

# Implementation and Testing For Paranoia

U-S-E  
CS169 – Software Engineering  
Dec 2 2003

Prepared by:

BERDAHL, EDGAR  
CHUNG, SANG  
GONG, GARY  
LE, TAM  
LEE, JOON YUL  
LI, SHENG  
MOZAFFARIAN, BEHRAD  
MURRAY, JASON  
NGUYEN, HIEP  
SHKOLNIKOV, YURIY

# Table of Contents

- Table of Contents ..... 2
- 1. Introduction ..... 3
  - 1.1 Abstract ..... 3
  - 1.2 Running Paranoia ..... 3
  - 1.3 Platform Independence..... 3
  - 1.4 API..... 4
  - 1.5 Subgroups..... 4
- 2. Revisions to the Design and Functionality Specifications ..... 4
  - 2.1 GUI Revisions ..... 4
  - 2.2 Security Revisions ..... 5
  - 2.3 Generic GIF/JPEG Revisions..... 5
  - 2.4 GIF Revisions ..... 5
  - 2.5 JPEG Revisions ..... 6
- 3. Testing..... 7
  - 3.1 GUI Tests ..... 7
  - 3.2 Security Tests ..... 7
  - 3.3 GIF Tests ..... 10
  - 3.4 JPEG Tests ..... 12
  - 3.5 Fine-Tuning JpegStegoImage..... 14
- 4. Known Bugs ..... 18
  - 4.1 GUI Bugs ..... 18
  - 4.2 Security Bugs ..... 19
  - 4.3 GIF Bugs ..... 19
  - 4.4 JPEG Bugs..... 19

# 1 INTRODUCTION

## 1.1 ABSTRACT

Paranoia allows an average user to securely transfer text messages by hiding them in a digital image file. A combination of steganography and encryption algorithms provides a strong backbone for Paranoia's security. Paranoia features innovative techniques for hiding text in a digital image file or even using it as a key to the encryption.

## 1.2 RUNNING Paranoia

Compile the package and run the module `paranoia.Application1`. The following will work on a UNIX type system, and the necessary files can be found within the directory.

- Change directories to outside the directory `paranoia`.
- If necessary, extract the libraries used by JBuilder from using `jar -xvf jbc1.jar`. It should create a directory called `com`. Make sure that the directories `paranoia` and `com` are next to each other in the same directory.
- Compile the Paranoia package using `javac paranoia/*.java`
- Run the GUI with `java paranoia.Application1`

## 1.3 PLATFORM INDEPENDENCE

Although we specified in our original design document that the program would run on Sun Java Virtual Machine for JDK 1.4.2 or higher, we surpassed this requirement by making it possible to use version 1.4.1 or higher. We tested that the application by running it on the following computers.

- Mac Powerbook G4 1.25GHz (Aluminum), 1GB RAM, OS X 10.3
- Mac Powerbook G3 (built-in Firewire), 384MB RAM, OS X 10.3
- UltraSparc E420 (271 Soda), 4-processor 450MHz, 4GB RAM, Solaris 8
- SunFire 280R (C199 in Cory), Dual 900MHz, 4GB RAM, Solaris 8
- Desktop, Pentium IV 2.4GHz, 512MB RAM, Windows XP
- Laptop, Pentium III 800Mhz, 512MB, Windows 2000
- Laptop, Pentium IV 2.0GHz, 512MB RAM, Windows XP
- Desktop, Pentium II 333MHz, Windows 98
- Desktop, Celeron 1.8GHz, 512MB RAM, Windows XP
- Laptop, Pentium III 750MHz, 512MB RAM, Windows 2000
- Desktop, Pentium III, 1GHz, 256MB, Windows 2000

Judging by the results, we recommend that users have at least 256MB of RAM and a processor running at at least 333MHz.

## 1.4 API

We used JavaDoc to generate an API in HTML format. The API can be found in the directory `API`.

## 1.5 SUBGROUPS

We worked mainly within four subgroups although near the end many of us crossed boundaries to aid in building and debugging the final application.

<b>GUI</b>	<b>Gong, Li, Murray, Shkolnikov</b>
<b>Security</b>	<b>Le, Nguyen</b>
<b>GIF</b>	<b>Berdahl, Lee</b>
<b>JPEG</b>	<b>Chung, Mozaffarian</b>

# 2 REVISIONS TO THE DESIGN AND FUNCTIONALITY SPECIFICATIONS

## 2.1 GUI REVISIONS

- The output image from the encoding process is fundamentally different than the input image. Thus, we decided that we didn't like the functionality of a user overwriting an input image with an output image during encoding. In order to implement this change, we simply removed the `save` function from the file menu. Now the encoded image can only be saved with `Save As...`
- We decided to streamline the interface by displaying the progress bar in the lower right-hand corner of the screen rather than popping up a dialog.
- One ambiguity in the Design Specification also needed to be clarified. We could have produced the result of the image key hash in the text key pane, but we decided that ambiguities could arise if the user then modified the text key but not the image key. Thus, we decided to define the text key and image key as two separate entities. Whichever pane is on top will be used for encoding or decoding unless it is empty, in which case, the reverse pane will be used if it contains content.

## 2.2 SECURITY REVISIONS

- According to the *Design Specification*, we should have used CRC to check the integrity of the received message. However, we wanted to make the encrypted text as short as possible, so we used the lower half of the MD5 hash instead.
- To help support for GIF transparency, the function `long setMaxCapacity(long[][] pixelArray)` was added. It allows the identification of pixels that are transparent and in which data should not be stored. This allows the GIF algorithm to easily skip over transparent pixels.
- The parameters were also removed from the constructors. It does not need any information about the progress bar because the JPEG and GIF modules handle the progress bar.

## 2.3 GENERIC GIF/JPEG REVISIONS

- Some of the exceptions were renamed although they retained the same meanings. Furthermore, a few exceptions were added. `AWTException` and `IOException` needed to be added due to methods in Java's API that threw exceptions. The exceptions `TextEmptyException` and `FileNotFoundException` were also added to make it easier for the GUI to handle the corresponding errors.
- A parameter was also added to the `encode()` function. Originally, we thought that we would pass the input text along inside of the `StegoSecurity` object. However, it later became clear that the amount of space available in the input image needs to be conveyed first. Since this information would only be available once the `encode()` function was invoked, we decided to pass the text along to `encode()`, which would pass the text to `StegoSecurity` after initializing the maximum space of the input image.
- The progress bar `JProgressBar` was added to the constructor to make it possible for the GIF and JPEG modules to update the progress bar accordingly.

## 2.4 GIF REVISIONS

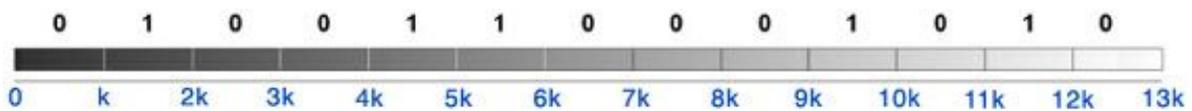
- All changes made to `GenericStegoImage` as described in section 2.3.
- Our implementation of GIF encoding used geometric Euclidean distance to find the closest matching color. This best match is generated only once, after the whole `BufferedImage` is processed. However, even if the image is saturated with 256 colors, this scheme may fail in extreme cases, where all the color's least significant bit ends in either 0 or 1. Then, encoder cannot find a match for any color in the colormap. Thus, GIF encoder can embed messages in GIF images with both odd and even colors.
- Implemented Extension: Often, GIF file's colormap does not use all 256 colors. When this is the case, our current version adds a color to colormap to accommodate GIF files with only odd or even colors. Current model only adds color when the search for the match fails for the

first time. Current version adds a color only to preserve the algorithm and withholds from filling up the color palette to maximum 256 colors.

- Modification: GIF file format prohibits multiple transparency colors. There can be at most one transparent color in the colormap. Thus, encoder may either convert transparent pixel to opaque pixel or exclude the transparent region as unwritable. Our module selected the latter scheme, and ignores the transparent region. However, this extra layer of representation of the data has increased the computational loads, since `opaqueMap[]` now maps each writable or opaque pixel to 2 integers, column and rows of corresponding pixel in `pixelMap[][]`.

## 2.5 JPEG REVISIONS

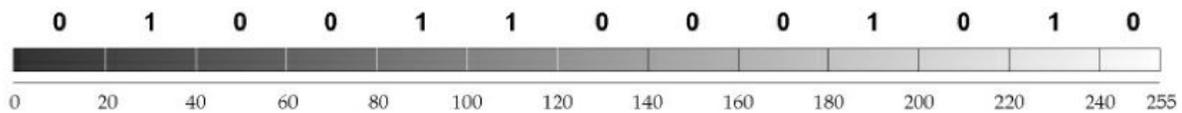
- All changes made to `GenericStegoImage` as described in section 2.3.
- At the beginning of the encoding process before squishing occurs, we found it helpful to carry out an additional step. We write the input image to the hard disk using our default JPEG compression quality and then load the image from the disk. Although this increases the amount of time required, it makes it possible for us to introduce most of the JPEG distortion before the process begins. The distortion produced during the final writing to disk is thus minimized—making our embedded information more robust.
- Furthermore, we needed to deal with the fact that a small subset of images cannot have information encoded in them with a given key. The chances of this occurring are small (see the section “Fine-Tuning `JpegStegoImage`“), and the class of images for which it occurs is not very useful—mostly images containing noise. However, we do not want to run the risk of supplying an output image, which supposedly contains information but cannot be encoded properly. As a result, at the end of the encoding process, we also attempt to decode the output image. If decoding fails, or the decoded information isn’t correct, then we throw an `UnsuitableImageException`. The user is informed to try again with another input image. By adding this step, we ensure that the user receives correct output images.
- We also changed the format of the stego-table so that all of the regions are the same size. The paper we were referencing<sup>1</sup> seemed to suggest cutting the last region short, but we found that the algorithm delivered more predictable results if all of the regions were the same size. This means that depending on the number of regions, the width of the regions may not be integer values anymore. However, this does not present a problem since we are working with floating point numbers at this point in the algorithm. See the figures below to better understand the difference.



Stego-table where the regions have uniform widths (our implementation).

---

<sup>1</sup> Yeuan-Kuen Lee and Ling-Hwei Chen. „Secure Error-Free Steganography for JPEG Images.“ Dept. Of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Rd., Hsinchu 30050, Taiwan, R.O.C.



Stego-table where the last region is cut off (Lee and Chen).

## 3 TESTING

### 3.1 GUI TESTS

Since testing for the GUI was not automatic, but rather carried out by multiple humans on multiple machines, it made more sense to print out a series of functionality and fault tests for the GUI. Several of us carried out these fault tests writing down the results on paper. In the process we found bugs and were able to alert those working on the GUI. As a result, the tests that were carried out later by humans resulted in fewer and fewer bugs being found. Please consult the paper hand-in for more information.

### 3.2 SECURITY TESTS

All tests harnesses can be found in `StegoTest.java`.

#### Test : Encryption Unit (See p. 45 of the *Design Specification*)

##### Test 1: Long plaintext (see `StegoSecurityTest.html`)

##### Test Approach

Encrypt and Decrypt text files of size 0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5MB in size. Compare the original `Strings` with the result `Strings`. Check if exceptions are thrown properly.

##### Goal and Result

To test how well this unit deals with `Strings` of different sizes.

- Removed limitation of 65535 bytes on plaintext, since `max{Int}` is around 2 million.
- MAXIMUM data this `StegoSecurity` can hold (in `cipherText` format) is **16777215 bits** or **2097151 bytes** (after compression and some overhead)

- 100% success rate
- Did throw error appropriately when compressed text is still larger than `maxCapacity` or when data is corrupted (one bit is flipped) or when the key is wrong.
- Run time increase linearly with plaintext size (2MB took about 16 sec)
- However, when `maxCapacity` is big enough, 5MB did not encrypt & decrypt correctly.
- Reason: 5MB was compressed down to about 3,064,372 bytes, while 24 bit to store `cipherLength` (in bits) only goes up to 2,097,151 bytes
  - absolute limit for `maxCapacity` is 0xFFFFFFFF bits or 2,097,151 bytes =>

CannotDecryptException was thrown.

The correctness of the algorithm was tested as follow:

- Get input from each of the file
- Encrypt in one `StegoSecurity` object (`steg1`)
- Copy bit by bit to another `StegoSecurity` object (`steg2`)
- Call `decrypt` on this object
- Then compare the `Strings`

**Test: Image to Key Conversion Unit (See p. 45 of the *Design Specification*)**

**Test 2: Deterministic Conversion (See `ImageKeyConversion.html`)**

### Test Approach

Take a pool of 100 images (different color set, size, and detail intensity) and convert each of these images to text key 100 times.

### Goal and Result

Check if every time we convert an image to a text key, we obtain the identical text key.

- 100% successful rate.
- For every image used as key, 100 hashing yield the same unique key hash.

**Test 3: Variation in Keys Produced (See p. 45 of the *Design Specification*)**

(See `ImageKeyVariation.html` and `Equations.doc`)

### Test Approach

Generate keys from 100 different images and compare the keys produced to see if they have acceptable amount of variation. We went over each bit to find the standard deviation of each bit and each byte. Also we compare each of the 100 keys against each other and observe the bit-wise differences between them.

### Goal and Result

To measure the obscurity ramifications of using hashed images for keys.

- All 100 keys generated from 100 images are considered **pseudo-random**
- The standard deviation of the bits is around 0.5 so we can see none of the bits are biased.
- Maximum number of same bits between 2 keys are 82 (considered safe since 128 of the same bits is a collision, meaning 2 images are hashed to the same key)
- The average bit-wise differences is around 64.1 (half of 128, as it should be since each bit only have 2 values zero and one)
- The standard deviation of numbers of same bits is 3.9, which mean the number of same bits between any 2 keys is CONSISTENTLY close to 64 as it should be.
- The distribution of the bytes are is probably uniformly distributed. We already know that the MD5 hash distributes the bytes already relatively well. Furthermore, few of the keys collide, and the standard deviation of the bytes is the same as the standard deviation for a uniform distribution  $128/\sqrt{3} \approx 74$ . At the end of section 3.2, we show that the standard deviation of a uniform distribution over  $[-128, 127]$  is roughly  $128/\sqrt{3} \approx 74$ .

**Test 4: Large Key Image (See pp. 45-46 of the *Design Specification*)**  
(See `LargeKeyImage.html`)

**Approach:**

Convert images of size 1, 5, 10, 15, 20 MB in size.

**Goal and Result:**

To test how well this unit deals with large file sizes.

Image files that are over 1MB don't work well because the buffer image size is too large and cause an `OutOfMemoryError`. (It only works with the smallest image in this test which is 2211x1202)

**Scope:**

This test is applied to the function that takes an image and returns the text key representing an image.

**Test 5: Huffman Unit**

**Extractability: Compression / Decompression (See p. 46 of the *Design Specification*)**  
(See `HuffmanUnit.html`)

**Test Approach**

Take 100 passages of text of different size, compress and decompress them. File inputs are in English, Spanish and other European languages and non-ASCII characters.

**Goal and Result**

Every time after we compressed a passage of text we could decompress it.

- 100% success rate.
- The text we received after decompression was 100% identical to the text that was compressed.
- We can compress/decompress any file size.
- The adaptive Huffman Algorithm is already known for saving the most space.

We want to know what the standard deviation of a uniform distribution is.

Let  $P_x$  be the power,  $\sigma_x$  be the standard deviation, and  $\mu_x$  be the mean of the signal  $x$ .  $p_x(x)$  is the probability density function (PDF) of  $x$ , which is shown in Figure S.1.  $x$  is uniformly distributed between  $a$  and  $-a$ . Since the area under the PDF must be 1, the height of the rectangle in  $p_x(x)$  is  $\frac{1}{2a}$ .

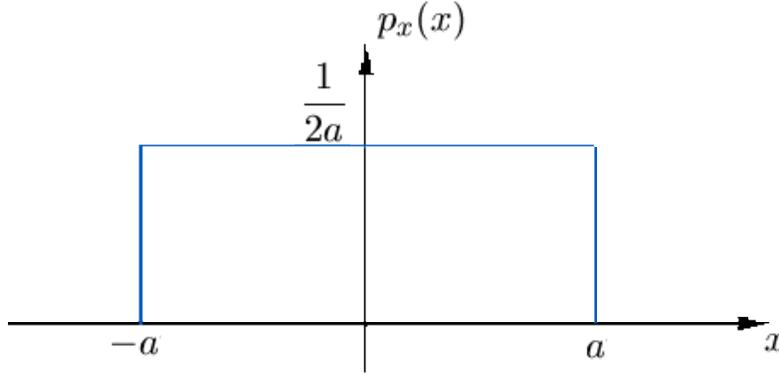


Fig. S.1. Probability density function (PDF) for a uniformly distributed signal  $x$ .

Since  $x$  is zero mean,  $\mu_x = 0$ . We can then use  $P_x = \sigma_x^2 + \mu_x^2$  to show that  $P_x = \sigma_x^2$ . Now it is easy to find  $\sigma_x$  from the definition of power.

$$\begin{aligned}
 P_x &= \int_{-\infty}^{+\infty} x^2 p_x(x) dx = \frac{1}{2a} \int_{-a}^{+a} x^2 dx \\
 &= \frac{1}{2a} \left[ \frac{x^3}{3} \right]_{-a}^{+a} = \frac{1}{2a} \left[ \frac{a^3 - (-a^3)}{3} \right] = \frac{a^2}{3} \\
 \sigma_x &= \frac{a}{\sqrt{3}}
 \end{aligned}$$

By substituting  $a \approx 128$ , we find that  $\sigma_x \approx 74$ .

### 3.3 GIF TESTS

We mainly used our test harness `GIFBatchTest.java` to test our module. It automates the process of encoding a group of gif files with randomly generated key and text and of decoding the encrypted image files to extract back the text and compare the results. The internal tests are all documented within the source code, but particular tests will be referenced in this document.

#### Test 1: Random Key and Random Stress Test

We gathered 100 test GIF images of various sizes and from various sources. Pictures from digital cameras and images from the internet, as well as personal archive of gif images were gathered. We stored them in the directory `gif_test_files` along with a few other files generated from the test harness. We made sure that all of the images were large enough to contain at least one character of text in addition to the header information. Random texts were generated for each test based on the size of the input image.

The output images from the final iteration of each approach can be found in the same `gif_test_files` directory with respective names like `output(x).gif`, where (x) refers to image number. Nicely formatted version of test results can be found in the

file\_text\_results.html.

### **Approach 1. (Using random text keys)**

Each image was tested 100 times using a randomly generated text key. Again,  $100 * 100 = 10,000$  encodings and decodings were carried out to determine if there was any data loss. Each iteration is saved under `files_text_results(x).html`, where (x) refers to the iteration number.

### **Approach 2. (Using each image as key)**

Each image was tested using each of the images as a key. That is,  $100 * 100 = 10,000$  encodings and decodings were carried out to determine if there was any data loss. Each iteration is saved under `image_key_results(x).html`, where (x) refers to the iteration number.

### **Goal**

Unlike the JPEG format, GIF file's encoding and decoding process is more deterministic. We strived to handle all cases including small, huge, transparent, opaque, and all combinations except the case of intended message being too large to be contained in the image.

### **Results**

We have reached the goal. The test was satisfactory with 0 error. GIF encoder failed only on cases where the intended text was near the estimated capacity of the file. It properly displays the error message and throws the `TextTooLongException`.

## **Test 2: Max Capacity Test**

### **Test Approach**

Java Virtual machine is platform and hardware dependent, however through these tests, we wish to provide certain metric guides. Our max capacity test was run on Windows XP over Pentium 2.0 GHz with 512 MB of memory.

### **Approach**

By first selecting a huge gif file, Java virtual machine is forced to display `OutOfMemory` error. Then, I would lower down the size of the file to estimate the maximum capacity of the file. If the maximum is found with small text message, then text message is gradually increased to offer a different set of metrics of capacity measurement.

First we selected a gif image with near 5 million pixels (2560 x 1920), and worked downward.

### **Result**

Working down from 5 million pixels and 80000 bits of data, the encoder was able to handle 2.77 million pixels (1925 x 1444). When the text message was lengthened, the java virtual machine failed. After the image was lowered to 2.2 million pixels (1700 x 1275), the encoder was able to hide 440,000 bits of information

### Hints for selecting good GIF images for embedding:

- Image must not be too huge, refer to the max capacity.
- Image with huge transparent region is not recommended, since the writable area is limited relative to the dimension of the GIF file.
- Images should contain more than 4 colors, more the better.
- In order to make the embedded text to be less apparent, select images with wide pool of similar colors.

## **3.4 JPEG TESTS**

We mainly used our test harness `JpegTest.java` to test our module. It is capable of carrying out roughly 10 different automated tests. The internal tests are all documented within the source code, but particular tests will be referenced in this document. The test numbers in `JpegTest.java`, which also correspond to the test numbers used in the directory structure of the submission, are given in parenthesis because they differ from those given in the *Design Specification*, which are given at the beginning of each title.

### **Test 1: Minimization of the Bit Error Rate (See p. 48 of the *Design Specification*)**

We gathered 100 test JPEG images of various sizes and from various sources. Pictures from digital cameras and images from the internet, as well as artificially generated images such as gradients and noise were gathered. We stored them in the directory `InImages` along with a few other files necessary for the test harness. We made sure that all of the images were large enough to contain at least one character of text in addition to the header information. Random texts were

generated for each test based on the size of the input image. The lengths of the texts were chosen to require modifying roughly every second 8x8 block of each image.

The output images from the final iteration of each approach can be found in the respective `OutImages` directories. Script of the outputs sent to the terminal can be found in the respective `Log.txt` files, and more nicely formatted versions can be found in the `JpegTestOutput.html`, although it has proven to be very difficult to open a 4.5MB HTML file that only contains a table. Thus, we recommend looking in `Log.txt` instead.

### **Approach (Part 1, Test 9 in the directory structure)**

Each image was tested using each of the images as a key. That is,  $100*100 = 10,000$  encodings and decodings were carried out to determine if there was any data loss.

The failure rate was quite small: 0.23%

### **Approach (Part 2, Test 8 in the directory structure)**

Each image was tested 100 times using a randomly generated text key. Again,  $100*100 = 10,000$  encodings and decodings were carried out to determine if there was any data loss.

The failure rate was quite small: 0.22%

### **Goal**

We achieved our goal. Almost all of the tests were successful. A small percentage of the images could not be encoded, but as explained below in “Fine-Tuning JpegStegoImage,” this was a necessary trade-off so that the encoded images did not look suspicious. For each image that cannot be encoded, an `UnsuitableImageException` is thrown, so that the GUI can inform the user. We originally thought that this exception happens mostly only in cases where the input image is particularly grainy—such as especially in the case of using pure white noise as an input. The files `Gaussian.jpg`, `GaussianGrayscale.jpg`, `Uniform.jpg`, and `UniformGrayscale.jpg` had caused the most problems. We had had a failure rate of about 3%, but we decided this was too high for our application and decreased  $z$  slightly. Due to this reduction, we reduced the failure rate to roughly 0.2% as shown above. Now although certain pictures are more likely to fail, it is no longer clear what they have in common. However, we think that whatever characteristic these images possess, it is probably similar to noise in some sense.

The images that caused problems more than once during the final 200,000 tests with the decreased  $z$  were `usual.jpg`, `SolidColors.jpg`, `newimage3.jpg`, and `dec2003-10.jpg`. It was not obvious to us exactly which characteristic these files possessed that made them different from the others, but determining this factor would be very difficult. Only one failed 8x8 block is required to destroy the entire encoding since the MD5 hash checksum implemented by `StegoSecurity` will cause an exception to be thrown. This also explains why these images work some of the time but not all of the time. Particular keys can influence the results.

## Test 2: Minimal Picture Distortion (See p. 48 of the *Design Specification*)

### Test Approach

We knew that the so-called “blocking“ effects were what we needed to avoid. Since we already had experience comparing input and output images, we simply investigated the contents of the `OutImages` directory from the previous test to make sure that the effects were only occasionally noticeable to eyes trained to see them.

### Goal

The goal was realized. Most images did not contain discernable traces of the “blocking“ effects. Those that did look slightly suspicious contained large monotonic surfaces that could be easily broken up by small changes in the DC values of the 8x8 blocks. However, on p. 3 of the *Design Specification*, we allowed for occasional problematic output. To help the user select good input images, we make the following suggestions. This information is also available in the online help system.

### Hints for selecting good JPEG images for embedding:

- Large digital photographs generally work well for the following reasons.
- The images should have rather large dimensions. 64 pixels are needed to encode one bit of information, and the header is about 20 bytes. As a result, if the dimensions of a given image are  $v$  by  $h$  pixels, then about  $(vh/64)/8 - 20$  characters can be stored. For example, a 400 x 400 image can store roughly 290 characters.
- Images should contain no large blocks of one color. Otherwise, the embedded information may be visible to the human eye. Pebbles on a beach or a group of buildings in a city would work well.
- The encoding may also not work for certain images. However, these images would generally not be used for steganography anyway. For example, images containing large variations in small regions such as pure noise are likely to cause problems.

## 3.5 FINE-TUNING JPEGSTEGOIMAGE

Consult pp. 24-9 of the *Design Specification*, for more detailed information about how the JPEG steganography algorithm works. There are a few crucial values that need to be set properly to optimize the behavior of the algorithm. The most important value is  $z$ .

Let the integer  $z$  be the number of entries in the `StegoTable`. In Fig. J.1.  $z = 13$  because there are 13 entries spanning the space between 0 and 255. If  $z$  is too small, then the presence of hidden information will be obvious. On the other hand, if  $z$  is too large, then sometimes information could possibly be lost. That is, since the information would be embedded in very small changes in the image, the following JPEG compression stage could damage the information. The tests below show that  $z = 70$  is roughly optimal for our purposes.

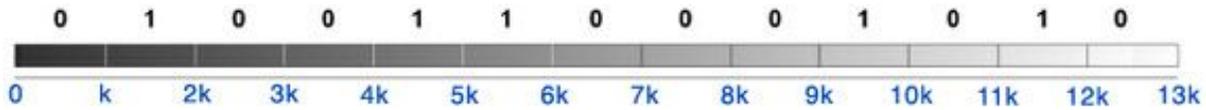


Fig. J.1. Stego-table with  $z = 13$ .

`quality` is also an important parameter. If it is too low, then the hidden information is more likely to be damaged during JPEG compression. However, if `quality` is too high, then third parties viewing the images might be suspicious because JPEGs usually aren't encoded at the highest quality. The tests below show that `quality = 0.8` is roughly optimal for our purposes.

The final parameter we experimented with is  $w$ , the standard deviation of the Gaussian noise we add to the image. This noise can help hide the embedded information. If there is too much noise, then the image will look grainy, but if there is too little, then the so-called “blocking“ effects are more likely to be visible. That is, third parties might notice that the borders between the 8x8 JPEG blocks were accentuated. The tests below show that  $w = 8$  is roughly optimal for our purposes.

Here we present some tests showing that we picked approximately optimal values. We vary one of the parameters at a time while leaving the others set to their defaults to show what the effect is. We also made .MOV files out of the output images to make it easier to visualize the outputs. QuickTime is required to view these movie files.

Before jumping into the tests themselves, it's also important to review the definition of the signal-to-noise (SNR) ratio. Because there were too many images to judge them all subjectively by humans, we used the more objective SNR measurement to help us understand the results. The definition utilizes logarithms to avoid exponents in the SNR parameter.  $P_u$  is the power of the input signal, and  $P_f$  is the power of the error, which can be described as the power of the difference between the input and output signals.

$$\text{SNR} = 10 \log (P_u/P_f) \text{ [dB]}$$

Lower signal-to-noise ratios are worse because it means that more noise is present. We thus optimized to increase the ratios as much as possible.

### Determining An Optimal Value For $z$ (Test 4 in the directory structure)

#### Test Approach

The file `InImages/TextToEncode.txt` containing 392 characters was encoded into `InImages/Lena.jpg`, a 512x512 color image. The default parameters for JPEG stenography were used except that  $z = 25:10:225$  (MATLAB notation for values from 10 to 225 in increments of 10). The output images can be found in `OutImage`, and a movie of them can be found at `Tweak z.mov`. A script of the output sent to the terminal can be found in `Log.txt`, and a more nicely formatted version can be found in `JpegTestOutput.html`.

#### Results

As expected, the encoding was successful for lower  $z$  values, while data loss resulted when using greater  $z$  values. (Here data loss means any failed encoding.) The border for this

particular image was about  $z = 125$ , but we chose the default value to be significantly smaller to stay on the safe side when using other images.

Smaller  $z$  values caused more noise to be added to the original image. We measured the noise using the SNR, but we did not include the values for  $z$  values resulting in data loss since we wouldn't be able to use them anyway. It can be seen that especially small values for  $z$  tend to greatly increase the noise, while the SNR levels off near higher values. Our choice of 70 is a good compromise: the SNR couldn't get much better at higher  $z$  values, but it is low enough to reduce the chance of encoding problems. These results can be seen in Fig. J.2.

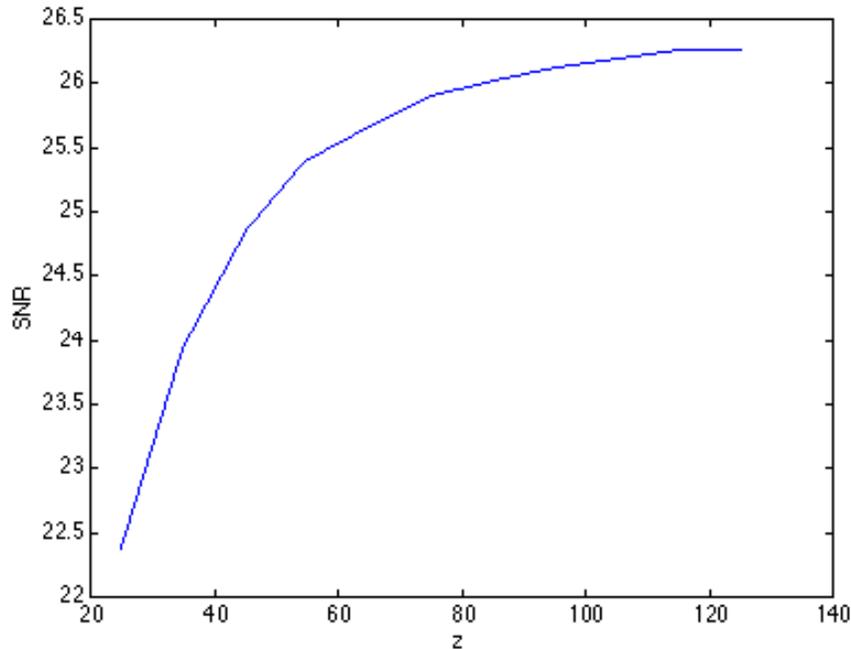


Fig. J.2. Increasing  $z$  causes the SNR to increase, although it levels off at higher values.

### Goal

We achieved our goal of finding a good value:  $z = 70$ .

### Determining An Optimal Value For *quality* (Test 5 in the directory structure)

#### Test Approach

The file `InImages/TextToEncode.txt` containing 392 characters was encoded into `InImages/Lena.jpg`, a 512x512 color image. The default parameters for JPEG stenography were used except that `quality = 0.00:0.05:1.0`. The output images can be found in `OutImage`, and a movie of them can be found at `Tweak quality.mov`. A script of the output sent to the terminal can be found in `Log.txt`, and a more nicely formatted version can be found in `JpegTestOutput.html`.

#### Results

Data loss resulted for `quality` values less than 0.5. This behavior is about what we expected. When more compression is used, then the image is corrupted so much that the hidden information is lost.

In Figure J.3, we plotted the SNR versus `quality` for the values that succeeded for the input image. At first the SNR increases with the quality as expected; then however, the SNR drops. We attribute this to the DC coefficient quantization tables being offset too far from the centers of the regions in the `StegoTable`. That is, the two quantizations are offset so that the input image is distorted more than necessary. Luckily, we wanted to use a lower quality setting anyway. Most JPEGs aren't compressed with the highest quality. For example, it would look suspicious for us to always use the highest quality JPEG compression setting. Additionally, it is important to note that the SNR really doesn't vary too much in the shown region anyway. Thus, our choice will not effect the SNR heavily.

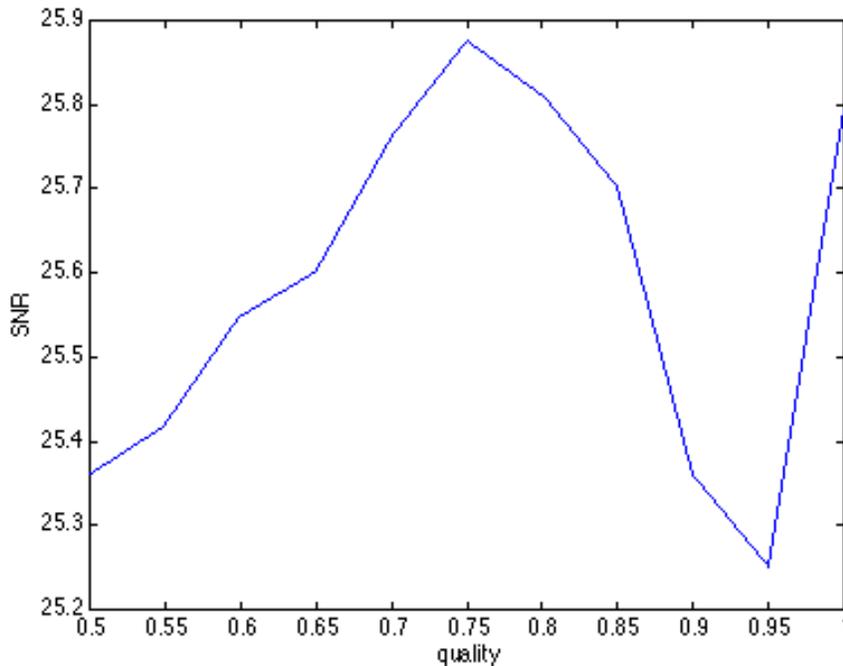


Fig. J.3. The shape of the curve is somewhat surprising, but the SNR really isn't varying that much so we don't need to worry too much about the SNR with respect to `quality`.

### Goal

We achieved our goal of finding a good value: `quality = 0.80`.

### Determining An Optimal Value For $w$ (Test 6 in the directory structure)

#### Test Approach

The file `InImages/TextToEncode.txt` containing 392 characters was encoded into `InImages/Lena.jpg`, a 512x512 color image. The default parameters for JPEG steganography were used except that  $w = 0:2:40$ . The output images can be found in `OutImage`, and a movie of them can be found at `Tweak quality.mov`. A script of the output sent to the terminal can be found in `Log.txt`, and a more nicely formatted version can be found in `JpegTestOutput.html`.

#### Results

We inspected the output images by hand to try to find an acceptable amount of noise. We found that starting at about  $w = 8$  there was enough noise to cover up the "blocking" effect

without making the image noticeably grainy. In addition, we plotted the SNR versus  $w$  just to verify that we were adding noise properly. This was the case. The more noise we added, the worse the SNR became. The plot in Figure J.5 illustrates why we did not want to add too much noise.

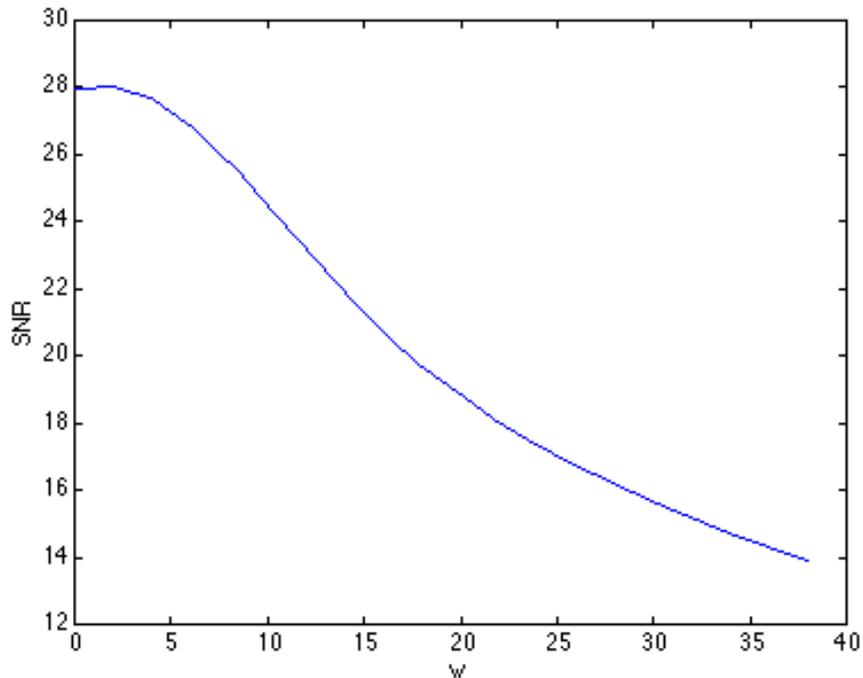


Fig. J.5. Adding more noise causes the SNR to drop.

### Goal

We achieved our goal of finding a good value:  $w = 8$ .

## 4 KNOWN BUGS

While we tried to eliminate as many bugs as possible, there are still a few bugs left that have not been resolved yet. However, these bugs have little significance with regards to the functionality of the program.

### 4.1 GUI BUGS

- The behavior of the progress bar is sometimes a little bit finicky. Following an exception or an encoding/decoding process, the progress bar is not always reset properly to 0%. Multiple files are affected.

## 4.2 SECURITY BUGS

- If the key image is too large, an `OutOfMemoryError` will be thrown. This behavior has been observed using images on the order of 1000 x 2000 pixels. Since errors of this sort usually cannot be caught, we recommend that the user simply use normally-sized images as input.

Affected file: `StegoSecurity.java`

## 4.3 GIF BUGS

- (None known.)

## 4.4 JPEG BUGS

- Part of the test harness seems to be platform dependent. The function for calculating the signal-to-noise ratio (SNR) sometimes needs to have the horizontal and vertical dimensions reversed so that the test code does not crash on rectangular images for different platforms. We did not resolve the issue because we had no problems using the SNR to find the optimal parameters at home, while the final application will not need to call the SNR code.

Affected file: `CompareImages.java`