ELE 201, Spring 2014
Laboratory No. 4
Compression, Error Correction, and Watermarking

1 Introduction

This lab focuses on the storage and protection of digital media. First, we'll take a look at ways to generate an equivalent (or near-equivalent) representation of a signal by discarding redundant or unimportant information. Next, we'll look at how we can guard against the influence of random errors in the storage or transmission of data by adding in some redundant (but highly structured) information. Finally, we'll look at ways to visibly or invisibly embed one signal into another for the purposes of verifying ownership.

2 Lab Procedure

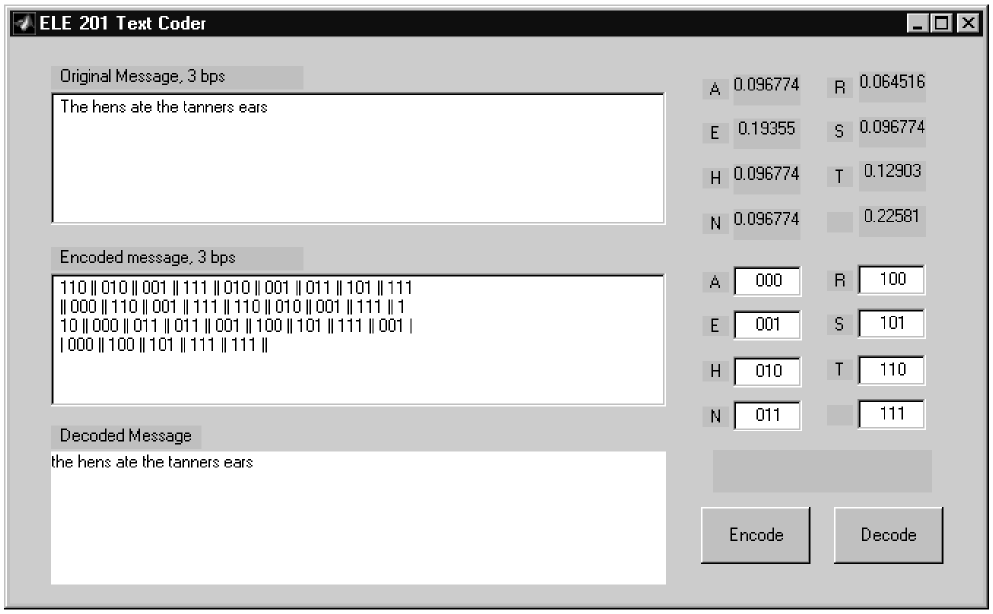
2.1 Compression

You've no doubt come across files with the extension `.zip`, which are compressed versions of larger files. Depending on the type of data, the size of the compressed version could be as little as 1 or 2 percent of the size of the original version, yet all the data is preserved! This tells us that there's often a lot of redundancy in signals, and that this redundancy can be removed to make shorter descriptions for the purpose of storage or transmission.

We'll start out by looking at text compression, in which it's important that the compressed message contain all the information required to reconstruct the original message exactly (this is called *lossless compression.*) Then we'll look at image compression, where a large portion of information can actually be removed without severely degrading the perceptual quality of the image (this is called *lossy compression.*)

2.1.1 Text compression

We'll take a look at text compression using the ELE 201 Text Coder, invoked by typing `coder` at the Matlab prompt. For this example, we'll restrict our attention to 7 frequently used letters of the alphabet, plus the space character. You can type an original message in the top box, and binary codewords in the eight boxes on the right-hand side. When you hit the "Encode" button, the alphabetic message is transformed into 0's and 1's using the code that you entered. When you hit "Decode", the binary message is converted back into text. Double bars (||) show how the binary string was parsed to produce the output. The interface checks to make sure that the code is a *prefix code*, i.e. that no codeword begins with any other codeword.



Original Message, 3 bps

```
The hens ate the tanners ears
```

Encoded message, 3 bps

```
110 || 010 || 001 || 111 || 010 || 001 || 011 || 101 || 111
|| 000 || 110 || 001 || 111 || 110 || 010 || 001 || 111 || 1
10 || 000 || 011 || 011 || 001 || 100 || 101 || 111 || 001 |
| 000 || 100 || 101 || 111 || 111 ||
```

Decoded Message

```
the hens ate the tanners ears
```

A	0.096774	R	0.064516
E	0.19355	S	0.096774
H	0.096774	T	0.12903
N	0.096774		0.22581

A	000	R	100
E	001	S	101
H	010	T	110
N	011		111

Figure 1: ELE 201 Text Coder

For this demo, we'll be using the following text string, which you should type into the top box:

$\boxed{\text{Annette and Deanne the heathens sedated Nathan the tense hen}}$

1. Encode the message using a set of eight 3-bit binary codewords. That is, use the eight possible 3-digit binary combinations. (Note that you can use the Tab key to jump to the next box.) What is the average number of bits per symbol for the encoded message using this code? $\boxed{\text{Q1}}$

Now, we'll take a brief look at the robustness of this code to errors. First, try flipping one of the bits in the encoded message (e.g. changing a 0 to a 1) and hitting "Decode" again. What happens to the decoded message and why? Next, instead of flipping a bit, insert a new bit somewhere in the middle of the decoded message and hit "Decode". What happens to the decoded message and why? $\boxed{\text{M1}}$ $\boxed{\text{D1}}$

2. Next, encode the message using a variable-length binary code. A good choice is the Huffman code you learned about in class. You can figure out what the Huffman code should be using the frequencies of each letter which appear in the upper right corner of the window. $\boxed{\text{M2}}$

   Include in your writeup the codeword you used for each symbol. What is the average number of bits per symbol for the encoded message, using this code? (This is computed for you above the Encoded Message window.) How does this compare with the average number of bits per symbol for the best possible block code, assuming the letters are independent of each other with these frequencies? (Hint: entropy!) $\boxed{\text{Q2}}$ $\boxed{\text{D2}}$

   Again, let's look at the robustness of this code to errors, using the bit flip and added bit. What happens in each case? Which code is more robust to bit-flipping? Which code is more robust to the added bit? $\boxed{\text{D3}}$

2.1.2 Image compression

From browsing the web, you've probably come upon different image file formats: `.bmp`, `.gif`, `.jpg`. These formats differ in how the image data is compressed and stored. In this section, we'll concentrate on the JPEG (Joint Photographic Experts Group) format. The way it works is (roughly):

1. Break the image into 8 x 8 blocks.
2. For each block, take the two-dimensional Discrete Cosine Transform (DCT). This is similar to the Fourier Transform, except the result is always real.
3. Quantize each DCT coefficient to some value. In general, this is a non-uniform quantization so that large values stay pretty much the same and small values get mapped to 0.
4. Code these quantized DCT coefficients by zig-zag scanning and using a combination of Huffman and run-length coding.
5. Store the coded data in a file.

To decompress the image, each image block is reconstructed by decoding to obtain the quantized DCT coefficients and taking the inverse DCT. Since the quantized coefficients are different from the original coefficients, the reconstructed image is not the same as the original. However, by using shrewd choices of quantization, the two images should look the same to our eyes.

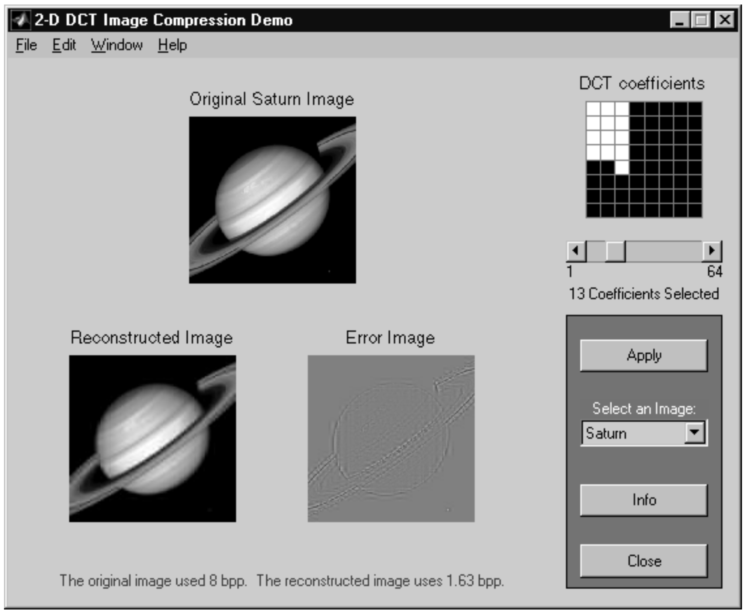
In this demo we'll explore a JPEG-like image compression scheme. The difference is that instead of quantizing all of the DCT coefficients, we'll just set some of them to 0 and transmit the rest. Start the demo by typing `jpegdemo` (this one is actually constructed around a built-in Matlab demo, which is why it looks so nice and works so well!)

The chart at the upper right of the window shows which coefficients are zeroed out (i.e. the black ones). The coefficients are zeroed out in order of increasing variance, generally creeping from the lower right hand corner (high frequencies in both directions) to the upper left corner (low frequencies in both directions).

Start out with the Saturn image and try moving the slider around to select fewer and fewer coefficients (remember to hit "Apply" each time). At what point do you feel like the reconstructed image is noticeably different from the original? How many bits per pixel are you using at this point? What does this tell you about the redundancy of the image? At what point do you think you would notice the reconstructed image had been processed in some way (without comparing it to the original)? $\boxed{\text{M3}}$ $\boxed{\text{D4}}$

Now answer the same questions for the Quarter image. Are your answers different? Why? $\boxed{\text{D5}}$

Finally experiment with the Line Art image, which consists of single-pixel-wide black lines on a white background. Why do you think JPEG compression works so poorly in this example? $\boxed{\text{D6}}$



13 Coefficients Selected

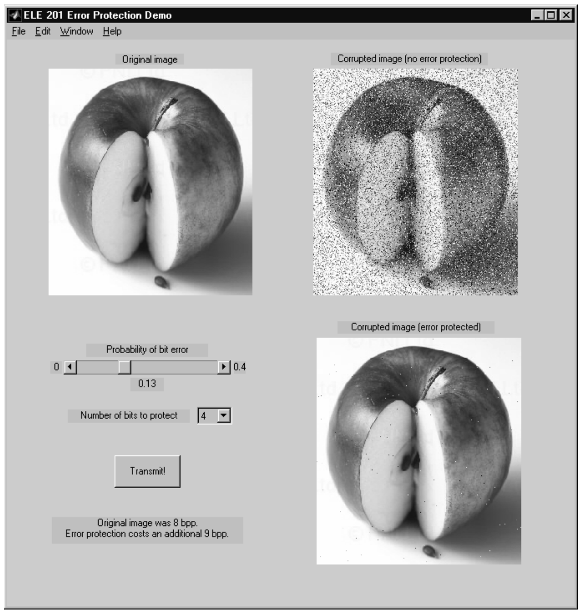
Select an Image: Saturn

The original image used 8 bpp. The reconstructed image uses 1.63 bpp.

Figure 2: Ripped-off JPEG demo

2.2 Error Correction

As you saw in the text coding section, small errors in the transmission of a message can make large errors at the receiving end. For this reason, it's standard to introduce some structured, redundant information into a signal to be transmitted so that the effects of errors can be fixed or mitigated. For example, when you bang your CD player, you introduce errors in the bits the laser reads from the CD that's playing, but with a good CD player, you shouldn't notice a difference in sound quality.


ELE 201 Error Protection Demo
File
Edit
Window
Help
Original image
Corrupted image (no error protection)
Corrupted image (error protected)
0
0.4
Transmit!

Probability of bit error: 0.13

Number of bits to protect: 4

Original image was 8 bpp.
Error protection costs an additional 9 bpp.

Figure 3: Error protection simulation

We won't get into the details of error detection and correction here but the results are simulated in a demo you invoke by typing `eprot`. At the upper left is an original image, coded using 8 bits per pixel. The slider controls the probability that during transmission, one of these bits is independently flipped. The corrupted image is shown at the upper right when you hit "Transmit". However, by paying a certain price (expressed as an expansion factor), you can protect the image against 1, 2, or more of these bit flip errors per pixel.

A math question: suppose that the original image is 300 x 270 pixels and that each pixel has 8 bits. Let the probability that a bit is corrupted during transmission be 0.01. On the average, how many pixels in the transmitted image have at least 1 bit error? You need to use probability for this question, i.e. compute $\boxed{\text{Q3}}$

$$(\text{\# of pixels})(\text{probability there is at least 1 error out of 8 bits})$$

Note this is the same as

$$(\text{\# of pixels})(1 - \text{probability there are exactly 0 errors out of 8 bits})$$

How many pixels in the transmitted image have at least 2 bit errors? Again, compute $\boxed{\text{Q4}}$

$$(\text{\# of pixels})(\text{probability there are at least 2 errors out of 8 bits})$$

Does this agree with what you obtain using the simulator? Keep in mind that a bit error at the Most Significant Bit (i.e. the one that controls whether the intensity is more or less than 128) makes a bigger perceptual difference than a bit error at the Least Significant Bit (which controls whether the intensity is even or odd). $\boxed{\text{Q5}}$

In fact, this points out the fact that some bits are more important than others, and that we might want to spend more on protecting these important bits. However, in this demo we protect all bits equally.

Hold the error-protection level steady at 1 bit, and increase the bit error probability. What happens to the transmitted images? You should be able to reach a point where the transmitted image looks pretty much the same in spite of the error protection. Why and roughly where does this phenomenon occur? $\boxed{\text{D7}}$

Now, spend a minute or two experimenting with error-protecting more bits and varying the bit error probability. Be sure to note how much you pay to protect more bits! Specifically, hold the error-protection level at 2 bits and vary the bit error probability. How does this differ from what you saw when only 1 bit could be corrected? What happens at the extreme case, when you have the power to correct 5 bit errors? $\boxed{\text{D8}}$

Suppose we compressed an 8-bit image by 50% using an algorithm like JPEG, and then we error-protected 2 bits per pixel, which expanded the data by a factor of 2. Then this compressed, protected image would also use roughly 8 bits per pixel. Which image would you choose for the purposes of transmission, the original or the compressed/protected one? $\boxed{\text{D9}}$

2.3 Watermarking

The word *watermarking* originated as an attribute of fancy paper. At the paper mill, a design would be pressed onto the paper while it was still wet. When the paper dried, this design would be visible if the paper was held to the light, but unobtrusive otherwise. Official Princeton documents are generally on watermarked stationary.

The term *digital watermarking* has been adopted to talk about the same kind of embedding process for audio and images. In the age of the Internet, the process of copying digital media is easy, quick, entirely accurate, and virtually unsupervised. This means you can download illegal copies of music and movies from around the world, or pull any image you like off the web and put it on your home page (even claiming it as your own!). Of course, the decent and good-hearted of us never do such things. But there are many nasty folks out there, and the people who produce and distribute digital media are increasingly concerned about the security of their information. Examples include the Napster and DeCSS lawsuits and the Hack SDMI challenge.

A digital watermark is a signal that is embedded in a media clip that is difficult to remove by anyone but the person doing the embedding. People use watermarks for different purposes, which include:

- Determining whether the data has been changed in any way
- Verifying ownership of data
- Authenticating data
- Hiding secret messages in data
- Captioning data
- Tracing illegal copies of data

2.3.1 Visible watermark

When the Vatican library put selected documents from their archives on-line a few years ago, they hired IBM to put a visible watermark on the images so that people could enjoy or study them but not use them in their own publications. We can do a similar thing in Matlab, using the command `visible`.

First, load the file **lab4.mat**, and take a look at the images *monkey* and *message*. (For the image viewing in this section, use the `imshow` command.) We'll embed this message image in the monkey image by taking the monkey pixels which correspond to white pixels of the message and darkening them by a certain amount. The effect should be an image that looks pretty much the same as the original, but with a visible "stamp" on it. The syntax for `visible` is:

```
≫ visible(monkey, message, factor);
```

where *factor* is the factor by which to darken the pixel colors. When *factor* is 1, the image is unchanged. If *factor* is less than 1, the message pixels get darker; if it's greater than 1, the message pixels get lighter.[1] Try the following values for *factor*: $\{.95, .9, .7, .5, 1.2\}$. A good watermark should be visible enough that you couldn't pass off the image as unmarked, but unobtrusive enough that the original image can still serve a useful purpose. Which value of *factor* gives a nice trade-off between the original image and visibility of the watermark? Feel free to experiment with other values. $\boxed{\text{M4}}$ $\boxed{\text{D10}}$

2.3.2 Invisible watermark

For obvious reasons, visible watermarks often detract from the usefulness of the carrier media, and the question of how to embed watermarks invisibly into images is an important research topic. You can do this in many ways, but here we'll just address one approach in the frequency domain. This scheme can be used to embed a binary image one-eighth the size of the original in each direction.

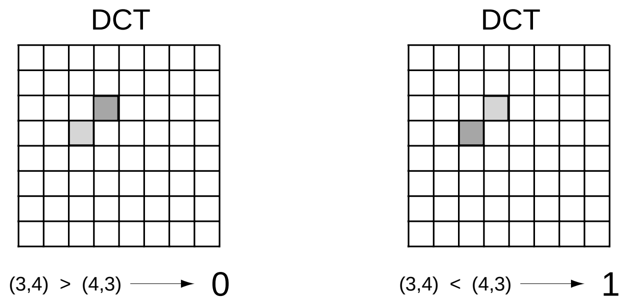


Figure 4: Embedding an invisible watermark

[1] The scheme we're using is: if *watermark(x,y)=1*, then *image(x,y)=image(x,y)*factor*.

1. Break the image into 8 x 8 blocks.
2. For each block, take the two-dimensional Discrete Cosine Transform (DCT).
3. If the watermark bit is a 0, switch the (3,4) and (4,3) entries of the DCT if necessary so that the (3,4) entry is larger. If the watermark bit is a 1, switch the (3,4) and (4,3) entries of the DCT if necessary so that the (4,3) entry is larger.
4. Take the inverse DCT of the block and replace this block in the original image.

This scheme is “secret” in the sense that unless you knew the magic coefficients were (3,4) and (4,3), you wouldn’t be able to find and remove the watermark.

You can add an invisible watermark to the monkey using the command

```
≫ wmonkey = invis2(monkey,lilmark);
```

Take a look at *wmonkey*. Can you see any differences? $\boxed{\text{Q6}}$

You can extract the watermark using the command

```
≫ verify2(wmonkey)
```

Finally, we’ll take a look at four monkey images, *monkey1* through *monkey4*, which should look more or less the same. These images are:

A. An image that first had noise with amplitude 75 added to it, and was then watermarked

B. An image that was first watermarked, and then had noise with amplitude 50 added to it

C. An image that was first watermarked, and then had noise with amplitude 75 added to it

D. A noisy image that has no watermark in it at all

Your job is to match images 1 through 4 with descriptions A through D above, using reasoning about the way watermarks might degrade with noise. In your write-up, present your conclusions and detailed reasoning! $\boxed{\text{D11}}$