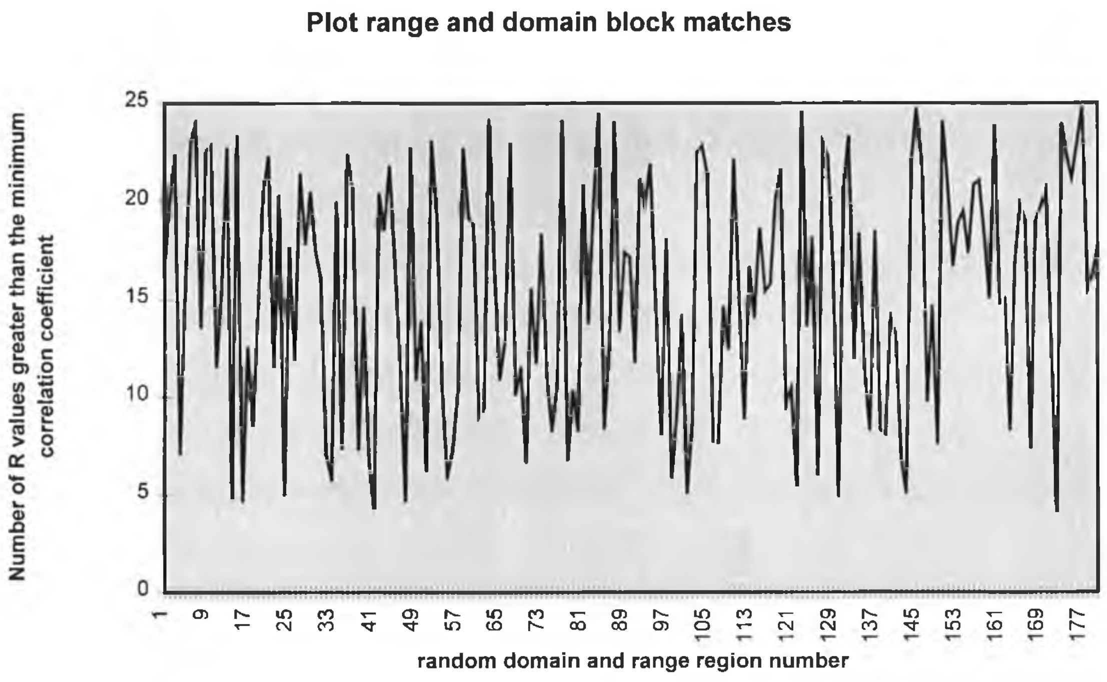


Figure 36A. Steganographic enhancements: Plot of spurious range and domain matches

The results were that for 50 matches of a random range block with a random domain block there were on average 14 matches.

These results show that a cryptanalyst can expect on average ≈ 30% of his range and domain blocks to give a match. These results are with $corr_{min}$ at too large a value to retrieve the steganographic data. In this case even if the cryptanalyst was to chose the correct range and domain blocks he would loose bits of the steganographic data.

Again several bogus messages were inserted into a test image. We then inserted our steganographic data and random noise. We found that the $corr_{min}$ value was 8.8. We then selected a series of range and corresponding domain regions in the image. We then tried to match 40 random range blocks against corresponding random domain blocks from the regions. For the matching process we set the $corr_{min}$ value to 8.8. We noted how many spurious matches occurred. The results are presented in the next graph.

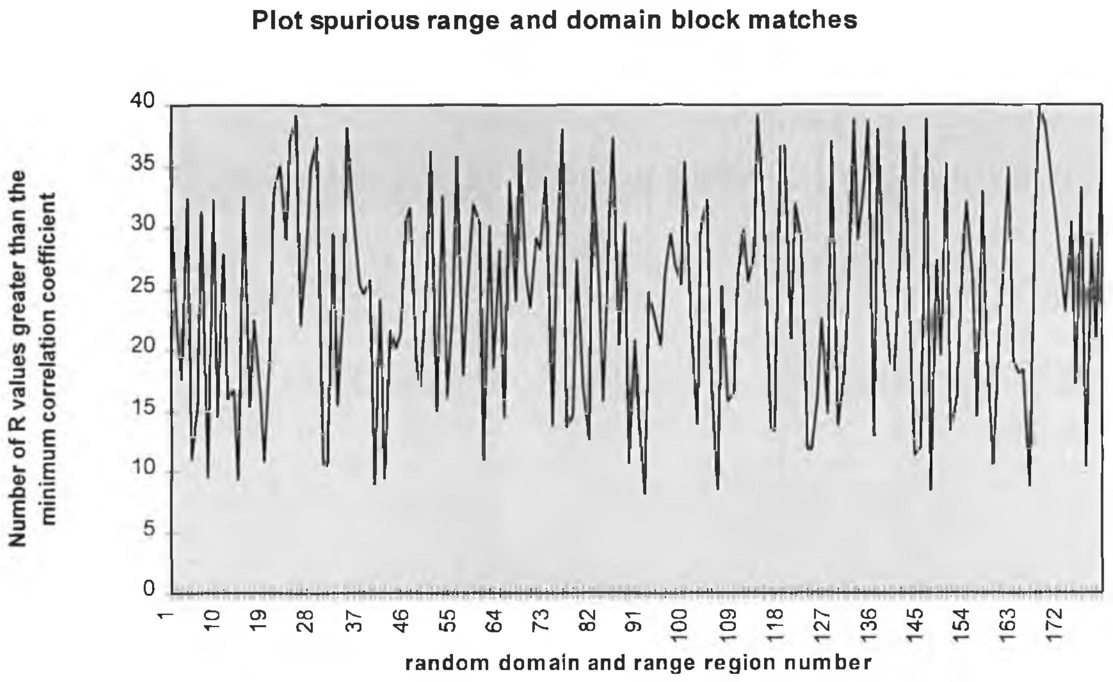


Figure 36B. Steganographic Enhancements: Plot of spurious range and domain matches

The results were that for 50 matches of a random range block with a random domain block, there were on average 22 spurious matches.

These results show that a cryptanalyst can expect on average $\approx$ 44% of his range and domain blocks to give a match. These results are with $corr_{min}$ at the correct value.

These results prove that while a cryptanalyst is searching for the correct range and domain regions then he will find many matching range and domain blocks. Each of these range and domain regions that the cryptanalyst tries will produce a bogus message. Some of these bogus messages have been placed by us but the majority are random.

10.3 Conclusions

If we are not using distributed blocks see 8.4 above then we draw the following conclusion:-
The algorithm gives excellent security provided the cryptanalyst does not by luck select the correct domain and range regions. If the cryptanalyst does select the correct domain and range regions then our only means of security is that the cryptanalyst has chosen the correct value for $corr_{min}$.

With distributed blocks combined with added noise to reduce the value of $corr_{min}$ then as described in 8.4 above the cryptanalyst would have

$$1024*1023*1022$$

possible choices for the pixels of a range block. This value would now have to be multiplied by the number of possible choices for $corr_{min}$. As $corr_{min}$ is a real number it potentially has many possible choices. For each value of $corr_{min}$ that is too high the retrieval process will loose bits of the steganographic data. Each value of $corr_{min}$ that is too low will have a spurious bits contained in it. The cryptanalyst has to be able to determine in each case that the steganographic data he has is invalid.

Chapter

11.

Conclusions

This chapter draws conclusions from the work on this thesis. We describe how useful our method is for data hiding. We also discuss the method's limitations.

The conclusions we draw from this research is as follows:-

- The method is excellent as a steganographic technique. It provides several means of security. Each of these are similar to a combination lock that is placed on the hidden information. Each lock will have be cracked. The following locks have to cracked in every cryptanalysis:-

  1. The position and shape of the range and domain regions. These can be selected by the user.
  2. The size and relationship between the range and domain regions.
  3. The use of distributed blocks.
  4. The correct selection of the $corr_{min}$ factor.

  Here we assume that the cryptanalyst is looking through the steganographic image file for the correct range and domain region. Given selected range and domain regions which are not the correct ones then the following locks have to be cracked :-

  1. The spurious block matches. This will cause the cryptanalyst to incorrectly retrieve a spurious hidden message. Each bogus message has to be discarded. These bogus messages would be simply noise.

2. The retrieval of actual bogus messages placed in random range regions by the user. These messages would not be noise and so the cryptanalyst would have a more difficult job in eliminating these messages.

- Computer based steganography is becoming increasingly important in this era of world-wide communications. One of the major advantages of steganography is that it can easily be combined with traditional cryptographic methods. This would give two levels of security. Our method can be easily combined with a cryptographic method. The data could be encrypted first before being hidden into an image.

- Multimedia document labelling is also of major importance in today's world. It is very easy to illicitly duplicate multimedia documents and distributed them. Hiding a steganographic label into an important multimedia document would ease this problem. Steganography would seem to be one of the only solutions to this problem as the document itself contains an innocuous label. Our method is suited to labelling because such a label is typically short. However it is not possible to adapt the method so that the steganographic information will survive through major lossy image processing.

- The amount of data one can store is small. It is only possible to store one bit per range block. This can be increased if the tolerance level is relaxed. Some ranges cannot be covered when the tolerance is low. Increasing the tolerance level would allow us to use all range blocks in the range region given an increase in the amount of steganographic data that can be stored. However we would normally require the tolerance to be low in order to give an image that is visually close to the original.

  The range region cannot overlap the domain region; because the range region gets modified and we need the domain region to remain unmodified. This restricts the amount of steganographic data that can be stored. However the bigger the range region is the more data that can be stored. But there has to be a compromise as to the size of the domain region. The smaller the domain region is the worse the quality of the produced image will be.

- The process of data hiding is slow. It depends on the amount of steganographic data and how high the tolerance is set. The time taken also depends very much on the size of the domain region. But the bigger the domain region the better the quality of the produced image. The data retrieving is even slower this is because the comparison processes is more intensive.

- It is possible to distribute the range and domain blocks throughout the regions. This means that a block is made up a pseudo random set of pixels taken from different parts of the region. The only limitation required is that a pixel only belongs to one range block. A range and domain block may be distributed. If this scheme of things was implemented then it would yield a vast amount of searches for the cryptanalyst.

Bibliography

1. Schnieder, *Applied Cryptography*,Wiley,1993
2. E. Franz et al, Computer Based Steganography, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
3. R. Anderson, Stretching the Limits of Steganography, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
4. M. Barnsley, *Fractal Image Compression*, AK Peters, 1993, 1-56881-000-8
5. E. Koch, J. Zhao, *Embedding Robust Labels into Images for Copyright Protection*, Proc. of the International Congress on Intellectual Property Rights for Specialized Information, Vienna, Austria, Aug 1995
6. David Kahn, *The Codebreakers*, Macmillan Publishing, 1967
7. David Kahn, *The History of Steganography*, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
8. Netcom.com , /pub/qwerty, FTP site for reviewed software
9. E. Franz et al, Computer Based Steganography, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
10. E. Koch, J. Rindfrey, J. Zhao, *Copright Protection for Multimedia Data*, Proceedings of the International Conference on Digital Media and Electronic Publishing, Dec 1994, Leeds UK
11. E. Koch , J. Zhao, *Towards Robust and Hidden Image Copyright Labelling*, Proc. of 1995 Workshop on Nonlinear Signal and Image Processing, Neos Marmaras, Greece, June, 1995
12. E. Koch, J. Zhao, *Embedding Robust Labels into Images for Copyright Protection*, Proc. of the International Congress on Intellectual Property Rights for Specialized Information, Vienna, Austria, Aug 1995
13. B. Kahin, *The strategic environment for protecting multimedia*, IMA Intellectual Property Project Proceeding, vol 1, no. 1, 1994
14. K. Matusi, K. Tanaka, *Video Steganography: How to secretly embed a signature in a picture*, IMA intellectual Property Project Proceedings, vol. 1, no. 1,1994
15. G. Wallace, *The JPEG still picture compression standard*, Communications of the ACM, vol 34, no.4, April 1991
16. R. Dixon, *Spread Spectrum Systems*, 2nd ed.,Wiley, New York, NY, 1984
17. J. Brassil, S.Low, N. Maxemchuk, *Electronic Marking and Identification Techniques to Discourage Document Copying*, AT&T Bell Labs, Murray Hill, NJ
18. A. Choudhury, N. Maxemchuk, S. Paul, *Copyright Protection for Electronic Publishing over Computer Networks*, AT&T Bell Labs, June 1994
19. T. Handel, *Hiding data in the OSI Network Model*, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8

20. E. Franz et al., *Computer Based Steganography* ,Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
21. D. Gruhl et al, *Echo Hiding*, Information Hiding Workshop, University of Cambridge, May 1996, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
22. Y. Fisher, *Fractal Image Compression Theory and Application*, Chapter 1, Springer-Verlag, 1995, 0-387942114
23. M. Barnsley, *Fractal Image Compression*, AK Peters, 1993, 1-56881-000-8
24. J. Hutchinson, *Fractals and Self Similarity*, Indiana University Mathematics Journal, Vol 35, No. 5, 1981
25. M. Barnsley, *Fractals Everywhere*, Academic Press, 1993, 0-12-079061-0
26. H. Peitgen, D.Saupe, H. Jurgens, Springer Verlag, *Fractals for the Class Room*, New York,1991
27. M.Barnsley, L. Hurd, *Fractal Image Compression*, 1993, AK Peters,1-56881-000-8
28. J. Hutchinson, *Fractals and Self Similarity*, Indiana University Mathematics Journal, Vol 35, No. 5, 1981
29. Y. Fisher, *Fractal Image Compression*, SIGGRAPH'92 Course Notes
30. A. Rosenfeld, *Multiresolution Image Processing and Analysis*, Springer-Verlag
31. A.Jacquin, *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding*, Phd Thesis, Georgia Institute of Technology, 1989
32. Y. Fisher, E.W. Jacobs, *Fractal Image Compression Using Iterated Transforms*, NOSC Technical Report, Naval Ocean Systems Center, San Diego CA 92152-5000
33. M. Spiegel, *Probability and Statistics*, Schaum's outline series, McGraw Hill, Chapter 8
34. W. Press et al, *Numerical Recipes*, Cambridge University Press, 0-521-30811-9, 1989, Chapter 14
35. E. Kreyszig, *Advanced Engineering Mathematics* , Wiley, 1989
36. F. Schied, Numerical Analysis, Schaum's Outline Series, Chapter 21, McGraw Hill, 07-055197-9, 1968
37. M. Spiegel, *Probability and Statistics*, Schaum's outline series, Chapter 5, page 160
38. M. Spiegel, *Probability and Statistics*, Schaum's outline series, Chapter 3, page 81
39. M. Spiegel, *Probability and Statistics*, Schaum's outline series, Chapter 8, page 263
40. M. Spiegel, *Probability and Statistics*, Schaum's outline series, Chapter 8,page 262
41. M. Nelson, The Data Compression Book, M&T books, 1991
42. Y. Fisher, *Fractal Image Compression Theory and Application*, Springer-Verlag, 0-387942114, 1995
43. I. Pitas & A. Venetsanopoulos, Nonlinear Digital Filters, Kluwer Academic, 1990
44. H. Myler & A. Weeks, *Computer Imaging recipes*, Prentice Hall,1993,0-13-189879-5
45. Y. Fisher, E.W. Jacobs, *Fractal Image Compression Using Iterated Transforms*, NOSC Technical Report, Naval Ocean Systems Center, San Diego CA 92152-5000
46. Y. Fisher, *Fractal Image Compression Theory and Application*, Chapter 1, Springer-Verlag,0-387942114
47. Y. Fisher, E.W. Jacobs, *Fractal Image Compression Using Iterated Transforms*, NOSC Technical Report, Naval Ocean Systems Center, San Diego CA 92152-5000

48. Y. Fisher, *Fractal Image Compression Theory and Application*, Chapter 3, Springer-Verlag, 0-387942114, 1995
49. T. Aura, Practical invisibility in digital communications, Information Hiding Workshop, University of Cambridge, May 1996, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
50. M. Sonka et al., Image Processing, Analysis and Machine Vision, Chapman & Hall,1994, 0-412-45570-6
51. Y. Fisher, *Fractal Image Compression Theory and Application*, Chapter 6, Springer-Verlag,0-387942114
52. I. Cox et al, *A Secure, Robust Watermark for Multimedia*, Workshop on Information Hiding, Univ. of Cambridge, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
53. I. Cox et al, *A Secure, Robust Watermark for Multimedia*, Workshop on Information Hiding, Univ. of Cambridge, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
54. J. Smith, Modulation and Information Hiding in Images, Workshop on Information Hiding, Univ. of Cambridge, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
55. I. Cox et al, *A Secure, Robust Watermark for Multimedia*, Workshop on Information Hiding, Univ. of Cambridge, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
56. J. Zhao et al, *Embedding robust labels into images for copyright protection*, Information, knowledge and New technologies conference, Vienna Aug 21-25
57. M. Sandford, The data embedding method, Proceedings of the SPIE Photonics Conference, Philadelphia, Sept 1995
58. T. Aura, Practical invisibility in digital communications, Information Hiding Workshop, University of Cambridge, May 1996, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
59. D. Kruglinski, *Inside Visual C++*, Microsoft Press,1996,155615-891-2
60. J. Smith, Modulation and Information Hiding in Images, Workshop on Information Hiding, Univ. of Cambridge, May '96, Lecture notes in Computer Science, Springer-Verlag, 1996, 3-540-61996-8
61. H. Myler & A. Weeks, *Computer Imaging recipes*, Prentice Hall,1993,0-13-189879-5
62. Shannon and Weaver, *The Mathematical Theory of Communication*, The University of Illinois Press, 1949.
63. Stallings, *Data and Computer Communications*, Macmillian.
64. D. Kruglinski, *Inside Visual C++*, Microsoft Press,1996,155615-891-2

Appendix A

Proof of Concept

In this appendix we look at implementation of our algorithm as a proof of concept. The application which we called *Frasteg* is used as a proof of concept. All testing was performed using Frasteg.

Frasteg allows a user to hide steganographic data into a 256 level grey scale bitmap. It builds a new bitmap which is visually identical to the original. This new bitmap will have the required steganographic data hidden in it. Frasteg also allows a user to retrieve steganographic data from a bitmap file.

Frasteg was designed in an object oriented manner. The implementation was in Visual C++. Frasteg uses full Visual C++ architectures. Therefore Visual C++ terminology will be used in the description. Visual C++ was chosen because it assists in developing industrial strength Windows based applications.

A.1 The Frasteg User Interface

Frasteg is a MS-Windows based application and as such has the standard menu driven user interface.

One must first of all open a bitmap file. You do this with the File Open menu option. Figure 4 shows an example bitmap (girl.bmp) opened and displayed in a child window.

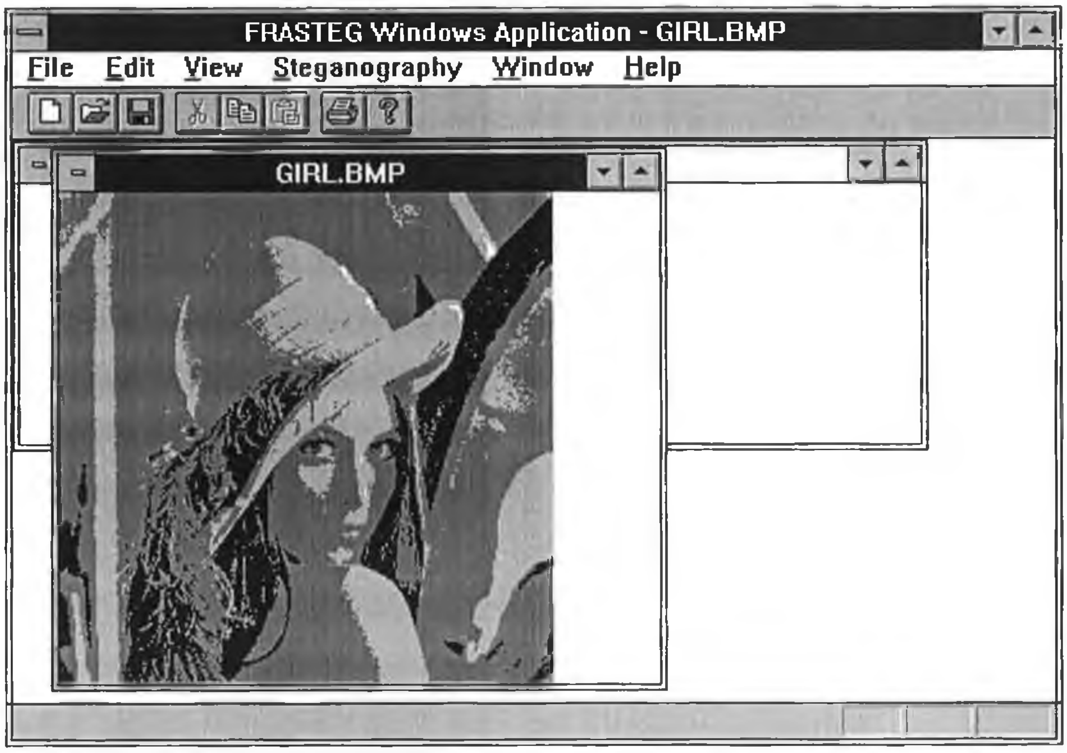

FRASTEG Windows Application - GIRL.BMP
File Edit View Steganography Window Help
GIRL.BMP

Figure 37. Frasteg User Interface

The user can now select the menu option *steganography*. This menu option has two sub-menu options associated with it.

- The *Add steganographic data* option allows a user to add steganographic data to the bitmap and store the result in a new bitmap. When this option is selected the user is presented with the standard Windows file save dialog. This dialog specifies the new bitmap file that is to be created. When the user has successfully selected a file name he/she is presented with another dialog box. This dialog allows the user to enter the steganographic data that is to be hidden into the new bitmap file.

  When the steganographic data is entered and the OK button selected the data hiding process begins. Status messages are displayed to the user to indicate progress. When data hiding is completed a dialog box is displayed to the user which indicates the number of bytes that were successfully hidden.
- The *Get steganographic data* menu option allows a user to retrieve steganographic data from the bitmap. When this option is selected the revealing process begins.

Status messages are displayed to the user to indicate progress. When the revealing process completes a dialog box is presented to the user which shows the steganographic data.

A.2 The CDib class

The CDib class contains the code and data for manipulating DIBs (Device Independant Bitmap). MFC has no class for manipulating DIBs. So we had to write code to manipulate the DIB. There are examples of how to do this in Kruglinski[59]. A CDib object represents a single device-independent bitmap. The image bits together with all necessary decoding information are contained within this object. The image bits are maintained in a HUGE array. Frasteg does not limit the size of the bitmap so this array can be very large.

The CDib object also contains information on the DIB. For example :- the number of colours, its width, its height etc. The reason these are stored here is that if we were to access them by means of the DIB header then we would have to lock the DIB memory get the information and then release the DIB memory again.

The CDib object contains a number of member functions. For example:-

- *Read* and *Write* members for serializing (reading/writing to disk) the DIB.
- Constructor and destructor for initialising and cleaning up the CDib object.
- *Display* function sends the CDib object to the display (or printer) without creating an intermediate GDI bitmap.
- There are member functions for returning the read only information from the CDib object. For example *GetLength* returns the length of the DIB.

A.3 Frasteg - The Code Format

Skeleton code for Frasteg was built using the AppWizard tool that is supplied with Visual C++. A Multiple Document Interface (MDI) application was chosen. A MDI application allows multiple documents and multiple views of each document. The skeleton's structure is based on a standard MFC based application.

A.3.1 MFC class hierarchy as applied to Frasteg

Taken together, the classes in MFC make up an *application framework.* That is the framework on which one builds an application for Windows. A programmers work with MFC is based largely on a few major classes and several C++ tools.

In any MFC based application a programmer will derive classes from several key MFC classes. In a running MFC based application a programmer will then have the following key objects:-

- The document(s)

  The document class (derived from MFC's *CDocument*) specifies the application's data. In Frasteg the data is an array which holds the pixel intensity values for the bitmap. A user of Frasteg may have many bitmaps open at any given time, each of these will have a CDocument object associated with it. See Appendix C, section C1.

- The view(s)

  The view class (derived from MFC's *CView*) is the user's *window on the data.* The view class specifies how the user sees the document's data and interacts with it. In Frasteg this class describes how to present the bitmap to the user. The OnDraw member of the view class redraws the current bitmap whenever the user does something which invalidates it. For example if the user minimises the window.

- The frame windows

  Views are displayed inside *document frame windows*. In an MDI application, document frame windows are child windows displayed inside a main frame window (derived from *CMDIFrameWnd*).

- The document template(s)

  A document template causes the creation of documents, views and frame windows. A particular document-template class creates and manages all open documents of one type. Derive from *CMultiDocTemplate* for MDI applications. There is one document template in Frasteg. This document template causes a document and a view object to be created whenever a user opens a bitmap file.

- The application object

  The application class (derived from *CWinApp*) controls all of the above objects.

A.3.2 Document-View model.

One of the most important features of an MFC based application is the division of labour between document and view objects. In a MFC based application the document-view programming model is used to separate the management of the data and the displaying of that data to the user.

A document object contains and manages the data in Frasteg. The document object contains many data members that are associated with the opened bitmap. From the user's standpoint the document is the data that user is manipulating. From the programmers standpoint the document is a C++ object of a class derived from the MFC library CDocument class. In Frasteg the document object contains message handling functions which are called when the user selects *Add steganographic data* and *Get steganographic data*; See Appendix C, section C1. MFC causes associated member functions to be called in response to theses menu options.

The view object displays the data and allows editing. From the user's standpoint, a *view* is an ordinary window that he/she can size, move etc. in the same way as any other Windows-based application window. From the standpoint of the programmer, a view is a C++ object of a class derived from the MFC library CView class.

A.3.4 Hiding and retrieving the steganographic data

Frasteg follows the algorithms as outlined in section 3.1 above. But generalises them into the following:-

- Selecting a range block r_i is in the order of the quadtree fractal compression algorithm.

- The *SelectedRanges* set is made up of the smallest range blocks defined by the quadtree algorithm, this allows us to hide more steganographic data. However if larger blocks were used it would make it more difficult for a cryptanalyst to crack the data hiding. This is because there would now be less likely that statistical analysis could be successfully applied to the images which have steganographic data hidden in them.

- We need to be able to get the next bit from this stream. In Frasteg *Steg* is an object of the C++ class StegData. It has an overloaded operator >> defined which gives the next bit in the stream. This operator takes a StegData object and a character variable as parameters. The result of this operation is 0 if the end of the bit stream is reached otherwise the result is 1. A class StegData also has an overloaded operator << defined which adds a bit to a stream. This operator is used when retrieving the steganographic data. See *Stegan Module* in Appendix C, section C4 for details.

Appendix B

Information Theory vs. Hiding Bandwidth

In this Appendix we document our attempt to apply information theory to our hiding algorithm. The main focus of this effort was to discover the theoretical limits on the data hiding bandwidth. The main source of the ideas in this section were presented in the paper by Smith[60].

B.1 Introduction

To apply information theory to our steganographic algorithm we consider:-

- The original image as a noisy channel.
- The steganographic data as the signal.
- The hiding algorithm as a modulator.
- The revealing algorithm as a demodulator.

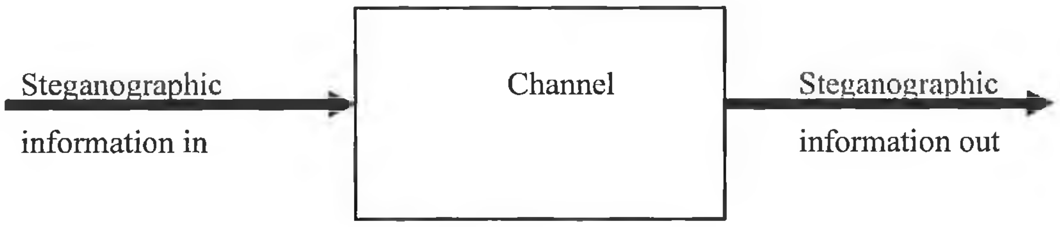


Figure 38. Steganographic algorithm as a communication system

B.2 Steganographic Algorithm as a Communications System

We have a number of ways to apply this communication system to our algorithm:-

- To the whole image, with all of the steganographic information as signal.

- To individual range blocks, with one bit of steganographic information as signal.
- To individual pixels in the range block and the amount information that an individual pixel can carry.

The channel capacity is the maximum rate at which a channel can carry information without error. We view the image as being subject to Gaussian noise[61], and for a channel subject to this type of noise the channel capacity is (from shannon's theorem[62]) given by:-

$$C = W * \log_2\left(1 + \frac{S}{N}\right)$$

Where, *W* is the bandwidth, and *S/N* is the signal to noise ratio.

For data hiding the noise is the original image and the signal is the steganographic data.

However the bandwidth here indicates the range of frequencies which can be successfully transmitted over a communications channel, and as such is measured in hertz. Therefore the image needs to be converted into the frequency domain in order to apply this formula in the traditional manner.

Our algorithm is a spatial domain method, and there seems to be no way of viewing the hiding algorithm as a modulator and the retrieval algorithm as a demodulator. This is because traditional modulation methods are amplitude, frequency and phase[63]. These are all based in the frequency domain.

Appendix C

Proof of Concept Code

This Appendix lists the main code sections for the proof of concept application (Frasteg). The code was written in Visual C++. Visual C++ was choosen as an implementation framework, because it assists in building industrial strength applications.

We give an index into the code main code sections. The pseudo code for our algorithm in Chapter 7, references this index. Appendix A gives a more high level overview of the proof of concept implementation.

All the code for Frasteg is on the attached floppy disk. The floppy disk also includes further code associated with the techniques of chapter 9.

C.1 Index Into The Code

C.2 The Document Code

/* The **CStegDoc module.**
This module has code:-
1. For converting the BMP file into a large array of pixel values each pixel value is 0-256 in grey level.
2. Menu handling code which is called by MFC when the user selects Hiding/retrieving options.*/

Note:- The non-updated MFC stuff has been removed.

```
#include "stdafx.h"
#include <afxdlgs.h>
#include "diblook.h"
#include <limits.h>
#include "proto.h"
#include "encgray.h"
#include "stegdoc.h"
#include <stdio.h>
#include "stegan.h"
#include "enterste.h"
#include "getstega.h"

int hsize,vsize,Nobits;
int shsize,svsize;
/* The steganographic data buffers. */
Stegan EncS(""),DecS(1000);
 RGBQUAD RQ;
   RGBTRIPLE RT;
   int alloced;
```

// **Document Constructor**

```
CStegDoc::CStegDoc()
{
        m_hDIB = NULL;
        m_palDIB = NULL;
        hsize = vsize = -1;
        m_sizeDoc = CSize(1,1);    // dummy value to make CScrollView happy
}
```

// **Document Destructor**

```
CStegDoc::~CStegDoc()
{
        if (m_hDIB != NULL)
        {
                ::GlobalFree((HGLOBAL) m_hDIB);
        }
        if (m_palDIB != NULL)
        {
                delete m_palDIB;
        }
}
```

```
/* BMP Initialise function */
void CStegDoc::InitDIBData()
{
	if (m_palDIB != NULL)
	{
		delete m_palDIB;
		m_palDIB = NULL;
	}
	if (m_hDIB == NULL)
	{
		return;
	}
	// Set up document size
	LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);
	cxDIB = (int) ::DIBWidth(lpDIB);	// Size of DIB - x
	cyDIB = (int) ::DIBHeight(lpDIB);	// Size of DIB - y

	m_sizeDoc = CSize(cxDIB, cyDIB);
	if (::DIBWidth(lpDIB) > INT_MAX || ::DIBHeight(lpDIB) > INT_MAX)
	{
		::GlobalUnlock((HGLOBAL) m_hDIB);
		::GlobalFree((HGLOBAL) m_hDIB);
		m_hDIB = NULL;
		MessageBox(NULL, "DIB is too large", NULL,
						MB_ICONINFORMATION | MB_OK);
		return;
	}

	::GlobalUnlock((HGLOBAL) m_hDIB);
	// Create copy of palette
	m_palDIB = new CPalette;
	if (m_palDIB == NULL)
	{
		// we must be really low on memory
		::GlobalFree((HGLOBAL) m_hDIB);
		m_hDIB = NULL;
		return;
	}
	if (::CreateDIBPalette(m_hDIB, m_palDIB) == NULL)
	{
		// DIB may not have a palette
		delete m_palDIB;
		m_palDIB = NULL;
		return;
	}
}

/* BMP Open function */
BOOL CStegDoc::OnOpenDocument(const char* pszPathName)
{
	CFile file;
	CFileException fe;
	if (!file.Open(pszPathName, CFile::modeRead | CFile::shareDenyWrite, &fe))
	{
		ReportSaveLoadException(pszPathName, &fe,
			FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
		return FALSE;
	}

	DeleteContents();
	BeginWaitCursor();

	// replace calls to Serialize with ReadDIBFile function
	TRY
	{
		m_hDIB = ::ReadDIBFile(file,bmfHeader);
	}
	CATCH (CFileException, eLoad)
	{
		file.Abort(); // will not throw an exception
		EndWaitCursor();
		ReportSaveLoadException(pszPathName, eLoad,
			FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);
		m_hDIB = NULL;
		return FALSE;
	}
	END_CATCH

	InitDIBData();
	EndWaitCursor();

	if (m_hDIB == NULL)
	{
		// may not be DIB format
		MessageBox(NULL, "Couldn't load DIB", NULL,
						MB_ICONINFORMATION | MB_OK);
		return FALSE;
	}
	SetPathName(pszPathName);
	SetModifiedFlag(FALSE);	// start off with unmodified
	return TRUE;
}

/* BMP Save function */
BOOL CStegDoc::OnSaveDocument(const char* pszPathName)
{
	CFile file;
	CFileException fe;

	if (!file.Open(pszPathName, CFile::modeCreate |
	  CFile::modeReadWrite | CFile::shareExclusive, &fe))
	{
		ReportSaveLoadException(pszPathName, &fe,
			TRUE, AFX_IDP_INVALID_FILENAME);
		return FALSE;
	}

	// replace calls to Serialize with SaveDIB function
	BOOL bSuccess = FALSE;
	TRY
	{
		BeginWaitCursor();
		bSuccess = ::SaveDIB(m_hDIB, file);
		file.Close();
	}
```

```
        CATCH (CException, eSave)
        {
                file.Abort(); // will not throw an exception
                EndWaitCursor();
                ReportSaveLoadException(pszPathName, eSave,
                        TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);
                return FALSE;
        }
        END_CATCH

        EndWaitCursor();
        SetModifiedFlag(FALSE);     // back to unmodified

        if (!bSuccess)
        {
                // may be other-style DIB (load supported but not save)
                //  or other problem in SaveDIB
                MessageBox(NULL, "Couldn't save DIB", NULL,
                                                MB_ICONINFORMATION | MB_OK);
        }

        return bSuccess;
}

/* Function to convert a BMP format file into internal representation. Internal representation
is a 2D array of pixel values. */
void CStegDoc::RGBtoImage(int Insert)
{
        WORD wNumColors;        // number of colors in color table
        LPSTR lpbi;             // pointer to packed-DIB
        LPBITMAPINFO lpbmi;     // pointer to BITMAPINFO structure (Win3.0)
        LPBITMAPCOREINFO lpbmc; // pointer to BITMAPCOREINFO structure (old)
        BOOL bWinStyleDIB;      // flag which signifies whether this is a Win3.0 DIB


        LPSTR   lpDIBHdr;           // Pointer to BITMAPINFOHEADER
  BYTE  *lpDIBBits;     // Pointer to DIB bits
  DWORD Height,Width;
  DWORD i,j;
  BYTE Red,Green,Blue;
  BYTE val;
  unsigned int cnt,tmp,nobits,index,bitcnt = 0;
  long l,Fpos;
  COLORREF C;
  int rgb;
  RGBQUAD * rq;
  RGBTRIPLE * rt;

        /* if handle to DIB is invalid, return FALSE */

        if (m_hDIB == NULL)
          return;

        lpbi = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);


        lpDIBBits = (BYTE *) ::FindDIBBits(lpbi);
 /* get pointer to BITMAPINFO (Win 3.0) */
```

```
  lpbmi = (LPBITMAPINFO)lpbi;

  /* get pointer to BITMAPCOREINFO (old 1.x) */
  lpbmc = (LPBITMAPCOREINFO)lpbi;

  noColours = DIBNumColors(lpbi);

  FILE *fp = fopen("input.bit","wb");

                    /* is this a Win 3.0 DIB? */
                    if (bWinStyleDIB = IS_WIN30_DIB(lpbi))
                         {
                         if ( Insert)
                           fwrite(lpbmi,sizeof(BITMAPINFOHEADER),1,output);
//                       fwrite(lpbmi->bmiColors,sizeof(RGBQUAD),noColours,output);
                      Height = lpbmi->bmiHeader.biHeight;
                      rq = (RGBQUAD *)((LPSTR)lpbmi + lpbmi->bmiHeader.biSize);
                      Width = lpbmi->bmiHeader.biWidth;
                         Nobits=nobits = lpbmi->bmiHeader.biBitCount;
                      }
                    else
                         {
                         if ( Insert )
                            fwrite(lpbmc,sizeof(BITMAPCOREHEADER),1,output);
//                       fwrite(lpbmc->bmciColors,sizeof(RGBTRIPLE),noColours,output);
                         rt = (RGBTRIPLE *)((LPSTR)lpbmc + lpbmc->bmciHeader.bcSize);
                      Height =  (DWORD)lpbmc->bmciHeader.bcHeight;
                      Width = (DWORD)lpbmc->bmciHeader.bcWidth;
                      Nobits=nobits = lpbmc->bmciHeader.bcBitCount;
                      }


/* Find colours in the colour map. */
     cnt =0 ;
     if ( Insert ) {
     for (i = 0; i < (int)noColours; i++)
                         {
                         if (bWinStyleDIB)
                              {
                              Red = rq[i].rgbRed;
                              Green = rq[i].rgbGreen;
                              Blue = rq[i].rgbBlue;
                              if ( !(Red == Green && Red == Blue ))
                                cnt++;
                              else
                                fwrite(rq+i,sizeof(RGBQUAD),1,output);
                              }
                         else
                              {
                              Red = rt[i].rgbtRed;
                              Green = rt[i].rgbtGreen;
                              Blue = rt[i].rgbtBlue;
                              if ( !(Red == Green && Red == Blue ))
                                cnt++;
                              else
                                 fwrite(rt+i,sizeof(RGBTRIPLE),1,output);
                              }
                         }
```

```
                for (i=0;i<cnt;i++)
                        if (bWinStyleDIB)
                                fwrite(&RQ,sizeof(RGBQUAD),1,output);
                        else
                        fwrite(&RT,sizeof(RGBTRIPLE),1,output);
                }

    svsize = hsize = Width;
    svsize = vsize = Height;

    if (!alloced) {
        matrix_allocate(EncImage, hsize, vsize, IMAGE_TYPE);
        if ( Insert)
      matrix_allocate(DecImage, hsize,vsize, IMAGE_TYPE);
    }
/* Go through the image data; image data consists of an index into the colour table and using index
find the appropriate colour. */
                for (i = 0; i < Height; i++)
                        {
                        for (j=0;j < Width;j++ )
                                {
                                switch (nobits) {
                                        case 1:
                                                if (  bitcnt == 8 ) {
                                                  val = *lpDIBBits++;
                                                  bitcnt = 0;
                                                  }
                                                index =  (val >> bitcnt) & 1;
                                                bitcnt++;
                                                break;
                                        case 4:
                                                if (  bitcnt == 0 ) {
                                                  val = *lpDIBBits++;
                                                  bitcnt = 8;
                                                  }
                                                bitcnt -= 4;
                                                index =  (val >> bitcnt) & 15;
                                                break;
                                        case 8:
                                                index = *lpDIBBits++;
                                                break;
                                        }/* switch */
                                if (bWinStyleDIB)
                                        Red = rq[index].rgbRed;
                                else
                                        Red = rt[index].rgbtRed;

                /*C = ((long)Red << 16) | (long)Green << 8 |
                        (long)Blue; */

                        EncImage[i][j] = Red;
                        if (fp)
                          fputc(Red,fp);
                        if ( Insert )
                    DecImage[i][j] = Red;
                        } /* for */

            for (;j%4;j++)
```

```
                        lpDIBBits++;
                    } /* for */
    if (fp)
      fclose(fp);
            ::GlobalUnlock((HGLOBAL) m_hDIB);
}

/* Function to convert internal format into a BMP format file. Internal representation
is a 2D array of pixel values. */
void CStegDoc::GetBMPFromRGBImage(IMAGE_TYPE **Image,char * bufhead,int siz)
{
        WORD wNumColors;        // number of colors in color table
        LPSTR lpbi;             // pointer to packed-DIB
        LPBITMAPINFO lpbmi;     // pointer to BITMAPINFO structure (Win3.0)
        LPBITMAPCOREINFO lpbmc; // pointer to BITMAPCOREINFO structure (old)
        BOOL bWinStyleDIB;      // flag which signifies whether this is a Win3.0 DIB

    char *lpDIBBits;    // Pointer to DIB bits
    DWORD i,j,nWidth;
    int index,bitcnt=8,val=0,sz,Red,x,r,cnt,diff,diffcnt;
    BYTE  * lpBits;
    BYTE  * pDIB;
    DWORD NoBits,Header;

    int nocol = 0;
    BYTE ind;

        if (m_hDIB == NULL)
          return;

        lpbi = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);

        lpDIBBits = (char *) ::FindDIBBits(lpbi);
   /* get pointer to BITMAPINFO (Win 3.0) */
   lpbmi = (LPBITMAPINFO)lpbi;

   /* get pointer to BITMAPCOREINFO (old 1.x) */
   lpbmc = (LPBITMAPCOREINFO)lpbi;

   noColours = DIBNumColors(lpbi);

                    /* is this a Win 3.0 DIB? */
                    bWinStyleDIB = IS_WIN30_DIB(lpbi);


    FILE * fp=fopen("output.bit","wb");

        for (i = 0; i < vsize; i++)
                        {
                        fprintf(fp,"\n");
                        for (j=0;j < hsize;j++ )
                                {
                                Red = Image[i][j];
                                if ( fp )
                                  fprintf(fp,"%d ",Red);
                                if (bWinStyleDIB)
                                   {
                                   diff = 254;
                    diffcnt = 0;

                    for(cnt=0;cnt<noColours;cnt++)
                              if ( Red == (r=lpbmi->bmiColors[cnt].rgbRed) )
                                break;
                              else
                                 if ((x=(Red-r < 0 ? r-Red: Red-r)) < diff)
                                   {
                                   diff = x;
                                   diffcnt = cnt;
                                   }
                            if ( cnt == noColours )
                               cnt = diffcnt;
                            }
                 else
                            {
                    diff = 254;
                    diffcnt = 0;
                    for(cnt=0;cnt<noColours;cnt++)
                              if ( Red == (r=lpbmc->bmciColors[cnt].rgbtRed) )
                                break;
                              else
                                 if ((x=(Red-r < 0 ? r-Red: Red-r)) < diff)
                                   {
                                   diff = x;
                                   diffcnt = cnt;
                                   }
                            if ( cnt == noColours )
                               cnt = diffcnt;
                            }

                 switch (Nobits) {
                            case 1:

                                      if (  bitcnt == 8 ) {
                                        fputc(val,output);
                                        bitcnt = 0;
                                        val = 0;
                                        }
                                      val =  (val << bitcnt) | (cnt & 1);
                                      bitcnt++;
                                      break;
                            case 4:
                                      if (  bitcnt == 0 ) {
                                        fputc(val,output);
                                        bitcnt = 8;
                                        val = 0;
                                        }
                                      bitcnt -= 4;
                                      val |= (cnt&15)<< bitcnt;
                                      break;
                            case 8:
                                      fputc(cnt,output);
                                      break;
                            }/* switch */

          } /* for */
   for (;j%4;j++)
```

```
                    fputc(0,output);
                  } /* for */

   if (fp)
      fclose(fp);
::GlobalUnlock((HGLOBAL) m_hDIB);
}



/* Some test steganographic messages. */
char *Keys[] = {
"Hello world",
"asasjahsahs",
"skdjksjsjdkjs",
";lpi886^^&&*()",
"}opspo785yth899oi",
"kjs879992ui9989oo",
"sjk892899998099-9",
"98989uiuiduisuiu",
".,/./.;//'6162jhhHJHJHHH",
"sdhp[p[p[ppweqwe[qwe",
"}}}{{{;'56535^*^^*^****"
};

/* Event handler for Add steganographic data. */
/* This is the driver code to add steganographic info into the image file. */
void CStegDoc::OnSteganographyAdddata()
{
        // TODO: Add your command handler code here
        Enterstegan dia;
        int sz,i;
        long Fsize,Fsize1;

        dia.m_stegan = EncS.Data();
        dia.DoModal();
        EncS.initchar(dia.m_stegan);

        CFileDialog BitmapFile(FALSE,NULL,"Fractal.bmp",
                                OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
                                "Bitmap File (*.bmp)||",NULL);


        BitmapFile.m_ofn.lpstrTitle = "Save Bitmap File";
        BitmapFile.DoModal();


/* Test code comment out. */
        for ( int ii=0;ii<1;ii++) {
 //  EncS.initchar(Keys[ii]);
        if ((output = fopen(BitmapFile.GetPathName(), "wb")) == NULL)
          {
          Print("Can't open fractal file.");
          return;
          }
        ::SetCursor(::LoadCursor(NULL,IDC_WAIT));
        fwrite(&bmfHeader,sizeof(BITMAPFILEHEADER),1,output);

        RGBtoImage(1);
        EncodeIt(1);
        }//for

        GetBMPFromRGBImage(DecImage,buf,sz);

        /* Delete off buffers */
        for (i=0;i<svsize;i++)
          delete [] DecImage[i];
        delete [] DecImage;
        DecImage = 0;
        fclose (output);


        ::SetCursor(::LoadCursor(NULL,IDC_ARROW));
        StatusMsg("");
        char buff[200];

        sprintf(buff,"Done no bits =%d",countbits);
        Print(buff);
}

void CStegDoc::OnUpdateSteganographyAdddata(CCmdUI* pCmdUI)
{
        // TODO: Add your command update UI handler code here
pCmdUI->Enable();
}



/* Event handler for Get steganographic data. */
/* This is the driver code to get steganographic info from an image file. */

void CStegDoc::OnSteganographyGetdata()
{
        // TODO: Add your command handler code here
        GetStegan dia;
        DecS.ClearIt();
        RGBtoImage(0);
//testfile = fopen("pack.out","w");
        ::SetCursor(::LoadCursor(NULL,IDC_WAIT));
        EncodeIt(0);
        ::SetCursor(::LoadCursor(NULL,IDC_ARROW));
        StatusMsg("");
        dia.m_stegan = DecS.Data();
        dia.DoModal();
}

void CStegDoc::OnUpdateSteganographyGetdata(CCmdUI* pCmdUI)
{
        // TODO: Add your command update UI handler code here

                pCmdUI->Enable();

}
```

```
/* Code for manipulating DIB (BMP file) not included. This code was taken from Kruginski64. */
```

C.3 The JPEG Convertion Code

```
double C(int n)
{
if (!n)
          return sqrt2;
return 1.0;
}

void fdct(IMAGE_TYPE **Image,int height,int width)
{
double dct[8][8];
int x,y,u,v,h,w;
double sum;

for (h=0;h<height;h+=JN)
for (w=0;w<width;w+=JN)   {
for (u=0;u<JN;u++)
   for (v=0;v<JN;v++)
          {
          sum =0.0;

          for (x=0;x<JN;x++)
             for (y=0;y<JN;y++)
                   sum += ((double)(Image[h+x][w+y])-128.0)*coses[x][u]*
                             coses[y][v];
          dct[u][v] = sqrt8*C(u)*C(v)*sum;
          }
for (u=0;u<JN;u++)
  for (v=0;v<JN;v++)
    Image[h+u][w+v] = dct[u][v];
          }
}



void idct(IMAGE_TYPE **image,int height,int width)
{
unsigned char pix[8][8];
int x,y,u,v,h,w;
double sum;

for (h=0;h<height;h+=JN)
for(w=0;w<width;w+=JN)            {
for (x=0;x<JN;x++)
   for (y=0;y<JN;y++)
          {
          sum =0.0;

          for (u=0;u<JN;u++)
            for (v=0;v<JN;v++)
                   sum += C(u)*C(v)*(double)(image[h+u][w+v])*coses[x][u]*
                             coses[y][v];

          pix[x][y] = 128.5+sqrt8*sum;
```

```
            }
for(u=0;u<JN;u++)
  for (v=0;v<JN;v++)
     image[h+u][w+v] = pix[u][v];
  }
}
```

C.4 The Hiding and Retrieving Code

```
#ifndef __ENCH__
#define __ENCH__


#define IMAGE_TYPE unsigned char /* may be different in some applications */
extern int countbits,Nobits,hsize,vsize,shsize,svsize;
extern IMAGE_TYPE ** EncImage;
extern IMAGE_TYPE ** DecImage;


/* used for creating the steganographic image file. */
#define matrix_allocate(matrix, hsize, vsize, TYPE) {\
   TYPE *imptr; \
   int _i; \
   matrix = (TYPE **)new char [((vsize)*sizeof(TYPE *))];\
   if (matrix == NULL) \
      Fatal("\nNo memory in matrix allocate."); \
   for (_i = 0; _i<vsize; ++_i, imptr += hsize){ \
      matrix[_i] = (TYPE *)new char [hsize * sizeof(TYPE)]; \
      if (matrix[_i] == NULL )\
         Fatal("\nNo memory in matrix allocate."); \
      }\
 }

#define DeleteMatrix(matrix, vsize) {\
   int _i; \
   for (_i = 0; _i<vsize; ++_i) { \
      delete matrix[_i]; \
      } \
   delete matrix; \
}

#endif

#define SBITS  5          /* number of bits used to store scale factor */
#define OBITS  7          /* number of bits used to store offset       */
#define SYM_OP 3
                 /* largest square image that fits in image   */
#define MINPART 4        /* min and max _part determine a range of   */
#define  MAXPART 5        /* Range sizes from hsize>>min to hsize>>max */
#define  DOMSTEP 1        /* Density of domains relative to size       */
#define DOMSTEPTYPE 0   /* Flag for dom_step a multiplier or divisor */
#define DOMTYPE 0        /* Method of generating domain pool 0,1,2..  */
#define POSTPROCESS 1     /* Flag for postprocessing.                  */
#define NUMITERATIONS 10  /* Number of decoding iterations used.       */

#define OUTPUTPARTITION 0 /* A flag for outputing the partition        */

#define ONLYPOSITIVE   0    /* A flag specifying use of positive scaling */
#define SUBCLASSSEARCH 1    /* A flag specifying classes searched        */
#define FULLCLASSSEARCH 1 /* A flag specifying classes searched        */
```

```
#define INSERTTOL 8.0          /* The tolerance value */

#define CORRmin 0.96           /* CORRmin value */
```

/* Encgray.cpp

Code for hiding and revealing algorithms. */

/* The basis of this module is taken from Yuval Fisher's Fractal compression code. We include his Copyright notice. */

```
/* Encode a byte image using a fractal scheme with a quadtree partition   */
/*                                                                        */
/*      Copyright 1993,1994 Yuval Fisher. All rights reserved.            */
/*                                                                        */
/* Version 0.03 3/14/94                                                   */



#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "diblook.h"
#include <limits.h>
#include <io.h>
#include "dibdoc.h"
#include "proto.h"
#include <math.h>
#include "encgray.h"
#include "stegan.h"

extern int alloced;
extern Stegan EncS;
extern Stegan DecS;
unsigned char Thebit;
int countbits = 0;

IMAGE_TYPE ** DecImage;
extern FILE *testfile,* output;
void list_free(struct classified_domain * nodenode);
void list_delete(struct classified_domain *node);
void apply_transformations(int);


#define DEBUG 0
#define GREY_LEVELS 255

#define bound(a)  ((a) < 0.0 ? 0 : ((a)>255.0? 255 : a))
#define IMAGE_TYPE unsigned char /* may be different in some applications */

/* various function declarations to keep compiler warnings away. ANSI   */
/* prototypes can go here, for the hearty.                              */
```

```
#define swap(a,b,TYPE)      {TYPE _temp; _temp=b; b=a; a= _temp;}

static IMAGE_TYPE **Encimage;

IMAGE_TYPE ** EncImage;
double     **domimage[4];   /* Decimated input image used for domains    */

int Finish = 0;
double max_scale = 1.0;     /* Maximum allowable grey level scale factor */
static int paptr = 1, /* how many bits are packed in sum so far */
           pasum = 0, /* packed bits */
           panum_of_packed_bytes = 0; /* total bytes written out */

static int
         FirstStegan,
         s_bits = SBITS,      /* Number of bits used to store scale factor */
         o_bits = OBITS,      /* Number of bits used to store offset       */
         min_part = MINPART,      /* Min and max _part determine a range of    */
         max_part = MAXPART,    /* Range sizes from hsize>>min to hsize>>max */
         dom_step = DOMSTEP,    /* Density of domains relative to size       */
         dom_step_type = DOMSTEPTYPE,    /* Flag for dom_step a multiplier or divisor */
         dom_type = DOMTYPE,    /* Method of generating domain pool 0,1,2..  */
         only_positive = ONLYPOSITIVE,    /* A flag specifying use of positive scaling */
         subclass_search = SUBCLASSSEARCH,  /* A flag specifying classes searched  */
         fullclass_search = FULLCLASSSEARCH, /* A flag specifying classes searched */
         *bits_needed,       /* Number of bits to encode domain position. */
         zero_ialpha,        /* The const ialpha when alpha = 0             */
         max_exponent;       /* The max power of 2 side of square image    */
                               /* that fits in our input image.          */

                               /* The class_transform gives the transforms  */
                               /* between classification numbers for       */
                               /* negative scaling values, when brightest   */
                               /* becomes darkest, etc...                   */
int    class_transform[2][24] = {23,17,21,11,15,9,22,16,19,5,13,3,20,10,18,
                                    4,7,1,14,8,12,2,6,0,
                                    16,22,10,20,8,14,17,23,4,18,2,12,11,21,5,
                                    19,0,6,9,15,3,13,1,7};

                               /* rot_transform gives the rotations for     */
                               /* domains with negative scalings.            */
int    rot_transform[2][8] = {7,4,5,6,1,2,3,0, 2,3,0,1,6,7,4,5};

struct transformation_node {
    int rx,ry,          /* The range position and size in a trans.  */
           xsize, ysize,
           rrx,rry,
           dx,dy;            /* The domain position.                   */
    int sym_op;     /* The symmetry operation used in the trans. */
    int depth;          /* The depth in the quadtree partition.     */
    double scale, offset; /* scalling and offset values.         */
    double scale1,offset1;
    double rms;
    int pos;
    struct transformation_node *next;       /* The next trans. in list */
```

```
} transformations, *trans,*prev;

struct domain_pixels {      /* This is a three (sigh) index array that   */
      int dom_x, dom_y;     /* dynamically allocated. The first index is */
      double sum,sum2;      /* the domain size, the other are two its    */
      int sym;              /* position. It contains the sum and sum^2   */
};

/* The Domain data structure. */
struct domain_data {
      int *no_h_domains,    /* The number of domains horizontally for   */
          *no_v_domains,    /* each size.                               */
          *domain_hsize,    /* The size of the domain.                  */
          *domain_vsize,    /* The size of the domain.                  */
          *domain_hstep,    /* The density of the domains.          */
          *domain_vstep;    /* The density of the domains.          */
        domain_pixels*** pixel;/* of the pixel values in the domains, which */
} domain;                   /* are computed just once.                  */


struct classified_domain {      /* This is a list which containes  */
      struct domain_pixels *the;      /* pointers to  the domain data    */
      struct classified_domain *next; /* in the structure above. There   */
} **the_domain[3][24];              /* are three classes with 24 sub-  */
                                        /* classes. Using this array, only */
                                        /* domains and ranges in the same  */
                                        /* class are compared..            */
                                        /* The first pointer points to the */
                                        /* domain size the the second to   */
                                        /* list of domains.                */
#define STEGSIZE 64
int Inserting;


/* Driver function for Hiding and Revealing. */
void CStegDoc::HideRevealIt(int inserting)
{
   /* Defaults are set initially */
   double          tol = 8.0;          /* Tolerance value for quadtree.  */
   char     inputfilename[200];
   char     outputfilename[200];
   int       i,j,k;
                                  /* size of the input image.      */
   long     stripchar=0;       /* chars to ignore in input file. */
   FILE     *input;
   int  num_iterations= NUMITERATIONS;

   inputfilename[0] = 1;  /* We initially set the input to this and */
   outputfilename[0] = 1; /* then check if the input/output names   */
                                /* have been set below.              */
   paptr = 1;                              /* how many bits are packed in sum so far */
   pasum = 0;                              /* packed bits */
   panum_of_packed_bytes = 0; /* total bytes written out */
         Inserting = inserting;
if ( Inserting ){
  trans = &transformations;
  trans->next = 0;
}
```

```
else
{


            trans = transformations.next;
FirstStegan = 1;
    if (hsize == -1)
            if (vsize == -1) hsize = vsize = 256;
            else hsize = vsize;
    else
            if (vsize == -1) vsize = hsize;

    /* allocate memory for the input image. Allocating one chunck saves  */
    /* work and time later.                                              */
if (!alloced ) {
    matrix_allocate(domimage[0], (hsize/2), (vsize/2), double)
    matrix_allocate(domimage[1], (hsize/2), (vsize/2), double)
    matrix_allocate(domimage[2], (hsize/2), (vsize/2), double)
    matrix_allocate(domimage[3], (hsize/2), (vsize/2), double)
    }
    Finish = 0;
    countbits = 0;
    if ( inserting )
      {
      testfile = fopen("ins.txt","w");
              if (!(EncS>>(Thebit)))
              Finish = 1;
      tol = INSERTTOL;
      }
    else
            {
            tol = RETRIEVETOL;
            testfile = fopen("ret.txt","w");
            }
    Inserting = inserting;
    PrintPix(EncImage,0,8);

    /* max_ & min_ part are variable, so this must be run time allocated */
    bits_needed = new int [(1+max_part-min_part)];

  /*  if (i < hsize*vsize)
            Fatal("Not enough input data in the input file.");
    else
            {
            char buf[200];
            sprintf(buf,"%dx%d=%d pixels read from %s.", hsize,vsize,i,inputfilename);
                        Print(buf);
                        }*/
    /* allcate memory for domain data and initialize it */
    compute_sums(hsize,vsize);



    /* output some data into the outputfile.                    */


            /* This is the quantized value of zero scaling.. needed later */
    zero_ialpha = 0.5 + (max_scale)/(2.0*max_scale)*(1<<s_bits);
  /* The following routine takes a rectangular image and calls the */
  /* quadtree routine to encode square sum-images in it.           */
  /* the tolerance is a parameter since in some applications different */
  /* regions of the image may need to be compressed to different tol's */
  StatusMsg("Encoding Image.....");
  partition_image(0, 0,STEGSIZE ,STEGSIZE, tol);
  StatusMsg("Done.");
  if ( Inserting )
    {
//                    for (i=0; i<10; ++i)
                              apply_transformations(0);
                    }
  /* stuff the last byte if needed */


if (testfile)
  fclose(testfile);

if (fpr2)
          fclose(fpr2);


for (i=0;i<10;i++){
  if ( fpR[i])
            fclose(fpR[i]);
  }
  /* Free allocated memory*/
  delete [] bits_needed;
/*    for (i=0;i<vsize/2;i++)
          {
          delete [] domimage[0][i];
          delete [] domimage[1][i];
          delete [] domimage[2][i];
          delete [] domimage[3][i];
          }
  delete [] domimage[0];
  delete [] domimage[1];
  delete [] domimage[2];
  delete [] domimage[3]; */
  delete [] domain.no_h_domains;

  delete [] domain.domain_hsize;
  delete [] domain.domain_vsize;
  delete [] domain.domain_hstep;
  delete [] domain.domain_vstep;
  for (i=0; i <= max_part-min_part; ++i)
          for (j=0;j< domain.no_v_domains[i];j++)
              delete [] domain.pixel[i][j];
  for (i=0; i <= max_part-min_part; ++i)
              delete [] domain.pixel[i];
  delete [] domain.no_v_domains;
  delete [] domain.pixel;
  /*for (i=0;i<vsize;i++)
          delete [] EncImage[i];
  delete [] EncImage;
  EncImage = 0;*/
```

```
/* not average 2x2 sub-images to pixels. This is needed for clas- */
/* sifying ranges rather than domain where decimation is needed.  */
/* ****************************************************************** */
int gsame  = 0;
int gsameflag = 0;
void average1(int x,int y,int xsize,int ysize, double * psum,double * psum2)
{
   register int i,j;
   register double pixel;
   *psum = *psum2 = 0.0;
   gsame = EncImage[y][x];
   gsameflag = 0;
   for (i=x; i<x+xsize; ++i)
   for (j=y; j<y+ysize; ++j) {
      if ( EncImage[j][i] != gsame )
            gsameflag++;
             pixel = (double)EncImage[j][i];
      *psum += pixel;
      *psum2 += pixel*pixel;
   }
}


/* ****************************************************************** */
/* Take a region of the image at x,y and classify it.        */
/* The four quadrants of the region are ordered from brightest to */
/* least bright average value, then it is rotated into one of the */
/* three cannonical orientations possible with the brightest quad */
/* in the upper left corner.                                  */
/* The routine returns two indices that are class numbers: pfirst */
/* and psecond; the symmetry operation that bring the square into */
/* cannonical position; and the sum and sum^2 of the pixel values */
/* ****************************************************************** */
void classify(int x, int y, int xsize, int ysize,int * pfirst,int * psecond,
                     int * psym, double * psum, double *psum2, int type)
  /* position, size of subimage to be classified */
  /* returned first and second class numbers      */
   /* returned symmetry operation that brings the */
  /* subimage to cannonical position.             */
 /* returned sum and sum^2 of pixel values       */
              /* flag for decimating (for domains) or not    */
{

   int order[4], i,j;
   double a[4],a2[4];
   void (*average_func)(int,int,int ,int , double * ,double * );

   if (type == 2) average_func = average; else average_func = average1;

   /* get the average values of each quadrant                    */


   (*average_func)(x,y,                xsize/2,ysize/2,   &a[0], &a2[0]);
   (*average_func)(x,y+ysize/2,        xsize/2,ysize/2,   &a[1], &a2[1]);
   (*average_func)(x+xsize/2,y+ysize/2, xsize/2,ysize/2,   &a[2], &a2[2]);
   (*average_func)(x+xsize/2,y,        xsize/2,ysize/2,   &a[3], &a2[3]);

   *psum = a[0] + a[1] + a[2] + a[3];
```

```
*psum2 =  a2[0] + a2[1] + a2[2] + a2[3];

for (i=0; i<4; ++i) {
        /* after the sorting below order[i] is the i-th brightest   */
        /* quadrant.                                                  */
        order[i] = i;
        /* convert a2[] to store the variance of each quadrant      */
        a2[i] -= (double)(1<<(2*type))*a[i]*a[i]/(double)(xsize*ysize);
}

/* Now order the average value and also in order[],   which will */
/* then tell us the indices (in a[]) of the brightest to darkest */
for (i=2; i>=0; --i)
for (j=0; j<=i; ++j)
        if (a[j]<a[j+1]) {
          swap(order[j], order[j+1],int)
          swap(a[j], a[j+1],double)
}

/* because of the way we ordered the a[] the rotation can be */
/* read right off of order[]. That will make the brightest   */
/* quadrant be in the upper left corner. But we must still   */
/* decide which cannonical class the image portion belogs    */
/* to and whether to do a flip or just a rotation. This is   */
/* the following table summarizes the horrid lines below     */
/* order      class              do a rotation          */
/* 0,2,1,3     0                 0                      */
/* 0,2,3,1     0                 1                      */
/* 0,1,2,3     1                 0                      */
/* 0,3,2,1     1                 1                      */
/* 0,1,3,2     2                 0                      */
/* 0,3,1,2     2                 1                      */

*psym = order[0];
/* rotate the values */
for (i=0; i<4; ++i)
        order[i] = (order[i] - (*psym) + 4)%4;

for (i=0; order[i] != 2; ++i);
*pfirst = i-1;
if (order[3] == 1 || (*pfirst == 2 && order[2] == 1)) *psym += 4;

/* Now further classify the sub-image by the variance of its   */
/* quadrants. This give 24 subclasses for each of the 3 classes */
for (i=0; i<4; ++i) order[i] = i;

for (i=2; i>=0; --i)
for (j=0; j<=i; ++j)
        if (a2[j]<a2[j+1]) {
          swap(order[j], order[j+1],int)
          swap(a2[j], a2[j+1],double)
}

/* Now do the symmetry operation */
for (i=0; i<4; ++i)
        order[i] = (order[i] - (*psym%4) + 4)%4;
if (*psym > 3)
        for (i=0; i<4; ++i)
```

```
            if (order[i]%2) order[i] = (2 + order[i])%4;

    /* We want to return a class number from 0 to 23 depending on */
    /* the ordering of the quadrants according to their variance  */
    *psecond = 0;
    for (i=2; i>=0; --i)
    for (j=0; j<=i; ++j)
          if (order[j] > order[j+1]) {
             swap(order[j],order[j+1], int);
             if (order[j] == 0 || order [j+1] == 0)
                   *psecond += 6;
             else if (order[j] == 1 || order [j+1] == 1)
                   *psecond += 2;
             else if (order[j] == 2 || order [j+1] == 2)
                   *psecond += 1;
          }
}

/* ************************************************************* */
/* Compute sum and sum^2 of pixel values in domains for use in  */
/* the rms computation later. Since a domain is compared with   */
/* many ranges, doing this just once saves a lot of computation */
/* This routine also fills a list structure with the domains    */
/* as they are classified and creates the memory for the domain */
/* data in a matrix.                                            */
/* ************************************************************* */
void compute_sums(int hsize,int vsize)
{
    int i,j,k,l,
          domain_x,
          domain_y,
          first_class,
          second_class,
          domain_size,
          domain_step_size,
          size,
          x_exponent,
          y_exponent;

    struct classified_domain *node;

    StatusMsg("Computing domain sums... ");


    /* pre-decimate the image into domimage to avoid having to     */
    /* do repeated averaging of 2x2 pixel groups.                  */
    /* There are 4 ways to decimate the image, depending on the    */
    /* location of the domain, odd or even address.                */
    for (i=0; i<2; ++i)
    for (j=0; j<2; ++j)
    for (k=i; k<hsize-i; k += 2)
    for (l=j; l<vsize-j; l += 2)
          domimage[(i<<1)+j][l>>1][k>>1] =
                      ((double)EncImage[l][k] + (double)EncImage[l+1][k+1] +
                      (double)EncImage[l][k+1] + (double)EncImage[l+1][k])*0.25;


    /* Allocate memory for the sum and sum^2 of domain pixels           */
/* We first compute the size of the largest square that fits in  */
/* the image.                                                    */
x_exponent = (int)floor(log((double)hsize)/log(2.0));
y_exponent = (int)floor(log((double)vsize)/log(2.0));

/* exponent is min of x_ and y_ exponent */
max_exponent = (x_exponent > y_exponent ? y_exponent : x_exponent);

/* size is the size of the largest square that fits in the image */
/* It is used to compute the domain and range sizes.              */
size =  1<<max_exponent;

if (max_exponent < max_part)
  Fatal("Reduce maximum number of quadtree partitions.");
if (max_exponent-2 < max_part)
  Print("\nWarning: so many quadtree partitions yield absurd ranges.");

i = max_part - min_part + 1;
domain.no_h_domains = new int [i];
domain.no_v_domains = new int [i];
domain.domain_hsize = new int [i];
domain.domain_vsize = new int [i];
domain.domain_hstep = new int [i];
domain.domain_vstep = new int [i];

domain.pixel= (struct domain_pixels ***)
               new char [(i*sizeof(struct domain_pixels **))];
if (domain.pixel == NULL) Fatal("No memory for domain pixel sums.");

for (i=0; i <= max_part-min_part; ++i) {
  /* first compute how many domains there are horizontally */
  domain.domain_hsize[i] = size >> (min_part+i-1);
  if (dom_type == 2)
          domain.domain_hstep[i] = dom_step;
  else if (dom_type == 1)
          if (dom_step_type == 1)
            domain.domain_hstep[i] = (size >> (max_part - i-1))*dom_step;
          else
            domain.domain_hstep[i] = (size >> (max_part - i-1))/dom_step;
  else
          if (dom_step_type == 1)
            domain.domain_hstep[i] = domain.domain_hsize[i]*dom_step;
          else
            domain.domain_hstep[i] = domain.domain_hsize[i]/dom_step;

  domain.no_h_domains[i] = 1+(hsize-domain.domain_hsize[i])/
                                            domain.domain_hstep[i];
  /* bits_needed[i][0] = ceil(log(domain.no_h_domains[i])/log(2.0)); */

  /* now compute how many domains there are vertically. The sizes  */
  /* are the same for square domains, but not for rectangular ones */
  domain.domain_vsize[i] = size >> (min_part+i-1);
  if (dom_type == 2)
          domain.domain_vstep[i] = dom_step;
  else if (dom_type == 1)
          if (dom_step_type == 1)
            domain.domain_vstep[i] = (size >> (max_part - i-1))*dom_step;
          else
```

```
            domain.domain_vstep[i] = (size >> (max_part - i-1))/dom_step;
    else
          if (dom_step_type == 1)
            domain.domain_vstep[i] = domain.domain_vsize[i]*dom_step;
          else
            domain.domain_vstep[i] = domain.domain_vsize[i]/dom_step;

    domain.no_v_domains[i] = 1+(vsize-domain.domain_vsize[i])/
                                                  domain.domain_vstep[i];

    /* Now compute the number of bits needed to store the domain data */
    bits_needed[i] = ceil(log((double)domain.no_h_domains[i]*
                                   (double)domain.no_v_domains[i])/log(2.0));

    matrix_allocate(domain.pixel[i], domain.no_h_domains[i],
                domain.no_v_domains[i], struct domain_pixels)

}

/* allocate and zero the list containing the classified domain data */
i = max_part - min_part + 1;
for (first_class = 0; first_class < 3; ++first_class)
for (second_class = 0; second_class < 24; ++second_class) {
  the_domain[first_class][second_class] =
                          (struct classified_domain **)
                          new char [(i*sizeof(struct classified_domain *))];
  for (j=0; j<i; ++j)
                the_domain[first_class][second_class][j] = NULL;
}

/* Precompute sum and sum of squares for domains                   */
/* This part can get faster for overlapping domains if repeated */
/* sums are avoided                                                */
for (i=0; i <= max_part-min_part; ++i) {
  for (j=0,domain_x=0; j<domain.no_h_domains[i]; ++j,
          domain_x+=domain.domain_hstep[i])
  for (k=0,domain_y=0; k<domain.no_v_domains[i]; ++k,
          domain_y+=domain.domain_vstep[i]) {
              classify(domain_x, domain_y,
               domain.domain_hsize[i],
               domain.domain_vsize[i],
                                        &first_class, &second_class,
                                        &domain.pixel[i][k][j].sym,
                &domain.pixel[i][k][j].sum,
                &domain.pixel[i][k][j].sum2, 2);

      /* When the domain data is referenced from the list, we need to */
      /* know where the domain is.. so we have to store the position  */
      domain.pixel[i][k][j].dom_x = j;
      domain.pixel[i][k][j].dom_y = k;
      node = (struct classified_domain *)
                               new char [(sizeof(struct classified_domain))];

      /* put this domain in the classified list structure */
      node->the = &domain.pixel[i][k][j];
      node->next = the_domain[first_class][second_class][i];
      the_domain[first_class][second_class][i] = node;
       }
  }

  /* Now we make sure no domain class is actually empty.  */
  for (i=0; i <= max_part-min_part; ++i)
  for (first_class = 0; first_class < 3; ++first_class)
  for (second_class = 0; second_class < 24; ++second_class)
    if (the_domain[first_class][second_class][i] == NULL) {
           node = (struct classified_domain *)
                                         new char [(sizeof(struct classified_domain))];
           node->the = &domain.pixel[i][0][0];
           node->next = NULL;
           the_domain[first_class][second_class][i] = node;
    }



}


double domval,rangeval,galpha,gbeta;

int gcmp;

int doit;
int cntit;
int findalpha(int i, int j,int i1,int j1)
{
static unsigned char DI,DI1;
static double pixel,pixel1;

          if ( doit == 2 ) {
            galpha=((double)DI - (double)DI1)/(pixel-pixel1);
            gbeta = (double)DI1 - galpha*pixel1;
            return 1;
                      /*fprintf(testfile,"DI=%d DI1 = %d s=%lf o=%lf",DI,DI1,galpha,gbeta);    */    \
            }
          cntit++;
          if ( cntit > 4 )
            return 0;
  DI = EncImage[j][i];
  if (DI <= 1 || DI >= 254 )
    if ( !doit )
           return 0;
    else
           return 2;
  if ( doit == 0 )
    {
    DI1 = DI;
    pixel1 = ((EncImage[j1][i1]+EncImage[j1][i1+1]+EncImage[j1+1][i1]+
                  EncImage[j1+1][i1+1])/4.0);
    doit++;
    }
  else
    if ( DI != DI1 )
           {
           pixel = ((EncImage[j1][i1]+EncImage[j1][i1+1]+EncImage[j1+1][i1]+
                                    EncImage[j1+1][i1+1])/4.0);
           if ( fabs((pixel - pixel1)) < 0.1)
```

```
            return 0;
          doit++;
          }

return 3;
}

int RetrieveCmp(int i, int j, int i1, int j1)
{
unsigned char en,en1;
double d,p;

  p = ((double)(EncImage[j1][i1]+EncImage[j1][i1+1]+EncImage[j1+1][i1]+
                      EncImage[j1+1][i1+1])/4.0);
  en = EncImage[j][i];
  en1 = bound(0.5+ galpha*p+gbeta);
  if ( en && en !=255 )
    {
    if ( gcmp = (en - en1 != 0) )
          return 1;
    }
return 0;
}
int randval=1;
int r2cnt=0,point9;
double MM,BB,R2,r2sums;


/* ************************************************************* */
/* Compare a range to a domain and return the correlation coefficient. This function
/* Is also used to build the RMS value for a domain and range.              */
/* ************************************************************* */
double compare(int atx, int aty, int xsize, int ysize, int depth,
                                   double rsum, double rsum2, int dom_x, int dom_y,
                  int sym_op, int * pialpha, int * pibeta)
{
   int i, j, amt,i1, j1, k,
          domain_x,
          domain_y;          /* The domain position                    */
   double pixel1,a,b,d,pixel=0.0,
              det,       /* determinant of solution */
              dsum,  /* sum of domain values */
              rdsum = 0.0,   /* sum of range*domain values   */
              dsum2,            /* sum of domain^2 values   */
              w2 = 0,       /* total number of values tested */
              rms = 0.0,      /* root means square difference */
              alpha,  /* the scale factor */
              beta;    /* The offset */
int first = 1,m=0,m1,m2,valsub,totvalsub,found=0;
int ret=0,DI,DI1,am,amount;
double x,y,r=0.0,N,t2=0.0,B=0.0,M=0.0,p=0.0,p1=0.0,s0=0.0,s1=0.0,s2=0.0,t0=0.0,t1=0.0;
unsigned char calcMB=1,dm,en,en1;
double R3,EV=0.0,EV1,TV=0.0,TV1,yhat;

   /* xsize = hsize >> depth; */
   /* ysize = vsize >> depth; */
   gcmp = 1;
   doit = 0;
```

```
    cntit=0;
    w2 = xsize * ysize;
    domval = rangeval = 0.0;
    dsum = domain.pixel[depth-min_part][dom_y][dom_x].sum;
            yhat = dsum/(double)w2;
    dsum2 = domain.pixel[depth-min_part][dom_y][dom_x].sum2;
    domain_x = (dom_x * domain.domain_hstep[depth-min_part]);
    domain_y = (dom_y * domain.domain_vstep[depth-min_part]);
    k = ((domain_x%2)<<1) + domain_y%2;
    am = 2;
    amt = 2;

                        // used as inner loop index as well
//    if ( Inserting)
      {
      am = 1;
      amt = 1;
      domain_x >>= 1;
      domain_y >>= 1;
      }

    /* The main statement in this routine is a switch statement which */
    /* scans through the domain and range to compare them. The loop   */
    /* center is the same so we #define it for easy modification      */


#define COMPUTE_LOOP     { if (Inserting)   {                              \
            if (!around){\
            pixel = domimage[k][j1][i1];                             \
            rdsum += EncImage[j][i]*pixel;                           \
              }\
            else{\/* Correlation Coefficient Calculation. */
            x = domimage[k][j1][i1];\
            y = EncImage[j][i];\
            EV1 = (alpha*x+beta-yhat);\
            EV += EV1*EV1;\
            TV1 = y-yhat;\
            TV += TV1*TV1;\
              }\
            }\
            else \
              {\
                if ( x && x < 254 )\
                        {\
                        s0++;  \
              y = domimage[k][j1][i1]+valsub;totvals+=domimage[k][j1][i1]; \
              if (found){dsum+=y;dsum2+=y*y;}\
                        t0 += y;\
                        t1 += y*x;\
                        t2 += y*y;\
                        s1 += x;\
                        s2 += x*x;\
                        }\
              }             \
            }


int totvals;
```

```
for ( amount=0;amount<1;amount++)  {
    switch(sym_op) {
             case 0:
            for (i=atx, i1 = domain_x; i<atx+xsize; ++i, i1+=am)
                        for (j=aty, j1 = domain_y; j<aty+ysize; ++j, j1+=am)
                            COMPUTE_LOOP
                                     break;
             case 1: for (j=aty, i1 = domain_x; j<aty+ysize; ++j, i1 += am)
                         for (i=atx+xsize-1, j1 = domain_y; i>=atx; --i, j1 += am)
                            COMPUTE_LOOP
                         break;
             case 2: for (i=atx+xsize-1, i1 = domain_x; i>=atx; --i, i1 += am)
                         for (j=aty+ysize-1, j1 = domain_y; j>=aty; --j,  j1 += am)
                            COMPUTE_LOOP
                         break;
             case 3: for (j=aty+ysize-1, i1 = domain_x; j>=aty; --j, i1 += am)
                         for (i=atx, j1 = domain_y; i<atx+xsize; ++i,  j1 += am)
                            COMPUTE_LOOP
                         break;
             case 4:  for (j=aty, i1 = domain_x; j<aty+ysize; ++j, i1 += am)
                         for (i=atx, j1 = domain_y; i<atx+xsize; ++i,  j1 += am)
                            COMPUTE_LOOP
                          break;
             case 5:   for (i=atx, i1 = domain_x; i<atx+xsize; ++i, i1 += am)
                         for (j=aty+ysize-1, j1 = domain_y; j>=aty; --j, j1 += am)
                COMPUTE_LOOP
                         break;
             case 6: for (j=aty+ysize-1, i1 = domain_x; j>=aty; --j, i1 += am)
                         for (i=atx+xsize-1, j1 = domain_y; i>=atx; --i,  j1 += am)
                            COMPUTE_LOOP
                        break;
             case 7:  for (i=atx+xsize-1, i1 = domain_x; i>=atx; --i, i1 += am)
                         for (j=aty, j1 = domain_y; j<aty+ysize; ++j, j1 += am)
            COMPUTE_LOOP
                         break;
    }

}


    if (!Inserting )
            {/* retrieving.
            IsRangeTransformed Pseudo code.*/
            int mm;
            double dem;
            double sqrtit;
            double sqr;
             N = s0;
            sqrtit = (N*s2-s1*s1)*(N*t2-t0*t0);
            if ( sqrtit < 0.0 )
              return 0.0;
            sqr = sqrt(sqrtit);
            s0++;
            dem = s0*s2 - s1*s1;
            if ( dem < 1.0E-10 || sqr < 1.0E-10)
              return 0.0;
            MM= M = (s0*t1 - s1*t0) / dem;
            BB =B = (s2*t0 - s1*t1) / dem;
            r = (N*t1 - s1*t0)/sqr;
```

```
                R2 = r;
                r2sums += fabs(r);


        if ( fabs(r) > CORRmin)
           gcmp = 0;


    }
/* hiding */
else{/* Hide Pseudo Code */
   det = (xsize*ysize)*dsum2 - dsum*dsum;

   if (det == 0.0)
         alpha = 0.0;
    else
         alpha = (w2*rdsum - rsum*dsum)/det;

   if  (only_positive && alpha < 0.0) alpha = 0.0;
   *pialpha = 0.5 + (alpha + max_scale)/(2.0*max_scale)*(1<<s_bits);
   if (*pialpha < 0) *pialpha = 0;
   if (*pialpha >= 1<<s_bits) *pialpha = (1<<s_bits)-1;


   alpha = (double)*pialpha/(double)(1<<s_bits)*(2.0*max_scale)-max_scale;

   /* compute the offset */
   beta = (rsum - alpha*dsum)/w2;

   /* Convert beta to an integer */
   /* we use the sign information of alpha to pack efficiently */
   if (alpha > 0.0)  beta += alpha*GREY_LEVELS;
   *pibeta = 0.5 + beta/
            ((1.0+fabs(alpha))*GREY_LEVELS)*((1<<o_bits)-1);
   if (*pibeta< 0) *pibeta = 0;
   if (*pibeta>= 1<<o_bits) *pibeta = (1<<o_bits)-1;

   /* Recompute beta from the integer */
   beta = (double)*pibeta/(double)((1<<o_bits)-1)*
            ((1.0+fabs(alpha))*GREY_LEVELS);
   if (alpha > 0.0) beta  -= alpha*GREY_LEVELS;

   }
  else{
            R2 = (EV/TV);
            if ( R2 > 0.95 )
                     R3 = 0.96;
  }
}
if (Inserting)
  /* Compute the rms based on the quantized alpha and beta! */
   rms= sqrt((rsum2 + alpha*(alpha*dsum2 - 2.0*rdsum + 2.0*beta*dsum) +
                   beta*(beta*w2 - 2.0*rsum))/w2);

else
  if ( found && trans)
     trans=trans->next;
    return(rms);
}
int bigcount = 0;
int nokey=1;
char gbest_ialpha,gbest_ibeta;



/* ************************************************************ */
/* Recursively partition an image, computing the best transfoms */
/* ************************************************************ */
void quadtree(int atx, int aty, int xsize,int ysize, double tol, int depth)
  /* the tolerance fit  */
{
   double alpha,beta;
   int fcount1,i,doit,count,found=0,count1,
           domain_x,
           domain_y,
           sym_op,           /* A symmetry operation of the square */
           ialpha,           /* Intgerized scalling factor         */
           ibeta,            /* Intgerized offset                  */
           best_ialpha,      /* best ialpha found                  */
           best_ibeta,
           best_sym_op,
           best_domain_x,
           best_domain_y,
           first_class,
           the_first_class,
           first_class_start,  /* loop beginning and ending values  */
           first_class_end,
           second_class[2],
           the_second_class,
           second_class_start,    /* loop beginning and ending values   */
           second_class_end,
           range_sym_op[2],            /* the operations to bring square to  */
           domain_sym_op;   /* cannonical position.         */

   long domain_ref;
   struct classified_domain *node;  /* var for domains we scan through */

   double rms, best_rms,     /* rms value and min rms found so far */
             best_m,best_b,best_r2,sum=0, sum2=0;      /* sum and sum^2 of range pixels      */

    char buf[200];

    if ( Finish )
           return;
    sprintf(buf,"quadtree to x=%d y=%d",atx,aty);
    StatusMsg(buf);

   /* keep breaking down the image until we are small enough */
   if (depth < max_part) {

           quadtree(atx,aty, xsize/2, ysize/2, tol,depth+1);
           quadtree(atx+xsize/2,aty, xsize/2, ysize/2,tol,depth+1);
           quadtree(atx,aty+ysize/2, xsize/2, ysize/2,tol,depth+1);
           quadtree(atx+xsize/2,aty+ysize/2, xsize/2, ysize/2,tol,depth+1);
           return;
   }

   /* now search for the best domain-range match and write it out */
   best_rms = 20.0;      /* just a big number */

          rms = 90.0;
   classify(atx, aty, xsize,ysize,
              &the_first_class, &the_second_class,
              &range_sym_op[0], &sum, &sum2, 1);


   /* sort out how much to search based on -f and -F input flags */
   if (fullclass_search) {
          first_class_start = 0;
          first_class_end = 3;
   } else {
          first_class_start = the_first_class;
          first_class_end = the_first_class+1;
   }

   if (subclass_search) {
          second_class_start = 0;
          second_class_end = 24;
   } else {
          second_class_start = the_second_class;
          second_class_end = the_second_class+1;
   }

   /* these for loops vary according to the optimization flags we set */
   /* for subclass_search and fullclass_search==1, we search all the  */
   /* domains (except not all rotations).                             */
 count = 0;
 found = 0;
 int thecount,thecount1;
 if ( ((Inserting && gsameflag > 2) || !Inserting) && depth == max_part ){
   for (first_class = first_class_start;
          first_class < first_class_end; ++first_class)
   for (second_class[0] = second_class_start;
          second_class[0] < second_class_end; ++second_class[0]) {

     /* We must check each domain twice. Once for positive scaling,  */
     /* once for negative scaling. Each has its own class and sym_op */
     if (!only_positive) {
          second_class[1] =
                 class_transform[(first_class == 2 ? 1 : 0)][second_class[0]];
          range_sym_op[1] =
            rot_transform[(the_first_class == 2 ? 1 : 0)][range_sym_op[0]];
     }

     /* only_positive is 0 or 1, so we may or not scan               */
     for (i=0; i<1/*(2-only_positive)*/; ++i)
     for (node = the_domain[first_class][second_class[i]][depth-min_part];
           node != NULL;
           node = node->next) {

         domain_x = (node->the->dom_x * domain.domain_hstep[depth-min_part]);
   domain_y = (node->the->dom_y * domain.domain_vstep[depth-min_part]);
```

```
        /* Set the positions of the regions */
    if ( domain_y >= STEGSIZE )
            count++;
    else
           if (domain_x >= STEGSIZE)
             ;
    }
  }

  if ( count < 2 )
    return;
  doit = 0;
  fcount1=count1 = 0;
  found = 0;
  if ( depth <= max_part)
  for (first_class = first_class_start;
          first_class < first_class_end; ++first_class)
  for (second_class[0] = second_class_start;
          second_class[0] < second_class_end; ++second_class[0]) {

    /* We must check each domain twice. Once for positive scaling,  */
    /* once for negative scaling. Each has its own class and sym_op */
    if (!only_positive) {
           second_class[1] =
                 class_transform[(first_class == 2 ? 1 : 0)][second_class[0]];
           range_sym_op[1] =
             rot_transform[(the_first_class == 2 ? 1 : 0)][range_sym_op[0]];
    }

    /* only_positive is 0 or 1, so we may or not scan               */
    for (i=0; i<1/*(2-only_positive)*/; ++i)
    for (node = the_domain[first_class][second_class[i]][depth-min_part];
            node != NULL;
            node = node->next) {
            domain_sym_op = node->the->sym;
            /* The following if statement figures out how to transform  */
            /* the domain onto the range, given that we know how to get */
            /* each into cannonical position.                          */
/* Pseudo Code : Reveal(1) */
{/* Pseudo Code : Hide(2) */
              if (((domain_sym_op>3 ? 4: 0) + (range_sym_op[i]>3 ? 4: 0))%8 == 0)
                    sym_op = (4 + domain_sym_op%4 - range_sym_op[i]%4)%4;
              else
                    sym_op = (4 + (domain_sym_op%4 + 3*(4-range_sym_op[i]%4))%4)%8;
   if ( atx == 0 && aty == 0 && node->the->dom_x == 8 && node->the->dom_y == 18)
          domain_sym_op = 1;
          domain_x = (node->the->dom_x * domain.domain_hstep[depth-min_part]);
  domain_y = (node->the->dom_y * domain.domain_vstep[depth-min_part]);


            doit = 0;
            if ( domain_y >= STEGSIZE )
              doit = 1;
            else
              if (domain_x >= STEGSIZE)
                    doit = 0;
            if ( doit )
              {
```

```
        count1++;

            if (Inserting)
              rms = compare(atx,aty, xsize, ysize,  depth, sum,sum2,
                            node->the->dom_x,
                            node->the->dom_y,
                            sym_op, &ialpha,&ibeta);
            else{
              double d,b;
                                    b = best_rms;
              for (sym_op=0;sym_op<8;sym_op++)  {
                     d = compare(atx,aty, xsize, ysize,  depth, sum,sum2,
                            node->the->dom_x,
                            node->the->dom_y,
                            sym_op, &ialpha,&ibeta);
                     if ( !gcmp )
                                          {
                                              b = d;
                       rms = d;
                                              break;
                                          }
                     }
            }
            /* Pseudo Code : Hide(3) */
    //      if ( rms < INSERTTOL && depth != max_part)
     //        return;
        if ( Inserting )
            {
           if ( rms < best_rms && rms< INSERTTOL
/* Pseudo Code : Hide(1) */
&& (Thebit ? (count1 < count >> 1) : (count1 >= count >> 1))  )

            {
            thecount1 = count1;
            thecount = count;
            best_ialpha = ialpha;
            best_r2 = R2;
            best_ibeta = ibeta;
            best_m = MM;
            best_b = BB;
            best_rms = rms;
            best_sym_op = sym_op;
            best_domain_x = node->the->dom_x;
            best_domain_y = node->the->dom_y;
            found =1;
            }
           }
       else/* Pseudo Code: Reveal. */

           if ( !gcmp ){
           nokey = 1;
           if ( count1 < count>> 1)
              DecS << 1;
             else
              DecS << 0; /* Pseudo Code: Reveal. */
            if ( DecS.LsFin() )
              Finish = 1;
            return;
```

```
                    }
                  }
  }
}
  if ( found )
         {
         found = 0;
/* if ( !Inserting ){/* Pseudo Code: Reveal (2) */
         nokey = 1;
                 if ( thecount1 < thecount>> 1)
                     DecS << 1;
                   else
                     DecS << 0;
                   if ( DecS.LsFin() )
                     Finish = 1;
                 return;
        }*/
        if ( Inserting )
          {

         alpha = ((double)((best_ialpha&(0x1f)))/(double)(1<<s_bits)*(2.0*max_scale)-max_scale);

        beta = ((double)((best_ibeta&(0x7f)))/(double)((1<<o_bits)-1)*
            ((1.0+fabs(alpha))*GREY_LEVELS));

        if (alpha > 0.0) beta -= alpha*GREY_LEVELS;

  //  if ( fabs(alpha *GREY_LEVELS + beta) > 30.0 )
  //      continue;
        struct transformation_node * t;
        t = (struct transformation_node *)
                     new char [(sizeof(struct transformation_node ))];
        t->next = NULL;
        t->scale = alpha;
        t->offset = beta;
        t->next = NULL;
        if (best_ialpha != zero_ialpha) {
          t-> sym_op = best_sym_op;
          t->dx = (double)best_domain_x * domain.domain_hstep[depth-min_part];
          t->dy = (double)best_domain_y * domain.domain_vstep[depth-min_part];
        } else {
          t-> sym_op = 0;
          t-> dx  = 0;
          t-> dy = 0;
        }
        t->rx = atx;
        t->ry = aty;
        t->depth = depth;

        t->rrx = atx + xsize;
        t->rry = aty + ysize;
        t->pos = 0;
        t->rms = best_rms;
       if (!(EncS>>(Thebit)))
                     Finish = 1;
        }
```

```
            return;
            }
            }
      }
   if (!found && depth < max_part) {
      /* We didn't find a good enough fit so quadtree down */
     /* This bit means we quadtree'd down */
      quadtree(atx,aty, xsize/2, ysize/2, tol,depth+1);
      quadtree(atx+xsize/2,aty, xsize/2, ysize/2, tol,depth+1);
      quadtree(atx,aty+ysize/2, xsize/2, ysize/2, tol,depth+1);
      quadtree(atx+xsize/2,aty+ysize/2, xsize/2, ysize/2, tol,depth+1);

    }
}

/* ************************************************************ */
/* Recursively partition an image, finding the largest contained */
/* square and call the quadtree routine the encode that square.  */
/* This enables us to encode rectangular image easily.            */
/* ************************************************************ */
void partition_image(int atx, int aty, int hsize, int vsize,double tol)
{
   int x_exponent,    /* the largest power of 2 image size that fits */
      y_exponent,    /* horizontally or vertically the rectangle.   */
      exponent,      /* The actual size of image that's encoded.    */
      size,
      depth;

   x_exponent = (int)floor(log((double)hsize)/log(2.0));
   y_exponent = (int)floor(log((double)vsize)/log(2.0));

   /* exponent is min of x_ and y_ exponent */
   exponent = (x_exponent > y_exponent ? y_exponent : x_exponent);
   size = 1<<exponent;
   depth = max_exponent - exponent;
   quadtree(atx,aty,size,size,tol,depth);
   if (size != hsize)
     partition_image(atx+size, aty, hsize-size,vsize, tol);

   if (size != vsize)
     partition_image(atx, aty+size, size,vsize-size, tol);
}

/* This code builds the new range blocks. */
void apply_transformations(int which)
{
    IMAGE_TYPE **tempimage;
            int         i,j,i1,cnt=0,j1,count=0;
    double pixel,scal,off,d;
    unsigned char firstval;
    int ch,ch1;
    char array[20];



    trans = transformations.next;
    while (trans) {
            ++count;
```

```
        cnt=0;
/* Since the inner loop is the same in each case of the switch below */
/* we just define it once for easy modification.                     */
#define COMPUTE_LOOP              {                         \
        if ( cnt && cnt < trans->pos-1 )\
          DecImage[j][i] = firstval;\
        else\ /* Pseudo Code : Hide(4) */
          { scal = trans->scale1;\
            off = trans->offset1;\
        pixel = ((EncImage[j1][i1]+EncImage[j1][i1+1]+EncImage[j1+1][i1]+      \
                EncImage[j1+1][i1+1])/4.0);                          \
        ch = DecImage[j][i];\
/* Pseudo Code : Hide(5) */
        DecImage[j][i] = bound(0.5 + scal*pixel+off);\
        ch1=((unsigned int)DecImage[j][i])-ch;\
        ch1 = ch1>0 ? ch1:-ch1;\
           if (!cnt )\
          firstval = DecImage[j][i];\
        } \
        cnt++;\
        }

        switch(trans->sym_op) {
          case 0: for (i=trans->rx, i1 = trans->dx;
                          i<trans->rrx; ++i, i1 += 2)
                    for (j=trans->ry, j1 = trans->dy;
                          j<trans->rry; ++j, j1 += 2)
                     COMPUTE_LOOP
                    break;
          case 1: for (j=trans->ry, i1 = trans->dx;
                          j<trans->rry; ++j, i1 += 2)
                    for (i=trans->rrx-1,
                          j1 = trans->dy; i>=(int)trans->rx; --i, j1 += 2)
                       COMPUTE_LOOP
                    break;
          case 2: for (i=trans->rrx-1,
                          i1 = trans->dx; i>=(int)trans->rx; --i, i1 += 2)
                    for (j=trans->rry-1,
                          j1 = trans->dy; j>=(int)trans->ry; --j, j1 += 2)
                       COMPUTE_LOOP
                    break;
          case 3: for (j=trans->rry-1,
                          i1 = trans->dx; j>=(int)trans->ry; --j, i1 += 2)
                    for (i=trans->rx, j1 = trans->dy;
                          i<trans->rrx; ++i, j1 += 2)
                       COMPUTE_LOOP
                    break;
          case 4: for (j=trans->ry, i1 = trans->dx;
                          j<trans->rry; ++j, i1 += 2)
                    for (i=trans->rx, j1 = trans->dy;
                          i<trans->rrx; ++i, j1 += 2)
                       COMPUTE_LOOP
                    break;
          case 5: for (i=trans->rx, i1 = trans->dx;
                          i<trans->rrx; ++i, i1 += 2)
                    for (j=trans->rry-1,
                          j1 = trans->dy; j>=(int)trans->ry; --j, j1 += 2)
                       COMPUTE_LOOP
                    break;
            case 6: for (j=trans->rry-1,
                            i1 = trans->dx; j>=(int)trans->ry; --j, i1 += 2)
                    for (i=trans->rrx-1,
                            j1 = trans->dy; i>=(int)trans->rx; --i, j1 += 2)
                      COMPUTE_LOOP
                    break;
            case 7: for (i=trans->rrx-1,
                            i1 = trans->dx; i>=(int)trans->rx; --i, i1 += 2)
                    for (j=trans->ry, j1 = trans->dy;
                            j<trans->rry; ++j, j1 += 2)
                      COMPUTE_LOOP
                    break;
          }
  if ( trans)
          trans = trans->next;
  }


}
```

C.5 The Steganographic Bit Stream Code

```
/*Stegan Module.*/
/* Code for manipulating the steganographic bit streams. */
#ifndef __STEGANH__
#define __STEGANH__
#include <string.h>
class Stegan {
        unsigned char * data;
        unsigned char * ls;
        int posls;
        unsigned char * rs;
        int posrs,lsfin,rsfin;
        int sz;
public:
        Stegan();
        Stegan(const char * stegan);
        Stegan(int size);
        ~Stegan();
        void ClearIt();
        void initzero();
        void initsize(int siz);
        void initchar(const char *init);
        int operator>>(unsigned char & bit);
        void operator<<(unsigned char bit);
        char * Data();
        int LsFin();
};

#endif


#include "stegan.h"
char * Stegan::Data()
{
return (char *)data;
}

Stegan::Stegan()
{
data = rs=ls = 0;
sz = 0;
posls = posrs = 1;
}

void Stegan::initzero()
{
memset(data,0,sz);
}

Stegan::Stegan(int size)
{
sz = size;
data = new unsigned char[size];
ls = rs = data;
lsfin = rsfin = 0;
posls = posrs = 1;
```

```
memset(data,0,size);
}

Stegan::Stegan(const char *stegan)
{
data = new unsigned char[sz = strlen(stegan)+1];
strcpy((char *)data,stegan);
ls = rs = data;
lsfin = rsfin = 0;
posls = posrs = 1;
}

void Stegan::ClearIt()
{
if ( data )
  {
  lsfin = rsfin = 0;
  ls = rs = data;
  posls = posrs = 1;
  memset(data,0,sz);
  }
}

void Stegan::initchar(const char * init)
{
if (data )
  delete data;
data = new unsigned char[sz = strlen(init)+1];
strcpy((char *)data,init);
ls = rs = data;
posls = posrs = 1;
lsfin = rsfin = 0;
}

void Stegan::initsize(int siz)
{
if (data )
  delete data;
data = new unsigned char[sz = siz];
ls = rs = data;
posls = posrs = 1;
lsfin = rsfin = 0;
}

Stegan::~Stegan()
{
delete data;
}

int Stegan::operator>>(unsigned char & bit)
{
unsigned char thechar;

if ( rsfin )
  return 0;

if ( posrs == 0x100 )
   {
    posrs = 1;
    if (!*rs)
      rsfin = 1;
    rs++;
    }

bit = 0;
if (*rs & posrs)
  bit = 1;

posrs <<= 1;
return 1;
}

int Stegan::LsFin()
{
return lsfin;
}

void Stegan::operator<<(unsigned char  bit)
{
unsigned char thechar;

if ( lsfin )
    return;

if ( posls == 0x100 )
    {
    posls = 1;
    if ( !*ls )
      lsfin = 1;
    ls++;
    }

if (bit)
  *ls |= posls;
posls <<= 1;
}
```