CS1114: Assignment 1

Issued: Monday 1/30/12 **Due: Friday 2/10/12 by 5PM**

1 Introduction

The purpose of the first assignment is to acquaint you with the basic software that is used in the class as well as to provide you the opportunity to learn about imaging fundamentals and about controlling the robots. In this assignment, we will assume that you have already completed the Matlab introduction that we provided you.

1.1 CS1114 procedure for submitting assignments

In CS1114, you will generally not need to submit code. Instead, when you are done with an assignment you must demo your functions to a TA. You can demo it at any time before the deadline. However, during the last few hours before the deadline they will only check you off if you make an **appointment** with them in advance. The TAs may ask you to demonstrate that you understand any code printed in this lab, and all code that you have written.

1.2 CS1114 procedure for writing code in Matlab

In this course you will often write Matlab procedures by editing a skeleton file that we will provide to you, unless you are asked to write the function from scratch. (Such a file is often called a "stub".) For assignment 1, these files will be in the directory `/courses/cs1114/student_files/A1` on the lab machines. The first thing that you should do is to copy the contents of A1 to your own directory (e.g. `cp -R /courses/cs1114/student_files/A1` /myfiles+).

Places where you need to do something in order to get credit are marked by a paragraph beginning with the sign "$\Longrightarrow$".

2 Thresholding

2.1 Experiment with the thresholding operation

Run `part1_gui`, which will allow you to see the output of the thresholding operation.

(a) Under 'Use Version' select 'Our Version'.

(b) To get an image, you can load one of our examples (Go to Image > Load Image From File and browse to `~/cs1114/student_files/A1`), or you may capture your own image (you can go to Image > Load Image from Camera).

(c) Good threshold values cause the wand, and only the wand, to be included in the output.

2.2 Implement thresholding

$\Longrightarrow$ Edit the `threshold_student.m` file. You will find the following code in the file:

```
function [ out_img ] = threshold_student(in_img, red_threshold, ...
                                         green_threshold, blue_threshold)
 % This function creates an output image which is thresholded by the three
 % values in red_threshold, green_threshold, and blue_threshold.
```

Recall that this code tells Matlab to create a new function called `threshold_student`. As the comment in the code says, this function returns an array which you have named `out_img`. It takes in four arguments: an input image, and the threshold values for red, green and blue. (The "..." simply tells Matlab that you have a statement that spans multiple lines. If the entire function declaration were on one line, then there would be no need for the "...".) You can tell the `part1_gui` program to execute your code by selecting 'Your Version' from the 'Use Version' dropdown box.

2.3 Writing your code

Your goal is to return a binary image with a black background and a white wand. Use the values for `red_threshold`, `green_threshold`, and `blue_threshold` to distinguish between 'red' and the 'background'.

2.3.1 Useful Matlab functions

`A = zeros(x1,x2,x3...)` creates an array 'A' of zeroes with dimensions x1, x2, x3 etc. `x == y` returns 1 if x has the same value as y, and 0 otherwise. Other operators of the same format include `<`, `>`, `<=`, `>=`.

3 Iteration

3.1 Comparing parity

To give you a taste of something completely different, we will now implement a function countSameParity.m. Parity is the quality of being even or odd. For part of this function you should use the `mod` function.

```
mod(x,y)
```

For positive x and y, this function will return the remainder of x/y. So if x is 10 and y is 3, `mod(x,y)`will return 1. How might you use this function to determine parity?

Recall the following code segment from lecture:

```
nzeros = 0;
[nrows,ncols] = size(C);
for row = 1:nrows
    for col = 1:ncols
        if (B(row,col) == 0)
            nzeros = nzeros + 1;
        end;
    end;
end;
```

This code counts the number of zeros in the matrix `C`.

For this assignment, we will count the number of cells whose cell value has the same parity (the quality of being odd or even) as the sum of the row and column value. For example, if the value 7 is in cell [2,1] (so the sum of the indices is 3), then we want to count this value because both 3 and 7 are odd. If the value 8 was in cell [2,1], we would not count it because 8 is even and 3 is odd. A test run might look something like this:

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

countSameParity should count cells [1,2] and [2,2], and return a count of 2.

You will get most of the points on this problem for getting your code to work. The obvious way to do this involves checking parity twice for each cell in the array. We will reserve a small number of points for writing a version of countSameParity that checks parity only once per cell.

$\Longrightarrow$ Write `countSameParity`. You will be implementing this function from scratch.

3.2 Color2Gray

As we discussed in class, intensity (grayscale) images can be represented with a single matrix in Matlab, and color images can be represented with three matrices (one each for red, green, and blue). Suppose we want to take a color image and turn it into a grayscale image (for instance, because we're implementing the "black and white" feature in Photoshop).[1] One way to do this would be to return just one of the color channels, for instance, just the red channel. This might have some unnatural results, however (e.g., a picture of a blue sky might end up looking very dark). How can we combine the red, green, and blue channels of a pixel into a single intensity? One simple way is to compute the *average* of the three color channels for the pixel. Here is an example conversion from color to grayscale:

[1]The opposite of this function, Gray2Color, is much more difficult, but think about how you might solve this problem.

Color image

Grayscale image

Write a function `color2gray` that takes three matrices (the red, green, and blue components of an image—you can assume the three matrices are of the same size), and returns a single matrix (the grayscale image) that is a *weighted* average of the three channels. Your weighting will use 30% of the red channel, 59% of the green channel, and 11% of te blue channel.[2] **You must use for loops to solve this problem**, and your function should work on images of any size. **Note:** because your function will compute weighted averages (resulting in floating point results), you will need to convert an image to double format with the builtin `im2double` function after reading it with `imread`.

You will be implementing this function from scratch. Several sample images (`color1.jpg`, `color2.jpg`, and `color3.jpg`) are included in the `~/cs1114/student_files/A1` directory.

To test your function, you can read in an image, convert it to double format, then extract out the red, green, and blue channels using the provided `image_rgb` function. You can then use the builtin `imshow` function to display the results. For instance:

```
>> C = imread('color1.jpg');
>> C = im2double(C);
>> [red, green, blue] = image_rgb(C);
>> G = color2gray(red, green, blue);   % Your function
>> imshow(G);   % Display the grayscale image
```

Finally, you can save the image to a file using the builtin `imwrite` function. This function takes two arguments: the image itself, and the desired filename:

```
>> imwrite(G, 'gray1.jpg');
```

If you are having trouble getting your image to show, it could be because you didn't convert the original image to double type, as explained in class. Either use `im2double()` on the image after reading it, or convert the output of your dilateImage function using `uint8()` or by dividing all values in your output by 255 so that they are between 0 and 1.

N.B. On the Linux command line, you can use the `display` command to show an image, for instance:

```
(Linux command line)> display gray1.jpg
```

(This is different from the Matlab `display` function.)

$\Longrightarrow$ Write `color2gray` (from scratch).

[2] This is a standard weighting, found at `http://en.wikipedia.org/wiki/Grayscale` (see "Converting color to grayscale). Any idea why green is weighted higher?

3.3 Color2Color

The operation above turns a color image with three channels (represented as three matrices) into a single matrix (an intensity or grayscale image), by taking a weighted average (or *linear combination*) of the three channels. We can extend this by turning the three color channels into three new color channels in an interesting way; in particular, you will write a function that takes a color image, and creates a new color image where each output color channel is a linear combination of the three input color channels. The linear combination will be specified by a 3×3 matrix (remember, matrices are what Matlab is good at), where each row specifies the weights of the linear combination, just like the weights you used above; row one is for the output red channel, row two for the output green channel, and row three for the output blue channel. We call this the "color-mixing matrix." This can be used to create Instagram-style image filters (at least for the simple filters). You should be able to use this `color2color` function as in the following example:

```
>> C = imread('color1.jpg');
>> C = im2double(C);
>> [red, green, blue] = image_rgb(C);
>> M =  [ .393, .769, .189; .349, .686, .168; .272, .534, .131 ];
>> [red2, green2, blue2] = color2color(red, green, blue, M); % Your function
>> I2 = rgb_image(red2, green2, blue2);
>> imshow(I2);   % Display the new color image
```

This would create a new image where the red channel is a mix of 39.3% of the original red channel, 76.9% of the original green channel, and 18.9% of the original blue channel, and the new green and blue channels are similarly computed from the weights in the second and third rows of `M`. (Incidentally, the particular matrix M above creates a nice sepia-toned result.) We provide you with a function `rgb_image` for putting three color channels back into a single image variable as shown above.

$\Longrightarrow$ Write `color2color` (from scratch). It will take red, green, and blue matrices, a color mixing matrix `M`, and will return new red, green, and blue matrices. You can use your `color2gray` function for inspiration. The sample color images above may be useful for testing.

$\Longrightarrow$ Figure out what matrix to pass to `color2color` to swap the red and blue channels, leaving the green channel the same.

$\Longrightarrow$ Create your own interesting color-mixing matrix, and save your result to a file for one of the sample images or your own images. You will show this image to a TA during your demo.

3.4 Image dilation

For this part of the assignment, you will be implementing a simple image processing filter called "dilate." To dilate an image, you replace each pixel's intensity with the maximum intensity in that pixel's neighborhood (where the neighborhood of a pixel is defined as the pixel itself, and it's four neighbors to the left, right, up, and down). For instance, if we have the following 3×3 images:

10	50	80
60	30	50
20	55	40

we would replace the intensity of the center pixel with the value 60. (Note that the neighborhood of a pixel on the boundary of the image is undefined. You can do anything you like with these boundary pixels.) Assume that the input image is grayscale for this function. You can use the image `otter.pgm` in the `~/cs1114/student_files/A1` directory for testing.

$\Longrightarrow$ Write a routine called `dilateImage` which takes in an image, and returns an image dilated as described above. You may find the Matlab built-in routine `max` useful here. `max` takes a vector as input, and returns the maximum element of the vector (for instance, `max([10, 40, 2, 16])` returns 40. A stub has been provided for you (see the file `dilateImage.m`).

$\Longrightarrow$ Next, write a routine called `dilateNTimes` which takes in an image, and an integer ≥ 1, which dilates the image repeatedly n times. This function should call your `dilateImage` function. A stub has been provided for you (see the file `dilateNTimes.m`).

4 Using the Robots

1. Get a robot (a Rovio or an Aibo) from the lab or one of the cabinets, and turn the robot on. Note that you can also use a virtual robot for testing this assignment. A tutorial of how to use the robots will be given in section.

2. Note the name of the robot (it will be on a label somewhere on the robot).

3. On your Matlab command line, type

   ```
   >> r = robotInit('robotname');
   ```

   (where `robotname` is the name of your robot). The `robotInit` function returns a handle to the robot (stored in the variable `r` above) that allows you to issue commands.

4. You should now be able to use the robot commands in Matlab on your computer.

4.1 Troubleshooting

If your robot isn't working, try "pinging" it. ("ping" is a network utility that checks if a host is connected to a network.) Open a Linux terminal, and at the command prompt, type: `ping robotname` where `robotname` is the name on your robot. You should see output that's something like

```
>> Reply from 192.168.3.42: bytes=32 time=18ms TTL=245
```

If you're still having problems, contact a TA.

4.2 Basic Robot Commands

We provide you with several commands to make robots do basic movement (more commands will be introduced in later assignments). We have done our best to make the interface simple and easy-to-use. Be very careful while playing with robots. Here is the current set of available Matlab commands to interact with robots:

1. `robotInit(robotName)` This sets up the robot control environment in Matlab and returns a handle to a robot. You must call this command before calling any of the commands below.

2. `robotDriveStraight(r, velocity, distance)` This makes the robot stored in `r` travel the specified distance, measured in millimeters, with the specified velocity (in millimeters per second). The `distance` and `velocity` arguments must be > 0.

3. `robotTurn(r, degrees)` This makes the robot `r` turn the specified number of degrees clockwise (`degrees` > 0) or counter clockwise (`degrees` < 0).

4. `robotIsDone()` This function will return 1 if the last movement function (A `robotDriveStraight` or a `robotTurn`) that has been issued has finished, or 0 otherwise. This includes the case in which no movement command has been issued.

5 Regular Polygonal Movement

Using the above commands, implement the function `robotPolyMove` in `robotPolyMove.m`. `robotPolyMove(n)` should cause your robot to move clockwise in a regular polygon with n sides of length 2.

You will be implementing this function from scratch.

6 Grading

We will be grading the following functions:

```
threshold_student.m
countSameParity.m
color2gray.m
color2color.m
dilateImage.m
dilateNTimes.m
robotPolyMove.m
```

Most of your grade will be directly related to whether or not the functions work. A small portion of your grade will be derived from how elegant your code is, whether or not you have used comments, if the `help` command displays anything useful when used on your functions.