CS1114 Assignment 4

out: March 12th, 2012
due: **5:00PM March 30th, 2012**

1 Previously, on Assignment 3...

...you wrote functions to drive the robot based on the position of the lightstick in the webcam image. Now, we'd also like the robot to respond to the *orientation* of the lightstick. In this assignment, we will determine a simple orientation for an object by first finding a bounding polygon (the convex hull), then choosing a major axis along the length of the object. Later, we will apply this algorithm to a real-time camera feed to provide a throttle control to the robots.

2 Convex Hull & Orientation

As defined in the lecture, a convex hull is the *smallest convex polygon* (see Fig. 1) containing a set of points. A convex polygon is a polygon for which it is possible to draw a straight line from any point in the polygon to any other point in the polygon without ever crossing the boundary or coming out of the shape. Given a set of points P (e.g. the big red blob you found using connected components in A3), your task is to implement an algorithm for finding the convex hull. We first consider the *Gift Wrapping* algorithm:

GiftWrappingAlgorithm(P):

1. Start with a point $p_1 \in P$ that is guaranteed to lie on the border of the hull (we chose the bottom-most point in class, but you can choose any extreme point). Add p_1 to the hull.

2. Find the point $p \in P$ that makes the largest "right-hand turn." i.e., if p_{prev} is the previous point on the convex hull, and ℓ is the line between the two previous points in the convex hull (or a horizontal line, in case the previous point is the bottom-most point) then the p maximizes the angle θ between ℓ and the line through p_{prev} and ℓ. Add p to the hull.

3. Repeat step 2 until we get back to p_1.

This algorithm marches around the convex hull in counter-clockwise order. Step 2 requires finding the angle between two line segments—you will have to use the vector and trigonometric operators described in class to implement this (the *acos* Matlab function may be helpful here).

The next algorithm to consider is quickhull, which, like quicksort, is a divide and conquer algorithm. We have provided you with an implementation of quickhull—you will call this function (with the same interface as your gift-wrapping algorithm code) in the next section of this assignment.

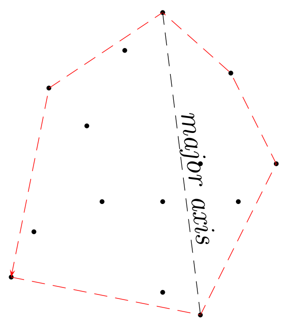

Figure 1: A convex hull. The black points form the point set, and the dotted red lines are the edges of the convex hull. The dotted black line is the major axis.

Orientation. We define the orientation of a polygon (a convex polygon, in our case) by defining a major axis. The major axis of a polygon is the pair of vertices that are the farthest from each other (the length of this axis is the *diameter* of the polygon. To find these points, find the distance between each pair of vertices and select the pair with the largest distance.

3 Running Time of Convex Hull

How long do these algorithms take to run? Your next task is to compare their runtimes on various types of input. In class, we discussed how the runtime of the gift-wrapping algorithm depends not only on the size of the input (n points, say), but also on the size of the output (the number of points on the convex hull, h). To test this out, you will first write functions to generate random points in various configurations. Then you will test your convex hull algorithms on different random point configurations:

1. **Random points inside of a square.** The first configuration is random points (with floating point coordinates) inside of the square with corners $(1, 1)$, $(1, -1)$, $(-1, -1)$, and $(-1, 1)$. The points should be uniformly distributed inside of this square, meaning that all points inside the square are equally likely to be generated. You may use the built-in Matlab function `rand(rows,cols)`, which generates a `rows` $\times$ `cols` matrix of random numbers between 0 and 1.

2. **Random points inside of a circle.** The second configuration is random points inside of the unit circle (the circle centered at the origin with radius $= 1$). Again, you should sample points from a uniform distribution.

3. **Random points on a circle.** The third configuration is random points *on* the unit circle (i.e., points with distance 1 from the origin). Again, you should sample points from a uniform distribution. This means you must be careful how you generate the points.

Once you have implemented the above, you will generate a plot of the runtime of the two convex hull algorithm on point sets of different size and in different configurations. You will generate three plots; the first two will contain three curves, one for each of the three configurations. The first plot will plot the size of the convex hull vs. the size of the input point set (for each of the three configurations). The second will plot the running time of the gift-wrapping algorithm vs. the size of the input point set (for each of the three configurations).

Finally, generate a third plot comparing the running time of gift-wrapping to quickhull using the **random points on a circle** configuration.

4 Preprocessing a binary image for convex hull

To find a bounding polygon of the lightstick, we can apply the convex hull algorithms you developed above to the big red blob you found in Assignment 3. However, it is somewhat silly to find the convex hull of a dense blob in the image, because most pixels in the blob with be in the interior, surrounded by other pixels. These pixels can't possibly be part of the convex hull—only pixels on the *boundary* of the blob may be on the convex hull (although not all boundary pixels will be). What does it mean for a pixel to be on the boundary of a blob? A boundary pixel is a pixel such that some of its neighbors are in the blob, and some are not in the blob (here we mean 4-connected neighbors, i.e., north, east, south, and west). Any blob pixel whose four neighbors are also blob pixels are not on the convex hull. In fact, any blob pixel with *three* neighbors that are in the blob also can be excluded from the convex hull. Thus, we can save a lot of time if we get rid of pixels with these properties.

You will solve this problem using a trick you recently learned: convolution. Your first task is to come up with a convolution kernel (i.e., a filter) that counts, for each pixel in a binary image, the number of (4-connected) neighboring 1's. That is, when you convolve the image with this kernel, every pixel will be replaced with the number of neighboring 1's. For instance, the output of your kernel on the binary image on the left should be the image on the right:

0	0	0	1	0
0	1	1	1	0
1	1	1	1	0
0	1	1	0	0
0	0	1	0	0

`output` =

0	1	2	1	1
2	2	3	3	1
1	4	4	2	1
2	2	3	2	0
0	2	1	1	0

Once you have found the right filter, you can apply it to an image `B` using the built-in Matlab convolution function `conv2`. The `conv2` function takes in two matrices, and convolves the first matrix with the second, returning the resulting convolved image. You may want to pass in the string 'same' as a third argument to conv2; otherwise, this function will return a slightly larger image whose width and height are the sum of the width and height of the two input matrices.

Now, only the pixels with value < 3 in the output convolved image can possibly be part of the convex hull. Your next task is to use the find function, combined with matrix operations such as $<$, to get the coordinates of these pixels. Note that you should also only pick out pixels that are 1's in the original input binary image (note that some 0 pixels in the input now have non-zero values in the output; we don't want to include such pixels, as only pixels that were 1 to begin with can be part of the convex hull). To exclude the pixels that were originally 0's, you will use the logical operator '&'. In Matlab, the '&' operator takes two numbers a and b, and returns 1 if both a and b are non-zero, and 0 if both a and b are zero. For instance:

```
>> 1 & 1

ans =
     1

>> 1 & 0

ans =
     0

>> 5 & 1

ans =
     1
```

The '&' operator is also defined on two matrices A and B (of the same size), where it applies the operator to each pair of corresponding elements of A and B.

You should combine all of the above into *one line* of Matlab code that takes a binary image and returns the coordinates of all the pixels that meet the criteria above: less than 3 neighboring 1's, and a 1 to begin with. You can declare the kernel on a separate line, if necessary.[1] This line will go into a function with the header:

```
function [bdry_rows, bdry_cols] = find_image_boundary(bimage)
```

For instance, if the input binary image shown above is stored in the matrix A, then a function call to filter_image_convex_hull(A) should return variables [bdry_rows, bdry_cols] corresponding to the coordinates:

```
(1, 4), (2, 2), (3, 1), (3, 4), (4, 2), (5, 3)
```

[1]While in the past we have discouraged such one-liners that use Matlab functions, this exercise can give you a sense of the power of Matlab.

(in these ordered pairs, the row comes first, followed by the column).

You will get one point of extra credit for coming up with a better way to filter pixels that can't possibly be part of the convex hull.

5 Task List

5.1 Functions to Write

We have provided template files for each of these functions, so go ahead and grab them from `~/cs1114/student_files/A4/` and copy them to your working directory to get started.

1. Write a function that implements the gift-wrapping algorithm, as described in Section 2 to find the shape of the points in the given binary image. Your function should be named `convex_hull_giftwrapping` (see the file `convex_hull_giftwrapping.m` for the function header). This function takes as input a list of rows and a list of columns (the $y-$ and $x-$coordinates of the points, respectively), and returns an array of *indices* of the points on the convex hull. The points on your hull may be listed in either clockwise or counterclockwise order, and may or may not contain colinear points. The starting point should appear both at the beginning and the end of the list. We will be providing a function for testing your convex hull code. You can also use the function `plotConvexHull(rows, cols, hullIndices)` to generate a plot of a point set and the polygon represented by `hullIndices`.

2. Write a function `poly_major_axis` that takes as input a list of rows and columns which represent vertices of a polygon and returns a 2-by-2 matrices of the form `[row1 col1; row2 col2]`; this matrix should contain the two points which form the major axis. Given arrays `rows` and `cols` representing the point set, and an array `hullIndices` of point indices representing the convex hull (e.g., the output from the `convex_hull_giftwrapping` above), the Matlab expressions `rows(hullIndices)` and `cols(hullIndices)` select the appropriate rows and columns for input to the `poly_major_axis` function.

3. Write a function `major_angle` that accepts a 2-by-2 matrix `[x1 y1; x2 y2]` (i.e. the result from `poly_axes_student`) and returns the angle of the major axis relative to the vertical (see Figure 2).
   Hint: this will require some trigonometry; you may find the `atan2` *function useful.*

4. Write functions `randPointsInSquare`, `randPointsInCircle`, and `randPointsOnCircle`, which take a number of points to generate n and return n random points with the appropriate distribution.

5. Write a function `plot_convex_hull_performance` that generates the plots described in Section 3.

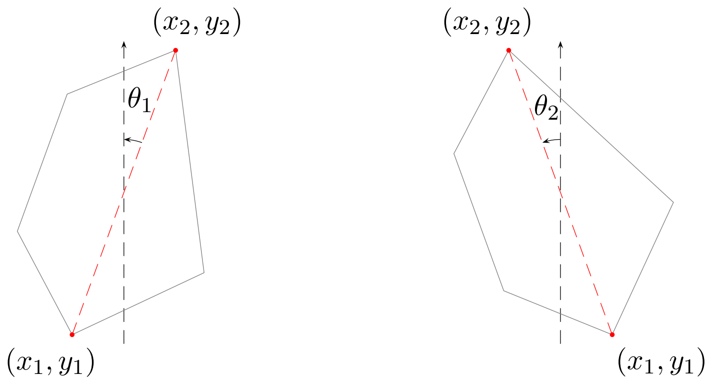

Figure 2: `major_angle` should measure the angle of the major axis relative to the vertical—in whatever direction will yield the smallest positive result; so, in this example, θ_1 and θ_2 would both be positive.

6. Write a one-line function `find_image_boundary` that takes as input a binary image `bimage` and returns arrays `rows` and `cols` representing the set of points on the boundary of the binary image as defined in Section 4. This function should be no more than one line (two, if you choose to create the filter matrix on a separate line), not including comments or the function header.

7. Write a function `lightstick_orientation` that takes as input a binary image `bimage` and returns the orientation (angle relative to the vertical) of the lightstick. This function should call many of the functions described above.