CS1114: Assignment 2

Assigned: Feb. 9, 2012 **First Part Due: Feb. 15 by 5PM**
Second Part Due: Feb. 22 by 5PM

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.

In this assignment, we are going to experiment with some of the interesting operations that we can perform on the binary images produced with your thresholding operator. The goal will be to support a useful application: tracking the movement of the lightstick 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, February 15**, and the second will be due one week later on **Friday, February 22**. 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:

```
* 2.1: sum, mean
* 4.1: Bounding box
* 4.2: Mid point
* 4.3: Centroid
* 5.3.2: Plot sorting time against size
* 5.3.3: Bubble sort
* 5.3.4: Compare sorting time of snail sort with bubble sort
```

Second Part:

```
* 6.1: partition
* 6.2: quicksort
* 6.3: Compare bubble sort to quicksort
* 6.4: quickselect

* 7.1: Median Vector
* 7.2: Top Left pixel finding
* 7.3: Dense Box (Black Diamond)
```

---------- BEGIN A2 PART 1 ----------

2 Two Simple Matlab Algorithms to Implement

We would like you to implement two 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 assignments, 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. As with A1, you should copy the '`/courses/cs1114/student_files/A2`' directory to your own directory so that you can work on these files. For instance, you may have made your own cs1114 directory in A1; if so you can copy the A2 files to that directory with the command:

```
cp -R /courses/cs1114/student_files/A2 ~/cs1114
```

Where we ask you to implement a function, please use the provided stub file.

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

2.1 `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.

3 Part2 GUI and Camera Calibration

For this assignment, you will want to open up `part2_gui`. This interface looks much like `part1_gui`, except that 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 `~/cs1114/A2/wand_thresholded1.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.

4 Binary Image Algorithms to Implement

We return to a very basic question which motivates this assignment: how can we estimate the position of the wand in the image? We'll start with a few simple techniques we discussed in class for this problem. Recall that we're dealing with binary images, so Matlab will provide us 2-dimensional arrays where each pixel is either 1 or 0 (on or off).

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 with two simple algorithm (bounding box and centroid); in the second part of this assignment, you will implement a more robust algorithm.

For all of the algorithms that we have provided here, **you can find function templates** in `/courses/cs1114/student_files/A2/` (which you should have copied over to somewhere in your home directory, as noted above). You should put your implementations in these template files.

A note on testing: remember to use different images to test your functions to be sure it will work for a variety of cases!

4.1 Bounding Box

One of the simplest ways to localize the wand is to use the bounding box algorithm. This algorithm computes the smallest (axis-aligned) rectangle that contains 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`.

For this part of the assignment you will 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 `bounding_box_student` in the file `bounding_box_student.m`. (You might want to use the `find` function we discussed in lecture.)

4.2 Midpoint of the Bounding Box

Once you have implemented the bounding box algorithm, write a function `midpoint_student` 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 `midpoint_student` in the file `midpoint_student.m`.

4.3 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 `centroid_student` in the file `centroid_student.m`. You should call the function `imean_student`, which you implemented above.

5 Sorting

Next, you will experiment with two sorting methods and compare their performance.

5.1 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;
```

5.2 Plotting in Matlab

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

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

For more details on `plot` or `semilogy`, use the `help plot` or `help semilogy` commands. In fact, both `plot` and `semilogy` can display multiple plots at once (simply give these functions multiple pairs of arrays, e.g., `plot(x,y,x,2*y)`.

5.3 Sorting and Run Times

In this part of the assignment, you will compare the performance of different sorting algorithms by using Matlab to time them, then plotting the times for arrays of a range of sizes.

5.3.1 Snail sort

Snail sort is a random sorting method. Given an array of unsorted numbers, snail sort randomly permutes all the numbers until the array is sorted (imagine sorting a deck of cards by shuffling them randomly until they are in order). A *permutation* is a rearrangement of the elements of an array (for instance, [3 1 2] is a permutation of [1 2 3]). 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 slow a manner as possible.

done = 0; n = length(A);

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

The function `randperm` generates a random permutation; if the permutation is stored in an array `p`, the syntax `A = A(p);` applies the permutation `p` to `A`, resulting in a rearranged array. The `issorted` function returns 1 if an array is sorted, and 0 otherwise. Snail sort has been implemented for you in the file `snailsort.m`. We will assign 1 extra credit point if you can come up with a sorting function that is even *slower* than snail sort on average.

5.3.2 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.

5.3.3 Bubble sort

Bubble sort is a simple sorting algorithm that works according to the following pseudocode:

```
while the array A is not sorted
    iterate through A
        if neighboring elements A(i) and A(i+1) are in the wrong order
            swap A(i) and A(i+1)
```

$\Longrightarrow$ Your job is to implement bubble sort in the file `bubblesort.m`. You may use the built-in `issorted` function to test whether an array is sorted.

5.3.4 Comparing Snail Sort with Bubble Sort

$\Longrightarrow$ Next, write a function comparing the running time of snail sort against your bubble sort function 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. You are allowed to use the built-in Matlab function `rand`, which generates random arrays. In particular, `rand(1,n)` generates a random array of length `n`.

```
-------------------------- END A2 PART 1 --------------------------
------------------------- BEGIN A2 PART 2 -------------------------
```

6 Quicksort

You will now implement your own version of quicksort, and compare its performance to bubble sort (later on we will modify quicksort to do selection for finding the median). Recall the three basic steps required for quicksort:

1. Choose an element to be the pivot (this could just be the first element, `A(1)`).
2. Partition the array into three subarrays `A1`, `A1`, and `A3`: one for elements smaller than the pivot (`A1`), elements equal to the pivot (`A2`), and elements larger than the pivot (`A3`).
3. Recurse: quicksort arrays `A1` and `A3` separately, then join them together.

Recall that there also must be a stopping condition, or else this will recurse forever. (Refer back to the slides on quicksort in lecture 6.)

6.1 Implementing the partition step

You first task is to implement step 2 of the quicksort algorithm (the partition step). You will write a function `partition_array` that takes in an array and a pivot value, and returns three arrays (the `A1`, `A2`, and `A3` described in step 2 above). The function header looks like:

```
function [ A1, A2, A3 ] = partition_array(A, pivot)
%PARTITION_ARRAY Partitions an array A into elements smaller than, equal to,
%  or larger than the pivot into output arrays A1, A2, and A3.
```

$\Longrightarrow$ Implement `partition_array` in the file `partition_array.m`.

6.2 Implementing quicksort

Now you will implement quicksort (you should call the `partition_array` function that you just wrote). Your `quicksort` function will have the following header:

```
function S = quicksort(A)
```

$\Longrightarrow$ Implement `quicksort` in the file `quicksort.m`. You may follow the basic pseudocode given in class. Try to make your quicksort function as fast as possible (1 point of extra credit will be given out for extremely efficient implementations). One way to test this function is to generate random arrays using Matlab's built-in `rand` function.

6.3 Comparing bubblesort to quicksort

$\Longrightarrow$ Now, similar to problem 5.3.4, write a function comparing the running time of your bubble sort function to your quicksort function on random arrays of different lengths, and plotting the results. Again, more points will be awarded for a robust comparison. You may want to include features like multiple trials, random array generation, an appropriate Matlab plot of the running time, and anything else you can think of.

6.4 Quickselect

Next, you will implement the modified version of quicksort used for selection (i.e., the "quickselect" algorithm). Recall that the quickselect algorithm takes two arguments, the array (`A`) and the rank of the element to be found (`k`, meaning the k^{th} largest element), and has the following steps (the first two are identical to quicksort):

1. Choose an element to be the pivot (this could just be the first element, `A(1)`).
2. Partition the array into three subarrays: one for elements smaller than the pivot (`A1`), elements equal to the pivot (`A2`), and elements larger than the pivot (`A3`).
3. If `k <= length(A3)`, then recursively find the k^{th} largest element in `A3`
4. If `k > length(A2) + length(A3)`, then let `j = k - (length(A2) + length(A3))`, and recursively find the j^{th} largest element in `A1`
5. Otherwise, return `x`.

Your `quickselect` function will have the following header:

```
function s = quickselect(A, k)
```

$\Longrightarrow$ Implement `quickselect` in the file `quickselect.m`. Your function will likely have some similarities to your `quicksort` function. (One way to test this function is to generate a random array using the `randperm` function described in Section 5.3.1).

7 Some Final Algorithms to Implement

7.1 Median Vector

Now, finally, we are going to do something more robust. Instead of using the midpoint or mean to localize the wand, we will now compute the median vector, as described in class. The header for this function is:

```
function [median_row, median_col] =
median_vector_student(nonzero_rows, nonzero_cols)
```

$\Longrightarrow$ Implement `median_vector_student` in the file `median_vector_student.m`. To compute the median vector, you will use the `quickselect` function you implemented above.

7.2 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.

7.3 ♦ 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.