C.2 Images

The purpose of this lab to explore images and colormaps. You will create synthetic images and movies, and you will process a natural image by blurring it and by detecting its edges.

C.2.1 Images in Matlab

Figure C.2 shows a black and white image where the intensity of the image varies sinusoidally in the vertical direction. The top row of pixels in the image is white. As you move down the image, it gradually changes to black, and then back to white, completing one cycle. The image is 200×200 pixels so the vertical frequency is 1/200 cycles per pixel. The image rendered on the page is about 10×10 centimeters, so the vertical frequency is 0.1 cycles per centimeter. The image is constant horizontally (it has a horizontal frequency of 0 cycles per centimeter).

We begin this lab by constructing the Matlab commands that generate this image. To do this, you need to know a little about how Matlab represents images. In fact, Matlab is quite versatile with images, and we will only explore a portion of what it can do.

An image in Matlab can be represented as an array with two dimensions (a matrix) where each element of the matrix indexes a colormap. Consider for example the image constructed by the `image` command:

```
>> v = [1:64];
>> image(v);
```

This should create an image like that shown in figure C.3.

The image is 1 pixel high by 64 pixels wide (Matlab, by default, stretches the image to fit the standard rectangular graphic window, so the one pixel vertically is rendered as a very tall pixel.) You could use the `repmat` Matlab function to make an image taller than 1 pixel by just repeating this row some number of times.

The pixels each have value ranging from 1 to 64. These index the default colormap, which has length 64 and colors ranging from blue to red through the rainbow. To see the default colormap numerically, type

```
>> map = colormap
```

To verify its size, type

```
>> size(map)

ans =

    64     3
```

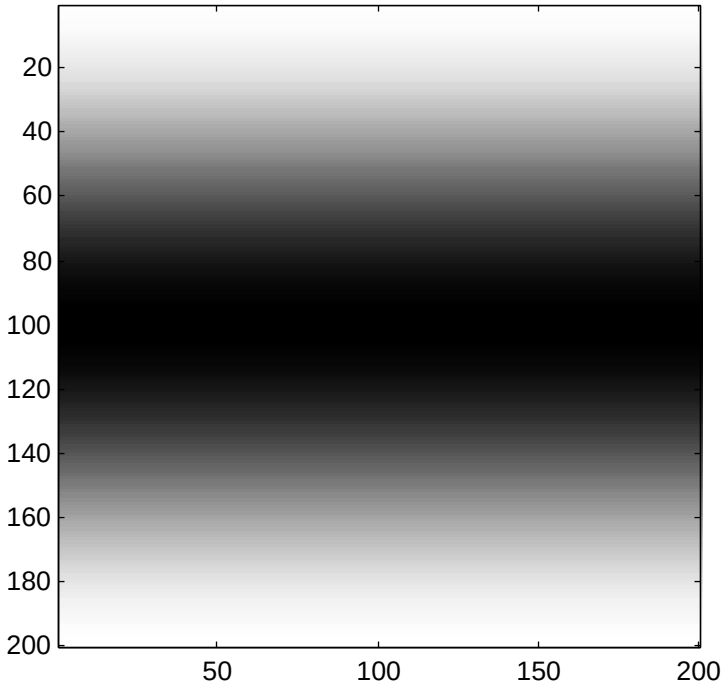

Figure C.2: An image where the intensity varies sinusoidally in the vertical direction.

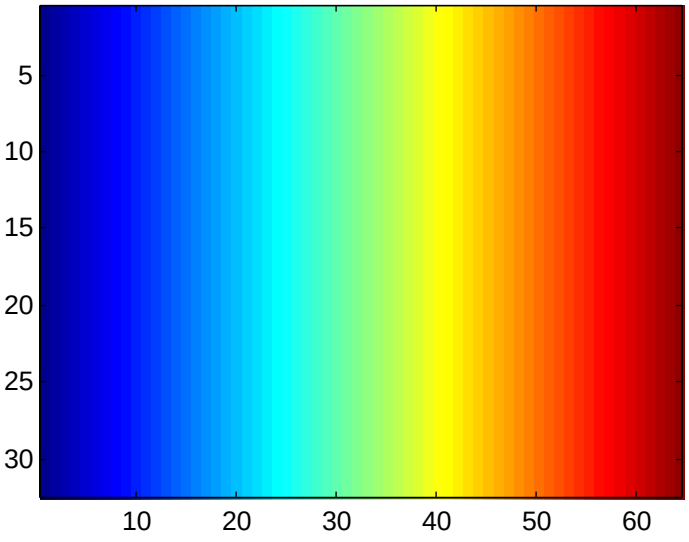

Figure C.3: An image of the default colormap.

Notice that it has 64 rows and three columns. Each row is one entry in the colormap. The three columns give the amounts of red, green, and blue respectively in the colormap. These amounts range from 0 (none of the color present) to 1.0 (the maximum amount of the color possible). Examine the colormap to convince yourself that it begins with blue and ends with red.

Change the colormap using the `colormap` command as follows:

```
>> map = gray(256);
>> colormap(map);
>> image([1:256]);
```

Examine the `map` variable to understand the resulting image. This is called a **grayscale colormap**.

C.2.2 In-lab section

1. What is the representation in a Matlab colormap for the color white? What about black?

2. Create a 200×200 pixel image like that shown in figure C.2. You will want the colormap set to `gray(256)`, as indicated above. Note that when you display this image using the `image` command, it probably will not be square. This is because of the (somewhat annoying) stretching that Matlab insists on doing to make the image fit the default graphics window. To disable the stretching and get a square image, issue the command `axis image`

   ```
   axis image
   ```

   As usual with Matlab, a brute-force way to create matrices is to use for loops, but there is almost always a more elegant (and faster) way that exploits Matlab's ability to operate on arrays all at once. Try to avoid using for loops to solve this and subsequent problems.

3. Change your image so that the sinusoidal variations are horizontal rather than vertical. Vary the frequency so that you get four cycles of the sinusoid instead of one. What is the frequency of this image?

4. An image can have both vertical and horizontal frequency content at the same time. Change your image so that the intensity at any point is the sum of a vertical and horizontal sinusoid. Be careful to stay with the numerical range that indexes the colormap.

5. Get the image file from

   `http://www.eecs.berkeley.edu/~eal/eecs20/images/helen.jpg`

   Save it in some directory where you have write permission with the name "helen.jpg". (**Note**: For reasons that only the engineers at Microsoft could possibly explain, Microsoft Internet Explorer does not allow you to save this file as a JPEG file, '.jpg'. It only allows you to save the file as a bit map, '.bmp', which is already decoded. So we recommend using Netscape rather than IE.)

   In Matlab, change the current working directory to that directory using the `cd` command. Then use `imfinfo` to get information about the file, as follows:

```
>> imfinfo('helen.jpg')

ans =

           Filename: 'helen.jpg'
        FileModDate: '27-Jan-2000 10:48:16'
           FileSize: 18026
             Format: 'jpg'
      FormatVersion: ''
              Width: 200
             Height: 300
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
```

Make a note of the file size, which is given in bytes. Then use `imread` to read the image into Matlab and display it as follows:

```
>> helen = imread('helen.jpg');
>> image(helen);
>> axis image
```

Use the `whos` command to identify the size, in bytes, and the dimensions of the `helen` array. Can you infer from this what is meant by `'truecolor'` above? The file is stored in JPEG format, where JPEG, which stands for Joint Pictures Expert Group, is an image representation standard. The `imread` function in Matlab decodes JPEG images. What is the compression ratio achieved by the JPEG file format (the compression ratio is defined to be size of the uncompressed data in bytes divided by the size of the compressed data in bytes).

6. The `helen` array returned by `imread` has elements that are of type `uint8`, which means unsigned 8-bit integers. The possible values for such numbers are the integers from 0 to 255. The upper left pixel of the image can be accessed as follows:

```
>> pixel = helen(1,1,:)

pixel(:,:,1) =

   205


pixel(:,:,2) =

   205
```

```
pixel(:,:,3) =

   205
```

In this command, the final argument is ':' which means to Matlab, return all elements in the third dimension. The information about the result is:

```
>> whos pixel
  Name        Size         Bytes  Class

  pixel       1x1x3            3  uint8 array

Grand total is 3 elements using 3 bytes
```

Matlab provides the `squeeze` command to remove dimensions of length one:

```
>> rgb = squeeze(pixel)

rgb =

   205
   205
   205
```

Find the RGB values of the lower right pixel. By looking at the image, and correlating what you see with these RGB values, infer how white and black are represented in truecolor images.

Matlab can only do very limited operations arrays of this type.

C.2.3 Independent section

1. Construct a mathematical model for the Matlab `image` function as used in parts 3 and 4 of the in-lab section by giving its domain and its range. Notice that the colormap, although it is not passed to `image` as an argument, is in fact an argument. It is passed in the form of a **global variable**, the current colormap. Your mathematical model should show this as an explicit argument.

2. In Matlab, you can create a movie using the following template:

```
numFrames = 15;
m = moviein(numFrames);
for frame = 1:numFrames;
   ... create an image X ...
   image(X), axis image
```

```
   m(:,frame) = getframe;
end
movie(m)
```

The line with the `getframe` command grabs the current image and makes it a frame of the movie. Use this template to create a vertical sinusoidal image where the sine waves appear to be moving upwards, like waves in water viewed from above. Try `help movie` to learn about various ways to display this movie.

3. We can examine individually the contributions of red, green, and blue to the image by creating **color separations**. Matlab makes this very easy on truecolor images by providing its versatile array indexing mechanism. To extract the red portion of the `helen` image created above, we can simply do:

```
red = helen(:,:,1);
```

The result is a 300×200 array of unsigned 8-bit integers, as we can see from the following:

```
>> whos red
  Name      Size         Bytes  Class

  red     300x200        60000  uint8 array

Grand total is 60000 elements using 60000 bytes
```

(Note that, strangely, the `squeeze` command is not needed whenever the last dimension(s) collapse to size 1.) If we display this array, its value will be interepreted as indexes into the current color map:

```
image(red), axis image
```

If the current colormap is the default one, then the image will look very off indeed (and very colorful). Change the colormap to grayscale to get a more meaningful image:

```
map = gray(256);
colormap(map);
```

The resulting image gives the red portion of the image, albeit rendered in black and white. Construct a colormap to render it in red. Show the Matlab code that does this in your report (you need not show the image). Then give similar color separations for the green and blue portions. Again, showing the Matlab code is sufficient. **Hint**: Create a matrix to multiply pointwise by the `map` matrix above (using the `.*` operator) to zero out two of its three columns. The `zeros` and `ones` functions might be useful.

4. A moving average can be applied to an image, with the effect of blurring it. For simplicity, operate on a black and white image constructed from the above red color separation as follows:

```
>> bwImage = double(red);
>> image(bwImage), axis image
>> colormap(gray(256))
```

The first line converts the image to an array of doubles instead of unsigned 8-bit integers because Matlab cannot operate numerically on unsigned 8-bit integers. The remaining two lines simply display the image using a grayscale colormap.

Construct a new image where each pixel is the average of 25 pixels in the original image, where the 25 pixels lie in a 5×5 square. The new image will need to be slightly smaller than the original (figure out why). The result should be a blurred image because the moving average reduces the high frequency content of a signal, and sharp edges are high frequency phenomena.

5. A simple way to perform edge detection on a black-and-white image is to calculate the difference between a pixel and the pixel immediately above it and to the left of it. If either difference exceeds some threshold, we decide there is an edge at that position in the image. Perform this calculation on the image `bwImage` given in the previous part. To display with the edges, start with a white image the same size or slightly smaller than the original image. When you detect an edge at a pixel, replace the white pixel with a black one. The resulting image should resemble a line drawing of Helen. Experiment with various threshold values. **Hint**: To perform the threshold test, you will probably need the Matlab `if` command. Try `help if` and `help relop`.

   Note: Edge detection is often the first step in **image understanding**, which is the automatic interpretation of images. A common application of image understanding is **optical character recognition** or **OCR**, which is the transcription of printed documents into computer documents.

   The difference between pixels tends to emphasize high frequency content in the image and deemphasize low frequency content. This is why it is useful in detecting edges, which are high frequency content. This is obvious if we note that frequency in images refers to the rate of change of intensity over space. That rate is very fast at edges.

Instructor Verification Sheet for C.2

Name: _____ Date: _____

1. Representation in a colormap of white and black.

   Instructor verification: _____

2. Vertical sinusoidal image.

   Instructor verification: _____

3. Horizontal higher frequency image. Give the frequency.

   Instructor verification: _____

4. Horizontal and vertical sinusoidal image.

   Instructor verification: _____

5. Compression ratio.

   Instructor verification: _____

6. Representation in truecolor of white and black.

   Instructor verification: _____