CS100R: Assignment 2

Issued: 9/06/07 **First Part Due: 9/14/07 by 6PM**

1 Introduction

In the first assignment, you had the opportunity to work with a basic image operation (thresholding). Recall that the goal was to create a binary image where each of the selected pixels satisfied some set of conditions you designed. While hopefully you succeeded in finding red pixels, this wasn't terribly useful by itself.

In this assignment, we are going to experiment with some of the interesting operations that we can perform on binary images. The goal will be to support a useful application: tracking the movement of the wand in the image. We will build on this for subsequent assignments.

Warning: this assignment is much longer than the first one, so start early! The assignment will be split into two parts; the first will be due **Friday, September 14**, and the second will be due one week later on **Friday, September 21**. Don't be fazed by the number of functions (some of them are not very long), but do make sure to plan your time. The course staff is always here to help you and answer your questions, and we encourage you to discuss general ways of approaching the problems among yourselves.

First Part:

```
* 3.1-3.2: min, max, sum, mean
* 5.1: Bounding box
* 5.2: Mid point
* 5.3: Centroid
* 6.3.1: Snail sort
* 6.3.2: plot sorting time against size (matlab plot)
* 6.3.3: Compare sorting time of matlab quicksort with snail sort
```

Second Part:

```
* 7.1: trim_max
* 7.2: trim_min
* 7.3: trim_max_ties
* 7.3: trim_min_ties
* 7.4: trim_max_qsort
* 7.4: trim_min_qsort
* 7.5: compare trimming time between trim_max_ties and trim_max_qsort
```

```
    * 8.1: Robust centroid
    * 8.2: Top Left pixel finding
    * 8.3: Dense Box (Black Diamond)
```

2 More Matlab Functions

2.1 Find

One important function to experiment with in Matlab is the `find` command. You can use the `find` command to determine the locations of all non-zero pixels:

```
>> bimage = imread('m:/wand/wand_threshold.bmp');
>> image(bimage);
>> [nonzero_rows, nonzero_cols] = find(bimage);
```

This will return the row and column indices of all nonzero pixels in two 1-dimensional arrays `nonzero_rows` and `nonzero_cols`, each of the same size. Each pixel will be recorded with the same index in each array – that is, `(nonzero_rows(1), nonzero_cols(1))` will correspond to the first pixel Matlab finds. For example:

```
>> total_nonzeros = length(nonzero_rows) %prints number of nonzero elements
>> length(nonzero_rows) %also prints number of nonzero elements
>> r1 = nonzero_rows(1);
>> c1 = nonzero_cols(1);
>> bimage(r1,c1) % prints the value of the first nonzero element found i.e. 1
```

2.2 Array Iteration and Early Termination

Recall the sum50 function from the Matlab introduction handout, which totals the elements in a 1D array that exceed 50:

```
>> sum50([10 501 3 109])
ans = 610
```

It's sometimes useful to break out of a loop early. Suppose that we want to stop as soon as the total exceeds 200. We can modify our code for sum50 to do this as follows. In a file named `sum50new.m` we put the Matlab code:

```
function [ total ] = sum50new(arry)
   total = 0;
   for i=1:length(arry)
      if (arry(i) > 50)
          total = total + arry(i);
      end;
      if (total > 200)
```

```
        break; % Break out of the loop early.
    end;
  end;
```

The `break` statement causes the `for` loop to exit early, as soon as the value of `total` exceeds 200. Let's try it out:

```
>> sum50new([10 501 3 109])
ans = 501
```

Note that when used inside nested loops, `break` will get you out of the *innermost* iteration.

3 Some Simple Matlab Algorithms to Implement

We would like you to implement some helper algorithms in Matlab. These algorithms should be useful to you when you are working on some of the more difficult algorithms described in next section and in future assignments.

When you are coding the algorithms in next section and in the future labs, look for places where you can use these algorithms to simplify your work. Wherever it is possible to reuse code in this assignment you should seek to do so. **This will be one of the grading criteria**, not only for this assignment, but for all of the assignments. Also, we want you to write the code yourself — don't use any built in Matlab functions unless we have shown you them in lecture!

We have provided stub files for you to place your implementation in. You should copy the `'m:/wand/part2_templates'` directory to your `H:` drive so that you can work on these files. Where we ask you to implement a function, you should do it in the provided file.

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

3.1 `min, max`

$\Longrightarrow$ Implement functions which find the minimum and maximum element in a 1D array. Place your implementation in the `imin_student.m` and `imax_student.m` files.

3.2 `sum, mean`

Implement a function to find the sum of all elements in a 1D array and a function to find the mean of all of the elements. Your `mean` function must use the `sum` function that you write.
$\Longrightarrow$ Place your implementation in the `isum_student.m` and `imean_student.m` files.

4 Part2 GUI and Camera Calibration

For this assignment, you will want to open up `part2_gui`. You will notice that this interface looks much like the interface that we saw for the last assignment, however on the bottom left

there is a button to Start Camera. When you click this button, the GUI will begin capturing from the camera, and will first run the input image through our thresholding algorithm, and then through an algorithm that you choose with the Algorithm Selection drop down box. The results of the algorithm will be shown overlaid on top of the thresholded image.

For debugging, you may find it useful to load a static image (see example images at `'m:/wand/wand_threshold1.bmp'`). As in the last assignment, you can load these from the Image menu. When you make changes to your algorithm, you can click the Re-Run Algorithm button. This will re-run the thresholding algorithm and the algorithm that you have selected from the drop down box.

If you are having trouble getting images that are easy to threshold, you may find that it helps to modify the camera settings to get darker images (this makes things that are bright stand out more). If this is the case, then you may wish to find the web-cam icon in the Windows tray, right click on it, and select Camera Settings. There are a variety of settings that you can modify here, but probably the most important to getting images that are easy to threshold is the Gain Control. You can find this by clicking the Driver Settings button and then going to the Advanced tab. You will have to uncheck automatic gain control, and you can then modify the exposure and gain.

To see how this affects your image, you can use the `preview` command in Matlab. This will allow you to see the output of the Camera in real-time while you adjust the settings.

5 Binary Image Algorithms to implement

We return to a very basic question which motivates this assignment: how can we actually provide a guess about the position of the wand in the image? We're going to look at some techniques used to localize the wand in the image. Recall that we're dealing with binary images, so Matlab will provide us 2-dimensional arrays where each pixel is either true or false (selected or not).

The input to our binary image algorithm will be the output of the thresholding function that we worked on in the last assignment. We will start here with one simple algorithm, and spend the next assignment looking at some more complex algorithms for localizing the wand in the image.

For all of the algorithms that we have provided here, **you can find function templates** in `m:/wand/part2_templates`. You should copy the contents of this folder to your `H:` drive and you should implement the functionality that we prescribe in these files.

5.1 Bounding Box

One of the simplest algorithms that can be used to localize the wand is the bounding box algorithm. The the goal of the algorithm is to find the smallest rectangle that covers all of the pixels in the image.

You can see the bounding box algorithm working by selecting Bounding Box from the Algorithm drop down box in the `part2_gui`.

We would like you to implement your own version of the bounding box algorithm. Your function should have the following declaration:

```
function [ top_row, bottom_row, left_col, right_col ] =
bounding_box_student(bimage)
```

$\Longrightarrow$ Implement your function `bounding_box_student` in the file `bounding_box_student.m`.

5.2 Midpoint of the Bounding Box

Once you have implemented the bounding box algorithm, we would like you to provide a function that finds the geometric center of the bounding box.

```
function [mid_row, mid_col] = midpoint_student(top_row, bottom_row,
left_col, right_col)
```

$\Longrightarrow$ Implement your function `midpoint_student` in the file `midpoint_student.m`.

5.3 Centroid

Implement a function that finds the mean of all the nonzero row and nonzero column values. You do not have to extract these from an image, `part2_gui` will supply the correct vectors to your algorithm.

```
function [centroid_row, centroid_col] = centroid_student(nonzero_rows,
nonzero_cols)
```

$\Longrightarrow$ Implement your function `centroid_student` in the file `centroid_student.m`. You should call the function `mean`, which you implemented above.

6 Sorting

For this part of the assignment, you will implement a sorting method and observe its performance compared to quicksort.

6.1 Matlab Commands: tic-toc

Matlab has in-built functions *tic* and *toc* to estimate running time of your algorithm. Here is an example of how to use them.

```
function out = YourAlgorithm[inputs]
% This starts a stopwatch timer
tic;

...

%your code

...
```

```
% This displays the elapsed time
toc;
```

6.2 Plotting in Matlab

Matlab can plot one variable against another. The two variables have to be vectors of the same length. To plot, you can enter:

```
>> x = [10, 25, 33, 41, 55]
>> t = [1, 2, 3, 4, 5]
>> plot(x,t) %Gives a linear plot.
>> semilogy(x,t) %Gives a logarithmic plot.
```

For more details on `plot` or `semilogy`, use the `help plot` or `help semilogy` commands.

6.3 Sorting and Run Times

Snail sort is a random sorting method. Given an array of unsorted numbers, you randomly permute all the numbers until the array is sorted. There are interesting ways to generate random permutations that can limit or boost snail speed.

```
function [ out ] = snailsort( A )
%SNAILSORT Sort an array, in as dumb a manner as possible.

done = 0; n = length(A); while (done == 0)
    p = randperm(n);
    A = applyperm(p,A);
    if (issorted(A))
        out = A;
        done = 1;
    end;
end;

function [ B ] = applyperm( p, A ) for i = 1:length(A)
    B(i) = A(p(i));
end;
```

`randperm` and `issorted` are built-in matlab functions that you can use.

6.3.1 Snail Sort

$\Longrightarrow$ Implement your own version of `snailsort` in the file `snailsort.m`.

6.3.2 plotting run-times

$\Longrightarrow$ Write a function that plots the run-times for sorting arrays of different lengths using snail sort. Don't make the arrays too long. Snail Sort can be *very* slow. Try it with less than 6 elements to begin with.

6.3.3 Comparing Snail Sort with the Matlab Built-in

$\Longrightarrow$ Then, write a suitable function comparing the running time of snail sort against Matlab's built-in quicksort (`sort`) on random arrays of different lengths and plotting the results. More points will be awarded for a robust comparison. You may want to include features like multiple trials, arrays of different lengths, random array generation, an appropriate Matlab plot of the running time, and anything else you can think of.

7 Fun with `trim`

7.1 `trim_max`

Now you must "remove" the largest few elements of a 1D array of positive numbers, by replacing them with -1. You should implement the function `trim_max_student`, which behaves as below:

```
>> A = [99 50 3 101 64];
>> B = trim_max(A, 2);
>> B
ans = [-1 50 3 -1 64]
```

Important note: for `trim_max_student`, you can assume there are no ties (i.e., every element in the input is distinct). Hint: You might wish to make use of one of the functions you just implemented above.
$\Longrightarrow$ Implement your function `trim_max_student` in the file `trim_max_student.m`.

7.2 `trim_min`

Now let's do the opposite (remove the smallest few elements). As before, the input 1D array will have positive numbers, and you can assume there are no ties. Here is the behavior you want:

```
>> A = [99 50 3 101 64];
>> B = trim_min(A, 2);
>> B
ans = [99 -1 -1 101 64]
```

However, to make life interesting, you are *required* to implement `trim_min_student` using `trim_max_student`. In other words, your code for `trim_min_student` must call `trim_max_student` and use the results.
$\Longrightarrow$ Implement your function `trim_min_student` in the file `trim_min_student.m`.

7.3 Handling ties

Now we need to consider the possibility that our input has ties (we still assume the values are positive). Write new code to handle this case correctly, by removing the largest few elements. When there is a tie, your code must remove the first such element (see the example below). **Hint:** the problem is much easier if you use the `break` statement we just introduced.

Your new code, which will be called `trim_max_ties_student`, should behave as follows:

```
>> A = [99 50 3 101 64 99 3];
>> B = trim_max_ties(A, 2);
>> B
ans = [-1 50 3 -1 64 99 3]
```

Note that the first element with value 99 was removed, not the second.
$\Longrightarrow$ Implement your function `trim_max_ties_student` in the file `trim_max_ties_student.m`.

Now similarly we want you to remove the smallest few elements while handling ties. Your new code, which will be called `trim_min_ties_student`, should behave as follows:

```
>> A = [99 50 3 101 64 99 3 1];
>> B = trim_min_ties(A, 2);
>> B
ans = [99 50 -1 101 64 99 3 -1]
```

Note that the first element with value 3 was removed, not the second.
$\Longrightarrow$ Implement your function `trim_min_ties_student` in the file `trim_min_ties_student.m`. As before, you are required to call `trim_max_ties_student` in your solution.

Finally, you will write a general-purpose `trim` function. The idea is that `trim` will behave like `trim_max_ties` if the second argument is positive and like `trim_min_ties` if the second argument is negative. Thus:

```
>> A = [99 50 3 101 64 99 3 1];
>> B = trim(A, 2);
>> B
ans = [-1 50 3 -1 64 99 3]
>> C = trim(A, -2);
>> C
ans = [99 50 -1 101 64 99 3 -1]
```

$\Longrightarrow$ Implement your function `trim_student` in the file `trim_student.m`. You are required to call both `trim_max_ties_student` and `trim_min_ties_student` in your solution.

7.4 Quicksort

In the earlier problems, you found the smallest few and largest few items in an array and removed them. There is a better and faster way of doing this: suppose our input is sorted and the elements in the array are in increasing order. Then trimming the max and min is just a matter of removing the first few and last few items from the array.
$\Longrightarrow$ Implement this using Matlab's built-in `sort` function. Place your implementation in the `trim_min_qsort_student.m` and `trim_max_qsort_student.m` files.

7.5 Trimming Time Comparison

You have implemented several different functions that do essentially the same thing. However, they do not necessarily have the same running time. One function may be faster or slower than the others.
$\Longrightarrow$ Compare the trimming time between `trim_max_ties_student` and `trim_max_qsort_student` function. Implement your function `compare_trim_max` in the file `compare_trim_max.m`.

8 More Algorithms to Implement

8.1 Robust Centroid

Now, finally, we are going to do something more robust. Instead of computing the mean, we are going to compute the trimmed mean. To compute the horizontal position of the center (which is the column index, or x-coordinate), we will discard the rightmost 5% and the leftmost 5% of the red pixels, and then compute the mean of the remaining 90%. Similarly, to compute the vertical position of the center, we will discard the uppermost 5% and the lowermost 5% of the red pixels.

```
function [centroid_row, centroid_col] =
robust_centroid_student(nonzero_rows, nonzero_cols)
```

$\Longrightarrow$ Implement your function `robust_centroid_student` in the file `robust_centroid_student.m`. You should call your function `trim` that you implemented above. When you demo this code to the lab TAs, show them different percentages to trim, as well as the 5% trimming we specified. Is there a value that gives better performance for your light stick tracker?

8.2 Top-leftmost pixel finding

You've already found the topmost red pixel and the leftmost red pixel. But suppose that you'd like to find the top-leftmost pixel.

```
function [row, col] =
upperleft_student(nonzero_rows, nonzero_cols)
```

$\Longrightarrow$ Implement your function `upperleft_student` in the file `upperleft_student.m`. In order to solve this problem, you will need to come up with a reasonable definition of what it means for a pixel to be the top-leftmost. To get checked off for this assignment, you will need to explain and justify the definition you used, and also to demonstrate your code.

8.3 ♦ Finding a dense box

Note: as the black diamond indicates, this problem is hard. Please be sure to complete the rest of the assignment first!

As some of you have noticed, the bounding box often gives poor results since it contains lots of non-red pixels. To fix this, you should try to find a box which is both large and dense, in that most of what it contains should be red pixels.

How you do this, and even precisely how you define the problem, is entirely up to you. Be sure to try your solution on some hard images (i.e., where the thresholding isn't working particularly well).

```
function [ top_row, bottom_row, left_col, right_col ] =
dense_box_student(bimage)
```

$\Longrightarrow$ Implement your function `dense_box_student` in the file `dense_box_student.m`. Demonstrate it on both a hard and easy example.