A Robust Algorithm for Information Hiding in Digital Pictures

by

Raymond W. Hwang

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree

Master of Engineering

At the

The Massachusetts Institute of Technology

May 21, 1999

© Raymond Hwang, 1999. All rights reserved.

The author hereby grants MIT permission to reproduce and distribute paper and electronic copies of this document, in whole or in part, and to grant other the right to do so.

Author

Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by

Walter Bender
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Acknowledgements

I would like to thank Walter Bender for providing me with the opportunity to participate in this research. It has been an extremely valuable educational experience. The chance to work at the Media Laboratory, with its extraordinary people and technology, has been most refreshing and eye opening.

This thesis would not have been possible without the insight and guidance of Daniel Gruhl. Thank you for your patience, knowledge and humor.

Thanks to Fernando Paiz, for just being around during this whole thing.

Lastly, thanks to my friends and family (Mom, Dad, and Lindi) who have accompanied me this far.
A Robust Algorithm for Information Hiding in Digital Pictures
by
Raymond W. Hwang

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science.

Abstract

A problem inherent in information hiding techniques for digital images is that of alignment during data extraction. This alignment problem can be solved using a search approach where a combination of coarse orientation detection, random search and gradient descent methods are employed. When used in conjunction with the Patch Track algorithm, it is possible to create an information hiding and retrieval system that is robust towards rotation, cropping and noise; successful data extraction can occur with a high degree of certainty from scanned images, rotated images and partial images. An image can be encoded with a desired piece of information (a reference number, URL etc.) with Patch Track by way of pseudo-random, imperceivable alterations of intensity throughout the image. This information can be then be extracted by the intended parties with a decoder. Various applications exist for this watermarking system including photo annotation, verification of image ownership on the Internet and data warehousing.

Thesis Supervisor: Walter Bender
Title: Assistant Director of Information Technology, Senior Research Scientist, The MIT Media Laboratory
Table of Contents

1 Introduction and Research Context
   1.1 Introduction ................................................................. 6
   1.2 Current Research .............................................................. 9

2 Patchwork Encoding and Decoding
   2.1 Information Hiding Ideology ............................................... 11
   2.2 Mathematical Basis ............................................................ 12
   2.3 Encoding Mechanism .......................................................... 16
   2.4 Hypothesis Testing ............................................................ 19
   2.5 Decoding Mechanism ......................................................... 21

3 Encoding Algorithm Parameters
   3.1 Brightness ................................................................. 25
   3.2 Patch Encoding ............................................................. 26
   3.3 Patch Size ................................................................. 28
   3.4 Patch Shape ............................................................... 28
   3.5 Number of Patches .......................................................... 28
   3.6 Patch Intensity and Contour ............................................. 29
   3.7 Visibility Mask ............................................................. 30

4 Orientation Detection and Correction
   4.1 Motivation ................................................................. 33
   4.2 Implementation ............................................................ 35

5 Improved Decoding by Gradient Search
   5.1 Motivation ................................................................. 39
   5.2 Gradient Descent ............................................................ 39
   5.3 Advantages and Implementation of Random Search/Gradient Descent.. 42

6 Multiple Bit Encoding/Decoding
   6.1 Implementation ............................................................ 47
   6.2 Error Correction ........................................................... 47

7 Conclusion
   7.1 Conclusion ................................................................. 50

References ............................................................................. 52

Appendix A, encode.c ......................................................... 53
Appendix B, decode.c .......................................................... 63
Appendix C, odetect.c .......................................................... 66
Appendix D, grad.c ............................................................... 82
Appendix E, mbencode.c ....................................................... 101
Appendix F, mbdecode.c ....................................................... 114
Table of Figures

Figure 2.1: Bender Buck in 8-bit Grayscale.........................................................13
Figure 2.2: Expected $S_{10,000}$ Distribution for a Bender Buck, $\sigma=10,400$........16
Figure 2.3a: Encoded Bender Buck.................................................................18
Figure 2.3b: Original Unencoded Bender Buck.................................................19
Figure 2.4: Hypothesis Test for an Image when $S_n >$ Threshold .........................20
Figure 2.5: Hypothesis Test for an Image when $S_n <$ Threshold .........................20
Figure 2.6a: Encoded Bender Buck with Points of Encoding Visualized.................25
Figure 2.6b: The Decode Window for Figure 2.6a with Decoding Points
Visualized........................................................................................................22
Figure 2.6c: Decode Window Misaligned Due to Rotation.....................................22
Figure 2.7: A Bender Buck with $n=10,000$ and $\delta=10$.................................23
Figure 2.8: $S_n$ Values Obtained From Decoding Figure 2.7...............................23

Figure 3.1: Bender Buck Encoded with $n=10,000$ and $\delta=80$.........................26
Figure 3.2: Visualization of Patches.....................................................................27
Figure 3.3: Bender Buck Encoded with $n=15,000$, $\delta=80$ and radius of 10.......29
Figure 3.4: Side View of Pseudo-Randomly Filled Circle.......................................29
Figure 3.5: Side View of a Negative Random Cone.............................................30
Figure 3.6: Block Diagram of the Double Sinc Blur Technique..............................31

Figure 4.1: One-bit Quantized Image Using a Brightness Threshold of 167.............36

Figure 5.1: Perfect Decode Window Orientation...................................................40
Figure 5.2: Imperfect Decode Window Orientation................................................40
Figure 5.3: Misleading Points Used in Calculating the Gradient............................41
Figure 5.4: Scanned Image of the Cut, Encoded Bender Buck.................................44
Figure 5.5: Aligned Bender Buck Using Random Search and Gradient Descent........45
Figure 5.6: Entire Bender Buck Before Random Search and Gradient Descent...45

Figure 6.1: "$0,1,1,0$ $S_n$ Values Under Error Correction Scheme.......................48
Chapter 1:
Introduction and Historical Background

1.1 Introduction
The emergence of the Internet as a popular means of communication has introduced new dilemmas regarding intellectual property. Certainly, the wide proliferation of URLs in print and television advertisements provides evidence towards the general popularity of this medium. It is common to find corporations and organizations offering web sites that complement other forms of advertising and publicity, providing consumers with vast amounts of company and product information, services and support. Publishing electronically disseminates information quickly and extensively, but given the nature of the World Wide Web, copyright protection is infringed upon with greater ease and/or frequency than with printed documents. This phenomenon can be attributed to the fact that text and images that exist in electronic form can easily be duplicated and republished without degradation, attribution and often without detection. It can be difficult to find such offenses when they occur because of the vast size of the Internet. And when such cases are found, it can be hard to prove ownership. A second problem also exists: with the advent of high-quality desktop printers, such images can be used offline as well, adding to the extent of the problem.

These problems exist because current popular formats for images on the Internet do not allow for any type of proprietary protection. Common Internet image formats, such as GIF\(^1\), do not contain provisions for copyright. TIFF\(^2\) and JPEG\(^3\), are compression techniques that have “out of band” provisions that place copyright information in

---

\(^{1}\) Graphic Interchange Format  
\(^{2}\) Tagged Image File Format  
\(^{3}\) Joint Photographic Experts Group
headers. This mechanism for copyright protection is not robust in that such copyright notifications can be manually removed with ease. Also, printing such images results in the loss of these headers. When a JPEG image is printed out and scanned back into digital form, header lines that were present in the original file will not be present in the scanned file. Any copyright information contained in the original image is therefore lost. Without more robust copyright mechanisms, everyday users have come to regard images that appear on the Internet as being in the public domain and therefore utilize them as such without malicious intent. Because of this belief, it is common to find the same picture or icon on multiple sites. But who was the photographer for this particular picture or which artist designed that icon? Do these images appear with permission or attribution? More often than not, these images do not appear with any reference or compensation to their originators. For both the corporation and the artist, this trend is not an appealing aspect of publishing on the Web.

This dilemma was one of the motivating factors for the research described in this thesis. One possible solution to this problem can be achieved with digital watermarking. The Patch Track algorithm, a multiple-bit extension of the Patchwork\(^4\) technique, discussed below, allows users to watermark images with selected data (such as an identification number) in a manner that is imperceivable to human eyes while being robust to noise, rotation and cropping. These characteristics make the Patch Track algorithm a good watermarking choice for the purpose of ownership identification in images. A web crawler could be used to detect watermarked images on the Internet, much like many Internet search engines, by scanning images for a particular mark. While images published on the web could be watermarked with an identification number using this algorithm, Patch Track’s resistance to noise also allows the watermark to exist in the image after being printed (a noisy process) from a desktop printer as well. A watermark in such a paper image could then be scanned (also a noisy process) with a desktop scanner into a computer and detected. In this way Patch Track offers a defense for both of the above Internet ownership problems. Additionally, Patch Track can be employed in an internal data warehouse scheme. Each image in an image warehouse could be

\(^4\) Bender, W. and Gruhl, D., *Information Hiding to Foil the Casual Counterfeiter*, (1998), pg. 3
watermarked with a pointer to important image information, such as the artist/photographer, subject, date, location of the high-resolution original, etc., in the company filesystem. This information would then be accessible as the image is copied, manipulated and used. In order to employ the Patchwork technique in a multiple-bit scheme for these uses, however, it is necessary to solve the alignment problem inherent in digital watermarking algorithms. Without a solution to this problem, multiple-bit encoding would be much less reliable.

The operation of applying a digital watermark to an image is referred to as “information hiding” or “encoding.” In order for the Patchwork and Patch Track decoders to operate, as will be discussed in Chapter 2, it is necessary to align the image correctly so that the decoder can visit the same points in the image that were altered by the encoder. A reliable mechanism for the alignment of the image is therefore necessary. This thesis will discuss one possible implementation for solving this alignment problem: the use of a random search and a gradient descent in conjunction with a coarse orientation detection system to obtain the correct alignment. Integrating this alignment mechanism into the Patchwork decoder then allows for the much more useful multiple-bit encoding scheme.

While the Patch Track algorithm could provide an element of protection for images published on the Web, this encoding algorithm is generalized and would allow users to embed data within an image for various purposes. These could include photo annotation, data warehousing and any uses that would take advantage of having data embedded within an image. The amount of data that could be encoded in a particular image is dependent on various factors including image size, image content, image resolution and desired data recoverability and will be discussed in Chapter 3.

The mechanism by which Patchwork and Patch Track algorithms watermark an image, discussed in Chapter 2, exists as alterations to the brightness values of the image itself. These changes are “masked” by various visual and textual elements, discussed in Chapter 3, that make them less visible to humans. Minimizing visibility is an important aspect of information hiding. If the encoded data interfered with the appearance of the image itself
in a way that was obvious, this would defeat the purpose of information hiding. Therefore, many efforts were made to minimize the visibility of the watermark. This and other desired characteristics of information hiding techniques are discussed briefly in Chapter 2.

This paper will first present and discuss some of the methods that are being researched or currently exist for digital watermarking. It will continue by introducing the fundamental basis for the encoding/decoding mechanism and discuss how the algorithm was implemented. Additional features of the Patchwork/Patch Track algorithm which enable multiple-bit encoding, including orientation detection and alignment correction by random search and gradient descent will then be discussed and evaluated.

1.2 Current Research
Digital watermarking is a fairly active field of research. The MIT Media Laboratory has been addressing digital watermarking, as well as information hiding in audio and ASCII text, under the direction of Walter Bender and Daniel Gruhl since 1994. Other Media Laboratory researchers such as Ted Adelson and Andrew Lippman developed information hiding methods in the 1980s during their efforts to develop Enhanced Definition Television systems (EDTV). A representative, but by no means exhaustive, list of current research groups involved in the field of information hiding include: IBM Research Tokyo Research Laboratory in Japan\(^5\) where research has been focused on watermarking technologies that offer various solutions for rights management of digital content. IBM has explored data hiding techniques in audio, video as well as still images. Explorations in these areas are also being conducted at the NEC Research Institute\(^6\) and at the Computer Laboratory at the University of Cambridge in England\(^7\).

The work being conducted at these laboratories approach information hiding in digital images from various technical angles. Similar to Patchwork, IBM Research uses accumulated pseudo-random noise to form a statistical inference about embedded data

\(^5\) http://www.trl.ibm.co.jp/projects/s7730/Hiding/index_e.htm
\(^6\) http://www.neci.nj.nec.com/tr/neci-abSTRACT-95-10.html
\(^7\) http://www.cl.cam.ac.uk/~fapp2/steganography/
with the goal of creating a tamper resistant watermark. Secure spread spectrum techniques for information hiding, numerically identical to the accumulated statistical method, are being explored at the NEC Research Institute in order to create a robust watermarking system for still images. The University of Cambridge has been involved in extensive investigations into digital information hiding including research on the robustness of existing watermarking techniques using benchmarking tools such as StirMark.\textsuperscript{8,9}

Across these different research groups, the emphasis of watermarking in digital images is still to achieve a robust algorithm that is resistant against various image modifications such as compression, noise and signal processing while maintaining data transparency. While Patch Track also focuses on robustness, a highlighting feature of this encoding method is that in addition to high resistance, its watermarks are low in visibility as well. By developing “masking” techniques (Chapter 3), the watermarks encoded by Patchwork is less visible than it would be otherwise, thus allowing for “stronger” encodings. In addition, the use of patches (Section 3.2) allows for a solution to the alignment correction problem that does not involve an exhaustive search. To the best knowledge of the author, this is what makes Patchwork unique.


Chapter 2:
Basic Encoding and Decoding

2.1 Information Hiding Ideology
Information hiding in images is a form of steganography in which the goal "is to hide messages inside other harmless messages in a way that does not allow an enemy to even detect that there is a second secret message present" [Markus Kuhn 7/3/95]. Placing information within images in this manner allows for various applications ranging from mere image annotation to data warehousing, as mentioned earlier. To be useful for these applications, information hiding, also known as watermarking, in digital images should be capable of embedding data in a carrier image with the following restrictions and features: 10

1. The host signal should be nonobjectionally degraded and the embedded data should be minimally perceptible or imperceivable (meaning an observer does not notice the presence of the data, even if they are perceptible.)
2. The embedded data should be directly encoded into the media, rather than into a file header. This insures that the data will remain within the image across file formats and media.
3. The embedded data should be immune to modifications ranging from intentional and intelligent attempts at removal to anticipated manipulations, e.g., channel noise, filtering, resampling, cropping, encoding, lossy compressing, printing and scanning, digital-to-analog (D/A) conversion, and analog-to-digital (A/D) conversion, etc. In addition, embedded data should be robust to "unwatermarking" systems such as StirMark.

4. Error correction coding\textsuperscript{11} should be used to ensure data integrity. It is inevitable that there will be some degradation to the embedded data when the host signal is modified.

5. The embedded data should be self-clocking or arbitrarily re-entrant. This ensures that the embedded data can be recovered when only fragments of the host signal are available, e.g., if a sufficiently large piece of an image is extracted from a larger picture, data embedded in the original picture can be recovered. This feature also facilitates automatic decoding of the hidden data, since there is no need to refer to the original host signal.

With these restrictions and characteristics in mind, the Patchwork algorithm was developed. In its most basic form, this algorithm is capable of encoding and decoding a single specific mark, or bit, in an image. This mark can be interpreted as an independent encoding that indicates whether the image in question contains a watermark (e.g. the watermark of the MIT Media Laboratory) or exist as a part of a larger encoding scheme, such as a reference number, as will be discussed in Chapter 6. In either case, this algorithm relies on a matched filter method as described below.

\section*{2.2 Mathematical Basis}

For purposes of illustration, it is assumed that the image under analysis is an eight-bit grayscale image; the brightness of the image is quantized to 256 levels. A grayscale image is defined only by the brightness, or intensity, of each pixel, such as the "Bender Buck"\textsuperscript{12} shown in Figure 2.1. Each pixel of such an image exists as a number in the image file, where the number, in this discussion, is an integer that ranges from 0 to 255. The number defines the intensity of that pixel, higher numbers being brighter. The amount of quantization therefore defines how many different brightness levels are available for the image. A simplifying assumption that will be made for the following analysis is that all brightness levels are equally likely and that all samples are

\footnotesize

\textsuperscript{12} Fabricated Media Laboratory Currency created by Fernando Paiz
independent of each other. These assumptions are not limiting, as was shown by Bender and Gruhl.  

Figure 2.1: Bender Buck in 8-bit Grayscale

The encoding algorithm begins by choosing two pixels randomly, with replacement, from the image, X and Y. These points have brightness levels of $B_x$ and $B_y$ (ranging from 0 to 255). Taking the difference between this pair of intensities gives:

$$S = B_x - B_y$$  \hspace{1cm} (1)

where $S$ can range from −255 to 255 for 8-bit quantization. Since $B_x$ and $B_y$ were assumed to be independent:

$$E(S) = E(B_x) - E(B_y) = 0$$  \hspace{1cm} (2)

where $E$ represents the expected value. Since each of the 256 brightness levels were assumed to be equally likely, $E(B_x)$ and $E(B_y)$ are both equal to 127.5. Therefore $E(S)$ is zero, i.e. the average value of $S$ after repeating this procedure many times is expected to be zero. The value for a specific trial of $S$, however, is dependent on its variance, $\sigma^2_S$.

---

How tightly the distributions of the individual values of $S$ will cluster around $E(S)$ depends on $\sigma^2_S$. Because $B_X$ and $B_Y$ were assumed to be independent, the following can be said:\(^{14}\)

$$\sigma^2_S = \sigma^2_{B_X} + \sigma^2_{B_Y}$$  \hspace{1cm} (3)

Assuming a uniform distribution of brightness, the variance of a particular pixel can be calculated by:

$$\sigma^2_{B_x} = \frac{(255-0)^2}{12} = 5418.75$$  \hspace{1cm} (4)

Since both pixel $X$ and $Y$ are chosen randomly from the same data set, selected with replacement:

$$\sigma^2_{B_X} = \sigma^2_{B_Y}$$  \hspace{1cm} (5)

$$\sigma^2_S = \sigma^2_{B_X} + \sigma^2_{B_Y} = 5418.75 + 5418.75 = 10837.5$$  \hspace{1cm} (6)

$$\sigma_S = \sqrt{10837.5} = 104$$  \hspace{1cm} (7)

where $\sigma_S$ is the standard deviation of $S$. Nothing very useful can be said of a particular value of $S$.\(^{15}\) However, if a statistic $S_n$ is constructed from numerous iterations of $S$:

$$S_n = \sum_{i=1}^{n} S_i = \sum_{i=1}^{n} (B_{X_i} - B_{Y_i})$$  \hspace{1cm} (8)

---


where \( n \) is the number of pairs of pixels taken. On average, as \( n \) approaches infinity, it can be expected that \( \frac{S_n}{n} \) will approach zero. From independence, it is seen that the expected value of \( S_n \):

\[
E(S_n) = n \times E(S) = n \times 0 = 0
\]

This result can be rationalized intuitively. Since pairs of points are chosen with replacement, and at random, one would expect that pixel \( X \) will be brighter than pixel \( Y \) about as many times that the reverse is true.

The variance of \( S_n \) can provide revealing information regarding the trend in \( S \) over many trials. This is crucial to the development of the Patchwork algorithm.

From independence and zero mean it can be seen that:

\[
\sigma_{S_n}^2 = n \times \sigma_S^2 \quad \text{(10)}
\]

\[
\sigma_{S_n} = \sqrt{n} \times \sigma_S = \sqrt{n} \times 104 \quad \text{(11)}
\]

With this formula, it is now possible to calculate the variance of \( S_n \) for a given number of iterations. It was arrived at through experimentation that an appropriate value of \( n \) for encoding a 1198x508 pixel image (e.g. one that would result from a 135dpi grayscale scan of a Bender Buck, as seen in Figure 2.1) is 10,000:

\[
\sigma_{S_{10,000}} = \sqrt{10,000 \times 104} = 10,400 \quad \text{(12)}
\]

By the Central Limit Theorem, \( S_{10,000} \) can be approximated as a Gaussian with a mean of zero, and a variance \( \sigma_{S_{10,000}} \). Figure 2.2 shows the expected \( S_{10,000} \) distribution for a Bender Buck. The Patchwork encoding method acts to shift this distribution through
modifications discussed in the next section. This shift in $S_n$ when an image is encoded is essentially what the Patchwork decoder (Section 2.5) uses to detect a watermark.

![Graph showing Expected $S_{10,000}$ Distribution for a Bender Buck, $\sigma_{S_{10,000}} = 10,400$](image)

**2.3 Encoding Mechanism**

In Patchwork, the first task for encoding an image is the generation of a set of points to alter in the image. These alterations are the physical manifestation of the digital watermark and cause the encoded distribution of $S_n$ to be shifted away from unencoded distribution of $S_n$. These alterations take the form of changes in brightness. In order for the decoder to recognize the watermark, it is crucial that it visit the same set of points in the image. If this does not occur, or if only some of these points are revisited, non-encoded points will be analyzed and the experimental value of $S_n$ obtained from an encoded distribution will not be shifted as far as intended. Depending on how far this value of $S_n$ ends up being shifted relative to the unencoded distribution in Figure 2.2, this may result in loss of data or at least a decrease in decode certainty. This is discussed further in Section 2.4. The ability to revisit the same points is therefore very important.

To generate a reproducible set of points, a pseudo-random number generator was used. Such a number generator requires a key, or a seed, in order to return a number. Successive use of the generator with the same seed results in a stream of pseudo-random numbers between 0 and 1. When two such pseudo-random numbers are taken and scaled

---

to the dimensions of the image, the position of a pixel can be obtained. In this way, the pixel positions that are needed in the algorithm can be generated pseudo-randomly. The important factor, however, is that when the pseudo-random number generator is supplied with the same seed later, e.g. during decode, the same stream of numbers is regenerated. A unique set of numbers can therefore be generated with a particular key and it is only necessary to know which key was used during encoding. Various encryption methods can be used to make this key secure in practical use if the need arises, such as RSA or PGP. When the same seed is used in conjunction with proper image alignment (discussed in Chapter 4 and 5), the watermark can be detected by the decoder.

The encoding then starts by obtaining a point \((X, Y)\), returned by scaling a pair of pseudo-random numbers by the dimensions of the image. In order to modify the image so that the encoded distribution of \(S_n\) can be separated from the unencoded distribution, the brightness at point \(X\) is raised by \(\delta\) while the brightness at point \(Y\) is lowered by the same amount (it is not necessary that the second point be lowered by the same amount, only that it be lowered). For simplicity, \(\delta\) is used for both \(X\) and \(Y\).

\[
S_n' = \sum_{i=1}^{n} ((B_{ai} + \delta) - (B_{bi} - \delta))
\]

\[
S_n' = 2\delta n + \sum_{i=1}^{n} (B_{ai} - B_{bi})
\]

where \(S_n'\) is the experimental value of \(S_n\) for an encoded image. In this way, each additional pair of pixels add \(2\delta\) to the accumulating sum. Therefore, after \(n\) repetitions:

\[
E(S_n') = 2\delta n
\]

where the standard deviation is the same as in the unaltered distribution. From this analysis, it can be seen that as \(n\) or \(\delta\) increases, the distribution of \(S_n\) shifts in the positive direction. As discussed in Section 2.4, shifting the \(S_n'\) distribution far enough in this direction makes any particular point that falls under one distribution highly unlikely to be
near the mean of the other. Therefore, an encoding can be successfully distinguished from noise by looking at the value of $S_n$ for that specific key. If it is sufficiently large in magnitude, where large depends on the certainty level that is desired, it can be said with confidence that encoding exists under that key. The level of certainty depends on the distance of $S_n$ from zero with respect to $\sigma_{S_n}$. A higher level of certainty requires that $E(S_n)$ be further (more standard deviations) from zero. An encoded image can then be distinguished from a non-encoded image using a Hypothesis Test (Section 2.4).17

Once the image has been encoded, it is saved and written to file. The resulting grayscale encoded image is then merged with its color components to produce the final encoded image, as seen in Figure 2.3a. The original color image is seen in Figure 2.3b. As expected, these images should appear identical. Extending the above algorithm for multiple-bit encoding is presented in Chapter 6. The C program, encode.c, which was written to perform the encoding procedure appears in Appendix A.

---

2.4 Hypothesis Testing

Hypothesis testing involves the use of a threshold to make a decision as to whether a situation is true. In this case, the hypothesis being tested is whether the image in question is encoded. Figure 2.4 plots both the encoded and non-encoded distributions on the same axis. As discussed earlier, Patchwork serves to shift $S_n$ to the right, thus separating the encoded distribution from that of the unencoded (where the shift is dependent on $\delta$ and $n$). A threshold is assigned so that an experimental value of $S_n$ greater than the threshold implies that the image is encoded. One possible threshold is shown in Figure 2.4. This threshold lies three standard deviations from 0 (the mean $S_n$ of the non-encoded image). Since $S_n$ is approximated to be Gaussian, there is a 0.13% chance that an experimental value of $S_n$ from a non-encoded image would occur at this value.\(^{18}\) As the threshold moves further to the right this percentage decreases. At the same time, the likelihood of that experimental value being obtained from an encoded image increases as the threshold moves closer the mean of $S_n'$. Therefore, an image that yields an experimental value of $S_n = z$ that falls in a region that is sufficiently far from 0 (as seen in Figure 2.4) can be declared with certainty (depending on how far $z$ is from 0) to not be from an unencoded image. The further $z$ is from 0, the less likely that such a value came from an unencoded image.

\(^{18}\) ibid. pg. 211
image. As mentioned before, if \( z \) was found to be \( 3 \sigma_{S_{n,\infty}} \), there is only a 0.13% chance that the image was non-encoded.

![Figure 2.4: Hypothesis Test for an Image when \( S_n > \) Threshold](image)

![Figure 2.5: Hypothesis Test for an Image when \( S_n < \) Threshold](image)

Figure 2.5 shows the case where the experimental value of \( S_n = z \) is less than the threshold because the same points were not visited. In this case the data would could not be recovered with the same level of certainty as before because it is not as far from the unencoded distribution. As established earlier, it is important for the decoder to revisit
the same points. This is the basic premise upon which the Patchwork encoding/decoding algorithm pair was created.

2.5 Decoding Mechanism
In order to decode the image, the correct seed is needed to generate the correct pixel pairs for analysis. This set of points and their position with respect to a set of axes will be called the decode window. This decode window is visualized in Figure 2.6 where two pixel pairs were decoded. Figure 2.6a shows the points where the image was encoded. Figure 2.6b shows the decode window. In order to decode the image, the decode window must coincide with these points precisely. Figure 2.6c shows the case where this window does not coincide. In this case, the image cannot be successfully decoded since the encoded points are not “seen” due to a rotation in the decode window. The points in the decode window are generated identically to the manner in which it was generated in the encoder. Using the same key, the pseudo-random number generator returns the same set of points for the decoder to visit.

Figure 2.6a: Encoded Bender Buck with Points of Encoding Visualized

The decode window revisits the points and calculates $S_n$ for that key and window. As mentioned above, using a hypothesis test on the value of $S_n$, a degree of certainty can be associated with whether the image was encoded or not. For example, if $S_n$ for the case where $n=10,000$ was found to be 93,600, one would conclude with great certainty (nine
standard deviations) that the image in question was encoded. The C program, decode.c, which was written to perform the decoding procedure appears in Appendix B.

![Figure 2.6b: The Decode Window for Figure 2.6a with Decoding Points Visualized](image)

Figure 2.6b: The Decode Window for Figure 2.6a with Decoding Points Visualized

![Figure 2.6c: Decode Window Misaligned Due to Rotation](image)

Figure 2.6c: Decode Window Misaligned Due to Rotation

Figure 2.7 shows the result when the Bender Buck in Figure 2.1 was encoded with $n=10,000$ and $\delta=10$ by seeding the pseudo-random number generator with the number 10. Figure 2.8 shows the $S_n$ values when this encoded image was decoded with seeds 0
through 20 under perfect alignment of the decode window. Since each key generates a unique set of points, only when the decoder is seeded with the same key that was used during encode will the decoder visit the same points. Therefore, it is expected that only key 10 will detect the encoding. It is clear from Figure 2.8 that the value of $S_n$ for key 10 is much larger than the other values that form a baseline around zero. For key 10, $S_n=201,541$. This is more than eight standard deviations ($\sigma_{s_{10,000}} = 10,400$) from zero. Therefore, it can be said with a very high degree of certainty that encoding exists in this image at key 10.

*Figure 2.7: A Bender Buck Encoded with $n=10,000$ and $\delta=10$*

*Figure 2.8: $S_n$ Values Obtained From Decoding Figure 2.7*
This is the basic mathematical model of the Patchwork encoding/decoding algorithm. It allows for the fulfillment of the six restrictions and features presented in Section 2.1. As has already been discussed, the Patchwork algorithm encodes watermarks into the media itself, rather than into a header. In this way the data can remain within the image across file formats and media. Also, the visibility of the encoding, as demonstrated by Figure 2.3, is very low. Using other modifications, such as a visibility mask (discussed in Section 3.7), will further decrease the appearance of the data. Using the gradient descent techniques discussed in Chapter 5 will extend Patchwork so that the decoder can recover alignment and thus recover data when only a fragment of the host is available. In addition, error correction coding will be incorporated into the Patch Track encoder and decoder to ensure data integrity in the presence of noise. Patchwork can be made more robust to several anticipated manipulations such as noise, cropping and lossy compression with the additions discussed in the next section.
Chapter 3:

Encoding Algorithm Parameters

3.1 Brightness

The choice of brightness as the method of encoding, rather than some other variable in
the image, was due in large part to the fact that the human visual system (HVS) has high
sensitivity to random changes in luminance, or brightness.\(^{19}\) By utilizing brightness, this
algorithm hides the data in regions of the picture that compression and reproduction
(changing formats, printing, etc.) techniques put the most effort into preserving. This
increases the likelihood that the mark is retained through format and media changes.

While the brightness level of certain pixels is actually being changed in the image, these
alterations are imperceivable to the human eye. The exact change in brightness (\(\delta\)) that is
applied to the image is dependent on the image itself and the desired encoding "strength".
Since the degree of certainty that is associated with an experimental value of \(S_n\) depends
on its value, a stronger encoding, or higher degree of certainty, can be achieved by
increasing \(\delta\), thus increasing \(S'_n\) (Equation 13). Increasing \(\delta\) will generally also have the
effect of making the encoding more obvious. In this way, a balance must be struck
between increasing certainty and decreasing perceivability. This tradeoff between
visibility of encoding and encoding strength arises many times in the use of this
algorithm.

To demonstrate this tradeoff, the Bender Buck is encoded with \(\delta=80\) and \(n=10,000\) in
Figure 3.1. The corresponding \(S'_{10,000}\) for this image was found to be 1,303,090.
Comparing Figure 3.1 to Figure 2.7 (\(\delta=10, n=10,000, S'_{10,000}=201,541\)) shows that while
the image with $\delta=80$ is more visibly encoded, its $S_{10,000}$ value is also much higher, meaning the data can be decoded with more confidence. In fact the same effect can be achieved by increasing $n$ instead, as will be seen later.

![Image](image_url)

*Figure 3.1: Bender Buck Encoded with $n=10,000$ and $\delta=80$*

### 3.2 Patch Encoding

In the development of the Patchwork encoding algorithm in Chapter 2, the watermark existed as points in the image. An improvement to this method uses patches, small regions in the image rather than individual points. This modification offers two specific advantages. One is the increased robustness of this encoding. Using patches decreases the spatial frequency of the noise introduced by this algorithm, thus making the encoding more resistance to lossy compression (such as JPEG compression) and finite input response filters. Patches make the encoding appear less like noise, as single points would, and more like important parts of the image in the eyes of the JPEG compressor. This helps to keep the encoding from being removed from the image during conversion to JPEG. In addition, patches help facilitate the decoding of the image under practical circumstances. By placing the encoding in a region larger than an individual point (or "chip"), while keeping the decoding mechanism the same, a more a forgiving alignment is required when attempting to detect the watermark in the image.

---

To implement the patch scheme, each point is replaced by a patch that is centered on that point, as visualized in Figure 3.2. Each circle represents a patch and each gray square represents the centers of each patch. These centers are also the points that would have been encoded had the single point scheme been used. The decoding points of the decode window are represented by boxes with crosses. Under perfect alignment of the decode window, each decoding point coincides with each patch center (gray square), as seen on the left. In this way, under perfect alignment, the image will decode in the same manner as under the single point scheme. On the right, a situation where the decode window is shifted to the right is shown. Despite being shifted, the decode points are still within the patch and therefore will still decode as if the decode window was perfectly aligned (the same is true with small rotations). With the single point scheme, small deviations such as this from perfect alignment would cause the decoding to fail. Using larger regions of encoding therefore adds more resistance to rotation during the decoding process.

Introducing patches to the encoding algorithm adds additional degrees of freedom with regard to controlling encoding visibility that will be discussed below. The Bender Buck seen in Figure 2.1 was encoded with \( n=10,000 \), \( \delta=10 \) and a patch radius of 10 pixels. This image decoded with \( S_n=201,541 \) under perfect decoding orientation. A 0.01 radian (0.57° rotation) caused \( S_n \) to decrease to 128,873, this is still within the range of high certainty (approximately 12 standard deviations).

![Figure 3.2: Visualization of Patches](image-url)
3.3 Patch Size
The size of the patches is an important parameter with distinct tradeoffs. Increasing the size of the patch can allow for easier alignment of the image in the decode stage. From Figure 3.2, it can be seen that a larger patch will resist more shift and rotation in the decode stage. Increasing the patch size will also cause patches to overlap sooner with respect to increasing patch number. In effect, patches in the encoding will begin to interfere with other patches sooner. It will be seen later in Chapter 4 and 5 that resistance against rotation and shift becomes less important with the addition of a gradient descent to the decoding system. For the example visited in the previous section, doubling the patch size increases the 0.01 radian rotation \( S' \) to 154,596.

3.4 Patch Shape
Another parameter to consider is the shape of the patches. This parameter in particular has an important impact on the visibility of the encoding. The HVS is particularly sensitive to regular or lattice-like placement (patterns) of objects with continuous edges.\(^{20}\) These patterned edges contain high frequencies, a feature that was described earlier as being particularly apparent to the HVS. In order to take advantage of this characteristic, the choice of circular patches was made. In addition, since patches are placed randomly in the image by this algorithm, lattice-like formations are avoided.

3.5 Number of Patches
Similar to the choice of \( \delta \), the number patches, \( n \), also impacts the encoding strength. From Equation 13, it can be seen that increasing \( n \) will also increase \( S' \). Again, having a higher value of \( S' \), means a higher certainty of encoding. However, as was mentioned with increased patch size, increasing the number of patches also tends to decrease the space for other patches. Too many encodings also has the effect of degrading the image as seen in Figure 3.3 where the Bender Buck is encoded with \( n=15,000 \), \( \delta=20 \) and radius of 10. A recommended \( n \) for a typical 1198x508 pixel image was found to be 10,000.

\(^{20}\) Bender, W. and Gruhl, D., Information Hiding to Foil the Casual Counterfeiter, (1998), pg. 7
3.6 Patch Intensity and Contour

As mentioned above, the HVS appears to be very sensitive to low frequency patterns\textsuperscript{21}. To take advantage of this characteristic, a good patch choice would be blob shaped, but one that did not have a constant intensity throughout its area. Instead, if the area was filled with pseudo-random intensity values, the patches would appear less patterned, and thus, less visible. Such a randomly filled circle is visualized in Figure 3.4. While this modification would make the patches less visible, it would have a deleterious affect on the encoding. A given patch may or may not have the desired $\delta$ change in brightness anymore at the point of decode. This would serve to undermine the basis of the encoding algorithm.

\begin{center}
\includegraphics[width=0.5\textwidth]{pseudo_random_filled_circle.png}
\end{center}

\textit{Figure 3.4: Side View of a Pseudo-Randomly Filled Circle}

A compromise can be achieved by using a "contour" that shapes the random intensities in the patch. Figure 3.5 visualizes a negative patch that has contoured random intensities. Here it can be seen that while a large degree of randomness is retained in the contoured

\textsuperscript{21} ibid.
image, encoding coherency is also maintained by applying an envelope to the random intensities. A perfectly aligned image will decode at the center of the patch, the apex or nadir of the contour, matching the result associated with a constant patch. This contour thus maintains the benefits of patch usage, such as rotation and shift resistance, while reducing visibility. Due to the shape of the contour, we refer to this type of patch as a “random cone.” Because humans tend to notice sharp edges and continuous patterns more readily than gradients and randomness, a patch contour that takes advantage of this fact will allow for “stronger” encoding while remaining well hidden. Since the random cone does not lend itself towards sharp edges, due to its contouring, nor patterns, because it is circular and placed randomly, random cones are an advantageous choice of patch type.

![Figure 3.5: Side View of a Negative Random Cone](image)

3.7 Visibility Mask
An additional refinement to the encoding algorithm involves the use of something that will be referred to as a visibility mask. By identifying regions in an image that are most suitable for encoding (regions in which the patches will be least noticeable) it is possible to conceal data in an image. Such regions are those that exhibit a large amount of high frequency content. Applying changes to regions of an image that contain a great deal of variation in brightness is less detectable to the HVS than applying the same changes to a region that varies less\(^\text{23}\). Adding a patch to a region of an image that is high frequency

\(^{22}\) ibid.
\(^{23}\) ibid.
(contains drastic changes in intensity) is much less visible than adding the same patch to a region of an image dominated by low frequencies. The visibility mask seeks to identify the high frequency regions in the target image and concentrate the encoding there.

The visibility mask was implemented using a double sinc blur technique diagramed in Figure 3.6. The image resulting from the first blur, which takes the average value of the points in a square window and replaces the center of that square with the average, was subtracted from the original image, resulting in a difference image. Since the blurring process attenuates high frequencies from the image, this difference image then contains a rough estimate of all the edges in the image (absolute value of high frequency areas). A second blurring, increases the width of these edges. In this way, high frequency regions in the image were identified for visibility masking of the encoding. The appropriate size of the blurring kernels depends on the size of the image. It was found that for 1198x508 pixel image, a good set of blur kernel sizes was 5x5 for the first blur and 7x7 for the second blur.

\[ \text{Target Image} \rightarrow \text{Sinc Blur} \rightarrow \text{Sinc Blur} \]

*Figure 3.6: Block Diagram of the Double Sinc Blur Technique*

This double blur process was added to the encoding algorithm after the patch locations were identified. Before applying these patches, however, the visibility mask was implemented by scaling the brightness change \((\delta)\) by a factor that measures the desirability of that location in terms of its frequency content. For example, if this position fell on a sharp edge, as specified by the visibility mask, a scale factor near 1 is used. If this position fell in a constant region, a scale factor near 0 is used. Intermediate cases were then assigned a scale factor relative to how much high frequency content was
present in that area. In this way, very blatant encodings can be avoided while decreasing the perceivability of the data.

These are the many elements of the basic Patchwork algorithm. Each aids in adding the desired robustness and/or decreased visibility as discussed in Chapter 2. In practice these elements of the encoding algorithm perform well. In order to encode with multiple bits, however, a further modification to the decoding process is necessary. This is the inclusion of an orientation detector, allowing for more resistance to rotation.
Chapter 4:
Orientation Detection and Correction

4.1 Motivation
The robust nature of the Patchwork algorithm allows watermarked images to be printed while retaining the embedded information. In order to retrieve this data, it is necessary to scan the paper image into digital format. A practical issue that must be addressed in this context is that of image alignment during the decoding process. The orientation of an image on a flatbed scanner is inherently imprecise. When using a desktop scanner, the mere act of closing the cover can drastically change the position of the image from where it was placed. Therefore, it is neither sufficient nor practical to require the user to accurately position the picture on the scanner in the correct orientation. One possible solution to this alignment problem is the use of an orientation detection and correction mechanism. By integrating such a mechanism into the decoder, an image can be placed on the scanner in an arbitrary orientation. The decoder would then determine the orientation and extract the data, thus maintaining the ability to decode the image.

The addition of orientation detection and correction also allows for a hierarchical approach to the decoding of a target image embedded by Patch Track (multiple bits). One appropriate use of digital watermarking, as mentioned earlier, is the encoding of images with reference numbers, perhaps indicating ownership by a particular company. In order to place such a watermark, as will be discussed further in Chapter 6, multiple bits will be required. Until this point, only the single bit case has been discussed. Multiple bits can be implemented as a sequence of positive and negative $S_n$ values that represent ones and zeros using Patch Track (discussed in Section 6.1). Since the encoding
algorithm embeds watermarks for different keys in a nearly orthogonal manner\textsuperscript{24} (due to the pseudo-random number generator), many bits can usually be encoded in an image before either degrading the image or interfering the encoding of other bits. For a Bender Buck, approximately 256 bits can be encoded before significant image degradation occurs.

In Patch Track, the first bit in the sequence of bits that constitute the multiple-bit encoding can serve as an encoding identifier, one that indicates if the image has been encoded at all. Thus, a hierarchy is created in the decoding process. First, it is determined whether encoding exists in the image. Then, given that a watermark is present in this image, the remainder of the watermark is read by the decoder.

This hierarchy is extremely useful in multiple bit encoding. When it is possible, it is certainly desirable to decrease $\delta$ or $n$ while maintaining the strength of encoding. Doing so would lower the visibility of the encoding. However, decreasing either $\delta$ or $n$ will have a direct effect on $S_n$. The Patch Track hierarchy, however, provides an alternate solution. A lighter encoding of the majority of the information is possible if the strong encoding of the first bit is used to signify the presence of data. In this way, the first bit of the stream of data bits is used to identify whether data is present in the image. If all of the information bits are encoded lightly, e.g. the 75% certainty level, there will be one failure in every four bits decoded. For some applications, this is not sufficient. For example, a much higher degree of certainty, perhaps 99.999\% may needed when tracing images on the Internet.\textsuperscript{25} Otherwise, because of the magnitude of images on the Internet, the process of tracking the encoded image will be overwhelmed by false alarms. Encoding all of the data at the 99.999\% certainty level may cause the data to become obvious in the image. If only the first bit is encoded at this strength, it can be known with 99.999\% certainty that there is encoding in this image. Far less encoding is then required for the remaining bits of the watermark since it only needs to be determined whether these bits are positive or negative. Error correction coding can then be used to increase

\textsuperscript{24} Ibid. pg. 9 \\
\textsuperscript{25} Ibid. pg. 11
the fidelity of these data points, as discussed in Chapter 5. This reduction in encoding power helps retain image quality while maintaining the accuracy of data recovery. It is therefore helpful to have a mechanism that aligns the image quickly and prepares it for decoding. Once this is done, the presence of the strong bit can be determined.

4.2 Implementation
Orientation detection and correction therefore is necessary with respect to both scanning practicality and multiple-bit encoding problems. To implement such a system, it was assumed that the encoded image was rectangular (as would be commonly expected). The scanned image would then contain the target image and background from the scanner. After scanning the image to file, the location of the target image in the scanned image must be determined. One computationally efficient manner in which to accomplish this is to quantize the image to one bit.

In order to quantize the image, it is necessary to first determine a threshold for quantization. Since the background color of scanned images is dependent on the scanner that is used, this is threshold is obtained through trials. Using an HP Scanjet 4C, the threshold quantization level was found to be 167. Intensities below 167 were quantized to 0, those above were quantized to 255. This results in an image that consists of a white background (255 in the eight-bit grayscale image) and a black rectangular region (0), which represents the image. The effect of quantizing is shown in Figure 4.1. Quantizing the image facilitates orientation detection. It is now possible to locate the four corners of the black rectangular image. Corner detection begins at each of the four sides of the scanned image and moves progressively inward toward the center analyzing either rows (for the top and bottom) or columns (for the sides). As we move from the borders to the center, the corners of the target image will be the first non-white object that is met. Using the location of the four corners, the angle of rotation can be easily calculated. Once the angle is determined, the image can be rotated and translated to the correct orientation using a subset of affine transforms: rigid body transforms.
Rigid body transforms allow for two dimensional translation and rotation. To apply a horizontal shift of $m$, a vertical shift of $n$ and a rotation of $\theta$ to a point $(x,y)$ the following equation is used:
\[
\begin{bmatrix}
\cos \theta & -\sin \theta & m \\
\sin \theta & \cos \theta & n \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix} =
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix}
\]  

(16)

which can be decomposed to:

\[
\begin{bmatrix}
1 & 0 & m \\
0 & 1 & n \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix} =
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix}
\]  

(17)

where the first matrix is applies a translation and the second matrix applies a rotation to point \((x,y)\). The resulting point \((x',y')\) is the shifted, rotated version of \((x,y)\).

The actual implementation of this transform is in the inverse direction. The points that are obtained from the scanned image form the set of \((x',y')\) and the goal is to obtain the original position of these points, \((x,y)\). Therefore, the following inverse affine transform is applied to the points in the scanned image:

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}^{-1}
\begin{bmatrix}
1 & 0 & m \\
0 & 1 & n \\
0 & 0 & 1
\end{bmatrix}^{-1}
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} =
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]  

(18)

In this way it is possible to obtain the original orientation of the image. To rotate and/or shift an image, this transformation is applied to all pixels in the image. Any rotation or shift that is detected by this algorithm can therefore be corrected with this inverse transformation. Using the dimensions obtained from the corner identification the image is cropped and written to file for decoding.

This method of orientation correction is useful when one requires a fast, yet accurate determination of whether a watermark exists in an image and when the image has been encoded in the normal viewing orientation. Since it depends only on the fact that the
image is rectangular (and fit on the scanner), this is the only requirement of the algorithm. In more general cases, including the decoding of randomly oriented encoding, data recovery from partial or cropped images and decoding the rest of the information in multiple-bit encoding, a more computationally intensive approach is required, and is described in the next chapter. Since this is a very rough estimation of the correct orientation of the image, as can be seen from the noise present in the quantization (Figure 4.1), a fairly large chip is required for data to be extracted. Through experimentation, it was found that a good patch size is 15 pixels at 135dpi. The C program, odetect.c, which was written to execute the above orientation correction appears in Appendix C.

This orientation detection and correction system, as stated earlier, can be used simply to aid the alignment of scanned images or as a part of the multiple-bit decoding scheme. In the latter, this system would account for the first stage of orientation correction, the rough alignment. It is then followed by a random search in the proximity of the roughly aligned image and a final adjustment of alignment by a gradient descent.
Chapter 5:

Improved Decoding by Random Search/Gradient Descent

5.1 Motivation
It has been assumed thus far that the image in question has been encoded in the expected manner, that is, in the normal viewing orientation. Certainly an image can easily be decoded at an arbitrary orientation, thus leaving the approach described in Chapter 4 less than useful. While the image may be aligned correctly visually, it will not be in the correct orientation for decoding. In addition, cropping of the image may have occurred, in which case, the shape of the target image may no longer be the same. Suppose only part of the image was available for decoding or part of the image was obscured. It would be desirable to have a mechanism that could handle these cases. One possible implementation is the use of a gradient descent, described below. Using this search method allows for orientation correction regardless of image shape and encoding orientation but at much higher cost in terms of decoding time.

5.2 Gradient Descent
Consider the representation of a negative patch in Figure 3.5. As described in Section 3.6, this is a side view of a non-random cone (a non-random cone is used here for clarity, the situation applies for a random cone also, as will be discussed below). The objective of the decoding algorithm is to have all of the decoding points ($X_i$ and $Y_i$ from Equation 3) correspond to the apex of the cone. This is perfect decoding alignment because it results in the largest possible decode value (in the correct alignment, when seeded with the same key, the decoder will pick the same points as the encoder). Under perfect alignment, $|S_n|$ is maximized, as seen in Figure 5.1 where the gray dot indicates the decode point. If, however, the orientation is not correct, a situation such as the one seen
in Figure 5.2 might occur. Here the decoding window is shifted to the left by six pixels (the same argument holds for rotation). When the decoding window is not correctly oriented, the decoding is much less successful since the full $\delta$ is not detected in all patches. In the patch seen in Figure 5.2, the situation is particularly bad since the decoding point falls on a value that has been nearly unaltered from the background brightness level. Depending on the original orientation, other decoding points may not even land in a patch, resulting in a significantly smaller $S_n$. This can obviously hinder the ability to recover the desired data. Orientation is therefore crucial.

![Perfect Decode Window Orientation](image1)

**Figure 5.1: Perfect Decode Window Orientation**

![Imperfect Decode Window Orientation](image2)

**Figure 5.2: Imperfect Decode Window Orientation**

In order to properly orient the decoding points, it is possible to follow the gradient of the cone (in gray images) toward the nadir. This is the essence of a gradient descent. To accomplish this, the gradient of brightness at the current, imperfect, decoding point is calculated and used to determine which the decode window should move. The gradient is
determined by calculating $S_n$ for the current window orientation as well as for all three possible degrees of freedom: horizontal shift, vertical shift and rotation. From these four $S_n$ calculations, the largest slope magnitude is identified. The window is then moved in the corresponding direction (depending on the sign of the slope) and the gradient analysis is performed again. This is then done repeatedly until the apex is reached. At this point, the gradient should be zero. In Figure 5.2, the window would initially move to the right. The amount that the window should move after each iteration of the gradient descent, or schedule size, is a complex topic and depends on various factors such as image size. Through trial and error, it was found that a good fixed schedule was a step size was 2 pixels and a rotation of 0.5°.

![Figure 5.3: Misleading Points Used in Calculating the Gradient](image)

Figure 5.3: Misleading Points Used in Calculating the Gradient

The use of a gradient descent in the case of random cones is slightly more involved. Since the random cones possess random intensities the straightforward gradient analysis is not sufficient. The random intensities that were inserted to decrease data visibility unfortunately add too much noise to the system for a simple slope calculation as might be performed in the non-random case. Calculating the slope of the surrounding points (gray) in a random cone can be misleading, as seen in Figure 5.3, where the slope calculated from the gray point indicates the wrong direction of traversal. To circumvent this problem, minimum absolute deviation regression analysis was used.$^{26}$ Rather than calculating the gradient of the intensities, a best-fit lines is calculated using the $S_n$ values that would result from moving ten points in the positive direction and ten points in the
negative direction from the current decoding point, in the two possible translation
directions. Rotation is treated similarly with steps in degrees. The slopes of these best-fit
lines are then used to determine the direction of window movement. In this way, the
contribution from noise introduced by the random intensities is reduced well enough to
reveal the underlying gradient. The decode window is then moved, as in the non-random
case, in the direction indicated. This process is repeated until the nadir is reached.

5.3 Advantages and Implementation of Random Search and
Gradient Descent
The gradient descent described above is implemented with regard to multiple-bit
encoding by locating the orientation that maximizes the $S_n$ of the strong marker bit. Since
the shape of the scanned image is not utilized in the orientation detection, this method
provides a way to orient a variety of differently shaped images (e.g. a rectangular image
that was cut in half in a jagged manner). The one requirement of the gradient search,
however, is the dimensions of the original image when it was encoded. These
dimensions are used to create the appropriate decoding window. Having such
information is well within reasonable expectations as the party (or parties with the rights
to the encoded information) that embedded the data would presumably be privy to this
information.

In this way, the orientation can be corrected for an image that has been partially lost,
intentionally disturbed or is just slightly different. With respect to the encoding, losing
part of the image only means that $n$, the number of encoded points, will decrease. This,
in turn, will decrease $S_n$ but given a strong encoding, such a loss will not affect the ability
to decode the image. This is unlike the unoriented case where $S_n$ is also decreased.
Under the unoriented case, the encoding is lost in the randomness of the cones. In the
partial image loss/perfect orientation case, the encoding is maintained, due to the
remaining available points that are aligned correctly. Since the number of encoding
points recovered is lower, $S_n$ will be lower.

The alignment detection is implemented on two levels, one proximity random search stage and one refinement stage. The scanned image will contain both the target image and additional blank space from the remaining area on the scanner bed. Once this image is acquired the orientation detection and correction described in Chapter 4 is performed. This orientation correction provides a good estimate of the correct alignment of the image. The random search stage consists of the acquisition of many \( S_n \) values corresponding to a variable number of random orientations of the decode window. These orientations are restricted to small displacements from the current orientation (produced by the orientation detection and correction stage). Through experimentation it was found that roughly 50 pixels of translation and 10 degrees of rotation perform well as limits to these displacements. The number of random orientations that are tested is important to the correct orientation detection. A sufficient number must be tested in order to generate a decode window orientation that is sufficiently close to the actual encoding orientation so that the gradient descent will work. Under testing conditions, this corresponds to an orientation that decodes with \( S_n > 5\sigma_5 \) and more than 6000 random orientations for a 1198x508 image. It required roughly 15 minutes to complete this random search on a Pentium II 450MHz PC with 512MB of RAM.

Given the size of the scanned image and the size of the original, a range for the shift values is set and random values within this range are generated (angles must range from 0 to \( 2\pi \)). The image is decoded for the strong bit at each of these random orientations. A variable set of the orientations corresponding to the highest \( S_n \) values is then cached and saved for refinement in the second stage. Again, it is crucial to create a set of orientations that are approximately correct at this stage. The refinement stage usually does not incur more than a 1% change in the orientation of the image. For this reason, a large number of orientations (>6000) should be used in the random search stage to insure the proper result.

The refinement stage is precisely the gradient search algorithm that was discussed in Section 5.2. When the image is correctly oriented, it is cropped and prepared for decoding by the multiple-bit decoder described in Chapter 6. A Bender Buck that was
encoded with \( n=12,000, \delta=40 \) and a random cone radius of 20. This image was then printed out at 135dpi, cut and scanned back in at 135dpi, as shown in Figure 5.4. Using the random search/gradient descent alignment correction method (using the top 10 decode windows from 10000 random orientations), it was possible to obtain the original alignment as seen in Figure 5.5. This image was then cut and decoded with \( S_n=56,581 \). The full image, when scanned in at an arbitrary orientation (Figure 5.6), decoded with \( S_n=75,076 \). The lower decode sum for Figure 5.5 is due to the loss of part of the image. Since the encoding exists in the pixels themselves, any loss of the image is accompanied by a decrease in \( S_n \). Nevertheless, both images were successfully decoded with high degrees of confidence.

![Figure 5.4: Scanned Image of the Cut, Encoded Bender Buck](image)

The random search and gradient descent approach to the alignment problem yields good alignment and high \( S_n \) values, but there are disadvantages. Perhaps the largest drawback of this implementation is the time required to complete an alignment adjustment. Because it uses a random search, it is computationally intensive. It requires roughly 15 minutes to complete a random search for 6000 orientations and the gradient descent of the ten best orientations on a Pentium II 450MHz PC with 512MB of RAM. This time cost could hinder the use of this implementation in certain circumstances. Nevertheless, images that are aligned with the random search and gradient descent method are done so
with good accuracy. It is believed that the random search and gradient descent algorithm could be optimized from both a theoretical and programming perspective. One such change would be an implementation in Perl rather than C, as was done in this research. This could lower the time required to run this implementation to a few minutes.

![Figure 5.5: Aligned Bender Buck Using Random Search and Gradient Descent](image)

Another problem that remains to be solved is that of image scaling during decode. The current implementation of the gradient search does not consider the possibility of
receiving a scaled version of the original image. With horizontal and vertical translation and rotation, this would increase the total number of degrees of freedom to four. By increasing the dimensions of the search space, the algorithm would also require more computation and more time requirements.

While this solution to the alignment problem is not optimal, it is sufficient to achieve proper orientation of the image for decoding given that the image is rectangular and the dimensions are known. Optimizing the search algorithm and implementing this optimization in Perl is an appropriate continuation of this work. In addition, future research could explore the problem of scaling presented above. The C implementation of both the random search and gradient descent, grad.c, appears in Appendix D.
Chapter 6:
Multiple-Bit Encoding/Decoding

6.1 Implementation
The multiple-bit encoder embeds a sequence of bits within an image in a specified order. This sequence of bits can then be extracted by the decoder and used accordingly. The decoder, for example, can convert the data into text. This is implemented, as discussed earlier, by first encoding the strong bit with a specified key (zero). The data is then encoded beginning at key one, with each successive key representing one bit. A one or a zero is encoded with either a positive or negative $S_n$. This can be achieved by modifying the basic encoding algorithm discussed in Chapter 2 with the ability to encode with both positive and negative $\delta$. This then allows the creation of a positive or negative $S_n$. After orienting with the strong bit, it is only necessary to identify the sign of the following bits. A positive $S_n$ represents a 1 while a negative $S_n$ represents a 0. The data is decoded in this manner by stepping through all of the keys and determining the corresponding $S_n$. The bits are then assigned according to whether the corresponding $S_n$ is positive or negative.

Error Correction
An additional feature that increases decoding accuracy while allowing a smaller $\delta$ to be used is error correction coding. One possible scheme, used here, encodes the data in triplicate. For example, if the data to be embedded is three bits long, keys 1, 2 and 3 would encode the first bit, keys 4, 5 and 6 would encode the second bit and keys 7, 8 and 9 would encode the third bit. This redundant coding allows for higher degrees of certainty when all three of the bit encodings agree or error correction when one fails due to noise. An example of this appears in Figure 6.1. Here the Bender Buck was watermarked with four bits, "0,1,1,0" using the triplicate error correction scheme.
discussed above. The C program, mbencode.c (Appendix E), was used to encode the image with the strong bit encoded with \( n=10,000 \) and the data bits encoded with \( n=2,000 \). All bits were encoded at \( \delta=30 \) with a radius of 15. The image was printed out and scanned back in at 135dpi. The random search and gradient descent alignment correction functions were then run to adjust the image. Finally, the image was decoded with the multiple-bit decoder implemented in C, mbdecode.c, seen in Appendix F. This particular set of data did not utilize the error correction since each encoded bit was extracted correctly.

<table>
<thead>
<tr>
<th>Seed</th>
<th>Decode Value (S_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Bit</td>
<td>39,896</td>
</tr>
<tr>
<td>1</td>
<td>-3,470</td>
</tr>
<tr>
<td>2</td>
<td>-3,154</td>
</tr>
<tr>
<td>3</td>
<td>-13,794</td>
</tr>
<tr>
<td>4</td>
<td>7,322</td>
</tr>
<tr>
<td>5</td>
<td>2,588</td>
</tr>
<tr>
<td>6</td>
<td>5,894</td>
</tr>
<tr>
<td>7</td>
<td>9,762</td>
</tr>
<tr>
<td>8</td>
<td>5,252</td>
</tr>
<tr>
<td>9</td>
<td>7,076</td>
</tr>
<tr>
<td>10</td>
<td>-7,950</td>
</tr>
<tr>
<td>11</td>
<td>-11,270</td>
</tr>
<tr>
<td>12</td>
<td>-8,568</td>
</tr>
</tbody>
</table>

*Figure 6.1: "0,1,1,0" S_n Values Under Error Correction Scheme*

For larger watermarks, each bit is encoded in this way until the desired data has been wholly encoded. The amount of data that can be stored in a particular image depends on the frequency content and size of the image. Typically this value ranges from 64 to 256 bits for most images.

The small amount of data that is contained in the image is not a significant limitation given the availability of network access. Rather than trying to place all the information that one would like to associate with a particular picture, a URL, filename or unique
serial number can be embedded instead, allowing for the dynamic updating information rather than the continual re-encoding of the image with the most recent data.
Chapter 7:

Conclusion

A complete multiple-bit encoder/decoder pair was created from the Patchwork approach. Beginning from the mathematical basis of the Patchwork algorithm for information hiding in digital images, gradual improvements were developed that allowed for the expansion to multiple-bit encoding (Patch Track) including: coarse orientation detection, random search, gradient descent and multiple-bit encoding/decoding. These improvements increased the usefulness of the Patchwork/Patch Track algorithm. Patch Track is a valuable technique for watermarking digital images for its robustness and allows the various watermarking applications discussed in Chapter 2.

Before the various uses for the Patch Track system can be successfully implemented, however, some additional improvements are necessary. For example, the random search and gradient descent is somewhat computationally intensive and does not currently perform the alignment correction in a reasonable amount of time for most applications. Solving this problem could be approached by optimization of the search algorithm and/or the programming. This can be used in conjunction with the acquisition of additional hardware. Unfortunately, an exhaustive search will always be one of the less efficient methods for alignment correction. However, exhaustive searches are particularly well suited for parallel computation. Increasing the number of CPUs involved in the random search will decrease the time required. While a form of exhaustive search, the method discussed in this thesis is nevertheless not a blind exhaustive search. Instead the process is sped by gaining information incrementally, the aggregate of which takes less time than a purely random search.
An additional problem that is left to be resolved is the extension of the decoder to account for the possibility that the incoming image may be scaled from the original size. This additional parameter will increase computing time and/or computation required.

This research has taken a significant step toward solving the alignment problem in digital information hiding in images. By eliminating a purely exhaustive search, the time required to properly align an image has been dramatically reduced. In addition, Patchwork/Patch Track has introduced various ways to decrease the visibility of watermarks in images. Both of these improvements are provide an element of direction to the development of more advanced watermarking techniques. It has been shown in this research that the various advancements contained in Patchwork/Patch Track are effective.
References

15. DataHiding Homepage™,
   http://www.trl.ibm.co.jp/projects/s7730/Hiding/index_e.htm
17. The information hiding homepage - digital watermarking & steganography,
   http://www.cl.cam.ac.uk/~fapp2/steganography/
Appendix A: encode.c

//C++ libraries/
#include <iostream>
#include <omanip>
#include <algorithm>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <math.h>

//for pseudo-random number generator
define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

//pseudo-random number generator 1 - for patch positions
//from numerical methods in c
float ran0(long *idum)
{
  long k;
  float ans;

  *idum ^= MASK;
  k = (*idum)/IQ;
  *idum = IA*(idum-k*IQ)-IR*k;
  if (*idum<0) *idum += IM;
  ans = AM(*idum);
  *idum ^= MASK;
  return ans;
}

//pseudo random number generator 2 - for random depth
float ran1(long *idum)
{
  long k;
  float ans;

  *idum ^= MASK;
  k = (*idum)/IQ;
  *idum = IA*(idum-k*IQ)-IR*k;
  if (*idum<0) *idum += IM;
  ans = AM(*idum);
  *idum ^= MASK;
  return ans;
}

//pgm structure
struct pgm{
  char* filename;
  long columns;
  long rows;
  double maxdepth;
  double* data;
};

//error message function
void error(char* s, char* s2 = "")
{
  cerr << s << ":" << s2 << \n
  exit(1);
}

//count the number of header lines in pgm files
int clines(char* filename)
{ 
    int count = 0;
    char ch;

    ifstream from(filename);

    do{
        from.get(ch);
    } while (ch != '\n');

    from.get(ch);

    while (ch == '#')
    {
        count++;
        do{
            from.get(ch);
        } while (ch != '\n');
        from.get(ch);
    }

    return count;
}

//get formatting of original file
int get_length(char* filename)
{
    char ch = '0';
    int length = 1;
    int first;
    int hlines;
    int temp;

    ifstream from(filename);

    //get to the image content, skip header lines
    hlines = clines(filename);

    for (int index = 0; index <= hlines; index++)
    {
        do{
            from.get(ch);
        } while (ch != '\n');
    }

    for (int index = 0; index <= 2; index++)
        from >> temp;

    //move to first line of image data
    from.get(ch);
    ch = '0';

    //get number of chars in first line
    while (ch != '\n')
    {
        from.get(ch);
        length++;
    }

    length = (int)(length - 3) / 4 + 1;

    return length;
}

//load picture data into structure
int get_image(char* filename, pgm* image)
{
    int length;
    long temp1;
    double temp2;
    int number;
}
double size;
char ch;

ifstream from(filename);
if (!from) error("cannot open input file ", filename);

number = clines(filename);

for (int index = 0; index <= number; index++)
{
    do{
        from.get(ch);
    } while (ch != "\n");
}

from >> temp1;
image->columns = temp1;
from >> temp1;
image->rows = temp1;
from >> temp1;
image->maxdepth = (double)temp1;

size = (image->columns) * (image->rows);
image->data = new double[size];

for (int index = 0; index < size; index++)
{
    from >> temp1;
    temp2 = (double)temp1;
    *((image->data)+index) = temp2;
}

length = getlength(filename);

return length;

//send image to file
void write_image(char* filename, pgm* image, int format)
{
    int pixel = 0;
    int index = 1;
    long total = 0;
    ofstream to(filename);
    if (!to) error("cannot open output file ", filename);

    to << "P2";
to << image->columns << " " << image->rows << "n";
to << image->maxdepth << "n";

double size = (image->columns) * (image->rows);

for (int rindex = 0; rindex < image->rows; rindex++)
{
    for (int cindex = 0; cindex < image->columns; cindex++)
    {
        if (index == 1)
        {
            to << setw(3) << (long)*((image->data)+cindex+(rindex*(image->
columns)));
            index++;
            total++;
        }
        else
        {
            to << setw(4) << (long)*((image->data)+cindex+(rindex*(image->
columns)));
            index++;
            total++;
        }
    }
}
if ((index == image->rows - 1) && (cindex == image->columns - 1))
   to << 'n';
else
   if (index == format + 1)
      [to << 'n';
       index = 1;
   ]
}
return;
}

//initialize the target data
void initialize(pgm* simage, pgm* timage)
{
timage->columns = simage->columns;
timage->rows = simage->rows;
timage->maxdepth = simage->maxdepth;
timage->filename = "# masked.pgm";

long size = (simage->columns) * (simage->rows);
timage->data = new double[size];

return;
}

//subtract, point by point, blurred image from original
void point_sub(pgm* simage, pgm* timage)
{
   long size = (simage->columns) * (simage->rows);

   for (int index = 0; index < size; index++)
      *((timage->data)+index) = *((simage->data)+index) - *((timage->data)+index);

   return;
}

//take abs of point_sub output image
void abs_image(pgm* timage)
{
double current_max = 0;
long size = timage->columns * timage->rows;

for (int index = 0; index < size; index++)
{
   if *((timage->data)+index) < 0
      *((timage->data)+index) = -*((timage->data)+index));

   if *((timage->data)+index) > current_max
      current_max = *((timage->data)+index);
}
timage->maxdepth = current_max;
return;
}

//for diagnostics only -- scales the image back up to viewable
//scales, eg. 255
void rescale_image(pgm* timage, double scale)
{
double temp = 0;
double current_max = 0; //record the new timage->maxdepth;
double maxdepth = 0;
long size = timage->columns * timage->rows;

//find max
for (int index = 0; index < size; index++)
{
   if *((timage->data)+index) > temp
      temp = *((timage->data)+index);
maxdepth = temp;

for (int index = 0; index < size; index++)
{
    temp = *((timage->data)+index) / maxdepth;
    *((timage->data)+index) = temp * scale;
    if (*((timage->data)+index) > current_max)
        current_max = *((timage->data)+index);
}
timage->maxdepth = current_max;
return;
}

//blurs the image with a square of size
void blur_image(pgm* simage, pgm* timage, int size)
{
    double new_depth;
    double pt_total;
    long yinit, xinit;
    long currenty, currentx;
    int max_dist = (size - 1)/2;

    long max_index = simage->columns * simage->rows;

    for (int yindex = 0; yindex < simage->rows; yindex++)
        for (int xindex = 0; xindex < simage->columns; xindex++)
        {
            yinit = yindex - max_dist;
            xinit = xindex - max_dist;

            pt_total = 0;

            for (int syindex = 0; syindex < size; syindex++)
            {
                currenty = yinit + syindex;

                if (currenty < 0)
                    currenty = currenty;

                if (currenty > simage->rows)
                    currenty = 2*simage->rows - currenty;

                if (currenty == simage->rows)
                    currenty = currenty - 1;

                for (int sxindex = 0; sxindex < size; sxindex++)
                {
                    currentx = xinit + sxindex;

                    if (currentx < 0)
                        currentx = currentx;

                    if (currentx > simage->columns)
                        currentx = 2*simage->columns - currentx;

                    if (currentx == simage->columns)
                        currentx = currentx - 1;

                    pt_total += *((simage->data)+currentx+(currenty*simage->columns));
                }
            }

            new_depth = pt_total/(size*size);

            *((timage->data)+xindex+(yindex*(timage->columns))) = new_depth;
        }
}

return;}
//create square patches in data
void squarein(pgm* image, pgm* mask, int n, long w, long h, long d, char rnd, long seed) {
    const long patches = n;

    long seed1 = seed;  //for random patch location
    long seed2 = 20;    //for random values inside patch
    double add_depth;
    long wth1, wth2;    //1 is positive patch, 2 is negative patch
    long hgt1, hgt2;
    long x, y;
    long random;
    double new_depth1 = 127;
    double new_depth2 = 127;
    long corners[4];   //[(x1, y1, x2, y2)]

    for (int index = 0; index < patches; index++)
    {
        wth1 = wth2 = w;
        hgt1 = hgt2 = h;

        corners[0] = (long)(ran0(&seed1)*((image->columns)));
        corners[1] = (long)(ran0(&seed1)*((image->rows)));
        corners[2] = (long)(ran0(&seed1)*((image->columns)));
        corners[3] = (long)(ran0(&seed1)*((image->rows)));

        //deal with image edges -- adjust dimensions of square to fit inside image
        if ((w > 1) && (h > 1))
        {
            if (((image->columns - corners[0]) < (wth1/2))
                wth1 = 2 * (image->columns - corners[0]);
            else
                if (corners[0] < (wth1/2))
                    wth1 = 2 * corners[0];

            if (((image->rows - corners[1]) < (hgt1/2))
                hgt1 = 2 * (image->rows - corners[1]);
            else
                if (corners[1] < (hgt1/2))
                    hgt1 = 2 * corners[1];

            if (((image->columns - corners[2]) < (wth2/2))
                wth2 = 2 * (image->columns - corners[2]);
            else
                if (corners[2] < (wth2/2))
                    wth2 = 2 * corners[2];

            if (((image->rows - corners[3]) < (hgt2/2))
                hgt2 = 2 * (image->rows - corners[3]);
            else
                if (corners[3] < (hgt2/2))
                    hgt2 = 2 * corners[3];

            for (long hindex = (-hgt1/2); hindex < (hgt1/2); hindex++)
                for (long windex = (-wth1/2); windex < (wth1/2); windex++)
                {
                    if (rnd == 'y')
                        add_depth = (double)(ran1(&seed2)*d);
                    else
                        add_depth = (double)d;

                    x = corners[0] + windex;  //current point
                    y = corners[1] + hindex;  //current point

                    new_depth1 = "((image->data)+y*image->columns+x) + " * (mask->data) + y*mask->columns+x) *

                    add_depth;

                    //center is max
                    if ((windex == 0) && (hindex == 0))
new_depth1 = *((image->data)+y*image->columns+x) + add_depth;

if (new_depth1 > 255)
    new_depth1 = 255;

*((image->data)+y*image->columns+x) = new_depth1;
}
for (long hindex = -(hgt2/2); hindex < (hgt2/2); hindex++)
for (long windex = -(wht2/2); windex < (wht2/2); windex++)
{
    if (rnd == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;

    x = corners[2] + windex; //current point
    y = corners[3] + hindex; //current point

    new_depth2 = *((image->data)+y*image->columns+x) - *((mask->data)+y*mask->columns+x) * add_depth;

    //center is max
    if (windex == 0) && (hindex == 0))
        new_depth1 = *((image->data)+y*image->columns+x) - add_depth;

    if (new_depth2 < 0)
        new_depth2 = 0;

    *((image->data)+y*image->columns+x) = new_depth2;
}

else //for the case of 1x1 patches
{
    x = corners[0]; //current point
    y = corners[1]; //current point
    if (rnd == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;

    new_depth1 = *((image->data)+y*image->columns+x) + add_depth;

    *((image->data)+y*image->columns+x) = new_depth1;

    x = corners[2]; //current point
    y = corners[3]; //current point
    if (rnd == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;

    new_depth1 = *((image->data)+y*image->columns+x) - add_depth;

    *((image->data)+y*image->columns+x) = new_depth1;
}

return;
}

//create circular patches in data
void circlein(pgm* image, pgm* mask, long n, long r, long d, char rnd, long seed)
{
    const int patches = n;

    int flag;
    int centers_error;

    long seed1 = seed; //for random patch location
    long seed2 = 20; //for random values inside patch
    double dist;
    long dx, dy;
    long rad1, rad2; //1 is positive patch, 2 is negative patch/
    long x, y;
    double rand_depth;
double add_depth;
double new_depth = 127;
long centers[4*patches]; //{x1, y1, x2, y2}/

for (int index = 0; index < patches; index++)
{
    float increment = (float)d/(float)r;
    rad1 = rad2 = r;

    //generate the centers for patches
    centers[index] = (long)(ran0(&seed1)*(image->columns));
    centers[index+patches] = (long)(ran0(&seed1)*(image->rows));
    centers[index+2*patches] = (long)(ran0(&seed1)*(image->columns));
    centers[index+3*patches] = (long)(ran0(&seed1)*(image->rows));

    for (long hindex1 = 0; hindex1 <= 2*rad1; hindex1++)
        for (long windex1 = 0; windex1 <= 2*rad1; windex1++)
        {
            dx = windex1 - rad1;
            dy = hindex1 - rad1;
            dist = sqrt(dx*dx+dy*dy); //distance from current point to center

            if (dist <= rad1)
            {
                x = centers[index] + windex1 - r; //create current point
                y = centers[index+patches] + hindex1 - r; //create current point

                //current point inside image
                if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
                {
                    add_depth = (double)(increment*(rad1 - dist)); //create conical depth

                    if (add_depth <= 0) //true inside the radius
                    {
                        if (add_depth == d) //true at center
                        {
                            new_depth = *(image->data)+y*(image->columns+x)+d;
                        }
                        else
                        {
                            if (rd == y) //randomize depth
                            {
                                rand_depth = (double)(ran1(&seed2)*add_depth);
                            }
                            else
                            {
                                rand_depth = add_depth;
                            }

                            new_depth = *(image->data)+y*(image->columns+x)+*(mask->data)+y*(mask-
                                            ->columns+x)*rand_depth; //assign new depth
                        }
                    }
                    else
                    {
                        new_depth = *(image->data)+y*(image->columns+x)+*(mask->data)+y*(mask-
                                            ->columns+x)*add_depth; //true at the radius

                        if (new_depth > 255)
                        {
                            new_depth = 255;
                            *(image->data)+y*(image->columns+x) = new_depth;
                        }
                    }
                }
            }
        }
}
if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
{
    add_depth = (double)(increment*(rad2 - dist)); //creates conical depth

    if (add_depth != 0) //true inside the radius
    {
        if (add_depth == d) //true at the center
            new_depth = *((image->data)+y*image->columns+x)-d;
        else
        {
            if (rnd == 'y')
                rand_depth = (double)(ran1(&seed2)*add_depth); //randomize the depth
            else
                rand_depth = add_depth;
            new_depth = *((image->data)+y*image->columns+x) - *((mask->data)+y*mask->columns+x) * rand_depth; //assign new depth
        }
    }
    else
        new_depth = *((image->data)+y*image->columns+x) - *((mask->data)+y*mask->columns+x) * add_depth; //assign new depth at the radius (no change)

    if (new_depth < 0)
        new_depth = 0;

    *((image->data)+y*image->columns+x) = new_depth;
}
}
return;
}

int main(int argc, char* argv[])
{
    pgm image;
    pgm mask, mask2;
    int format;

    char* type = argv[3];
    char* rnd = argv[4];

    if (*type == 'c')
    {
        if (argc != 9) error("Wrong number of arguments"); //check arguments/

        long number = atol(argv[5]);
        long radius = atol(argv[6]);
        long depth = atol(argv[7]);
        long seed = atol(argv[8]);

        cerr << "Initializing Image in":
        format = get_image(argv[1], &image);

        //for mask/
        double scale = 1;
        int size1 = 7; //blur sizes/
        int size2 = 5;

        initialize(&image, &mask);
        initialize(&image, &mask2);

        cerr << "Creating Visibility Mask in":
        blur_image(&image, &mask2, size1);
        point_sub(&image, &mask2);
        abs_image(&mask2);
        initialize(&mask2, &mask); //mask is now the current altered version.
        blur_image(&mask2, &mask, size2);
        rescale_image(&mask, scale); //scale to 0-1 level
    }
cerr << "Encoding image in";
circlein(&image, &mask, number, radius, depth, *rnd, seed);

cerr << "Writing image in";
write_image(argv[2], &image, format);
}

if (*type == 's')
{
    if (argc != 10) error("Wrong number of arguments"); //check arguments

    long number = atol(argv[5]);
    long width = atol(argv[6]);
    long height = atol(argv[7]);
    long depth = atol(argv[8]);
    long seed = atol(argv[9]);

cerr << "Initializing image in";
format = get_image(argv[1], &image);

    //for mask/
    double scale = 1;
    int size1 = 7; //blur sizes/
    int size2 = 5;

    initialize(&image, &mask);
    initialize(&image, &mask2);

cerr << "Creating Visibility Mask In";
blur_image(&image, &mask2, size1);
point_sub(&image, &mask2);
abs_image(&mask2);
initialize(&mask2, &mask); //mask is now the current altered version.
blur_image(&mask2, &mask, size2);
rescale_image(&mask, scale); //scale to 0-1 level

cerr << "Encoding Squares in";
squarein(&image, &mask, number, width, height, depth, *rnd, seed);

cerr << "Writing image in";
write_image(argv[2], &image, format);
}
return 0;
Appendix B: decode.c

//C++ libraries*

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <time.h>  //for rand()//

//for pseudo-random number generator
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

//pseudo-random number generator
float ran0(long *idum)
{
    long k;
    float ans;

    *idum ^= MASK;
    k = (*idum)/IQ;
    *idum = IA*(idum-k*IQ)-IR*k;
    if (*idum<0) *idum += IM;
    ans = AM*(idum);
    *idum ^= MASK;
    return ans;
}

//error message function
void error(char* s, char* s2 = "")
{
    cerr << s << s2 << "\n";
    exit(1);
}

//pgm structure
struct pgm{
    char* filename;
    long columns;
    long rows;
    double maxdepth;
    double* data;
};

//count number of header lines
int clikes(char* filename)
{
    int count = 0;
    char ch;

    ifstream from(filename);

    do{
        from.get(ch);
    } while (ch != '\n');

    from.get(ch);

    while (ch == '#')
    {
        count++;
        do{
            from.get(ch);
        } while (ch != '\n');
    }
from.get(ch);    
}            
return count;     
}                  

//get image from file  
void get_image(char* filename, pgm* image)   
{    
    ifstream from(filename);   
    if (!from) error("cannot open input file ", filename);   
    char ch;   
    long temp1;   
    double temp2;   
    double size;   
    int number = clines(filename);   
    for (int index = 0; index <= number; index++)   
    {   
        do{    
            from.get(ch);   
        } while (ch != "n");   
    }    
    from >> temp1;    
    image->columns = temp1;    
    from >> temp1;    
    image->rows = temp1;    
    from >> temp1;    
    image->maxdepth = (double)temp1;    
    size = (image->columns) * (image->rows);    
    image->data = new double[size];    
    for (int index = 0; index < size; index++)   
    {   
        from >> temp1;   
        temp2 = (double)temp1;   
        *((image->data)+index) = temp2;   
    }                  
    return;            
}                  

//get sum for seed s   
double get_points(pgm* image, long n, long s)   
{    
    long seed1 = s;    //** DECODE SEED **/    
    const long summations = 1;    
    double point_a, point_b;    
    long ar, ac, br, bc;    
    double S = 0;    
    int col = image->columns;    
    
    for (long sum_index = 1; sum_index < (summations+1); sum_index++)   
    {    
        for (long pt_index = 0; pt_index < n; pt_index++)   
        {   
            ac = (long)(ran0(&seed1)*(image->columns));   
            ar = (long)(ran0(&seed1)*(image->rows));   
            bc = (long)(ran0(&seed1)*(image->columns));   
            br = (long)(ran0(&seed1)*(image->rows));   
            point_a = *((image->data)+ac+ar*col);   
            point_b = *((image->data)+bc+br*col);   
            S = S+(point_a - point_b);   
        }  
    }                  
    return(S);                 
}                  

64
```c
int main(int argc, char* argv[]) {
    int seedr = 20;  //number of seeds from 0 to decode
    long seed;
    pgm image;
    int hightindex = 0;
    double highsum = 0;
    double sechigh = 0;
    double sum = 0;
    long number;

    if (argc != 4) error("Wrong number of arguments");
    number = atol(argv[3]);

    ofstream to(argv[2]);
    if (fio) error("cannot open input file ", argv[2]);

    cerr << "Acquiring Image\n";
    get_image(argv[1], &image);

    cerr << "Decoding Image\n";
    for (int index = 0; index <= seedr; index++)
    {
        seed = (long)index;
        cerr << " Seed: " << seed << " ";
        sum = get_points(&image, number, seed);
        cerr << " Finished: " << sum << "\n";
        to << seed << " " << sum << "\n";
        if (sum > highsum)
        {
            sechigh = highsum;
            highsum = sum;
            hightindex = seed;
        }
        else
        if (sum > sechigh)
            sechigh = sum;
    }

    if (highsum > 2*sechigh)
        cerr << "Data found at Seed: " << hightindex << "\n\n";
    else
        cerr << "No Data found\n\n";

    return 0;
}
```
Appendix C: odetect.c

// C++ libraries

#include <iostream.h>
#include <iomanip.h> //for setw()
#include <fstream.h> //for file output
#include <stdlib.h>
#include <time.h> //for rand()
#include <math.h>
#include <cassert.h>
#include <stdio.h>
#include <string>
#include <sstream>

#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

// for pseudo-random number generator
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

// pseudo-random number generator
float ran0(long *idum)
{
  long k;
  float ans;

  *idum ^= MASK;
  k = (*idum)/IQ;
  *idum = IA*(idum-k*IQ)-IR*k;
  if (*idum<0) *idum += IM;
  ans = AM*(*idum);
  *idum ^= MASK;
  return ans;
}

// pgm structure
struct pgm{
  char* filename;
  long columns;
  long rows;
  double maxdepth;
  double* data;
};

// error message function
void error(char* s, char* s2 = "")
{
  cerr << s << s2 << "\n";
  exit(1);
}

// print matrix
void prntxt(float* mat, int size)
{
  cerr << "\n";

  for (int index = 0; index < size; index++)
    cerr << mat[index] << " ,";

cerr << "\n";
return;
// counts the number of comment lines - written by Fernando Paiz
int clines(const char* filename, int& img_type)
{
    int count = 0;
    char* ch = new char[1];

    ifstream from(filename);
    from.get(ch[0]);
    from.get(ch[0]);

    img_type = atoi(ch);

    do{
        from.get(ch[0]);
    } while (ch[0] != '\n');
    from.get(ch[0]);

    while (ch[0] == '#')
    {
        count++;
        do{
            from.get(ch[0]);
        } while (ch[0] != '\n');
        from.get(ch[0]);
    }

    return count;
}

// load picture data into structure
void get_pgm(const string& filename, pgm* image) {
    ifstream from(filename.c_str());
    if (!from) error("cannot open input file ");

    char ch;
    long temp;
    long size;

    int img_type;
    int number = clines(filename.c_str(), img_type);

    if (img_type == 2) {
        for (int lines = 0; lines <= number; lines++)
        {
            from >> temp;
            image->columns = temp;
            from >> temp;
            image->rows = temp;
            from >> temp;
            image->maxdepth = (double)temp;

            cout << image->columns << "x" << image->rows << " ascii PGM file read. " << endl;

            size = (image->columns) * (image->rows);
            image->data = new double[size];

            for (int index = 0; index < size; index++)
            {
                from >> temp;
                *(image->data)+index = (double)temp;
            }
        }
    }
}
return;
} else if (img_type == 5) {

    ifstream is(filename.c_str());
    char buffer[128];
    is.getline(buffer, 128);
    is.getline(buffer, 128);
    while(buffer[0] == '#') is.getline(buffer, 128);

    istream str(buffer, 128);
    long width, height;
    str >> width;
    str >> height;
    is.getline(buffer, 128);
    cout << width << "x" << height << " binary PGM file read." << endl;
    image->columns = width;
    image->rows = height;
    image->maxdepth = (double)255;
    long size = (image->columns) * (image->rows);
    image->data = new double[size];

    unsigned char mybuf[1024];
    long num;
    long index = 0;
    // while(is.read(mybuf, 1024)){
    while (1) {
      is.read(mybuf, 1024);
      for (long i=0;i<1024; i++)
        long tlong = mybuf[i];

        image->data[index] = (double) mybuf[i];
        index++;
        //cout << index << " to " << size << endl;
        if (index >= size) return;
    }
}
return;
}

} else {
    cerr << "ERROR! Unknown PGM file format." << img_type << endl;
    exit(1);
}
}

//get the sums for clockwise and counterclockwise rotation
void get_points(pgm* imagecw, pgm* imageccw, long n, long s, double S[2])
{
    long seed1 = s; //** DECODE SEED **/
const long summations = 1;
double ppointcw, npointcw;
double ppointccw, npointccw;
long arcw, accw, brcw, bccw;
long arccw, acccw, brcw, bcccw;
double Scw = 0;
double Sccw = 0;
int colcw = imagecw->columns;
int colccw = imageccw->columns;

    for (long sum_index = 1; sum_index < (summations+1); sum_index++)
    {
      for (long pt_index = 0; pt_index < n; pt_index++)
        {

68
float temp = ran0(&seed1);

accw = (long)(temp*(imagecw->columns));
acccw = (long)(temp*(imageccw->columns));
temp = ran0(&seed1);
arcw = (long)(temp*(imagecw->rows));
arccw = (long)(temp*(imageccw->rows));
temp = ran0(&seed1);
bccw = (long)(temp*(imagecw->columns));
bcccw = (long)(temp*(imageccw->columns));
temp = ran0(&seed1);
brcw = (long)(temp*(imagecw->rows));
brccw = (long)(temp*(imageccw->rows));

ppointcw = "((imagecw->data)+accw+arcw*colcw);
npointcw = "((imagecw->data)+bccw+brcw*colcw);

//c and r switched for ccw
ppointccw = "((imageccw->data)+accw+arcw*colccw);
npointccw = "((imageccw->data)+bccw+brcw*colccw);

Scw = Scw+(ppointcw - npointcw);
Sccw = Sccw+(ppointccw - npointccw);

}

S[0] = Scw;
S[1] = Sccw;
return;
}

//initialize pgm file
void initialize(pgm* simage, pgm* timage)
{
timage->columns = simage->columns,
timage->rows = simage->rows;
timage->maxdepth = simage->maxdepth;
timage->filename = "# quantized.pgm";

long size = (simage->columns) * (simage->rows);
timage->data = new double[size];

return;
}

//quantize image to 1 bit
void quantimage(pgm* image, pgm* quant)
{
int quantlevel = 167; //the threshold for quantization for BW
int index = 0;
int flag = 0;
int maxindex = (image->columns)*(image->rows);

while (index <= maxindex)
{
if (*((image->data)+index) >= quantlevel)
{
*((quant->data)+index) = 255;
flag = 1;
}
else
*((quant->data)+index) = 0;

index++;
}

if (flag == 1)
image->maxdepth = 255;
}
//return the value of the pixel at the position desired after reframing
double imgpos(int x, int y, pgm* image, float* rot, float* sft)
{
    int rows = image->columns;
    int cols = image->rows;
    double out;

    out = *((image->data) + x*cols + y*cols);

    return out;
}

//clean the edges of the quantized image
void cleanedges(pgm* image)
{
    int side = 4; //how much to clean on the side
    int top = 4; //how much to clean on the top/bot
    int rows = image->rows;
    int cols = image->columns;

    //clean sides
    for (int rindex = 0; rindex < rows; rindex++)
    {
        for (int cindex = 0; cindex < side; cindex++)
        {
            *((image->data) + cindex + rindex*cols) = 255;
            *((image->data) + cols - cindex + rindex*cols) = 255;
        }
    }

    //clean top/bot
    for (int rindex = 0; rindex < top; rindex++)
    {
        for (int cindex = 0; cindex < cols; cindex++)
        {
            *((image->data) + cindex + rindex*cols) = 255;
            *((image->data) + cols - cindex + (rows - rindex)*cols) = 255;
        }
    }

    return;
}

//send image to file
void write_image(char* filename, pgm* image, int format)
{
    int pixel = 0;
    int index = 1;
    long total = 0;
    ofstream to(filename);
    if (to) error("cannot open output file", filename);

    to << "P2\n";
    to << image->columns << " " << image->rows << "\n";
    to << image->maxdepth << "\n";

    double size = (image->columns) * (image->rows);

    for (int rindex = 0; rindex < image->rows; rindex++)
    {
        for (int cindex = 0; cindex < image->columns; cindex++)
        {
            if (index == 1)
            {
                to << setw(3) << (long)*((image->data) + cindex + (rindex*(image->columns)));
                index++;
                total++;
            }
            else
            {
                if (index == format)
                {

```
```c
#include "setw(4)" << (long)((image->data)+cindex+(rindex*(image->columns))));
index++;
total++;
}
if ((rindex == image->rows - 1) && (cindex == image->columns - 1))
to << 'n';
else
  if (index == format + 1)
  {
    to << 'n';
    index = 1;
  }
}
return;

//set view frame for corner detection to 0
int allzero(double frame[5])
{
  int cond = 0;
  int time = 0;

  for (int index = 0; index < 5; index++)
  {
    if (frame[index] == 0)
    {
      if (time == 0)
      {
        cond = index;
        time = 1;
      }
      else
      {
        if (frame[index - 1] == 0 && (time == 1))
          cond = index;
      }
    }
  }
  return cond;
}

//scan for the top corner
void hscantop(pgm* image, int corner[4])
{
  double frame1[5], frame2[5];
  int place;
  double size;
  int top, find = 0;
  int ref, chk;
  int col;

  col = image->columns;
  size = image->columns * image->rows;

  for (int n = 0; n++; n < 5)
    frame2[n] = *((((image->data) + n));
  for (int n = 0; n++; n < 5)
    frame1[n] = *((((image->data) + n + 5));

  //next unread number
  place = 10;
  while (find && (place < size))
  {
    for (int index = 0; index < 5; index++)
    {
      frame2[index] = frame1[index];
      frame1[index] = *((((image->data) + place + index));
    }
  }
```
place = place + 5;
find = allzero(frame1);

ref = place + find - 5;
chk = col * 30;

if (find && (*((image->data) + ref + chk) == 255))
{
    find = 0;
}

corner[0] = ref;
return;

//scan for the bottom corner
void hscanbot(pgm* image, int corner[4])
{
    double frame1[5], frame2[5];
    int place;
    double size;
    int top, find = 0;
    int ref;
    int col, chk;

    col = image->columns;
    size = image->columns * image->rows;

    //read backwards from end
    for (int n = 0; n++ < 5)
        frame2[n] = *((image->data) + ((int)size - 4 + n));
    for (int n = 0; n++ < 5)
        frame1[n] = *((image->data) + ((int)size - 9 + n));

    //next unread number
    place = (int)size - 9;

    while (find && (place < size))
    {
        for (int index = 0; index < 5; index++)
            {frame2[index] = frame1[index];
             frame1[index] = *((image->data) + place - 4 + index);
            }
        place = place - 5;
        find = allzero(frame1);
        ref = place + find + 1;
        chk = col * 30;

        if (find && (*((image->data) + ref - chk) == 255))
        {
            find = 0;
        }
        corner[1] = ref;
        return;
    }

    //scan for the right corner
    void vscanrt(pgm* image, int corner[4])
    {
        double frame1[5], frame2[5];
        int place;
        double size;
        int top, find = 0;
int ref, pos, mul;
int col, chk;

col = image->columns;
size = image->columns * image->rows;

for (int n = 0; n++ < 5)
    frame2[n] = *((image->data) + (n*col)+col);
for (int n = 0; n++ < 5)
    frame1[n] = *((image->data) + (n+5)*col+col);

//next unread number
place = 16;

while (find && (place < size))
{
    pos = place;
    mul = 0;
    while (pos > image->rows)
    {
        pos = pos - image->rows;
        mul++;
    }

    for (int index = 0; index < 5; index++)
    {
        frame2[index] = frame1[index];

        if ((pos+index) < image->rows)
        {
            frame1[index] = *((image->data) + (pos+index)*col+col+mul);
        }
        else
        {
            pos = pos - image->rows;
            mul++;
            frame1[index] = *((image->data) + (pos+index)*col+col+mul);
        }
    }
    place = place + 5;
    find = allzero(frame1);

    pos = place;
    mul = 0;

    while (pos > image->rows)
    {
        pos = pos - image->rows;
        mul++;
    }

    ref = (pos)*col+col.mul;

    //check point
    chk = 5;

    if (find && *((image->data) + ref - chk) == 255))
    {
        find = 0;
    }
}
corner[3] = ref;
return;
}

//scan for the left corner
void vscanlt(pgm* image, int corner[4])
{
    double frame1[5], frame2[5];
int place;
double size;
int top, find = 0;
int ref, pos, mul;
int col, chk;

col = image->columns;
size = image->columns * image->rows;

for (int n = 0; n++ < 5)
    frame2[n] = "((image->data) + (n*col));
for (int n = 0; n++ < 5)
    frame1[n] = "((image->data) + (n+5)*col);"

//next unread number
place = 10;

while (find && (place < size))
{
    pos = place;
    mul = 0;
    while (pos > image->rows)
    {
        pos = pos - image->rows;
        mul++;
    }
    for (int index = 0; index < 5; index++)
    {
        frame2[index] = frame1[index];
        if (pos+index < image->rows)
            frame1[index] = "((image->data) + (pos+index)*col+mul);
        else
            { pos = pos - image->rows;
              mul++;
              frame1[index] = "((image->data) + (pos+index)*col+mul);
            }
    }
    place = place + 5;
    find = allzero(frame1);
    pos = place;
    mul = 0;
    while (pos > image->rows)
    {
        pos = pos - image->rows;
        mul++;
    }
    ref = mul + (pos)*col;

    //check point
    chk = 8;

    if (find && ("((image->data) + ref + chk) == 255))
    {
        find = 0;
    }
    corner[2] = ref;
    return;
}

//for testing, label corners
void update(pgm* quant, int corner[4], float cnter[2])

{
    int center[2];
    int col = quant->columns;

    center[0] = (int)center[0];
    center[1] = (int)center[1];

    for (int indx = 0; indx < 4; indx++)
    {
        *((quant->data)+corner[indx]) = 128;
        for (int pt = 0; pt < 10; pt++)
        {
            *((quant->data)+corner[indx]+pt) = 128;
            *((quant->data)+corner[indx]+pt*col) = 128;
            *((quant->data)+corner[indx]+pt*col+pt) = 128;
            *((quant->data)+corner[indx]+pt*col+pt*col) = 128;
        }
    }

    *((quant->data)+center[0]+center[1]*col) = 128;
    for (int pt = 0; pt < 10; pt++)
    {
        *((quant->data)+center[0]+center[1]*col+pt) = 128;
        *((quant->data)+center[0]+center[1]*col+pt*col) = 128;
        *((quant->data)+center[0]+center[1]*col+pt*col+pt) = 128;
    }

    return;
}

//find image center
double find_center(pgm* quant, int corner[4], float center[2])
{
    int CoordCorner[8];
    int side[2];
    int temp = 0;
    int xtmp1 = 0, xtmp2 = 0, xtmp3 = 0;
    int ytmp1 = 0, ytmp2 = 0, ytmp3 = 0;
    int col = quant->columns;
    float beta1, beta0, alpha1, alpha0;
    double angle;

    for (int index = 0; index < 4; index++)
    {
        temp = (int)(corner[index] / quant->columns);
        CoordCorner[index * 2 + 1] = temp;
        CoordCorner[index * 2] = (corner[index] % quant->columns);
    }

    ytmp1 = (int)((CoordCorner[5] - CoordCorner[3])/2 + CoordCorner[3]);

    xtmp2 = (int)((CoordCorner[0] - CoordCorner[6])/2 + CoordCorner[6]);
    ytmp2 = (int)((CoordCorner[1] - CoordCorner[7])/2 + CoordCorner[7]);

    xtmp3 = (int)((xtmp2 - xtmp1)/2 + xtmp1);
    ytmp3 = (int)((ytmp2 - ytmp1)/2 + ytmp1);

    center[0] = (float)xtmp3;
    center[1] = (float)ytmp3;

    alpha0 = center[0];
    beta0 = center[1];

    alpha1 = (float)xtmp2;
    beta1 = (float)ytmp2;

    angle = atan2(beta1-beta0, alpha1-alpha0);
}
return angle;
}

//shift matrix
float* makeshift(float xyshift[2])
{
    float* S = new float[9];
    S[0] = 1;
    S[1] = 0;
    S[3] = 0;
    S[4] = 1;
    S[6] = 0;
    S[7] = 0;
    S[8] = 1;
    S[2] = xyshift[0];
    S[5] = xyshift[1];
    return S;
}

//rotation matrix clockwise
float* makerot(float angle)
{
    float* R = new float[9];
    R[0] = (float)cos((double)angle);
    R[1] = (float)sin((double)angle);
    R[2] = 0.0;
    R[3] = (float)sin((double)angle);
    R[4] = (float)cos((double)angle);
    R[5] = 0.0;
    R[6] = 0.0;
    R[7] = 0.0;
    R[8] = 1.0;
    return R;
}

//multiply 3x3 matrix by matrix a*b
float* mulmatmult(float* b, float* a)
{
    float* temp = new float[9];
    temp[0] = a[0]*b[0]+a[1]*b[3]+a[2]*b[6];
    return temp;
}

//multiply a 3x3 matrix by a vector a*b
float* mulmatvec(float* b, float* a)
{
    float* temp = new float[3];
    temp[0] = a[0]*b[0]+a[1]*b[1]+a[2]*b[2];
    return temp;
}
//inverse 3x3 matrix
float* invmtx(float* T)
{
    float* l = new float[9];

    l[0] = T[8]*T[4] - T[5]*T[7];


    for (int i = 0; i < 9; i++)
        l[i] = tmp;

    return l;
}

//get dimensions of the desired image
float* getdim(pgm* image, int corner[4])
{
    float* dist = new float[2];
    int coord[6];

    //upper left then lower
    int temp = (int)(corner[0] / image->columns);
    coord[1] = temp;
    coord[0] = (corner[0] % image->columns);

    temp = (int)(corner[2] / image->columns);
    coord[3] = temp;
    coord[2] = (corner[2] % image->columns);

    temp = (int)(corner[1] / image->columns);
    coord[5] = temp;
    coord[4] = (corner[1] % image->columns);


    return dist;
}

int index(pgm* image, int x, int y){
    return image->columns * y + x;
}

//transforms image and interpolates
void transform(pgm* image, pgm* output, float* l)
{
    float* in = new float[3];
    float* out;

    for (int i=0; i<output->columns; i++)
        for (int j=0; j<output->rows; j++)
        {
            in[0] = (float) i;
            in[1] = (float) j;
            in[2] = 1.0;

            out = mulmatvec(in, l);

            int x0 = (int) (floor((double) out[0]));
            int x1 = (int) (ceil((double) out[0]));

            //interpolate
            float y0 = (out[0] - x0);
if (x0 == x1) x1++;  

int y0 = (int) (floor((double) out[1]));  
int y1 = (int) (ceil((double) out[1]));  
if (y0 == y1) y1++;  

float xscale = out[0] - (float) x0;  
float yscale = out[1] - (float) y0;  

if (x0 < 0 || x1 >= output->columns ||  
y0 < 0 || y1 >= output->rows)  
    output->data[index(output, i, j)] = 0.0;  
} else  
    float dx0y0 = image->data[index(output, x0, y0)];  
    float dx0y1 = image->data[index(output, x0, y1)];  
    float dx1y0 = image->data[index(output, x1, y0)];  
    float dx1y1 = image->data[index(output, x1, y1)];  
    float tmp1 = xscale * (dx1y0 - dx0y0) + dx0y0;  
    float tmp2 = xscale * (dx1y1 - dx0y1) + dx0y1;  
    float lev = yscale * (tmp2 - tmp1) + tmp1;  
    output->data[index(output, i, j)] = lev;  

}  

delete [] out;  
}  

delete [] in;  
}  

//initialize new file to fit cropped picture  
void specialinit(pgm* output, float x, float y)  
{  
    output->columns = (long)x;  
    output->rows = (long)y;  

    return;  
}  

//crop image for write  
void trunc(pgm* output2, pgm* output3)  
{  
    int col3 = output3->columns;  
    int col2 = output2->columns;  

    for (int i = 0; i < output3->rows; i++)  
    {  
        for (int j = 0; j < output3->columns; j++)  
        {  
            *((output3->data)+i*col3+j) = *((output2->data)+i*col2+j);  
        }  
    }  

    return;  
}  

int main(int argc, char* argv[])  
{  
    const double pi = 3.14159265359;  

    int corner[4];  
    float center[2];  
    double negcenter[2];  
    int newcenter[2];  
    pgm image;  
    pgm quant;  
    pgm outputcw, outputccw;  
    pgm outputcw2, outputccw2;  
    pgm outputcw3, outputccw3;  
    float angle;  
    int corner1;
int format = 0;
int total = 0;
float* M = new float[9];
float* M2 = new float[9];
float* S = new float[9];
float* R = new float[9];
float* R2 = new float[9];
float* B = new float[9];
float* T = new float[9];
float* T2 = new float[9];
float* I = new float[9];
float* X = new float[9];

if (argc == 1)
{
    cerr << "orient input-file cw-output ccw-output cw-data-output ccw-data-output number-of-patches";
    return 0;
}
else
    if (argc != 8) error("Wrong number of arguments"); //check arguments

cerr << "Initializing Image\n";
format = 14; //get_length(argv[1]);
get_pgm(argv[1], &image);
initialize(&image, &quant);
initialize(&image, &outputcw);
initialize(&image, &outputcw2);
initialize(&image, &outputcw3);
initialize(&image, &outputccw);
initialize(&image, &outputccw2);
initialize(&image, &outputccw3);

cerr << "Quantizing Image\n";
quantimage(&image, &quant);

cerr << "Finding Position\n";
cleanedges(&quant);

hscantop(&quant, corner);
hscanbot(&quant, corner);
vscanit(&quant, corner);
vscanit(&quant, corner);

cerr << "Finding Rotation\n";
angle = find_center(&quant, corner, center);
write_image(argv[2], &quant, format);
float* initial = new float[2];
initial[0] = -center[0];
initial[1] = -center[1];

cerr << "Calculating Transformation\n";
//shift to 0,0
S = makeshift(initial);

//rotate by correct angle, either cw or ccw
R = makerot((float)angle);
R2 = makerot((float)(-angle-pi/2));

//shift back to original position
B = makeshift(center);

//multiply S and R
M = mulmatmat(S, R);
M2 = mulmatmat(S, R2);

//multiply M and B
T = mulmatmat(M, B);
T2 = mulmatmat(M2, B);

//invert for transformation of rotated image
float* SS = invmtx(T);
float* SS2 = invmtx(T2);
transform(&image, &outputcw, SS);
transform(&image, &outputccw, SS2);
delete [] SS;

// get dimensions of desired image
float* length = getdim(&image, corner); // height, width of image

// calculate shift need to move image to upper left to truncate
float* isolatecw = new float[2];
float* isolateccw = new float[2];

isolatecw[0] = -center[0] + .5 * length[0];
isolatecw[1] = -center[1] + .5 * length[1];
S = makeshift(isolatecw);
SS = invmtx(S);

// lengths are switched for ccw
isolateccw[0] = -center[0] + .5 * length[0];
isolateccw[1] = -center[1] + .5 * length[1];
S = makeshift(isolateccw);
SS2 = invmtx(S);

cerr << "Performing Transformation\n";

// shift to upper left
transform(&outputcw, &outputcw2, SS);
transform(&outputccw, &outputccw2, SS2);

// truncate image
specialinit(&outputcw3, length[0], length[1]);
trunc(&outputcw2, &outputcw3); // crop outputcw2 to just the bill
specialinit(&outputccw3, length[1], length[0]);
trunc(&outputcw2, &outputccw3); // crop outputccw2 to just the bill

write_image(argv[3], &outputcw3, format);
write_image(argv[4], &outputccw3, format);

// DECODE COMMANDS
int seedr = 20; // ** DECODE RANGE **/
long seed;
int highindexcw = 0;
int highindexccw = 0;
double highsumcw = 0;
double highsumccw = 0;
double sehichgcw = 0;
double sehichgccw = 0;
double sum[2];
long number;

number = atol(argv[7]);
ostream locw(argv[5]);
ostream loccw(argv[6]);

// decode for counterclockwise and clockwise turn
cerr << "Decoding Image\n";
cerr << " Seed CW CCW\n";
for (int index = 0; index <= seedr; index++)
{
    seed = (long)index;

    // sum[0]: cw, sum[1]: ccw
    get_points(&outputcw3, &outputccw3, number, seed, sum);
cerr << "* * * * setw(4) * seed * * * * setw(4) * sum[0] * * * * setw(4) * sum[1] " << "\n\n); 
tocw << seed << " \n; 
toccw << seed << " \n; 

if (sum[0] > highsumcw) 
    
    sechighcw = highsumcw; 
    highsumcw = sum[0]; 
    highindexcw = seed; 

else 
    if (sum[0] > sechighcw) 
        sechighcw = sum[0]; 

if (sum[1] > highsumccw) 
    
    sechighccw = highsumccw; 
    highsumccw = sum[1]; 
    highindexccw = seed; 

else 
    if (sum[1] > sechighccw) 
        sechighccw = sum[1]; 

if (highsumcw > highsumccw) 
    
    if (highsumcw > 1.4*sechighcw) 
        cerr << "\nLet me guess...was you number * " << (long)highindexcw << "?\n\n; 
        cerr << "\nNo Data found\n\n; 
    else 
    
    else 
        cerr << "\nLet me guess...was you number * " << (long)highindexccw << "?\n\n; 
        cerr << "\nNo Data found\n\n; 

return 0; 
}
Appendix D: grad.c

//C++ libraries/
#include<iostream.h>
#include<io manip.h> //for setw()//
#include<fstream.h> //for file output//
#include<stdio.h>
#include<time.h> //for rand()//
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include<string>
#include<stringstream>

//for pseudo-random number generator
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876
#define pi 3.14159265359
#define refnum 10 //number of gradients stored to refine

//pseudo-random number generator 1 - for patch position
float ran0(long *idum)
{
  long k;
  float ans;

  *idum ^= MASK;
  k = (*idum)/IQ;
  *idum = IA*(idum-k*IQ)-IR*k;
  if (*idum<0) *idum += IM;
  ans = AM *(*idum);
  *idum ^= MASK;
  return ans;
}

//pseudo-random number generator 2 - for window orientations
float ran1(long *idum)
{
  long k;
  float ans;

  *idum ^= MASK;
  k = (*idum)/IQ;
  *idum = IA*(idum-k*IQ)-IR*k;
  if (*idum<0) *idum += IM;
  ans = AM *(*idum);
  *idum ^= MASK;
  return ans;
}

//pgm structure
struct pgm
{
  char* filename;
  long columns;
  long rows;
  double maxdepth;
  double* data;
};

//returns the reference position for the x,y coordinate
int index(pgm* image, int x, int y)
{
  return image->columns * y + x;
}
// error message
void error(const char* s, const char* s2 = "")
{
    cerr << s << ": " << s2 << ":\n";
    exit(1);
}

// prints matrix mat
void prmtx(float* mat, int size)
{
    cerr << \n[
;
    for (int index = 0; index < size; index++)
        cerr << mat[index] << \",\n";
    cerr << \n;
    return;
}

// counts comment lines
int clines(const char* filename, int& img_type)
{
    int count = 0;
    char* ch = new char[1];
    ifstream from(filename);
    from.get(ch[0]);
    from.get(ch[0]);
    img_type = atoi(ch);
    do{
        from.get(ch[0]);
    } while (ch[0] != \n); 
    from.get(ch[0]);
    while (ch[0] == \#)
    {
        count++;
        do{
            from.get(ch[0]);
        } while (ch[0] != \n);
        from.get(ch[0]);
    }
    return count;
}

// load picture data into structure
void get_pgm(const string& filename, pgm* image)
{
    ifstream from(filename.c_str());
    if (!from) error("cannot open input file \", filename.c_str());
    char ch;
    long temp;
    long size;
    int img_type;
    int number = clines(filename.c_str(), img_type);
    if (img_type == 2) {
        for (int lines = 0; lines <= number; lines++)
            do{
                from.get(ch);
```c++
} while (ch != 'n');

from >> temp;
image->columns = temp;
from >> temp;
image->rows = temp;
from >> temp;
image->maxdepth = (double)temp;

cout << image->columns << 'x' << image->rows << " ascii PGM file read." << endl;

size = (image->columns) * (image->rows);
image->data = new double[size];

for (int index = 0; index < size; index++) {
  from >> temp;
  *((image->data)+index) = (double)temp;
}

return;
}
else if (img_type == 5) {
  ifstream is(filename.c_str());
  char buffer[128];

  is.getline(buffer, 128);
  is.getline(buffer, 128);
  while(buffer[0] != '#') is.getline(buffer, 128);
  ifstream str(buffer, 128);
  long width, height;

  str >> width;
  str >> height;
  is.getline(buffer, 128);
  cout << width << 'x' << height << " binary PGM file read." << endl;

  image->columns = width;
  image->rows = height;
  image->maxdepth = (double)255;

  long size = (image->columns) * (image->rows);
  image->data = new double[size];

  unsigned char mybuf[1024];
  long num;
  long index = 0;

  while (1) {
    is.read(mybuf, 1024);
    for (long i=0;i<1024; i++) {
      long ilong = mybuf[i];
      image->data[index] = (double) mybuf[i];
      index++;

      if (index >= size) return;
    }
  }
  return;
}
else {
  cerr << "ERROR! Unknown PGM file format: " << img_type << endl;
  exit(1);
}

//initialize pgm file for write
void initialize(pgmm* simage, pgmm* timage) {
  timage->columns = simage->columns;
```

84
timage->rows = simage->rows;
timage->maxdepth = simage->maxdepth;
timage->filename = "# quantized.pgm";

long size = (simage->columns) * (simage->rows);
timage->data = new double[size];

return;
}

//send image to file
void write_image(char* filename, pgm* image, int format)
{
    int pixel = 0;
    int index = 1;
    long total = 0;
    ofstream to(filename);
    if (fio) error("cannot open output file", filename);
    to << "P2\n# patch.pgm\n";
    to << image->columns << " " << image->rows << '\n';
    to << image->maxdepth << '\n';

double size = (image->columns) * (image->rows);
for (int rindex = 0; rindex < image->rows; rindex++)
{
    for (int cindex = 0; cindex < image->columns; cindex++)
    {
        if (index == 1)
        {
            to << setw(3) << (long)(image->data[index(image, cindex, rindex)]);
            index++;
            total++;
        }
        else
        if (index <= format)
        {
            to << setw(4) << (long)(image->data[index(image, cindex, rindex)]);
            index++;
            total++;
        }
        if (rindex == image->rows - 1) && (cindex == image->columns - 1)
        to << '\n';
        else
        if (index == format + 1)
        {
            to << '\n';
            index = 1;
        }
    }
    return;
}

//creates a 3x3 shift matrix
float* makeshift(float xshift[2])
{
    float* S = new float[9];
    S[0] = 1;
    S[1] = 0;
    S[3] = 0;
    S[4] = 1;
    S[6] = 0;
    S[7] = 0;
    S[8] = 1;
    S[2] = xshift[0];
    S[5] = xshift[1];
return S;
}

//creates a rotation matrix that rotates the image clockwise by angle radians
float* makeRot(float angle)
{
    float R = new float[9];
    R[0] = (float)cos((double)angle);
    R[1] = (float)-sin((double)angle);
    R[2] = 0.0;
    R[3] = (float)sin((double)angle);
    R[4] = (float)cos((double)angle);
    R[5] = 0.0;
    R[6] = 0.0;
    R[7] = 0.0;
    R[8] = 1.0;

    return R;
}

//multiplies a 3x3 matrix by matrix, a*b
float* mulmatmat(float* a, float* b)
{
    float* temp = new float[9];
    temp[0] = a[0]*b[0]+a[1]*b[3]+a[2]*b[6];

    return temp;
}

//multiplies a 3x3 matrix and a vector b, a*b
float* mulmatvec(float* a, float* b)
{
    float* temp = new float[3];
    temp[0] = a[0]*b[0]+a[1]*b[1]+a[2]*b[2];

    return temp;
}

//finds the inverse of a 3x3 matrix
float* invmtx(float* T)
{
    float* I = new float[9];

    for (int i = 0; i < 9; i++)
        I[i] /= tmp;

    return I;
}
return 1;
}

//replaces the minimum value in the vector with the newval if it is bigger
int find_min(float* vec, float newval, int length)
{
    float current = vec[0];
    float tempmin = current;
    int tempindex = 0;

    // finds the smallest element in the vector
    for (int index = 1; index < length; index++)
    {
        current = vec[index];
        if (current < tempmin)
        {
            tempmin = current;
            tempindex = index;
        }
    }
    if (newval < tempmin)
        return tempindex;
}

// takes a pt desired and the transform matrix and returns
// the corresponding point in the transformed image
float pptrans(float* in, pgm* input, float* T)
{
    float* out;
    float value;

    out = mulmatvec(in, T);

    int x0 = (int) (floor((double) out[0]));
    int x1 = (int) (ceil((double) out[0]));
    if (x0 == x1) x1++;

    int y0 = (int) (floor((double) out[1]));
    int y1 = (int) (ceil((double) out[1]));
    if (y0 == y1) y1++;

    float xscale = out[0] - (float) x0;
    float yscale = out[1] - (float) y0;

    if (x0 < 0 || x1 >= input->columns ||
        y0 < 0 || y1 >= input->rows)
        value = 202.0; // set to background value to min bias
    else
    {
        float dx0y0 = input->data[index(input, x0, y0)];
        float dx0y1 = input->data[index(input, x0, y1)];
        float dx1y0 = input->data[index(input, x1, y0)];
        float dx1y1 = input->data[index(input, x1, y1)];

        float tmp1 = xscale * (dx1y0 - dx0y0) + dx0y0;
        float tmp2 = xscale * (dx1y1 - dx0y1) + dx0y1;
        float lev = yscale * (tmp2 - tmp1) + tmp1;

        value = lev;
    }
    delete [] out;

    return value;
}

// return the value of the pixel at the position desired after reframing
float imapos(int x, int y, pgm* image, float* M)
{
    float val;
float* in = new float[3];

in[0] = x;
in[1] = y;
in[2] = 1.0;

val = pptrans(in, image, M);

delete [] in;

return val;
}

//transforms image with transformation matrix l and interpolates data
void transform(pgm* image, pgm* output, float* l)
{
float* in = new float[3]; //current data point to trans/interp
float* out; //vector containing new coordinates of current data

for (int i=0; i<output->columns; i++){
    for (int j=0; j<output->rows; j++){

        //create data point vector
        in[0] = (float) j;
in[1] = (float) j;
in[2] = 1.0;

        //transform data point by matrix l
        out = mulmatvec(in, l);

        //round x down and up to nearest integer values
        int x0 = (int) (floor((double) out[0]));
in[0] = (int) (ceil((double) out[0]));
if (x0 == x1) x1++;

        //round y down and up to nearest integer values
        int y0 = (int) (floor((double) out[1]));
in[1] = (int) (ceil((double) out[1]));
if (y0 == y1) y1++;

        //difference between actual and down rounded values
        float xscale = out[0] - (float) x0;
in[0] = out[1] - (float) y0;

        //in the case that the point has been moved off the image, make the data equal to zero (back)
        if (x0 < 0 || x1 >= output->columns ||
y0 < 0 || y1 >= output->rows)
        {
            output->data[index(output, i, j)] = 202.0; //set to background to min bias
        }

        //otherwise interpolate the data
        else {

            //find the four data values that surround the new coordinate
            float dx0y0 = image->data[index(output, x0, y0)];
in[0] = image->data[index(output, x0, y1)];
in[1] = image->data[index(output, x1, y0)];
in[1] = image->data[index(output, x1, y1)];

            //scale the new data point linearly with adjacent points
            //two x points are scaled
            float tmp1 = xscale * (dx1y0 - dx0y0) + dx0y0;
in[0] = xscale * (dx1y1 - dx0y1) + dx0y1;

            //scale y
            float lev = yscale * (tmp2 - tmp1) + tmp1;
output->data[index(output, i, j)] = lev;
        }

delete [] out;
float get_points(pgm * image, int imgx, int imgy, long n, long seed, float* E) {
    const long summations = 1; //number of summations to perform
    double pos; //positive point data value
    double neg; //negative point data value
    long posrow, negrow; //row points for summation of image
    long poscol, negcol; //col points for summation of image
    const int imgx = imgx; //size of window, x
    const int imgy = imgy; //size of window, y
    float sum; //decode sums for image
    int col = image->columns;

    for (long sum_index = 1; sum_index < (summations+1); sum_index++)
    {
        for (long pt_index = 0; pt_index < n; pt_index++)
        {
            //obtain pseudorandom value
            float temp = ran0(&seed);

            //assign points to check
            poscol = (long)(temp*imgx);
            temp = ran0(&seed);
            posrow = (long)(temp*imgy);

            negcol = (long)(temp*imgx);
            temp = ran0(&seed);
            negrow = (long)(temp*imgy);

            //obtain data values given the transformation E of image
            pos = imgpos(poscol, posrow, image, E);
            neg = imgpos(negcol, negrow, image, E);

            sum = sum+(pos - neg);
        }
    }
    return sum;
}

void specialinit(pgm* output, float x, float y)
{
    output->columns = (long)x;
    output->rows = (long)y;
    return;
}

void truncate(pgm* output2, pgm* output3)
{
    int col3 = output3->columns;
    int col2 = output2->columns;

    for (int i = 0; i < output3->rows; i++)
    {
        for (int j = 0; j < output3->columns; j++)
        {
            *(output3->data)+i*col3+j) = *(output2->data)+i*col2+j);
        }
    }
    return;
void showme(pgm* image, float shiftx, float shifty, float angle, float* center, char* outfile) {
    float* G = new float[9]; //random rotation matrix
    float* P = new float[9]; //shift to center matrix
    float* S = new float[9]; //shift to 0,0 matrix
    float* H = new float[9]; //composite matrix 1
    float* J = new float[9]; //random shift matrix
    float* C = new float[9]; //composite matrix 2
    float* W = new float[9]; //composite matrix 3
    int format = 14; //points per line in data file (formatting)

    pgm output;
    initialize(image, &output);

    //shift to 0,0 for rotation
    float* zerozero = new float[2];
    zerozero[0] = -center[0];
    zerozero[1] = -center[1];

    S = makeshift(zerozero);
    G = makerot(angle);
    H = mulmatmat(S,G);

    P = makeshift(center);
    C = mulmatmat(H,P);

    float shift[2];
    shift[0] = shiftx;
    shift[1] = shifty;
    J = makeshift(shift);
    W = mulmatmat(C,J);

    transform(image, &output, W);

    cerr << "writing image\n";
    write_image(outfile, &output, format);

    delete [] G;
    delete [] P;
    delete [] S;
    delete [] H;
    delete [] J;
    delete [] C;
    delete [] W;
    delete [] zerozero;

    return;
}

//returns a transformation matrix with desired movements
float* give_matrix(float* center, float angle, float xshift, float yshift) {
    float* R = new float[9]; //random rotation matrix
    float* B = new float[9]; //shift to center matrix
    float* S = new float[9]; //shift to 0,0 matrix
    float* M = new float[9]; //composite matrix 1
    float* D = new float[9]; //random shift matrix
    float* C = new float[9]; //composite matrix 2
    float* E = new float[9]; //composite matrix 3

    float shift[2];
    shift[0] = xshift;
    shift[1] = yshift;

    //shift to 0,0 for rotation
    float* zerozero = new float[2];
    zerozero[0] = -center[0];
    zerozero[1] = -center[1];
zerozero[1] = -center[1];
S = makeshift(zerozero);
R = makerot(angle);
M = mulmatmat(S,R);
B = makeshift(center);
C = mulmatmat(M,B);
D = makeshift(shift);
E = mulmatmat(C,D);
delete [] R;
delete [] B;
delete [] S;
delete [] M;
delete [] D;
delete [] C;
delete [] zerozero;
return E;
}

// regression function 1 - Minimized Absolute Deviation
// found in Numerical Recipes, pg 703
// include "nrutil.h"
#define SIGN(a, b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
#define NR_END 1
#define FREE_ARG char

// Numerical Recipes standard error handler
void perror(char error_text[])
{
    fprintf(stderr,"Numerical Recipes run time error...\n");
    fprintf(stderr,"%s\n",error_text);
    fprintf(stderr,"...now exiting to system...\n");
    exit(1);
}

// allocate a float vector with subscript range v[nl..nh]
float *vector(long nl, long nh)
{
    float *v;
    v=(float *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(float)));
    if (lv) perror("allocation failure in vector()");
    return v-nl+NR_END;
}

// free a float vector allocated with vector()
void free_vector(float *v, long nl, long nh)
{
    free(FREE_ARG) (v-nl-NR_END));
}

int ndatat;
float *xt,*yt,aa,abdevt;

// fits data to y=a+bx, where ndata is amount of data and abdev is the abs. deviation in y
void medfit(float x[], float y[], int ndata, float *a, float *b, float *abdev)
{
    float rofunc(float b);
    int j;
    float bb,b1,b2,del,f,f1,f2,sigb,temp;
    float sx=0.0,sy=0.0,sxy=0.0,sxx=0.0,sxq=0.0;
    ndatat=ndata;
    xt=x;
    yt=y;

91
for (j=1;j<=ndata;j++) {
    sx += x[j];
    sy += y[j];
    sxy += x[j]*y[j];
    sxx += x[j]*x[j];
}

del=ndata*sxx-sx*sx;
aa=(sxx-sy*sxx*sxy)/del;
bb=(ndata*sxy-sx*sy)/del;
for (j=1;j<=ndata;j++)
    chisq += (temp=x[j]-((aa+bb*x[j]),temp*temp));
sigb=sqrt(chisq/del);
b1=bb;
f1=rofunc(b1);
b2=bb+SIGN(3.0*sigb,f1);
f2=rofunc(b2);
while (f1*f2 > 0.0) {
    bb=2.0*b2-b1;
    b1=b2;
    f1=f2;
    b2=bb;
    f2=rofunc(b2);
}

*sigb=0.01*sigb;
while (fabs(b2-b1) < sigb) {
    bb=0.5*(b1+b2);
    if (bb == b1 || bb == b2) break;
    f=rofunc(bb);
    if (f*f1 > 0.0) {
        f1=f;
        b1=bb;
    } else {
        f2=f;
        b2=bb;
    }
}
*a=aa;
*b=bb;
*abdev=abdev/ndata;
}

#define EPS 1.0e-7

extern int ndatat;
extern float *xt,*yt,aa,abdevt;

#define SWAP(a,b) temp=(a);(a)=(b);(b)=temp;

//select function needed in rofunc
float select(unsigned long k, unsigned long n, float arr[])
{
    unsigned long i,r,j,1,mid;
    float a,temp;

    l=1;
    ir=n;
    for (;j;i++)
    {
        if (ir <= l+1) {
            if (ir == l+1 & arr[ir] < arr[l]) {
                SWAP(arr[l],arr[ir])
            }
            return arr[k];
        } else {
            mid=(l+ir) >> 1;
            SWAP(arr[mid],arr[l+1])
            if (arr[l+1] > arr[ir]) {
                SWAP(arr[l+1],arr[ir])
            }
            if (arr[l] > arr[ir]) {
                SWAP(arr[l],arr[ir])
            }
        }
    }
}
if (arr[i+1] > arr[i]) {
    SWAP(arr[i+1],arr[i])
}
i=i+1;
j=i;
a=arr[i];
for (; ; ) {
    do i++ while (arr[i] < a);
    do j-- while (arr[j] > a);
    if (j < i) break;
    SWAP(arr[i],arr[j])
}
arr[i]=arr[j];
arr[j]=a;
if (j >= k) ir=j-1;
if (j <= k) i=1;
}
}
#endif

//regression function 2
float rofunc(float b)
{
    float select(unsigned long k, unsigned long n, float arr[]);
    int j;
    float *arr,d,sum=0.0;

    arr=vector(1,nmdatat);
    for (j=1;j<=ndatat;++) arr[j]=yf[j]-b*x(j);
    if (ndatat & 1) {
        aa=select((ndatat+1)>>1,ndatat,arr);
    } else {
        j=ndatat >> 1;
        aa=0.5*(select(j,ndatat,arr)+select(j+1,ndatat,arr));
    }
    abdev=0.0;
    for (j=1;j<=ndatat;++) {
        d=yf[j]-b*x(j)+aa;
        abdev += fabs(d);
        if (fabs(d) == 0.0) d /= fabs(yf[j]);
        if (fabs(d) > EPS) sum += (d >= 0.0 ? x[j] : -x[j]);
    }
    free_vector(arr,1,ndatat);
    return sum;
}
#endif

//regress values in each +/-x direction to find direction to shift
float regressx(pgm* image, int patchnum, int imgxdim, int imgydim, float shiftx, float shifty, float angle, float* center)
{
    int regresslim = 21; //use 15 points in pos and neg direction to regress
    float* sumcache = new float[regresslim]; //record of all sums regressed
    float* T = new float[9]; //transformation matrix
    float slope; //slope from the regression
    float intercept; //intercept from the regression
    float dev; //mean absolute deviation from the regression
    float sum = 0; //sum for iteration
    float* position = new float[regresslim]; //create the y axis for regression

    //get sums
    for (int index = -(regresslim-1)/2; index <= (regresslim-1)/2; index++)
    {
        T = give_matrix(center, angle, shift+index, shifty);
        sumcache[index+(regresslim-1)/2] = get_points(image, imgxdim, imgydim, patchnum, 0, T); position[index+(regresslim-1)/2] = index;
    }
//get slope of the regressed line
medfit(position, sumcache, regresslim, &intercept, &slope, &dev);

delete [] T;
delete [] sumcache;
delete [] position;

return slope;
}

//regress values in each +/-y direction to find direction to shift
float regressy(pgm* image, int patchnum, int imgxdim, int imgydim, float shiftx, float shifty, float angle, float* center)
{
    int regresslim = 21; //use 15 points in pos and neg direction to regress
    float* sumcache = new float[regresslim]; //record of all sums regressed
    float* T = new float[9]; //transformation matrix
    float slope; //slope from the regression
    float intercept; //intercept from the regression
    float dev; //mean absolute deviation from the regression
    float sum = 0; //sum for iteration
    float* position = new float[regresslim]; //create the y axis for regression

    //get sums
    for (int index = -(regresslim-1)/2; index <= (regresslim-1)/2; index++)
    {
        T = give_matrix(center, angle, shiftx, shifty+index);
        sumcache[index+(regresslim-1)/2] = get_points(image, imgxdim, imgydim, patchnum, 0, T);
        position[index+(regresslim-1)/2] = index;
    }

    //get slope of the regressed line
    medfit(position, sumcache, regresslim, &intercept, &slope, &dev);

delete [] T;
delete [] sumcache;
delete [] position;

return slope;
}

//regress values in each +/- rotation to find direction to shift
float regressa(pgm* image, int patchnum, int imgxdim, int imgydim, float shiftx, float shifty, float angle, float* center)
{
    int regresslim = 21; //use 15 points in pos and neg direction to regress
    float ang_inc = pi/360; //increment angle at .5 degrees
    float* sumcache = new float[regresslim]; //record of all sums regressed
    float* T = new float[9]; //transformation matrix
    float slope; //slope from the regression
    float intercept; //intercept from the regression
    float dev; //mean absolute deviation from the regression
    float sum = 0; //sum for iteration
    float* position = new float[regresslim]; //create the y axis for regression

    //get sums
    for (int index = -(regresslim-1)/2; index <= (regresslim-1)/2; index++)
    {
        T = give_matrix(center, angle+index*ang_inc, shiftx, shifty);
        sumcache[index+(regresslim-1)/2] = get_points(image, imgxdim, imgydim, patchnum, 0, T);
        position[index+(regresslim-1)/2] = index;
    }

    //get slope of the regressed line
    medfit(position, sumcache, regresslim, &intercept, &slope, &dev);

delete [] T;
delete [] sumcache;
delete [] position;

return slope;
//get index of maximum element magnitude
int getmax(float* vec, int length)
{
  float temp = vec[0];
  int tempindex = 0;

  for (int index = 1; index < length; index++)
  {
    if (fabs(vec[index]) > fabs(temp))
    {
      temp = vec[index];
      tempindex = index;
    }
  }

  //move in x:1, move in y:2, move in angle:3
  tempindex++;

  //if the slope is negative, want to move in negative direction
  if (temp < 0)
    tempindex = -tempindex;

  return tempindex;
}

//add a slope to the record
void add_slope(float* slopes, float value)
{
  slopes[0] = slopes[1];
  slopes[1] = slopes[2];
  slopes[2] = slopes[3];
  slopes[3] = value;

  return;
}

//refine the gradient search aligned image
void refine(pgm* image, int patchnum, int imgxdim, int imgydim, float* shiftxrec, float* shiftyrec, float* anglerc, float* sumrec, float* center, char* outile)
{
  int recmem = 4; //number of moves to keep in memory to compare to const float slopethresh = 50; //the threshold slope to stop at const float angle_inc = pi/360; //max angle to rotate each iteration const float xshift_inc = 2; //2 * (scanxdim * imgydim);
  const float yshift_inc = 2; //2 * (scanydim - imgydim);
  float* slopes = new float[3]; //slopes from regression: x, y, angle
  int whichdir; //which direction to move
  float* lastx = new float[recmem]; //the last x slopes
  float* lasty = new float[recmem]; //the last y slopes
  float* lastslope = new float[recmem]; //the last slopes
  int* done = new int[3]; //when all ones, indicates that finished refinement
  float* finals = new float[refnum]; //the record of all final sums
  float* T = new float[9]; //transformation matrix used in final summing
  int maxsumindex; //the index in finals that has the max sum
  pgm* refined = new pgm[9]; //the final refined image

  //perform refinement for all of the top sums from coarse gradient
  for (int index = 0; index < refnum; index++)
  {
    cerr << "starting refinement on number " << index << " with";

    //reset flags
    done[0] = 0;
    done[1] = 0;
    done[2] = 0;

    //reset move records
    for (int i = 0; i < recmem; i++)
      {
lastx_slope[i] = 0;
lasty_slope[i] = 0;
last_slope[i] = 0;
}

//continue until all three parameters are maxed
while (done[0] != 1 || done[1] != 1 || done[2] != 1) {
    //regress for all three parameters and get slopes after stepping
    //then add the slopes to the records
    slopes[0] = regressx(image, patchnum, imgxdim, imgydim, shiftxrec[index], shiftyrec[index], anglexrec[index],
    center);
    slopes[1] = regressy(image, patchnum, imgxdim, imgydim, shiftxrec[index], shiftyrec[index], anglexrec[index],
    center);
    slopes[2] = regressa(image, patchnum, imgxdim, imgydim, shiftxrec[index], shiftyrec[index], anglexrec[index],
    center);

    // if all slopes under a threshold then stop
    if ((fabs(slopes[0]) < slopethresh) && (fabs(slopes[2]) < slopethresh) && (fabs(slopes[3]) < slopethresh)) {
        done[0] = 1;
        done[1] = 1;
        done[2] = 1;
        cerr << "all slopes under slopethresh\n";
    } else {
        whichdir = getmax(slopes, 3);

        switch(abs(whichdir)) {
            case 1:
                //for x shift, check to see if we are looping
                if ((whichdir > 0) && (slopes[0] != lastx_slope[0]) && (slopes[0] != lastx_slope[2])) {
                    shiftxrec[index] = shiftxrec[index] + xshift;  // xshift
                    addslope(lastx_slope, slopes[0]);
                    cerr << "moving: " << whichdir << "\n";
                } else {
                    if ((whichdir < 0) && (slopes[0] != lastx_slope[0]) && (slopes[0] != lastx_slope[2])) {
                        shiftxrec[index] = shiftxrec[index] - xshift;  // xshift
                        addslope(lastx_slope, slopes[0]);
                        cerr << "moving: " << whichdir << "\n";
                    } else {
                        done[0] = 1;  // refinement done due to looping
                    }
                }
                break;
            case 2:
                //for y shift, check to see if we are looping
                if ((whichdir > 0) && (slopes[1] != lasty_slope[0]) && (slopes[1] != lasty_slope[2])) {
                    shiftyrec[index] = shiftyrec[index] + yshift;  // yshift
                    addslope(lasty_slope, slopes[1]);
                    cerr << "moving: " << whichdir << "\n";
                } else {
                    if ((whichdir < 0) && (slopes[1] != lasty_slope[0]) && (slopes[1] != lasty_slope[2])) {
                        shiftyrec[index] = shiftyrec[index] - yshift;  // yshift
                        cerr << "moving: " << whichdir << "\n";
                    } else {
                    }
                }
            }
        }
    }

addslope(lastyslope, slopes[1]);
carr << "moving: " << whichdir << "n";
} else 
done[0] = 1; //refinement for y done
}
break;
case 3:
//for rotation, check to see if we are looping
{
anglicrc[index] = anglicrc[index] + angle_inc;
addslope(lastaslope, slopes[2]);
carr << "moving: " << whichdir << "n";
} else 
{
anglicrc[index] = anglicrc[index] + angle_inc;
addslope(lastaslope, slopes[2]);
carr << "moving: " << whichdir << "n";
} else 
done[0] = 1; //refinement for angle done
break;
} //end else 
} //end while

carr << " refinement number " << index << " finishedn";
}

carr << "nthe refined sums are:n";
for (int index = 0; index < refnum; index++)
{
carr << " computing number " << index << "n";
T = give_matrix(center, anglicrc[index], shiftrec[index], shiftyrec[index]);
finalsums[index] = get_points(image, imgxdim, imgydim, patchnum, 0, T);  //final sum of finalsums[index]
carr << "n";

maxsumindex = getmax(finalsums, refnum) - 1; //because of the +1 in getmax

carr << "number " << maxsumindex << " has the largest sumn";
T = give_matrix(center, anglicrc[maxsumindex], shiftrec[maxsumindex], shiftyrec[maxsumindex]);

//truncat and write the refined image to outfile

carr << " creating final image in " << outfile << "n";
initialize(image, &refined);
transform(image, &refined, T);

pgm truncref;
truncref.columns = imgxdim;
truncref.rows = imgydim;
truncref.maxdepth = image->maxdepth;
truncref.filename = "refined.pgm";

long size = (truncref.columns) * (truncref.rows);
truncref.data = new double[size];

trunc(&refined, &truncref);
write_image(outfile, &truncref, 14);

//for debugging, writes out the "refined" version of one of the other orientations
maxsumindex = 2;
cerr << "number 2 is a random sum\n";
T = give_matrix(center, anglerec[maxsumindex], shiftxrec[maxsumindex], shiftyrec[maxsumindex]);

//for debugging only
cerr << "creating random image in random.pgm\n";
transform_image(&refined, T);
write_image("random.pgm", &refined, 14);
delete [] slopes;
delete [] lastx_slope;
delete [] lasty_slope;
delete [] lastx_slope;
delete [] done;
delete [] T;
delete [] finalsums;
return;
}

//sets all members of vector to zero
float* zerovec(float* vec, int length)
{
    for (int index = 0; index < length; index++)
    {
        vec[index] = 0;
    }
    return vec;
}

//coarse gradient search loop
void gradloop(pgm* original, int gradnum, int patchnum, int imgxdim, int imgydim, int scanxdim, int scanydim, float anglelim, char* datfile, char* outfile)
{
    const int seedmax = 1; //number of seeds to search
    long oseed = 1; //start seed for random orientation generation
    const float angle_range = pi/4; //anglelim; //max angle to rotate
    const float xshift_range = 200;//2 * (scanxdim - imgxdim);
    const float yshift_range = 200;//2 * (scanydim - imgydim);
    float* R = new float[9]; //random rotation matrix
    float* B = new float[9]; //shift to center matrix
    float* S = new float[9]; //shift to 0,0 matrix
    float* M = new float[9]; //composite matrix 1
    float* D = new float[9]; //random shift matrix
    float* C = new float[9]; //composite matrix 2
    float* E = new float[9]; //composite matrix 3
    float* anglerec = new float[refnum]; //record of angles used
    float* shiftxrec = new float[refnum]; //record of x shifts used
    float* shiftyrec = new float[refnum]; //record of y shifts used
    float* sumrec = new float[refnum]; //number of sums recorded for refinement
    int repl; //the index value to replace in sumrec
    float shift[2]; //x,y shifts used in gradient search
    float temp2;
    float angle;
    float sum;
    float center[2]; //center of the scanbed x,y
center[0] = (original->columns)/2;
center[1] = (original->rows)/2;

    //shift to 0,0 for rotation
    float* zerozero = new float[2];
    zerozero[0] = -center[0];
    zerozero[1] = -center[1];
    S = makeshift(zerozero);

    float highnum = 0; //the highest sum found through gradient search
    int highindex = 0; //the index for this sum in the records

    //zero the sums record
    sumrec = zerovec(sumrec, refnum);
for (int index = 0; index < gradnum; index++)
{
    cerr << "starting " << index << ": ";

    // make random rotation matrix
    temp2 = 0.5 - ran1(&seed); // in order to obtain negative values
    angle = temp2 * angle_range;
    R = makerot(angle);

    // shift back to original location
    B = makeshift(center);

    // shift random amount
    temp2 = 0.5 - ran1(&seed); // in order to obtain negative values
    shift[0] = temp2 * xshift_range;
    temp2 = 0.5 - ran1(&seed);
    shift[1] = temp2 * yshift_range;
    D = makeshift(shift);

    // create composite matrix
    M = mulmatmat(S,R);
    C = mulmatmat(M,B);
    E = mulmatmat(C,D);

    // get the sum for this orientation
    sum = get_points(original, imgxdim, imgydim, patchnum, 0, E);

    // save sum in records if it is greater the minimum member
    repl = find_min(sumrec, sum, refnum);

    if (repl != -1)
    {
        sumrec[repl] = sum;
        anglerec[repl] = angle;
        shiftxrec[repl] = shift[0];
        shiftyrec[repl] = shift[1];
    }

    cerr << sum << " at index " << index << ", angle " << angle << " and shift " << shift[0] << " " << shift[1] << "n";
}

for (int index = 0; index < refnum; index++)
cerr << sumrec[index] << " " << shiftxrec[index] << " " << shiftyrec[index] << " " << anglerec[index] << "n";

cerr << "writing results to file";
ofstream to(datfile);
if (!to) error("cannot open output file", datfile);
for (int i = 0; i < refnum; i++)
to << sumrec[i];

cerr << "starting refinementn";
refine(original, patchnum, imgxdim, imgydim, shiftxrec, shiftyrec, anglerec, sumrec, center, outfile);

// clean up
delete [] S;
delete [] R;
delete [] B;
delete [] M;
delete [] D;
delete [] E;
delete [] C;
delete [] sumrec;
delete [] anglerec;
delete [] shiftxrec;
delete [] shiftyrec;
delete [] zerozero;
return;
/the main procedure, duh
int main(int argc, char* argv[]) 
{
    pgm image;
    int gradnum = 10000; //number of tries in gradient search
    
    if (argc != 10)
        error("grad input-file output-image img-xdim img-ydim scan-xdim scan-ydim angle_range data-output number-of-patches");
    
    int patches = atoi(argv[9]);
    int imgxdim = atoi(argv[3]);
    int imgydim = atoi(argv[4]);
    int scanxdim = atoi(argv[5]);
    int scanxdim = atoi(argv[6]);
    float angle = atof(argv[7]);
    
    cerr << "Initializing
";
    get_pgm(argv[1], &image);
    
    cerr << "Starting Gradient Search\n";
    gradloop(&image, gradnum, patches, imgxdim, imgydim, scanxdim, scanxdim, angle, argv[6], argv[2]);
    
    cerr << "nGradient Search Complete\n";
    return 0;
}
Appendix E: mbencode.c

//C++ libraries/

#include <iostream.h>
#include <iomanip.h> // for setw(/)
#include <fstream.h> // for file output /
#include <stdlib.h>
#include <time.h> // for rand(/) 
#include <math.h>

#define IA 16807
#define IM 2147483647
#define AM (1.0f/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

//pseudo-random number generator for patch position
float ran0(long *idum)
{
    long k;
    float ans;

    *idum ^= MASK;
    k = (*idum)/IQ;
    *idum = IA*pow(idum-k*IQ)-IR*k;
    if (*idum<0) *idum += IM;
    ans = AM*pow(idum);
    *idum ^= MASK;
    return ans;
}

//pseudo-random number generator for depths
float ran1(long *idum)
{
    long k;
    float ans;

    *idum ^= MASK;
    k = (*idum)/IQ;
    *idum = IA*pow(idum-k*IQ)-IR*k;
    if (*idum<0) *idum += IM;
    ans = AM*pow(idum);
    *idum ^= MASK;
    return ans;
}

//image data structure
struct pgm
{
    char* filename;
    long columns;
    long rows;
    double maxdepth;
    double* data;
};

//error message function
void error(char* s, char* s2 = "")
{
    cerr << s << s2 << "\n";
    exit(1);
}

//count the number of header lines
int clen(char* filename)
{
    int count = 0;
}
char ch;

ifstream from(filename);

do{
    from.get(ch);
} while (ch != 'n');

from.get(ch);

while (ch == '#')
{
    count++;
    do{
        from.get(ch);
    } while (ch != 'n');
    from.get(ch);
}
return count;

//get formatting of original physical file
int getlength(char* filename)
{
    char ch = '0';
    int length = 1;
    int first;
    int hlines;
    int temp;

    ifstream from(filename);

    //get to the image content, skip header lines
    hlines = clines(filename);

    for (int index = 0; index <= hlines; index++)
    {
        do{
            from.get(ch);
        } while (ch != 'n');
    }

    for (int index = 0; index <= 2; index++)
        from >> temp;

    //move to first line of image data
    from.get(ch);
    ch = '0';

    //get number of chars in first line
    while (ch != 'n')
    {
        from.get(ch);
        length++;
    }
    length = (int)(length-3)/4 + 1;
    return length;
}

//load picture data into structure
int get_image(char* filename, pgm* image)
{
    int length;
    long temp1;
    double temp2;
    int number;
    double size;
    char ch;

    ifstream from(filename);
if (errno) error("cannot open input file ", filename);

number = clines(filename);

for (int index = 0; index <= number; index++)
{
    do{
        from.get(ch);
    } while (ch != 'n');
}

from >> temp1;
image->columns = temp1;
from >> temp1;
image->rows = temp1;
from >> temp1;
image->maxdepth = (double)temp1;

size = (image->columns) * (image->rows);
image->data = new double[size];

for (int index = 0; index < size; index++)
{
    from >> temp1;
    temp2 = (double)temp1;
    *((image->data)+index) = temp2;
}

length = getlength(filename);
return length;

//send image to file
void write_image(char* filename, pgm* image, int format)
{
    int pixel = 0;
    int index = 1;
    long total = 0;
    ofstream to(filename);
    if (fio) error("cannot open output file ", filename);

to << "P2\n" patch.pgm\n";
to << image->columns << * image->rows << \n;
to << image->maxdepth << \n;

double size = (image->columns) * (image->rows);

for (int rindex = 0; rindex < image->rows; rindex++)
{
    for (int cindex = 0; cindex < image->columns; cindex++)
    {
        if (index == 1)
        {
            to << setw(3) << (long)((image->data)+cindex+(rindex*(image->
            columns))));
            index++;
            total++;
        }
        else
            if (index <= format)
            {
                to << setw(4) << (long)((image->data)+cindex+(rindex*(image->
                columns))));
                index++;
                total++;
            }
        if ((rindex == image->rows - 1) && (cindex == image->columns - 1))
            to << '\n';
        else
            if (index == format + 1)
            {
            }
to << 'n';
   index = 1;
 }
}
}
return;
}

//initialize the target data
void initialize(pgm* simage, pgm* timage) {
   timage->columns = simage->columns;
   timage->rows = simage->rows;
   timage->maxdepth = simage->maxdepth;
   timage->filename = "# masked.pgm";
   long size = (simage->columns) * (simage->rows);
   timage->data = new double[size];
   return;
}

//subtract, point by point, blurred image from original
void point_sub(pgm* simage, pgm* timage) {
   long size = (simage->columns) * (simage->rows);
   for (int index = 0; index < size; index++)
      *((timage->data)+index) = *((simage->data)+index) - *((timage->data)+index);
   return;
}

//take abs of point_sub output image
void abs_image(pgm* timage) {
   double current_max = 0;
   long size = timage->columns * timage->rows;
   for (int index = 0; index < size; index++)
   {
      if (*((timage->data)+index) < 0)
         *((timage->data)+index) = -*((timage->data)+index);
      if (*((timage->data)+index) > current_max)
         current_max = *((timage->data)+index);
   }
   timage->maxdepth = current_max;
   return;
}

//for diagnostics only -- scales the image back up to viewable
//scales, eg. 255
void rescale_image(pgm* timage, double scale) {
   double temp = 0;
   double current_max = 0; //record the new timage->maxdepth;
   double maxdepth = 0;
   long size = timage->columns * timage->rows;

   //find max
   for (int index = 0; index < size; index++)
   {
      if (*((timage->data)+index) > temp)
         temp = *((timage->data)+index);
   }
   maxdepth = temp;
   for (int index = 0; index < size; index++)
   {
temp = *((timage->data)+index) / maxdepth;
*((timage->data)+index) = temp * scale;

if *((timage->data)+index) > current_max
    current_max = *((timage->data)+index);
}
timage->maxdepth = current_max;
return;

//blurs the image with a square of size
void blur_image(pgm* simage, pgm* timage, int size)
{
double new_depth;
double pt_total;
long yinit, xinit;
long currenty, currentx;
int max_dist = (size - 1)/2;

long max_index = simage->columns * simage->rows;
for (int yindex = 0; yindex < simage->rows; yindex++)
    for (int xindex = 0; xindex < simage->columns; xindex++)
    {
yinit = yindex * max_dist;
xinit = xindex * max_dist;

    pt_total = 0;
    for (int syindex = 0; syindex < size; syindex++)
    {
        currenty = yinit + syindex;

        if (currenty < 0)
            currenty = -currenty;

        if (currenty > simage->rows)
            currenty = 2*simage->rows - currenty;

        if (currenty == simage->rows)
            currenty = currenty - 1;

        for (int sxindex = 0; sxindex < size; sxindex++)
        {
            currentx = xinit + sxindex;

            if (currentx < 0)
                currentx = -currentx;

            if (currentx > simage->columns)
                currentx = 2*simage->columns - currentx;

            if (currentx == (simage->columns))
                currentx = currentx - 1;

            pt_total += *((simage->data)+currentx+(currenty*(simage->columns)));
        }
    }
new_depth = pt_total/(size*size);
*((timage->data)+xindex+(yindex*(timage->columns)))= new_depth;
}
return;

//create square patches in data
void squarein(pgm* image, pgm* mask, int n, long w, long h, long d, char md, long seed)
{
    const long patches = n;
long seed1 = seed; //for random patch location
long seed2 = 20; //for random values inside patch
double add_depth;
long wth1, wth2; //1 is positive patch, 2 is negative patch/
long hgt1, hgt2;
long x, y;
long random;
double new_depth1 = 127;
double new_depth2 = 127;
long corners[4]; //x1, y1, x2, y2/

for (int index = 0; index < patches; index++)
{
    wth1 = wth2 = w;
    hgt1 = hgt2 = h;

    corners[0] = (long)(ran0(&seed1)*(image->columns));
    corners[1] = (long)(ran0(&seed1)*(image->rows));
    corners[2] = (long)(ran0(&seed1)*(image->columns));
    corners[3] = (long)(ran0(&seed1)*(image->rows));

    //deal with image edges -- adjust dimensions of square to fit inside image
    if (((w > 1) && (h > 1))
    {
        if (((image->columns - corners[0]) < (wth1/2))
            wth1 = 2 * (image->columns - corners[0]);
        else
            if (corners[0] < (wth1/2))
                wth1 = 2 * corners[0];

        if (((image->rows - corners[1]) < (hgt1/2))
            hgt1 = 2 * (image->rows - corners[1]);
        else
            if (corners[1] < (hgt1/2))
                hgt1 = 2 * corners[1];

        if (((image->columns - corners[2]) < (wth2/2))
            wth2 = 2 * (image->columns - corners[2]);
        else
            if (corners[2] < (wth2/2))
                wth2 = 2 * corners[2];

        if (((image->rows - corners[3]) < (hgt2/2))
            hgt2 = 2 * (image->rows - corners[3]);
        else
            if (corners[3] < (hgt2/2))
                hgt2 = 2 * corners[3];

    for (long hindex = (-hgt1/2); hindex < (hgt1/2); hindex++)
        for (long windex = (-wth1/2); windex < (wth1/2); windex++)
        {
            if (rnd == 'y')
                add_depth = (double)(ran1(&seed2)*d);
            else
                add_depth = (double)d;

            x = corners[0] + windex; //current point
            y = corners[1] + hindex; //current point

            new_depth1 = *((image->data)+y*image->columns+x) + *((mask->data)+y*mask->columns+x) *

            add_depth;

            //center is max
            if ((windex == 0) && (hindex == 0))
                new_depth1 = *((image->data)+y*image->columns+x) + add_depth;

            if (new_depth1 > 255)
                new_depth1 = 255;

            *((image->data)+y*image->columns+x) = new_depth1;
for (long hindex = (-hgt2/2); hindex < (hgt2/2); hindex++)
for (long windex = (-wth2/2); windex < (wth2/2); windex++)
{
    if (rn1 == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;
    x = corners[2] + windex; //current point
    y = corners[3] + hindex; //current point
    new_depth2 = *(image->data) + y*image->columns + x - *(image->data) + y*mask->columns + x) * 
    //center is max
    if ((windex == 0) && (hindex == 0))
        new_depth1 = *(image->data) + y*image->columns + x) + add_depth;
    if (new_depth2 < 0)
        new_depth2 = 0;
    *(image->data) + y*image->columns + x) = new_depth2;
}

else //for the case of 1x1 patches
{
    x = corners[0]; //current point
    y = corners[1]; //current point
    if (rn1 == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;
    new_depth1 = *(image->data) + y*image->columns + x) + add_depth;
    *(image->data) + y*image->columns + x) = new_depth1;
    x = corners[2]; //current point
    y = corners[3]; //current point
    if (rn1 == 'y')
        add_depth = (double)(ran1(&seed2)*d);
    else
        add_depth = (double)d;
    new_depth1 = *(image->data) + y*image->columns + x) - add_depth;
    *(image->data) + y*image->columns + x) = new_depth1;
}

return;

//create circular patches in data for data = 1
void circlein1(pgm* image, pgm* mask, long n, long r, long d, char md, long seed)
{
    const int patches = n;
    int flag;
    int centers_error;
    long seed1 = seed; //for random patch location
    long seed2 = 20; //for random values inside patch
double dr;  //for random patch location
double dx, dy;
    long rad1, rad2; //1 is positive patch, 2 is negative patch//
    long x, y;
    double rand_depth;
    double add_depth;
double new_depth = 127;
    long centers[4*patches]; //[(x1, y1, x2, y2)]//
    for (int index = 0; index < patches; index++)
    {

float increment = (float)d/(float)r;

rad1 = rad2 = r;

//generate the centers for patches
centers[index] = (long)(ran0(&seed1)*(image->columns));
centers[index+patches] = (long)(ran0(&seed1)*(image->rows));
centers[index+2*patches] = (long)(ran0(&seed1)*(image->columns));
centers[index+3*patches] = (long)(ran0(&seed1)*(image->rows));

//make positive point
for (long hindex1 = 0; hindex1 <= 2*rad1; hindex1++)
    for (long windex1 = 0; windex1 <= 2*rad1; windex1++)
        {
            dx = windex1 - rad1;
            dy = hindex1 - rad1;
            //distance from current point to center
            dist = sqrt(dx*dx+dy*dy);

            if (dist <= rad1)
                {
                    //create current point
                    x = centers[index] + windex1 - r;
                    y = centers[index+patches] + hindex1 - r;

                    //current point inside image
                    if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
                        {
                            //creates conical depth
                            add_depth = (double)(increment*(rad1 - dist));

                            if (add_depth != 0) //true inside the radius
                                {
                                    if (add_depth == d) //true at center
                                        new_depth = *((image->data)+y*image->columns+x)+d;
                                    else
                                        {
                                            //randomize depth
                                            if (rn == 'Y')
                                                rand_depth = (double)(ran1(&seed2)*add_depth);
                                            else
                                                rand_depth = add_depth;

                                            //assign new depth
                                            new_depth = *((image->data)+y*image->columns+x)+((mask->data)+y*mask->columns+x)*rand_depth;
                                        }
                                    else
                                        new_depth = *((image->data)+y*image->columns+x)+((mask->data)+y*mask->columns+x)*add_depth; //true at the radius

                                    if (new_depth > 255)
                                        new_depth = 255;

                                    *((image->data)+y*image->columns + x) = new_depth;
                                }
                            }
                        }
        }

//make negative point
for (long yindex2 = 0; yindex2 <= 2*rad2; yindex2++)
    for (long xindex2 = 0; xindex2 <= 2*rad2; xindex2++)
        {  
            dx = xindex2 - rad2;   //x distance
            dy = yindex2 - rad2;   //y distance

            dist = sqrt(dx*dx+dy*dy); //distance from current point to center
if (dist <= rad2)
{
    //create current point
    x = centers[index+2*patches] + xindex2 - rad2;
    y = centers[2+index+3*patches] + yindex2 - rad2;

    //current point inside image
    if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
    {
        //create conical depth
        add_depth = (double)increment*(rad2 - dist);

        if (add_depth != 0) //true inside the radius
        {
            if (add_depth == d) //true at the center
                new_depth = *((image->data)+y*image->columns+x)-d;
            else
                { //randomize the depth
                    if (md == y)
                        rand_depth = (double)(rand1(&seed2)*add_depth);
                    else
                        rand_depth = add_depth;

                    new_depth = *((image->data)+y*image->columns+x) - *(image->data)+y*mask-
                    >=columns+x)*rand_depth;
                }
            }
        }
    }
    //assign new depth at the radius (no change)
    else
    { new_depth = *((image->data)+y*image->columns+x) - *(mask->data)+y*mask-
        >=columns+x);
        add_depth = new_depth;

        if (new_depth < 0)
            new_depth = 0;

        *(image->data)+y*image->columns+x = new_depth;
    }
}
return;
}

//create circular patches in data for data = 0
void circlein0(pgm* image, pgm* mask, long n, long r, long d, char md, long seed)
{
    const int patches = n;
    int flag;
    int centers_error;

    long seed1 = seed; //for random patch location
    long seed2 = 20; //for random values inside patch
    double dist; //for random values inside patch
    long dx, dy;
    long rad1, rad2; //1 is positive patch, 2 is negative patch//
    long x, y;
    double rand_depth;
    double add_depth;
    double new_depth = 127;
    long centers[4*patches]; //x1, y1, x2, y2//

    for (int index = 0; index < patches; index++)
    {
        float increment = (float)(d/((float)r));
        rad1 = rad2 = r;

        //generate the centers for patches
centers[index] = (long)(ran0(&seed1) * (image->columns));
centered[+index+patches] = (long)(ran0(&seed1) * (image->rows));
centers[index+2*patches] = (long)(ran0(&seed1) * (image->columns));
centers[index+3*patches] = (long)(ran0(&seed1) * (image->rows));

//make positive point
for (long hindex1 = 0; hindex1 <= 2 * rad1; hindex1++)
    for (long windex1 = 0; windex1 <= 2 * rad1; windex1++)
    {
        dx = windex1 - rad1;
        dy = hindex1 - rad1;
        //distance from current point to center
        dist = sqrt(dx*dx + dy*dy);
        if (dist <= rad1)
        {
            //create current point
            x = centers[index] + windex1 - r;
            y = centers[+index+patches] + hindex1 - r;

            //current point inside image
            if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
            {
                //creates conical depth
                add_depth = (double)(increment*(rad1 - dist));

                if (add_depth != 0) //true inside the radius
                {
                    if (add_depth == d) //true at center
                        new_depth = "((image->data)+y*image->columns+x)-d;
                    else
                    {
                        //randomize depth
                        if (rnd == 'y')
                            rand_depth = (double)(ran1(&seed2)*add_depth);
                        else
                            rand_depth = add_depth;

                        //assign new depth
                        new_depth = "((image->data)+y*image->columns+x)-((mask->data)+y*mask->columns+x)*rand_depth;";
                    }
                }
                else
                
                    new_depth = "((image->data)+y*image->columns+x)-((mask->data)+y*mask->columns+x)*add_depth;" //true at the radius;

                    if (new_depth < 0)
                        new_depth = 0;

                        "((image->data)+y*image->columns + x) = new_depth;
                }
            }
        }
    }

//make negative point
for (long yindex2 = 0; yindex2 <= 2 * rad2; yindex2++)
    for (long xindex2 = 0; xindex2 <= 2 * rad2; xindex2++)
    {
        dx = xindex2 - rad2; //x distance
        dy = yindex2 - rad2; //y distance
        dist = sqrt(dx*dx + dy*dy); //distance from current point to center

        if (dist <= rad2)
        {
            //create current point
            x = centers[index+2*patches] + xindex2 - rad2;
            y = centers[+index+3*patches] + yindex2 - rad2;
        }
//current point inside image
if (x >= 0 && x < image->columns && y >= 0 && y < image->rows)
{
    //creates conical depth
    add_depth = (double)(increment*(rad2 - dist));

    if (add_depth != 0) //true inside the radius
    {
        if (add_depth == d) //true at the center
            new_depth = *((image->data)+y*image->columns+x)+d;
        else
            { //randomize the depth
                if (rnd == 'y')
                    rand_depth = (double)(ran1(&seed2)*add_depth);
                else
                    rand_depth = add_depth;
                //assign new depth
                new_depth = *((image->data)+y*image->columns+x) + *((mask->data)+y*mask->columns+x) +
                *columns+x) * rand_depth;
            }
        //assign new depth at the radius (no change)
        else
            new_depth = *((image->data)+y*image->columns+x) + *((mask->data)+y*mask->columns+x) *
            add_depth;
    }
    if (new_depth > 255)
        new_depth = 255;
    *((image->data)+y*image->columns + x) = new_depth;
}
}
return;

//that main procedure thingy
int main(int argc, char* argv[])
{
    int bits = 4; //number of bits to encode
    int currant = 3; //amount of encoding redundancy
    pgm image;
    pgm mask, mask2;
    int format;
    char* type = argv[3];
    char* md = argv[4];

    if (argc == 1)
        error("encode input output shape random number-strong number-data dimensions strong-depth data-depth data");
    if ("type == 'c'")
    {
        if (argc != 11) error("Wrong number of arguments"); //check arguments

        long number = atoi(argv[5]);
        long datanum = atoi(argv[6]);
        long radius = atoi(argv[7]);
        long depth = atoi(argv[8]); //of strong bit
        long datadepth = atoi(argv[9]); //of data bits
        int dataradius = 5; //radius of data patches
        char* dat = argv[10];

        cerr << "Initializing Image\n";
        format = get_image(argv[1], &image);

        //for mask
}
double scale = 1;
int size1 = 7; //blur sizes/
int size2 = 5;

initialize(&image, &mask);
initialize(&image, &mask2);

cerr << "Creating Visibility Mask\n";
blur_image(&image, &mask2, size1);
point_sub(&image, &mask2);
abs_image(&mask2);
initialize(&mask2, &mask); //mask is now the current altered version.
blur_image(&mask2, &mask, size2);
rescale_image(&mask, scale); //scale to 0-1 level

cerr << "Encoding Image\n";
cerr << "encoding strong bit\n";
circle1(&image, &mask, number, radius, depth, *rnd, 0);

//encode the data
for (int index = 0; index < bits; index++)
{
    cerr << "\n" << "encoding bit number \n" << "data is \n" << dat[index] << " \n";
    cerr << "encoding seeds: \n";
    //error correction coding
    if (dat[index] == '1')
    for (int i = 0; i < corramt; i++)
    {
        circle1(&image, &mask, number, dataradius, depth, *rnd, index*corramt+i+1);
        cerr << index*corramt+i+1 << "(+) \n";
    }
    else
    for (int i = 0; i < corramt; i++)
    {
        circle0(&image, &mask, number, dataradius, depth, *rnd, index*corramt+i+1);
        cerr << index*corramt+i+1 << "(-) \n";
    }
}
cerr << "\n" << "Writing Image\n";
write_image(argv[2], &image, format);
}

if (*type == 's')
{
    if (argc != 10) error("Wrong number of arguments"); //check arguments/

    long number = atoi(argv[5]);
    long width = atoi(argv[6]);
    long height = atoi(argv[7]);
    long depth = atoi(argv[8]);
    long seed = atoi(argv[9]);

    cerr << "Initializing image\n";
    format = get_image(argv[1], &image);

    //for mask
double scale = 1;
int size1 = 7; //blur sizes/
int size2 = 5;

initialize(&image, &mask);
initialize(&image, &mask2);

cerr << "Creating Visibility Mask\n";
blur_image(&image, &mask2, size1);
point_sub(&image, &mask2);
abs_image(&mask2);
initialize(&mask2, &mask); //mask is now the current altered version.
blur_image(&mask2, &mask, size2);
rescale_image(&mask, scale); //scale to 0-1 level

cerr << "Encoding Squaresin";
squarein(&image, &mask, number, width, height, depth, *rnd, seed);

cerr << "Writing imagein";
write_image(argv[2], &image, format);
//write_image(argv[2], &mask, format);
}
return 0;
}
Appendix F: mbdecode.c

//C++ libraries
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <time.h> //for rand()
#include <sstream>
#include <iomanip.h>
#include <stdio.h>
#include <string>
#include <sstream>

#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

//pseudorandom number generator for patch position
float ran0(long *idum)
{
    long k;
    float ans;

    *idum ^= MASK;
    k = (*idum)/IQ;
    *idum = IA*((*idum-k*IQ)-IR)*k;
    if (*idum<0) *idum += IM;
    ans = AM*(*idum);
    *idum ^= MASK;
    return ans;
}

//error message function
void error(const char* s, const char* s2 = "")
{
    cerr << s << ' ' << s2 << 'n';
    exit(1);
}

//data structure for pgm
struct pgm{
    char* filename;
    long columns;
    long rows;
    double maxdepth;
    double* data;
};

// counts the number of comment lines
int clines(const char* filename, int& img_type)
{
    int count = 0;
    char* ch = new char[1];

    ifstream from(filename);
    from.get(ch[0]);
    from.get(ch[0]);
    from.get(ch[0]);

    img_type = atoi(ch);
    do{
        from.get(ch[0]);
    } while (ch[0] != 'n');
from.get(ch[0]);

while (ch[0] == '#')
{
    count++;  
    do{
        from.get(ch[0]);
    } while (ch[0] != 'n');
    from.get(ch[0]);
}
return count;
}

// load picture data into structure
void get_pgm(const string& filename, pgm* image) {

    ifstream from(filename.c_str());
    if (!from) error("cannot open input file " , filename.c_str());

    char ch;
    long temp;
    long size;
    int img_type;

    int number = clines(filename.c_str(), img_type);

    if (img_type ==2) {
    for (int lines = 0; lines <= number; lines++)
        do{
            from.get(ch);
        } while (ch != 'n');

    from >> temp;
    image->columns = temp;
    from >> temp;
    image->rows = temp;
    from >> temp;
    image->maxdepth = (double)temp;

    cout << image->columns << "x" << image->rows << " ascii PGM file read." << endl;
    size = (image->columns) * (image->rows);
    image->data = new double[size];

    for (int index = 0; index < size; index++) {
        from >> temp;
        "((image->data)+index) = (double)temp;
    return;
    }else if (img_type ==5) {

    ifstream is(filename.c_str());
    char buffer[128];

    is.getline(buffer, 128);
    is.getline(buffer, 128);
    while(buffer[0] == '#') is.getline(buffer, 128);

    istringstream str(buffer, 128);
    long width, height;

    str >> width;
    str >> height;
    is.getline(buffer, 128);
    cout << width << "x" << height << " binary PGM file read." << endl;

    image->columns = width;
    image->rows = height;
    image->maxdepth = (double)255;
long size = (image->columns) * (image->rows);
image->data = new double[size];

unsigned char mybuf[1024];
long num;
long index = 0;

while (1) {
    if (read(mybuf, 1024)) {
        long long = mybuf[i];
        image->data[index] = (double) mybuf[i];
        index++;
        if (index >= size) return;
    }
    return;
} else {
    cerr << "ERROR! Unknown PGM file format: " << img_type << endl;
    exit(1);
}

// get the sum
double get_points(pgm* image, long n, long s)
{
    long seed1 = s; //** DECODE SEED **/
    const long summations = 1;
    double point_a, point_b;
    long ar, ac, br, bc;
    double S = 0;
    int col = image->columns;

    for (long sum_index = 1; sum_index < (summations+1); sum_index++) {
        for (long pt_index = 0; pt_index < n; pt_index++)
        {
            ac = (long)(ran0(&seed1) * (image->columns));
            ar = (long)(ran0(&seed1) * (image->rows));
            bc = (long)(ran0(&seed1) * (image->columns));
            br = (long)(ran0(&seed1) * (image->rows));

            point_a = *((image->data) + ac + ar * col);
            point_b = *((image->data) + bc + br * col);

            S = S + (point_a - point_b);
        }
        return(S);
    }

    // the main function, duh
    int main(int argc, char* argv[])
    {
        int bits = 4; // amount of information
        int corramt = 3; // amount of redundancy in error correction
        int startseed = 1; // starting decode seed
        long seed; // current seed being decoded
        int* info = new int[bits]; // the information decoded
        char ch = '0'; // the character represented by byte
        pgm image; // float strongsum; // sum for the strong bit
        float* sum = new float[bits*corramt]; // record of sums
        int* transl = new int[bits]; // the information recovered

        if (argc == 1) error("mbdecode input file datafile number-strong number-data");
        if (argc != 5) error("Wrong number of arguments");
long number = atol(argv[3]);
long datanum = atol(argv[4]);
ofstream to(argv[2]);
if (!to) cerr("cannot open output file ", argv[2]);

cerr << "Acquiring Image\n";
get_pgm(argv[1], &image);

cerr << "Decoding Image\n";
//get the sum for the strong bit
strongsum = get_points(&image, number, (long)0);
cerr << "strong bit: " << strongsum << '\n';

int index2 = 0;
for (int index = 1; index <= bits*coramt; index++)
{
    sum[index-1] = get_points(&image, datanum, (long)index);
    cerr << "seed " << index << ", value: " << sum[index-1] << '\n';
}
cerr << "n extracted data: \n"
for (int index = 0; index < bits*coramt; index++)
{
    if (sum[index] > 0)
        trans[index] = 1;
    else
        trans[index] = 0;
    if ((index+1)%3 == 1)
        cerr << '*' << trans[index];
    else
        cerr << trans[index];
}
cerr << '\n';
cerr << "n encoded data: \n"
for (int index = 0; index < bits; index++)
{
    if ((trans[index*3] + trans[index*3+1] + trans[index*3+2]) > 1)
        info[index] = 1;
    else
        info[index] = 0;
    cerr << info[index];
}
cerr << '\n';
delete[] sum;
delete[] trans;
delete[] info;
return 0;
}