Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Department of Mechanical Engineering

6.050J/2.110J **Information and Entropy** **Spring 2006**

Issued: February 14, 2006 **Due: February 17, 2006**

Problem Set 3

Laboratory Assignment

In 6.050J/2.110J, MATLAB will be used frequently in problem sets and demonstrations. MATLAB is a mathematics software package that is efficient at matrix operations as well as a good tool for data visualization. Access MATLAB on any Athena workstation by typing `add matlab` at your `athena%` prompt. Then type `matlab &` to enter the program. A separate window will appear, in which you can enter MATLAB commands.

Whenever a problem requires MATLAB, write a text file that holds the commands or functions that you would normally type into the MATLAB command line. This text file, also known as an M-file, should have a `.m` extension (e.g., `filename.m`). This file can be executed in the MATLAB window by typing the filename without the `.m` extension. This takes away the drudgery of retyping commands over and over again. Programs like vi, textedit, emacs, and pico can be used to create and edit the M-file on Athena. Moreover, we also request that you type `diary` (in MATLAB) which keeps track of all your commands and outputs from MATLAB. The results are placed in a file called diary. Please create a separate M-file for each problem and edit the diary for readability.

Go through the MATLAB Tutorial (http://www-mtl.mit.edu/Courses/6.050/2006/notes/matlab.pdf) given out in class to familiarize yourself with the syntax and environment. Remember that at any time, you can type `help` at the MATLAB prompt (before typing this, type `more on` to turn on page-by-page viewing). It is our intention that MATLAB be used as a helpful tool in this course, and that it not be a barrier to anyone's success in the course. We also hope that you'll develop skill with MATLAB which may be useful to you in other classes. To that end, we're here to help you if you run into problems, so feel free to ask us for assistance. If you have any questions, please email 6.050-staff@mit.edu.

Problem 1: Is it Over-Compressed or is it Modern Art?

This exercise demonstrates some of the basic ideas behind image compression. You will write your own video image compressor in MATLAB![1] This will be as simple as possible, so don't sweat. Write all your MATLAB commands in `ps3p1.m`. For simplicity the images you will work with in this exercise have no color. Each pixel is represented by a number: 0 if the pixel is black, 1 if the pixel is white, or a number between 0 and 1 for shades of gray. Thus a picture is represented by a matrix (a two-dimensional array) of numbers.

Usually pixel values are represented by a small number of bits, often 8 bits, in which case only a finite number of shades can be represented. For the purpose of this exercise you may assume that the shades are represented by real numbers, so any shade can be represented.

Most modern video compression algorithms use a form of the Discrete Cosine Transformation (DCT). The original image is broken up into blocks, in our case 8 pixels high and 8 pixels wide. For each block, the DCT is applied to the matrix of pixel values. The result is another matrix of the same size, with values

[1] The 2.110/6.050 staff would like to thank Joe Huang, the Spring 2000 class TA, for developing the image compression exercise used in this assignment. It is, in our opinion, really cool because it truly links what we discuss in class to the real world; once you've completed it, you will understand and have experience with the way JPEG images are created. For this reason, the problem has not changed much since last few years, and we are trusting you to complete it without consulting last year's solutions. We hope you enjoy it!

(DCT coefficients) that are numbers, not necessarily in the range from 0 to 1. One of these coefficients is (to within a scale factor) the average value of all 64 pixels. Other coefficients describe the variation of shade across the block.

The DCT is reversible, in the sense that the original image can be calculated exactly from the matrix of coefficients. No information is lost, but in turn no compression has occurred because the DCT coefficients require as many bits as the original image. What irreversible compression algorithms such as JPEG do is discard small DCT coefficients, thereby saving on the number of bits needed. When it is time to render the image, the inverse DCT transformation is applied, and the result, which is not quite the same as the original image, is displayed. If the choice of which coefficients to discard is done well, the changes in the image as perceived by the human eye are minimal.

In MATLAB, type `dctdemo` to view a demonstration of the DCT. Click on "Info" for detailed information. Here's a brief description of what is going on.

1. The original image you choose is divided into 8 x 8 blocks of pixels.
2. DCT is applied to each block. The resulting matrix of coefficients has at the upper left the coefficient that gives the average. Coefficients down and to the right measure how rapidly the pixel values change, i.e., whether the block has pixels with high spacial frequencies. For example, the coefficient at the lower right is large only if the block had pixels with sort of a checkerboard pattern. If the image is relatively smooth over the block in question, only a few coefficients toward the upper left will be significant.
3. Compression occurs when small values of DCT coefficients are set to zero. You can do this by moving the horizontal slider to block out certain values and then clicking "Apply". MATLAB will render the resulting compressed image.
4. The error image shows the result of subtracting the original pixel values from the reconstructed values.

Play with this demonstration long enough to observe the trade-off between compression (e.g., the percentage of coefficients discarded) and visual fidelity.

a. Start off by loading the vertigo image by typing the following in MATLAB.

```
load imdemos vertigo;   % Load vertigo matrix from image demos library
vertigo=double(vertigo);% Convert the matrix to double precision
colormap('gray');       % Grayscale for any pictures you want to display
                        % (This creates a blank window that we'll use soon)
imshow(vertigo,[0 255]);% Display vertigo in the window
```

b. Perform a 2D DCT operation on 8×8 blocks of the image matrix. You might find the commands `blkproc` and `dct2` useful. Use the first form of `blkproc` explained in its help page (`help blkproc`), where FUN is `'dct2'` (put single quotes around dct2). To keep help information from scrolling off the screen, type `more on` at the MATLAB prompt.

c. Set to zero all coefficients with absolute value less than 10. An example to do this can be found in the help of dct2. We will later want to vary this cutoff value and count the number of non-zero values. After this step, many codecs apply run length and variable length encoding to transmit or store the image efficiently for later use.

d. Perform a 2D Inverse DCT operation on each of the 8×8 blocks of the image matrix. The values of the resulting matrix will be in floating point, but you should round the elements to the nearest integer since image pixel values are integers. Use the `blkproc` and `idct2` commands similarly to

the way you performed the forwards DCT in Part b. Then use the `round` command to complete the reconstruction of the image.

e. Determine the mean squared error between the original and reconstructed image. The definition of mean squared error to use is `mean2((x - y).^2)`. Values will appear larger than in the demo because our error definition does no normalization. The answer you should receive is 10.2970. Note that you can see your new image with the `imshow` command the same way you displayed the original image. If you want a new window for the image, so it doesn't just paint over the previous one, type figure at the MATLAB prompt.

Now compose a graph, having along the x-axis the number of non-zero values of the matrix after the cutoff procedure (before you did the inverse DCT), and the mean squared error (after you did the inverse DCT and rounding) on the y-axis. The number of non-zero values is the number of bytes required to store the image (ignoring overhead), so smaller means more compression. Range the cutoff value from 0 to 100 in increments of 4. The use of `nnz` and `plot` will do the trick. It will be easiest to use them if you create a vector with the numbers of non-zero values and another vector with the mean squared errors.

Write in `ps3diary` the largest byte size for which you can detect the difference between the original image and the reconstructed image by eye. Include any comments in the diary file. Be sure the script in `ps3p1.m` is executable in MATLAB.

Problem 2: Compression is Fun and Easy

Now you can get your hands dirty with the LZW (Lempel-Ziv-Welch) compression algorithm. Although this problem will not require the use of MATLAB, you may use it if you wish. There are many derivatives of LZW that are more powerful, but we'll stick to what was explained in lecture and in the notes.

a. Use the 7 bit ASCII table that can be found in Chapter 2 of the notes for your basic alphabet. Notice that the ASCII table ends at HEX `7F` (Hexadecimal is Base 16 using A - F for values 11 - 15, so HEX `7F` is 127). Your dictionary will therefore range from HEX `80` through HEX `FF`. Encode the following message in HEX using the LZW technique:

`yubba dubba dubba dubba dubba doo`[2]

Be sure that you can decode the message properly. Use the examples in Section 3.2 of the notes as a guide for formatting your solution although some details would be different as the example uses DEC instead of HEX. Put your results in ps2diary and include your dictionary entries. Note: HEX `20` is the space. For this exercise use HEX `03` (ETX) to signify the end of the message and HEX `02` (STX) for the start of the message and note that these are not the same as the ones of the example in the notes.

b. **Optional (just for fun):** Implement the LZW coding and decoding operations of part (a) above, using any computer language you know and have access to such as Matlab. Assume that any message you handle will be short so that the dictionary will not extend beyond HEX FF (naturally a robust implementation would need to handle dictionary overflow). If you have such an implementation, it would be easy to try to encode longer phrases such as

`yubba dubba doo, yubba dubba dubba doo, yubba dubba dubba dubba dubba doo.`

---

[2] Chosen in honor of "Fred Flintstone"!

Turning in Your Solutions

Name the M-file `ps3p1.m` and name the diary `ps3diary`. You can rename your files at your `athena%` prompt using the following command.

```
mv oldfilename newfilename
```

Turn in this problem set by e-mailing your M-files and diary along with your answers to any problem(s) not done using MATLAB, to 6.050-submit@mit.edu. You may do this either by attaching them to the e-mail as text files, or by pasting their content directly into the body of the e-mail (if you do this, please somehow indicate where each file begins and ends). Alternatively, you may turn in your solutions on paper in Room 38-344. The deadline for submission is the same no matter which option you choose.

Your solutions are due 5:00 PM on Friday, February 17, 2006. Later that day, solutions will be posted on the web.