2006

Using the singular value decomposition for image steganography

Daniel Wengerhoff
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/1388

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Using the singular value decomposition for image steganography

by

Daniel Wengerhoff

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Information Assurance

Program of Study Committee:
Jennifer Davidson, Major Professor
Clifford Bergman
Leslie Hogben

Iowa State University
Ames, Iowa
2006

Copyright © Daniel Wengerhoff, 2006. All rights reserved.
DEDICATION

I would like to dedicate this thesis to Erin Siem without whose patience, love and support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and financial assistance during the writing of this work.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................. v

**LIST OF FIGURES** .............................................. vii

**CHAPTER 1. Background on Steganography and Literature Review** ............................... 1

1.1 Literature Review ................................................. 2

**CHAPTER 2. Linear Algebra Background for the Singular Value Decomposition** ............... 4

**CHAPTER 3. Basic Embedding and Extraction Algorithms** ........................................... 10

3.1 The Basic Embedding Algorithm ......................................... 10

3.2 The Basic Extraction Algorithm ........................................ 11

**CHAPTER 4. IMPROVING THE EMBEDDING ALGORITHM TO REDUCE ERROR** ..................... 13

4.1 Introduction .................................................... 13

4.2 Choosing a Good Block Size and the Right Number of Columns to Protect .................. 13

4.3 Adjusting Singular Values ........................................ 18

4.4 The Benefits of Iteration ......................................... 19

4.5 Redundancy ..................................................... 19

**CHAPTER 5. Conclusions and further research** .................................................. 25

5.1 Ideas for Further Research ......................................... 25

5.2 Conclusions .................................................... 25

**APPENDIX A. Additional charts and pictures** ........................................... 27

**APPENDIX B. MatLab Code** ........................................ 48
BIBLIOGRAPHY
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Image: Tank.tiff, Independent variable: columns protected, Dependent variables: error rate, message size, Constant: 8 x 8 block size</td>
<td>16</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Image: Tank.tiff, Independent variable: Block Size, Dependent variables: error rate, message size, Constant: 1 column protected, 100 trials per block size</td>
<td>17</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>All images were tested using a block size of 8 x 8 with 2 columns protected, 1 iteration and no redundancy. This table shows that the recovery error rate is reduced when the singular values are evenly spaced.</td>
<td>18</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Image: Tank.tiff, Independent variable: Number of Iterations, Dependent variables: error rate, Constants: 8x8 block size, 2 columns protected.</td>
<td>19</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Image: Pentagon.tiff, Independent variable: Number of Iterations, Dependent variables: error rate, Constants: 8 x 8 block size, 2 columns protected, message size of 344,064 bits.</td>
<td>20</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>Image: Tank.tiff, Independent variable: redundancy Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iteration, 2 columns protected</td>
<td>20</td>
</tr>
<tr>
<td>Table 4.7</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 15 times redundancy, 2 columns protected</td>
<td>21</td>
</tr>
<tr>
<td>Table 4.8</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iterations, 5 times redundancy, 2 columns protected</td>
<td>21</td>
</tr>
<tr>
<td>Table 4.9</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iterations, 5 times redundancy, 2 columns protected</td>
<td>22</td>
</tr>
<tr>
<td>Table 4.10</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 21 times redundancy, 2 columns protected</td>
<td>22</td>
</tr>
<tr>
<td>Table 4.11</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 21 times redundancy, 2 columns protected</td>
<td>22</td>
</tr>
<tr>
<td>Table A.1</td>
<td>Independent variable: Image, Dependent variables: error rate, message size, Constant: 8x8 block size, 1 iteration, 21 times redundancy, 2 columns protected</td>
<td>27</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 4.1  Original image of the pentagon. .......................................................... 14
Figure 4.2  Image of the pentagon with a random message embedded using a block
size of 8 × 8 and 1 column protected. ................................................................. 15
Figure 4.3  Image of the pentagon with a random message embedded using a block
size of 8 × 8 and 2 columns protected. ................................................................. 23
Figure 4.4  Image of the pentagon with a random message embedded using a block
size of 16 × 16 and 9 column protected. ................................................................. 24
Figure A.1  Image of the pentagon with a random message embedded using a block
size of 4 × 4 and 1 columns protected. ................................................................. 28
Figure A.2  Image of the pentagon with a random message embedded using a block
size of 8 × 8 and 0 columns protected. ................................................................. 29
Figure A.3  Image of the pentagon with a random message embedded using a block
size of 8 × 8 and 1 column protected. ................................................................. 30
Figure A.4  Image of the pentagon with a random message embedded using a block
size of 8 × 8 and 2 columns protected. ................................................................. 31
Figure A.5  Image of the pentagon with a random message embedded using a block
size of 16 × 16 and 5 columns protected. ................................................................. 32
Figure A.6  Image of the pentagon with a random message embedded using a block
size of 16 × 16 and 9 columns protected. ................................................................. 33
Figure A.7  Image of a boat with a random message embedded using a block size
of 8 × 8 and 2 columns protected. ................................................................. 34
Figure A.8  Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 5 iterations. .......................... 35
Figure A.9  Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 15 times redundancy. ............... 36
Figure A.10 Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, 15 times redundancy and 5 iterations. 37
Figure A.11 Original aerial photo. Aerial photo with block size, columns protected, iteration and redundancy available. ................................. 38
Figure A.12 Original image of a stream. .................................................... 39
Figure A.13 This image is a good example to show that not every image is a good candidate for embedding a secret message. The black holes in the railing are a result of $8 \times 8$ in an area where there is very little color variation. Image of a stream with a random message embedded using a block size of $8 \times 8$ and 1 columns protected. ................................. 40
Figure A.14 Image of a stream with a random message embedded using a block size of $8 \times 8$ and 2 columns protected. ................................. 41
Figure A.15 Image of a stream with a random message embedded using a block size of $16 \times 16$ and 9 columns protected. ................................. 42
Figure A.16 Original image of a tank. .................................................... 43
Figure A.17 Image of a tank with a random message embedded using a block size of $8 \times 8$ and 1 columns protected. ................................. 44
Figure A.18 Image of a tank with a random message embedded using a block size of $8 \times 8$ and 2 columns protected. ................................. 45
Figure A.19 Image of a tank with a random message embedded using a block size of $16 \times 16$ and 9 columns protected. ................................. 46
Figure A.20 Image of a tank with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 3 iterations. ................................. 47
CHAPTER 1. Background on Steganography and Literature Review

Steganography, the study of data hiding with the intent of sending a secret message, has become increasingly important over the last few years as the use of digital media, which provides ample space to hide the message, has become more popular. Generally, methods of image steganography hide messages by using redundancy in the image. The message will be visually imperceptible as long as the insertion of the message does not cause any noticeable changes to the original image. Of course, it may be possible to detect the message using other means. The study of detecting secret messages is called Steganalysis.

Of course cryptography already provides a means to send a secret message, but cryptography and steganography have different goals. Consider that the main goal of cryptography is to render a message unintelligible to a receiver who does not know the secret key required to decrypt the message. This differs from the goal of steganography which is to conceal the fact that a message even exists. A common example to illustrate this difference, is that two prisoners Bob and Carl are conspiring to escape, but all their communications are being monitored by a warden Wendy. If Bob sends an encrypted message to Carl, Wendy will wonder what they are secretly discussing and cut off all communications between them. Obviously, Bob and Carl need to send messages to each other that will go undetected. If Bob and Carl have access to computers, they might choose to use an image from a website to hide their messages. Of course, Bob and Carl can still encrypt the message before hiding it in the image to ensure that if Wendy finds the hidden message she cannot read it.
1.1 Literature Review

There have been several attempts made to use the singular value decomposition to hide data. This section discusses several of the previous attempts. First in [Liu, R. (2002)], the authors embed a watermark into the singular values of the matrix by perturbing . This work is different from what is presented in this thesis because here the data to be embedded is not a watermark and the information is embedded into the U matrix of the of the singular value decomposition, not in the singular values themselves. It is important to note the distinction between watermarking and steganography. The difference is that a watermark is a small message that provides proof of ownership which needs to be imperceptible and robust against attempts to remove or modify the watermark. One other crucial property of a watermark is that it needs to be trustworthy, that is that nobody else could have produced a counterfeit watermark. In this thesis, the messages discussed are not intended to provide proof of ownership nor do they need to be robust against attacks. They do however need to be imperceptible.

Another attempt to use the singular value decomposition is in [Bao, P. (2005)]. In that paper the authors propose a method for embedding a watermark in the singular value decomposition in the wavelet domain of an image. This differs from the work in this thesis because instead of embedding in the wavelet domain, the message is embedded in the spatial domain.

All the images presented in this thesis are available from at http://sipi.usc.edu/database/ [USC-SIPI, (2004)]. The images were chosen because they are square, grayscale images which are freely available in .tiff formats. These images often appear in academic papers. In fact, several of the images are found in [Bao, P. (2005)] and [Liu, R. (2002)].

Finally, the original embedding and extraction algorithms discussed in this thesis originally appeared in [Davidson, D. (2005)]. This thesis expands upon their work by attempting to identify and eliminate sources of error in the embedding algorithm. To this extent parameters that eliminate error while maintaining visual fidelity are discussed.

Chapter 1 provides a brief background in linear algebra needed to understand the method for using the singular value decomposition of a matrix to hide a message in a digital image which is presented in Chapter 2. Chapter 2 also covers the method used to recover the embedded
message. Chapter 3 addresses how the embedding method can be improved. Chapter 4 provides
conclusions and ideas for further research. While there are some images and charts throughout
many chapters, appendix 1 provides many more and appendix 2 provides the Matlab code used
to produce the results presented.
CHAPTER 2. Linear Algebra Background for the Singular Value Decomposition

Before discussing the algorithm that will be used to hide data with the singular value decomposition of a matrix some basics from linear algebra shall be covered. The following definitions and theorems come from [Lay, D. (2000)]. Note that what follows applies to $m \times n$ real matrices, however for ease of notation I will be focusing on $n \times n$ real matrices.

**Definition.** $x$ is an *eigenvector* of $A$, an $n \times n$ matrix, if $x$ is a nonzero vector such that $Ax = \lambda x$ for some some scalar $\lambda$. $\lambda$ is called the *eigenvalue* of $A$ corresponding to $x$.

**Definition.** A scalar $\lambda$ is an *eigenvalue* of $A$, an $n \times n$ matrix if and only if $\lambda$ is a solution to $\det(A - \lambda I) = 0$. $\det(A - \lambda I)$ is called the *characteristic polynomial* of $A$.

**Definition.** The *null space* of a matrix $A$, denoted by $(\text{Nul } A)$, is the set of all vector solutions to the equation $Ax = 0$. This means that Nul $A$ is the set of eigenvectors corresponding to the eigenvalue 0.

**Definition.** A *subspace* of $\mathbb{R}^n$ is any subset $H$ in $\mathbb{R}^n$ that satisfies the following properties:
1. The zero vector is in $H$;
2. If $u, v \in H$ then $(u + v) \in H$;
3. For every $u \in H$ and for every scalar $c$ the vector $(cu) \in H$.

**Theorem 2.0.1** Nul $A$ is a subspace of the column space of $A$ (Col $A$).

Eigenvalues can also be characterized as the scalar solutions ($\lambda$) to the equation $(A - \lambda I)x = 0$. Note that the vector solutions to $(A - \lambda I)x = 0$ belong to the null space of $A - \lambda I$. We call
the subspace generated by the eigenvectors associated with each eigenvalue the eigenspace of A. Notice that the eigenspace of A is a subspace of \( \mathbb{R}^n \).

**Theorem 2.0.2** Let \( A \) be an \( n \times n \) matrix and let \( \{v_1, \ldots, v_r\} \) be eigenvectors corresponding to distinct eigenvalues \( \lambda_1, \ldots, \lambda_r \) of \( A \). Then \( \{v_1, \ldots, v_r\} \) is linearly independent.

**Definition.** Let

\[
a = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}^\tau
\]

and

\[
b = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix}^\tau
\]

be vectors in \( \mathbb{R}^n \), where \( \tau \) denotes the transpose. We define the *dot product* of \( a \) and \( b \) by

\[
a \cdot b = a_1 b_1 + \cdots + a_n b_n.
\]

Note that \( a \cdot b = a^\tau \cdot b \).

**Definition.** We say two vectors \( u \) and \( v \) are *orthogonal* if \( u \cdot v = 0 \). Also a set of vectors \( \{u_1, \ldots, u_n\} \) is an *orthogonal set* if \( u_i \cdot u_j = 0 \) for all \( i \neq j \). The set is *orthonormal* if it is orthogonal and all of the vectors in the set are unit vectors.

**Theorem 2.0.3** If \( S = \{v_1, \ldots, v_r\} \) is an orthogonal set of nonzero vectors, then \( S \) is linearly independent.

**Definition.** \( A \) is a *symmetric* matrix if \( A^\tau = A \).

**Theorem 2.0.4** If \( A \) is a real symmetric then any two eigenvectors corresponding to different eigenvalues are orthogonal.

**Proof.** Let \( v_1 \) be an eigenvector for the eigenspace of \( \lambda_1 \) and let \( v_2 \) be an eigenvector for the
eigenspace of $\lambda_2$ with $\lambda_1 \neq \lambda_2$. Now compute

$$\lambda_1 v_1 \cdot v_2 = (\lambda_1 v_1)^\top v_2 = (Av_1)^\top v_2 \quad (v_1 \text{ is an eigenvector})$$

$$= (v_1^\top A^\top) v_2 = v_1^\top (Av_2) \quad (A \text{ is symmetric})$$

$$= v_1^\top (\lambda_2 v_2) \quad (v_2 \text{ is an eigenvector})$$

$$= \lambda_2 v_1^\top v_2 = \lambda_2 v_1 \cdot v_2$$

So we have $\lambda_1 v_1 \cdot v_2 = \lambda_2 v_1 \cdot v_2$. Thus $(\lambda_1 - \lambda_2)v_1 \cdot v_2 = 0$. Since we know $\lambda_1 \neq \lambda_2$ we have $v_1 \cdot v_2 = 0$.

**Theorem 2.0.5** *The Spectral Theorem for Symmetric Matrices.* If $A$ is an $n \times n$ symmetric matrix over the real numbers then it has the following properties:

i) $A$ has $n$ real eigenvalues counting multiplicity;

ii) The dimension of the eigenspace corresponding to eigenvalue $\lambda$ is the multiplicity of $\lambda$;

iii) Eigenvectors from different eigenspaces are orthogonal;

iv) $A$ is orthogonally diagonalizable.

**Theorem 2.0.6** If $A \in \mathbb{R}^{n \times n}$, the eigenvalues of $A^\top A$ are non-negative.

**Definition.** The *singular values* of a matrix are the square roots of the eigenvalues of $A^\top A$ and $AA^\top$.

Generally the singular values are ordered with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

**Definition.** The *right singular vectors*, $v_1, \ldots, v_n$ of $A$ are the eigenvectors of $A^\top A$ corresponding to $\sigma_1, \ldots, \sigma_n$.

**Proposition.** The singular value, $\sigma_i$, is the length of the vector $Av_i$.

**Proof.** Let $A$ be a real $n \times n$ matrix. Then $A^\top A$ is symmetric and can be orthogonally diagonalized. Let $\{v_1, \ldots, v_n\}$ be an orthonormal basis for $\mathbb{R}^n$ consisting of eigenvectors of
$A^T A$ and let $\sigma_1^2, \ldots, \sigma_n^2$ be the associated eigenvalues of $A^T A$. Then,

$$\|A v_i\|^2 = (A v_i)^T A v_i = v_i^T A^T A v_i$$

$$= v_i^T (\lambda_i v_i) \quad (v_i \text{ is an eigenvector of } A^T A)$$

$$= \lambda_i \quad \text{(Since } v_i \text{ is a unit vector)}$$

As defined earlier the singular values are the square roots of the eigenvalues of $A^T A$ and they are arranged in nonincreasing order. That is $\sigma_i = \sqrt{\lambda_i} = \sqrt{\|A v_i\|^2} = \|A v_i\|$ which is the length of the vector $A v_i$.

**Theorem 2.0.7** Suppose that $\{v_1, \ldots, v_n\}$ is an orthonormal basis of $\mathbb{R}^n$ and each $v_i$ is a right singular vector of $A^T A$, arranged so that the corresponding eigenvalues of $A^T A$ satisfy $\sigma_1^2 \geq \cdots \geq \sigma_n^2$. Also suppose that $A$ has $r$ nonzero singular values. Then $\{A v_1, \ldots, A v_r\}$ is an orthogonal basis for Col $A$ and rank $A = r$.

**Proof.** First show $\{A v_1, \ldots, A v_n\}$ is an orthogonal set. $v_i$ and $\lambda_j v_j$ are orthogonal for $i \neq j$. So,

$$(A v_i)^T (A v_j) = v_i^T A^T A v_j = v_i^T (\lambda_j v_j) = 0$$

and we can conclude that $\{A v_1, \ldots, A v_n\}$ is an orthogonal set.

Also, the lengths of the vectors $A v_1, \ldots, A v_n$ are the singular values of $A$. Since there are $r$ nonzero singular values of $A$, $A v_i \neq 0$ if and only if $1 \leq i \leq r$. So we have $A v_1, \ldots, A v_r$ are linearly independent vectors and they are in Col $A$. Now let $y \in \text{Col } A$, say $y = A x$. Now we can write $x = c_1 v_1 + \cdots + c_n v_n$. Thus,

$$y = A x = c_1 A v_1 + \cdots + c_r A v_r + c_{r+1} A v_{r+1} + \cdots + c_n A v_n = c_1 A v_1 + \cdots + c_r A v_r + 0 + \cdots + 0$$

and $y \in \text{Span}\{A v_1, \ldots, A v_r\}$. Thus, rank $A = \dim \text{Col } A = r$. 
Theorem 2.0.8 The columns of a real matrix \( V \in \mathbb{R}^{n \times n} \) are orthogonal if and only if \( V^T V = I \).

Theorem 2.0.9 The Singular Value Decomposition Let \( A \) be an \( m \times n \) matrix with rank \( r \). Then there exists an \( m \times n \) matrix \( \Sigma \) where the first \( r \) diagonal entries are the first \( r \) singular values of \( A \) where \( \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0 \) and all the other entries are 0. Also, there exist an \( m \times m \) orthogonal matrix \( U \) and an \( n \times n \) orthogonal matrix \( V \) such that \( A = U \Sigma V^T \). This is called the singular value decomposition (SVD) of \( A \).

Proof. Let \( \sigma_i^2 \) and \( v_i \) be as in the previous theorem so that \( \{Av_1, \ldots, Av_r\} \) is an orthogonal basis for Col \( A \). Since the length of \( Av_i \) is \( \sigma_i \), the singular value corresponding to \( v_i \), we can normalize \( \{Av_1, \ldots, Av_r\} \) to obtain an orthonormal set \( \{u_1, \ldots, u_r\} \). Thus \( u_i = \frac{1}{\sigma_i} Av_i \). Also \( Av_i = \sigma_i u_i \) (for \( 1 \leq i \leq r \)). Now use the Gram-Schmidt process to extend \( \{u_1, \ldots, u_r\} \) to an orthonormal basis of \( \mathbb{R}^m \), \( \{u_1, \ldots, u_m\} \). Now let \( U = [u_1 \cdots u_m] \) and \( V = [v_1 \cdots v_n] \). By construction \( U \) and \( V \) are orthogonal. Also \( AV = [Av_1 \cdots Av_r 0 \cdots 0] = [\sigma_1 u_1 \cdots \sigma_r u_r 0 \cdots 0] = U \Sigma \).

Since \( V \) is an orthogonal matrix \( U \Sigma V^T = AVV^T = A \).

Theorem 2.0.10 Let \( A \) be an \( n \times n \) matrix and let \( U \Sigma V^T \) be the singular value decomposition of \( A \). If \( A \) has \( n \) singular values \( \sigma_1, \ldots, \sigma_n \) with \( \sigma_1 > \sigma_2 > \cdots > \sigma_n > 0 \) then the columns of \( U \) and the rows of \( V^T \) are unique up to a sign.

Proof. Since each of the singular values are distinct, their squares (the eigenvalues of \( A^T A \)) \( \lambda_1, \ldots, \lambda_n \) are distinct. Also, since each eigenvalue has multiplicity 1 the corresponding eigenvector is unique, up to multiplication by a scalar and therefore unique up to a sign. Since each of the eigenvectors are from different eigenspaces they are pairwise orthogonal. The rows of \( V^T \) are the unit eigenvectors of \( A^T A \) which are unique up to multiplication by \(-1\). Also since since each column of \( U \) is given by \( u_i = \frac{1}{\sigma_i} Av_i \), each \( u_i \) will be uniquely determined up to a sign.

In order to discuss the algorithm a definition is needed. First, let \( A \) be a \( m \times n \) matrix where \( m > n \) and let \( A \) have \( n \) distinct nonzero singular values. Also, let \( A = U \Sigma V^T \) be the
SVD of $A$. Then the rows of $V^\tau$ are unique up to a sign and the first $n$ columns of $U$ are unique up to a sign. However, columns $n + 1$ through $m$ are not unique since they are the basis of the null space of $AA^\tau$.

**Definition.** A column of a matrix is *greater than zero* if the first nonzero entry is greater than zero. A matrix is said to be *standard* if each column is greater than zero.

Notice that if an $n \times n$ matrix $U$ is not standard then it can be made to be standard by multiplying $U$ on the right by an $n \times n$ diagonal matrix, $D$ with $\pm 1$ as the entries along the diagonal. Also, notice that $D$ is its own inverse and $UDD = U$.

$$A = U\Sigma V^\tau = (UD)D\Sigma V^\tau = (UD)\Sigma(DV^\tau)$$

**Theorem 2.0.11** If $A$ is an $n \times n$ matrix where, $A = U\Sigma V^\tau$, the singular values of $A$ are $\sigma_1 > \sigma_2 > \ldots > \sigma_n$ and $U$ is standard then $V^\tau$ is unique.

The concepts presented in this chapter show that the singular value decomposition can be calculated for any matrix which we take advantage of in the next chapter when we describe the data embedding algorithm.
CHAPTER 3. Basic Embedding and Extraction Algorithms

This chapter discusses the basic algorithm to embed a message in an image using the singular value decomposition. In later chapters modifications to this algorithm will be discussed. The algorithm for extracting the message is also discussed in this chapter.

3.1 The Basic Embedding Algorithm

Let $M = m_1 m_2 m_3 \ldots m_k$, where $m_i \in \{0, 1\}$ be the message we want to embed in an image. The basic idea of the algorithm is to embed $M$ into $U$ by changing the signs of certain entries in $U$ to correspond with the bit values $\{m_i\}$. Also assume that an image, $I$, is broken up into blocks of size $8 \times 8$. The message will be broken into pieces and each consecutive piece will be embedded into an $8 \times 8$ block. Let $A = U \Sigma V^T$, where $U$ is standard, be one of these blocks. Note that the entries of $A$ will be integer valued from 0 to 255. Then we can embed $M$ using the following embedding algorithm.

$$U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{18} \\ u_{21} & u_{22} & u_{23} & \cdots & u_{28} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{81} & u_{82} & u_{83} & \cdots & u_{88} \end{pmatrix}$$

$M$ is just a string with bit values of $\pm 1$ in each position. We now form a new matrix $\hat{U}$ in the following way. We let the first column of $\hat{U}$ be the same as the first column of $U$. Starting with the first non-zero entry of column 2, say $u_{21}$, we let $\hat{u}_{21} = m_1 * |u_{21}|$. We continue down the column in this way until $u_{28}$. Here do not place a bit value entry in $\hat{u}_{28}$ but some value.
that will make column 2 of $\hat{U}$ orthogonal to column 1 of $\hat{U}$.

We embed more bits of $M$ in column 3 in the same way, except that in column 3 we can only embed message bits through $u_{36}$ because we need to make column 3 orthogonal to columns 1 and 2. In each successive column we can embed one less bit than we did in the column before it. So in an 8×8 block, starting in column 2, we can embed $6 + 5 + 4 + 3 + 2 + 1 = 21$ bits our message.

The next step is to find $\hat{A} = \hat{U}\Sigma V^\dagger$. Note that the values of $\hat{A}$ are no longer necessarily integer values from 0 to 255. Instead they are real numbers, possibly outside of that range. We make $\hat{A}$ into $A'$ by first rounding all the entries in $\hat{A}$ so that they are now all integers. Next we clip the rounded entries of $A'$ to get $\hat{A}$. This means that if $e$ is an entry of $A'$ and $e < 0$ we set $e = 0$ in $\hat{A}$ and if $e > 255$ we set $e = 255$ in $\hat{A}$. Thus $\hat{A} = \text{Clip}(A')$.

After repeating this process on every 8×8 block in the image, we have the marked image $\tilde{I}$.

### 3.2 The Basic Extraction Algorithm

In the previous section, the general algorithm to embed data in an image was presented. Of course, it is necessary for the receiver of the message to be able to extract the message from the image. In order for the receiver to extract the message he must know the block size, $k$, that the sender used. The receiver then breaks the image into $k$ by $k$ blocks. Next, the SVD of each block is computed so that $A = USV^\dagger$, where $U$ is standard. Now the message can be read from the matrix $U$ by looking at the signs of the entries in $U$ which have had information hidden in them. For example, if the second column of $U$ is

$$
\begin{pmatrix}
3 & -4 & 15 & 38 & 7 & -46 & 9 & 6
\end{pmatrix}^T
$$

the first six bits of the message will be 011101.
Ideally these algorithms would work every time for every bit and we would never incorrectly recover a bit. In practice though, error can be introduced from several sources. The next chapter mentions the source of errors and provides several methods used to reduce errors.
CHAPTER 4. IMPROVING THE EMBEDDING ALGORITHM TO REDUCE ERROR

4.1 Introduction

If the embedding and recovery algorithms are used as presented in the last chapter, several problems arise. Firstly, the image is very badly distorted, which is contrary to our goal of imperceptibly changing the image. Secondly, not all of the bits of the embedded message are recovered accurately. This chapter discusses techniques to maintain the original image fidelity and to increase the accuracy of the received message.

4.2 Choosing a Good Block Size and the Right Number of Columns to Protect

An important fact when working with the SVD of an image is that the largest singular values contains the most visual information about the image. Remember that the singular values are arranged largest to smallest. This means that if the first column of $U$ is modified then the resulting section of the image will appear dramatically different.

In the earlier description of the embedding algorithm, we let the first column of $\hat{U}$ be the same as the first column of $U$. As figures 4.1 and A.3 show, the problem with this is that the stegoimage has a visibly lower fidelity than the original image. Figure A.3 can be identified as the stegoimage because the bottom of the pentagon and the roads are blurry.

The image fidelity can be improved for example, by letting the first $m$ columns of $\hat{U}$ be the same as the first $m$ columns of $U$. This is called protecting the first $m$ columns of $U$. By doing this the stegoimage looks more like the cover image (see figure A.4 at the end of this chapter),
Figure 4.1  Original image of the pentagon.
Figure 4.2 Image of the pentagon with a random message embedded using a block size of $8 \times 8$ and 1 column protected.
however the capacity of the image is reduced. For example, using a block size of 8 × 8 and
protecting 1 column allows us to embed 21 bits in each block. If we protect 2 columns, we can
only embed 15 bits per block. Another concern was whether or not changing the number of
columns protected has an effect on the error rate of message recovery (the percentage of bits
recovered incorrectly).

To see if the number of columns protected influences the error rate, the same experiment
was run on several images. In the experiment a random message was embedded in the image,
which was broken into 8 × 8 blocks. Next the embedding algorithm was run while protecting
1 column. After the message is extracted from the stegoimage it is compared to the original
message and the percentage of bits recovered incorrectly is calculated. Next the same test
was performed while protecting 2 columns, then 3 columns and so on. This experiment was
repeated 100 times, with a different message each time, for each number of columns protected,
Table 4.1 shows the results when the test is run on an image of a tank.

<table>
<thead>
<tr>
<th>Columns Protected</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.1020</td>
<td>86,016</td>
</tr>
<tr>
<td>2</td>
<td>9.0097</td>
<td>61,440</td>
</tr>
<tr>
<td>3</td>
<td>10.0637</td>
<td>40,960</td>
</tr>
<tr>
<td>4</td>
<td>10.4628</td>
<td>24,576</td>
</tr>
<tr>
<td>5</td>
<td>10.9528</td>
<td>12,288</td>
</tr>
<tr>
<td>6</td>
<td>13.5313</td>
<td>4,096</td>
</tr>
</tbody>
</table>

Table 4.1 Image: Tank.tiff, Independent variable: columns protected, Dependent variables: error rate, message size, Constant: 8 x 8 block size

Notice that as the number of columns protected increases so does the error rate. This is
very important, because while having greater visual fidelity between the cover and stegoimages
is a priority, so is being able to recover the message accurately. From this data it is clear that
if a block size is 8x8, then protecting 2 columns will provide the best balance between message
size, error rate and visual fidelity of the stegoimage for this image.

In the discussion of the basic embedding algorithm, the cover image was broken into 8 × 8
blocks, but why was that size chosen? To answer that question first consider the pros and cons
of various block sizes. Using a block size that is very small will allow for a lower error rate as well as greater visual similarity between the cover image and the stegoimage. However, smaller block sizes also mean lower capacity and therefore smaller messages. On the other hand, large block sizes have higher data capacities but are more prone to higher error recovery rates. Also, if a block size that is too large is used then more columns must be protected, so that the fidelity of the cover image and stegoimage is preserved, which lowers the capacity.

To determine what block size should be used, a message was inserted into an image first using a block size of $4 \times 4$. After the message is extracted from the stegoimage, it is compared to the original message and the percentage of bits recovered incorrectly is calculated. Then block sizes of $8 \times 8$, $16 \times 16$, and so on up to $512 \times 512$ were used. With all block sizes only 1 column was protected. The experiment was repeated 100 times, with a different message each time, for each block size, on one image. Table 4.2 shows the error rate and data capacity of different block sizes with 1 column protected.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5.8813</td>
<td>49,152</td>
</tr>
<tr>
<td>8</td>
<td>6.1020</td>
<td>86,016</td>
</tr>
<tr>
<td>16</td>
<td>7.2844</td>
<td>107,520</td>
</tr>
<tr>
<td>32</td>
<td>22.7138</td>
<td>119,040</td>
</tr>
<tr>
<td>64</td>
<td>44.6655</td>
<td>124,992</td>
</tr>
<tr>
<td>128</td>
<td>49.4186</td>
<td>128,016</td>
</tr>
<tr>
<td>256</td>
<td>49.9442</td>
<td>129,540</td>
</tr>
<tr>
<td>512</td>
<td>49.9918</td>
<td>130,305</td>
</tr>
</tbody>
</table>

Table 4.2  Image: Tank.tiff, Independent variable: Block Size, Dependent variables: error rate, message size, Constant: 1 column protected, 100 trials per block size

It is also important to notice that the size of the message (capacity) that can be embedded into an image is dependent on the block size. Using a block size of $8 \times 8$ provides the best balance between error rate, message size and visual fidelity of the stegoimage for this image. Below are 2 images of the pentagon, one with a random message embedded and one with no hidden message. One image (figure A.4) uses an $8 \times 8$ block size with 2 columns protected. The other (figure A.6) uses a $16 \times 16$ block size with 9 columns protected. Notice that the
image with 16x16 blocks is noticeably different from the original image, while the image using 8x8 blocks looks identical to the original. For the original image of the pentagon see figure 4.1.

Another question that can be asked is, does protecting some columns on the right help reduce the error rate or improve the similarity of the stegoimage to the cover image? In fact, the answer is no. The error rate is not reduced, nor is the stegoimage more similar to the cover than if only columns on the left are protected. The only real effect of protecting columns on the right is that the data capacity of each block is reduced.

4.3 Adjusting Singular Values

A problem with using the singular value decomposition is that some of the singular values can be closely grouped together. This occurs in areas of the image where there is not much difference in color from one pixel to the next. It is a problem because it causes a greater error rate. However, if the singular values are evenly distributed, the error rate decreases. Table 4.3 shows the effect of evenly spacing the singular values. Thus, in the embedding algorithm the distance the singular values should be from each other is computed by adding the first singular value and the last singular and dividing by the block size. The singular values are evenly spaced between the first and last singular value.

<table>
<thead>
<tr>
<th>Image</th>
<th>Error - No Singular Value Adjustment</th>
<th>Error - Singular Values Evenly Spaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank.tif</td>
<td>12.3652</td>
<td>8.9388</td>
</tr>
<tr>
<td>Aerial.tif</td>
<td>7.9922</td>
<td>6.1081</td>
</tr>
<tr>
<td>Boat.tif</td>
<td>13.6955</td>
<td>10.2210</td>
</tr>
<tr>
<td>Chemical.tif</td>
<td>10.7318</td>
<td>7.3125</td>
</tr>
</tbody>
</table>

Table 4.3 All images were tested using a block size of 8 x 8 with 2 columns protected, 1 iteration and no redundancy. This table shows that the recovery error rate is reduced when the singular values are evenly spaced.
4.4 The Benefits of Iteration

One of the main sources of error is that there is a large difference between the original $U$ matrix and $U_E$ ($U$ with the message embedded) with respect to the magnitudes of $U$. What if a message is embedded into a block $A$ to get $A_{E_1}$ and then the same message is embedded into $A_{E_1}$ to get $A_{E_2}$? There will probably be a smaller change from $U_{E_1}$ to $U_{E_2}$ than there was from $U$ to $U_{E_1}$. This means that the entries of $U_{E_2}$ are closer to $U_{E_1}$ than the entries from $U_{E_1}$ are to $U$. As the number of times the message is re-embedded increases, the percentage of bits that are incorrectly recovered by the receiver decreases. How many iterations are necessary? Tables 4.4 and 4.5 below shows the error rate as the number of iterations varies in images where $8 \times 8$ blocks are used and 2 columns are protected.

<table>
<thead>
<tr>
<th>Columns Protected</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>9.0097</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4.7244</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3.7226</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>3.4030</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>3.2980</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>3.2617</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>3.2471</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>3.2258</td>
<td>61,440</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>3.2352</td>
<td>61,440</td>
</tr>
</tbody>
</table>

Table 4.4 Image: Tank.tiff, Independent variable: Number of Iterations, Dependent variables: error rate, Constants: $8 \times 8$ block size, 2 columns protected.

It is important to note that the error rate will decrease very slightly if more than 5 iterations are used, but it will also take much longer to run the algorithm a larger number of times.

4.5 Redundancy

Another way to reduce errors in the received message is to use a basic error correcting code called redundancy. Redundancy means that the same set of bits will be embedded in $h$ consecutive blocks where $h$ is odd. Then when the receiver receives the message he can look at
<table>
<thead>
<tr>
<th>Iterations</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.7456</td>
</tr>
<tr>
<td>3</td>
<td>2.8282</td>
</tr>
<tr>
<td>5</td>
<td>2.1806</td>
</tr>
<tr>
<td>7</td>
<td>2.0031</td>
</tr>
<tr>
<td>9</td>
<td>1.9391</td>
</tr>
<tr>
<td>11</td>
<td>1.9312</td>
</tr>
<tr>
<td>13</td>
<td>1.8909</td>
</tr>
<tr>
<td>15</td>
<td>1.9041</td>
</tr>
</tbody>
</table>

Table 4.5  Image: Pentagon.tiff, Independent variable: Number of Iterations, Dependent variables: error rate, Constants: 8 x 8 block size, 2 columns protected, message size of 344,064 bits.

Table 4.6 Image: Tank.tiff, Independent variable: redundancy Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iteration, 2 columns protected

If redundancy is used with iteration the error rate is dramatically reduced. This can be seen in table 4.10.

This chapter has shown that for the best results with regards to error rate, visual fidelity and message size it is best to use a block size of 8 x 8 while protecting 2 columns. Also, message redundancy of 15 all but eliminates error while severely limiting the message size, so if a very
Table 4.7 Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 15 times redundancy, 2 columns protected

<table>
<thead>
<tr>
<th>Image</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aerial.tiff</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1,024</td>
</tr>
<tr>
<td>airport.tiff</td>
<td>1</td>
<td>0.0012</td>
<td>5</td>
<td>0</td>
<td>16,384</td>
</tr>
<tr>
<td>boat.tiff</td>
<td>1</td>
<td>0.0024</td>
<td>5</td>
<td>0</td>
<td>4,096</td>
</tr>
<tr>
<td>chemical.tiff</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1,024</td>
</tr>
<tr>
<td>pentagon.tiff</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>16,384</td>
</tr>
<tr>
<td>straw.tiff</td>
<td>1</td>
<td>0.2734</td>
<td>5</td>
<td>error</td>
<td>4,096</td>
</tr>
<tr>
<td>tank.tiff</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>4,096</td>
</tr>
</tbody>
</table>

Table 4.8 Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iterations, 5 times redundancy, 2 columns protected

<table>
<thead>
<tr>
<th>Image</th>
<th>Columns Protected</th>
<th>Redundancy</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aerial.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.2474</td>
<td>3,072</td>
</tr>
<tr>
<td>airport.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.4744</td>
<td>49,152</td>
</tr>
<tr>
<td>boat.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.9595</td>
<td>12,288</td>
</tr>
<tr>
<td>chemical.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.3678</td>
<td>3,072</td>
</tr>
<tr>
<td>pentagon.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.4970</td>
<td>49,152</td>
</tr>
<tr>
<td>straw.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4.1691</td>
<td>12,288</td>
</tr>
<tr>
<td>tank.tiff</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0.6626</td>
<td>12,288</td>
</tr>
</tbody>
</table>

small amount of error is acceptable it is better to use a redundancy of 5. It is also important to iteratively embed the message and the optimal number of iterations is 5. The combination of iterations and provide the greatest error reduction.
<table>
<thead>
<tr>
<th>Image</th>
<th>Columns Protected</th>
<th>Redundancy</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aerial.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.0033</td>
<td>3,072</td>
</tr>
<tr>
<td>airport.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.4814</td>
<td>49,152</td>
</tr>
<tr>
<td>boat.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.0871</td>
<td>12,288</td>
</tr>
<tr>
<td>chemical.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.0326</td>
<td>3,072</td>
</tr>
<tr>
<td>pentagon.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.0397</td>
<td>49,152</td>
</tr>
<tr>
<td>tank.tiff</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0.0602</td>
<td>12,288</td>
</tr>
</tbody>
</table>

Table 4.9  Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 1 iterations, 5 times redundancy, 2 columns protected

<table>
<thead>
<tr>
<th>Image</th>
<th>Columns Protected</th>
<th>Redundancy</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank.tiff</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>0</td>
<td>4096</td>
</tr>
<tr>
<td>Pentagon.tiff</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>0</td>
<td>16384</td>
</tr>
</tbody>
</table>

Table 4.10  Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 21 times redundancy, 2 columns protected

<table>
<thead>
<tr>
<th>Image</th>
<th>Columns Protected</th>
<th>Redundancy</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank.tiff</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>0</td>
<td>4,096</td>
</tr>
<tr>
<td>Tank.tiff</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>0.0024</td>
<td>12,288</td>
</tr>
<tr>
<td>Tank.tiff</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>0.1180</td>
<td>28,672</td>
</tr>
<tr>
<td>Pentagon.tiff</td>
<td>2</td>
<td>21</td>
<td>5</td>
<td>0</td>
<td>16,384</td>
</tr>
<tr>
<td>Pentagon.tiff</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>1.0e-3 x 0.6104</td>
<td>49,152</td>
</tr>
<tr>
<td>Pentagon.tiff</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>0.1391</td>
<td>114,688</td>
</tr>
</tbody>
</table>

Table 4.11  Independent variable: Image, Dependent variables: error rate, message size, Constant: 8 x 8 block size, 5 iterations, 21 times redundancy, 2 columns protected
Figure 4.3  Image of the pentagon with a random message embedded using a block size of $8 \times 8$ and 2 columns protected.
Figure 4.4  Image of the pentagon with a random message embedded using a block size of $16 \times 16$ and 9 column protected.
CHAPTER 5. Conclusions and further research

5.1 Ideas for Further Research

While this thesis has presented a complete working algorithm using the singular value decomposition for steganography, there is further research that can be done. First of all, the images used for this thesis were tiff files so adapting this method for different image file formats would make it more practical given that jpeg and other compressed file formats are much more readily than tiff files. Also, all of the files used in this thesis were grayscale images and since few people trade grayscale images modifying the algorithm to work with color images would help to better conceal hidden messages.

Another way to improve the usefulness of this technique is to improve the data capacity. With a greater data capacity different error correcting codes can be used to further reduce the error rate. Also a greater capacity allows for audio or video messages to be hidden. One way to increase capacity is to hide a large message across multiple images. Also, once the algorithm can work with compressed color images, it could be extended to be used in video file, which could allow for a much greater data capacity than images alone.

Finally, a steganographic algorithm is good only so long as the hidden messages cannot be detected. A detected message can be read (if it is not encrypted), tampered with or destroyed so thorough steganalysis should be performed.

5.2 Conclusions

This thesis has presented a linear algebra background and it has explained a detailed algorithm to embed and recover a message using the singular value decomposition. Many ideas to maintain visual fidelity and reduce the message error recovery rate from spacing the
singular values evenly apart to simple error correcting codes have been evaluated and discussed.
Finally, some areas of further research were presented, so that the work presented here can be
built upon.
APPENDIX A. Additional charts and pictures

More Charts and Pictures

Table A.1 shows that redundancy alone is enough to reduce the error rate to zero. However to achieve such a high redundancy, in this case 21 times, only one column can be protected. As discussed in an earlier section, protecting only one column is not acceptable because the resulting image is visibly different from the original image.

<table>
<thead>
<tr>
<th>Image</th>
<th>Columns Protected</th>
<th>Redundancy</th>
<th>Iterations</th>
<th>Error Rate</th>
<th>Message Size(bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>4,096</td>
</tr>
<tr>
<td>Pentagon.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>16,384</td>
</tr>
<tr>
<td>Couple.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>4,096</td>
</tr>
<tr>
<td>Man.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>16,384</td>
</tr>
<tr>
<td>Stream.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>4,096</td>
</tr>
<tr>
<td>House.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>3,072</td>
</tr>
<tr>
<td>Airplane.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>12,288</td>
</tr>
<tr>
<td>Sailboat.tif</td>
<td>1</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>12,288</td>
</tr>
</tbody>
</table>

Table A.1 Independent variable: Image, Dependent variables: error rate, message size, Constant: 8x8 block size, 1 iteration, 21 times redundancy, 2 columns protected
Figure A.1 Image of the pentagon with a random message embedded using a block size of $4 \times 4$ and 1 columns protected.
Figure A.2 Image of the pentagon with a random message embedded using a block size of $8 \times 8$ and 0 columns protected.
Figure A.3  Image of the pentagon with a random message embedded using a block size of $8 \times 8$ and 1 column protected.
Figure A.4 Image of the pentagon with a random message embedded using a block size of $8 \times 8$ and 2 columns protected.
Figure A.5  Image of the pentagon with a random message embedded using a block size of $16 \times 16$ and 5 columns protected.
Figure A.6 Image of the pentagon with a random message embedded using a block size of $16 \times 16$ and 9 columns protected.
Figure A.7  Image of a boat with a random message embedded using a block size of $8 \times 8$ and 2 columns protected.
Figure A.8  Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 5 iterations.
Figure A.9  Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 15 times redundancy.
Figure A.10  Image of a boat with a random message embedded using a block size of $8 \times 8$, 2 columns protected, 15 times redundancy and 5 iterations.
Figure A.11  Original aerial photo. Aerial photo with block size, columns protected, iteration and redundancy available.
Figure A.12  Original image of a stream.
Figure A.13  This image is a good example to show that not every image is a good candidate for embedding a secret message. The black holes in the railing are a result of $8 \times 8$ in an area where there is very little color variation. Image of a stream with a random message embedded using a block size of $8 \times 8$ and 1 columns protected.
Figure A.14  Image of a stream with a random message embedded using a block size of $8 \times 8$ and 2 columns protected.
Figure A.15 Image of a stream with a random message embedded using a block size of $16 \times 16$ and 9 columns protected.
Figure A.16  Original image of a tank.
Figure A.17  Image of a tank with a random message embedded using a block size of $8 \times 8$ and 1 columns protected.
Figure A.18  Image of a tank with a random message embedded using a block size of $8 \times 8$ and 2 columns protected.
Figure A.19  Image of a tank with a random message embedded using a block size of $16 \times 16$ and 9 columns protected.
Figure A.20  Image of a tank with a random message embedded using a block size of $8 \times 8$, 2 columns protected, and 3 iterations.
APPENDIX B.  MatLab Code

This appendix contains all of the MatLab Code used to produce the images and charts in this thesis.

simple iterate

%simple_iterate.m
%This is a function which embeds a random message into an image and then
%recovers the message. The error rate is then calculated and returned in
%avg_error_rates. Necessary inputs are the image url
%without the file type (ex: C:\images\test.tiff is C:\images\test), the
%file type entered without a '.' (ex tiff not .tiff), the block size to be
%used, the number of columns to protect, the amount of redundancy and the
%number of iterations to be used.

function avg_error=simple_iterate(image_addr, image_type, dim,
cols_protected, redun, iterations)

tests = 10;
error_rates=zeros(1,tests); %an array to hold the error rate of each test
msg_sizes=zeros(1,tests); %an array to hold the size of the message embeded in test

%Turn off common Matlab warnings
warning off MATLAB:divideByZero warning off
% MATLAB: nearlySingularMatrix warning off MATLAB:singularMatrix

for t=1:tests
    t % display the test number

    % embed a random message into the image
    msg=mark_funct2(image_addr, image_type, dim, cols_protected, redun, iterations);
    msg_size=size(msg,2) % display the size of the embedded message

    % recover the message from the marked image
    recover_addr = [image_addr,'_marked'];
    rcvd_msg=recover_sing(recover_addr,image_type,dim,cols_protected,redun);

    % compute the error rate between the original and recovered message
    % and put it in the t-th slot of error_rates
    rcvd_msg_size=size(rcvd_msg,2);
    diff=abs(msg-rcvd_msg);
    error=sum(diff)/msg_size;
    error_rates(l,t)=error
end

% compute the average error rate for the given image and variables
avg_error=sum(error_rates)/tests*100

embed1

% embed1.m
% This is a function that embeds a random message into an image and returns
% the message that was embedded. Necessary inputs are the image url
without the file type (ex: C:\images\test.tiff is C:\images\test), the
file type entered without a '.' (ex tiff not .tiff), the block size to be
used, the number of columns to protect, the amount of redundancy and the
number of iterations to be used.

function msg = embed(image_addr, image_type, dim, cols_protected, redun, iterations)

%read the input image
input_image = imread(image_addr, image_type);

%make input_image into a real valued matrix so SVD can be computed
working_image = double(input_image)+1;

%make a copy of the image so that the original data isn’t lost
junk = working_image;

%get the dimensions of the image
[m,n]=size(junk);

%Calculate the number of blocks and bits per block
bpb = ((dim-cols_protected-1)\(dim-cols_protected))/2;
num_blocks=m*n/(dim^2);

%computes the largest message that can be embedded
msg_size=ceil(bpb*num_blocks/redun);

%Generate a random message of size msg_size to embed
msg=round(rand(1,msg_size));

% Concatenate msg to itself redun times to form redun_msg
for t=1:redun
    if t==1
        redun_msg=msg;
    else
        redun_msg=[redun_msg,msg];
    end
end %end t

for a=1:iterations
    % Display the iteration
    a

    % Make a copy of redun_msg
    temp_msg=redun_msg;

    for j=1:(m/dim)
        for i=1:(n/dim)

            % Get the block that will be worked on
            block=junk(dim*j-(dim-1):dim*j, dim*i-(dim-1):dim*i)-127;

            % Compute the SVD of the block
            [U,S,V]=svd(block);
%used to make U standard and create V_prime
U_std = U;
V_prime = V;

%make U std and modify V into V_prime
for k=1:dim

    %If the first entry of a column in U is negative multiply
    %the column by -1
    if U(1,k)<0
        U_std(1:dim,k) = -1*U(1:dim,k);
        V_prime(1:dim,k) = -1*V(1:dim,k);
    end %end if
end %end k

%used to create s_prime
S_prime = S;

%evenly space the Singular Values between the largest and smallest.
avg_dist = (S(2,2)+S(dim,dim))/(dim-2);
for k = 3:dim-1
    S_prime(k,k)=S(2,2)-(k-2)*avg_dist;
end %end k

%get a part of message to be embedded
msg_chunk = temp_msg(1:bpb);
% remove the first bpb bits of the message so that the next pass
% allows us to get the next bpb bits
temp_msg = temp_msg(1,bpb+1:end);

% U_mk will be where the embedding occurs
U_mk=U_std;

% index is the bit of msg_chunk to embed
index = 1;

% embed the msg_chunk using U_std to make U_mk
% k varies from cols_protected+1 to dim since message bits can only
% be embedded in those columns
for k=cols_protected+1:dim
    % the last column of the block will have no bits embedded in
    % it, but it still needs to be made orthogonal to the rest
    % of the columns
    if k < dim
        % m+1-k comes from the fact that we can only embed data
        % in rows 2 through dim-(k-1)
        for l=2:dim+1-k
            if msg_chunk(index) == 0
                % replace the entry with -1 times the absolute
                % value of the entry
                U_mk(l,k)= -1*abs(U_std(l,k));
            else
                % replace the entry with its absolute value
                U_mk(l,k)=abs(U_std(l,k));
            end
        end
    end
end
end %end if
%advance the index by 1 bit
index = index + 1;
end %end 1
end %end if

%we need to make column orthogonal to other columns solve a system %of column # - 1 equations in col # - 1 unknowns.
coeffs = zeros(k-1); \%will become coefficeint matrix
sols = zeros(k-1,1); \%will become solutions matrix

%fills the coefficeint matrix
for x=1:k-1
   for y=1:k-1
      coeffs(x,y)=U_mk(y+dim+1-k,x);
   end %end y
%fill solutions matrix
sols(x,1)=-dot(U_mk(1:dim+1-k,k),U_mk(1:dim+1-k,x));
end %end x

%compute the new entries that make the column orthogonal to %the ones that came before it.
new_entries = coeffs\sols;
sz=size(new_entries,1); \%the number of new entries

%put the new entries in U_mk
for p=dim+1-sz:dim
   U_mk(p,k) = new_entries(p-(dim-sz),1);
end %end p

%normalize U_mk

norm_factor = sqrt(dot(U_mk(1:dim,k),U_mk(1:dim,k)));
for q=1:dim
    U_mk(q,k) = U_mk(q,k)/norm_factor;
end %end q
end %end k

%A_tilde will be the new matrix to put back in place of block
A_tilde = round(U_mk*S_prime*V_prime');

%make sure that the values of A_tilde are between 0 and 127 if
%not then make any negative entries 0 and any entries that
%are greater than 255 equal to 255
for r=1:dim
    for s=1:dim
        if A_tilde(r,s) < -127
            A_tilde(r,s) = -127;
        end %end if
        if A_tilde(r,s) > 128
            A_tilde(r,s) = 128;
        end %end if
    end %end r
end %end s

%replace the block in junk with A_tilde+127 so that the values
%of a tilde are between 0 and 255
junk(dim*j-(dim-1):dim*j, dim*i-(dim-1):dim*i)=A_tilde(1:dim,1:dim)+127;
end %end j
end %end i
end %end a

working_image2=uint8(junk-1);
%append_marked to the input file name
output_addr = [image_addr,'_marked.'];
%append the file type to the output address
output_addr = [output_addr,image_type];
%write out input_file_marked.image_type
imwrite(working_image2,output_addr,image_type);

recover

%recover1.m
%This function recovers a message that has previously been hidden in an
%image and returns it in rcvd_msg. Necessary inputs are the image url
%without the file type (ex: C:\images\test.tiff is C:\images\test), the
%file type entered without a '.' (ex tiff not .tiff), the block size to be
%used, the number of columns to protect, the amount of redundancy and the
%number of iterations to be used.

function rcvd_msg = recover1(image_addr, image_type, dim,
cols_protected,redun)

%read in the file
input_image = imread(image_addr, image_type);

%make input_image into a real valued matrix so SVD can be computed
recovery_image = double(input_image)+1;

% make a copy of the image so that the original data isn't lost
junk = recovery_image;

% get the dimensions of the image
[m,n]=size(junk);

% Calculate the number of bits per block
bpb = ((dim-cols_protected-1)*(dim-cols_protected))/2;

% computes the largest message that could have been embedded
msg_size=floor((m*n)/(dim*dim)*bpb);

% a place to keep a recovered message chunk of size bpb
temp_rcvd_msg=zeros(1,bpb);

% recover the message
for j=1:(m/dim)
    for i=1:(n/dim)
        % get the block that will be worked on
        block=junk(dim*j-(dim-1):dim*j, dim*i-(dim-1):dim*i)-127;

        % compute the SVD of the block
        [U,S,V]=svd(block);

        % used to make U standard
        U_std = U;
If the first entry of a column in U is negative multiply
the column by -1
for k=1:dim
    if U(1,k)<0
        U_std(1:dim,k) = -1*U(1:dim,k);
    end %end if
end %end k

next_spot keeps track of where to place the bit read from
the U matrix, reset each time through the j loop
next_spot=1;

only read data from columns that are not protected,
excluding the last one
for p=cols_protected+1:(n/dim)-1

the first row is always protected so this loop
%starts at 2
for q=2:(dim+1)-p
    %if the entry being examined is < 0 then the bit
    %is a 0 otherwise the bit is a 1
    if U_std(q,p)<0
        temp_rcvd_msg(1,next_spot)=0;
    else
        temp_rcvd_msg(1,next_spot)=1;
    end %end if
%increment next_spot
next_spot = next_spot + 1;
end %end q
end %end p

if i == 1 & j == 1
    rcvd_msg = temp_rcvd_msg;
else
    rcvd_msg = [rcvd_msg temp_rcvd_msg];
end %end if
end %end j
end %end i

%calculate the size of the embedded message
msg_size = size(rcvd_msg,2)/redun;

%The following takes into account the redundancy and finds the original
%embedded message using a simple error correcting code. If the redundancy
%was 5 then the message we have recovered so far is really the same message
%concatenated with itself 5 times. If we look at the first bit of each of
%these copies and have more 1s than 0s then we assume the bit is a 1. We do
%this for each bit of the message.
act_msg = zeros(1,msg_size); q=0; for w=1:msg_size
    for y=0:redun-1
        q=q + rcvd_msg(1,w+y*msg_size);
    end % y loop
    if q > redun/2
        act_msg(1,w)=1;
else
    act_msg(1,w)=0;
end  %end if
q=0;
end  %end w

%return rcvd_msg
rcvd_msg=act_msg;
BIBLIOGRAPHY


