EE 341, Spring 2010

Handout for Lab 2: Introduction to Image Processing

This handout is intended to prevent you from getting stuck on some Matlab issues that you may encounter while working on Lab 2. The explanations in the handout are suggestions only; you don't need to follow the advice, and you don't have to implement your solution according to these hints to get full credit. But you may find this handout helpful if you get stuck.

Learning about the functions used in Assignment 1

The description of Assignment 1 contains the names of all of the functions needed to complete the assignment. To learn more about each function, remember to use the `help` command (for example, type `help imread` at Matlab's command prompt (>>) to learn more about the `imread` command to read an image from a file.

How to display images properly with imshow, even after using conv2

In Assignment 1, you are asked to display several images using the `imshow` command. The `imshow` command is an excellent and very convenient way to quickly display an image in Matlab, but it has one peculiarity: it behaves slightly different for gray-scale images whose pixels are byte-values (0,1,2,…,255) and for gray-scale images whose pixels are floating-point values. For byte-valued image, 0 is mapped to black, 255 is mapped to white, and anything in between is mapped to shades of gray. For floating-point images however, all values less than or equal to 0 are mapped to black, all values greater or equal to 1 are mapped to white, and all values between 0 and 1 are mapped to shades of gray.

When we read in an image from a file with the `imread` command, its pixels will have byte values. After conversion to gray-scale with `rgb2gray`, they will still be byte-values (although now there's only 1 value per pixel, whereas for color images there are 3 byte values per pixel). In some versions of Matlab, when we use an image as input to the `conv2` function, its pixels should be floating-point values or else the `conv2` function will issue a warning and the result will be completely wrong. In this case, it makes sense to convert the image to floating-point values after conversion to gray-scale but before using conv2 on it. We can use the `double` function to accomplish that.

To display such an image properly, we could adjust the values in the image so they will fit in the range from 0 to 1. Or we could tell `imshow` that it should map the smallest value in the image to black, and the highest value in the image to white, instead of using the range from 0 to 1 to map from black to white. The second option may sound difficult at first, but luckily `imshow` has a trick for that. We can use the command

```
imshow(yourfloatingpointimage,[]);
```

The two square brackets, [ ], denote the empty vector in Matlab. Using the empty vector as the second argument to `imshow` tells it exactly what we want: map the lowest value in the image to black, and the highest in the image value to white. See help `imshow` for details on this. In some versions of Matlab, it is recommended to convert your gray-scale images to floating-point values to avoid problems with conv2, and then use the `imshow` command as shown above.

Some hints on scaling

We are going to scale images in Assignment 3. There are many ways to do that. For example, you could use for-loops to accomplish it. But Matlab also has a couple of very powerful ways to index vectors and matrices that allow us to do the scaling very efficiently and in only a few lines of Matlab code. This section illustrates some of those mechanisms by using a simple example.

For instance, let's consider the vector a = [0 1/4 1/2 3/4 1], which we can create in Matlab as follows:

```
>> a = [0, 0.25, 0.5, 0.75, 1];
```

or similarly by typing

```
>> a = [0:0.25:1];
```

We can select individual elements like this

```
>> a(1)
      ans = 0
>> a(3)
      ans = 0.5
>> a(5)
      ans = 1
```

and we can select the last element of the vector by using the end keyword:

```
>> a(end)
      ans = 1
```

We can also select a set of elements at the same time

```
>> a([1 3 5])
      ans = 0 0.5 1
```

which we also could have written as

```
>> a(1:2:end)
      ans = 0 0.5 1
```

The last statement shows how to select every other element from a vector, starting at element 1, and continuing until the last element of the vector. If you're new to this, experiment a little bit with it by using other, perhaps longer, vectors. Try different step sizes, for example a(1:3:end) instead of a(1:2:end). You can try it also for matrices. You will find that this works for matrices too and is a very quick way to select rows and columns of a matrix. You can use this technique when doing the scaling of images.

Hint: put scaling in a function

It may be helpful to write two functions that do the scaling for you. Both functions take an image and a scaling factor as an input, and produce a thumbnail as an output. One of the functions scales images without averaging, and the other function scales the input image with averaging.

The m-file of each function would start with almost identical function definitions, similar to this:

```
function thumbnail = scaleimage(image, scalefactor)
```

and

```
function thumbnail = averagethenscaleimage(image, scalefactor)
```

The body of the functions would perform scaling by `scalefactor` on the image `image`, and assign the scaled image to the variable `thumbnail`.

Hint: test your scaling function

The reason to put the scaling in a function is that it makes it easier for you to test your Matlab code. For example, if your scaling functions were called `scaleimage` and `averagethenscaleimage`, then you could use the following code to verify your scaling algorithms:

```
>> scaletest = reshape(1:36,6,6)
scaletest =
     1  7 13 19 25 31
     2  8 14 20 26 32
     3  9 15 21 27 33
     4 10 16 22 28 34
     5 11 17 23 29 35
     6 12 18 24 30 36
>> scaleimage(scaletest,2)
ans =
     1 13 25
     3 15 27
     5 17 29
>> averagethenscaleimage(scaletest,2)
ans =
     4.5 16.5 28.5
     6.5 18.5 30.5
     8.5 20.5 32.5
>> averagethenscaleimage(scaletest,3)
ans =
     8 26
     11 29
```

It is recommended that you test your scaling algorithms in a similar manner to make sure that they give meaningful results. In the case of scaling without averaging, any of the other sub-matrices (such as `[7 19 31; 9 21 33; 11 23 35]`) is also acceptable.

Hint: fixing the size of the image before scaling may avoid problems

Depending on the input to your scaling algorithm, you may encounter problems that are caused by the fact that the height or the width of the image (or both) are not an integer multiple of the scaling factor. For example, how does your scaling algorithm react to the following input:

```
>> scaletest = reshape(1:20,4,5)
scaletest =
     1  5  9 13 17
     2  6 10 14 18
     3  7 11 15 19
```

```
     4  8 12 16 20
>> scaleimage(scaletest,3)
     ans = ???
>> averagethenscaleimage(scaletest,3)
     ans = ???
```

If your Matlab code gives an error on input like this, or if you want to avoid those errors altogether, you may want to "fix" the size of the image before scaling it. By fixing the image we mean padding it with boundary pixel values on the right and bottom such that the height and width of the image are both integer multiples of the scaling factor. For scaling by 3 in the example above, the padded version of the matrix would look like this:

```
     paddedscaletest =
     1  5  9 13 17 17
     2  6 10 14 18 18
     3  7 11 15 19 19
     4  8 12 16 20 20
     4  8 12 16 20 20
     4  8 12 16 20 20
```

Turning in your work

Follow instructions from the assignment and include all required figures and answers in the report. Please turn in the hard copy report. Please also do not forget to submit the Matlab codes and report via E-Submit.

Good luck!