CS 270 – Fall 2013
Assignment 2
Machine Vision
Date Due: Wednesday, 2 October

Objectives

For Assignment 2 you will implement in Java a classic 2D machine vision algorithm often called *Blob Analysis*. You will explore its accuracy in providing certain geometric measurements of objects under translation and rotation. You will assess the effects of noise and the use of hard vs. fuzzy thresholds.

Blob Analysis

Blob analysis refers to a class of machine vision methods that attempts to segment an image into individual objects and measure various properties of those objects. It is a very old method with lots of variations and names. It was one of the first methods provided in the earliest commercial vision systems, and is still used to this day, although it is far less important.

Blob analysis is primarily a 2D method, in that it is intended for locating and measuring 2D objects constrained to lie in a plane. For example, it might be used to locate individual characters on a printed page, or manufactured parts lying flat on a conveyer belt.

Blob analysis has two principal components, at least in the flavor that we will study, which I'll call *connectivity analysis* and *geometric analysis*.

Connectivity Analysis

The purpose of connectivity analysis is to segment an image into blobs and background. A blob is a set of pixels that are connected according to rules that I define below. A blob exists in an image as a result of a computation, and hopefully corresponds to a physical object in a scene viewed by a camera. If the computation goes well, there will be a one-to-one correspondence between image blobs and scene objects. If the scene is more complex than connectivity analysis can handle, blobs may correspond to no or many objects, and objects may have no or many corresponding blobs.

Assume that every pixel in an image is classified as *object* or *background* by some means. The simplest and probably most widely used means is a simple threshold: a pixel is object if its gray value is above (for light objects) or below (for dark objects) some threshold value. The threshold may be fixed (a terrible idea) or computed from the image (a less terrible idea). Far more sophisticated (but slower) pixel classification methods based on local gray scale or color thresholding operators can be devised, but we will stick with a fixed gray-level threshold for A2.

Connectivity analysis uses a *neighborhood function* that specifies a set of relative locations that define the neighbors of any given pixel. The function must have the property that pixel *a* is the neighbor of pixel *b* if and only if *b* is the neighbor of *a*. The two most common neighborhood functions on a square tessellation are *4-neighbors* (N, S, E, W, neighbors share edges) and *8-neighbors* (N, NE, E, SE, S, SW, W, NW, neighbors share edges or vertices). On a hexagonal tessellation one would use 6-neighbors.

Pixels *a* and *b* are *connected* if and only if *a* and *b* are classified as *object* and either *a* and *b* are neighbors or there exists some pixel *c* such that *a* and *c* are connected and *b* and *c* are neighbors. This is a recursive definition. A blob is a set of all pixels that are mutually connected. Don't let this formal definition confuse you—your intuition of what is a set of connected pixels will serve you well. Note that one can just as well define connected sets of background pixels, but we will not do so here.

There are three distinct algorithms (that I know of) for computing connectivity, of varying complexity and speed. The simplest, and the one that you will implement, uses a stack and closely follows the above recursive definition. Another method walks around the perimeter of blobs, and the fastest and most complex method builds 2D blobs from connected 1D runs while applying various splitting and merging rules. This method gets its speed primarily because it proceeds in strict raster order and so uses cache memory very efficiently.

The recursive connectivity algorithm uses two primary data structures:

- A stack of (x,y) integer pixel coordinates.

- A 2D boolean array called `mark` that is two rows taller and two columns wider than the image, and which indicates which pixels have already been explored as the algorithm progresses. The two extra rows and columns form a border around the elements of `mark` that correspond to the image. These border elements are initialized to `true` to so that the recursive algorithm does not wander off the image and trigger an array access exception. One could avoid the border pixels by checking every image coordinate to prevent running off the edge, but since you have to check the `mark` array anyway, checking coordinates adds lots of unnecessary execution time.

Consider a function `explore(x, y)` whose purpose is to see if the pixel at `(x,y)` should be added to a blob, and if so to add it. Here is what explore does:

```
if (the pixel at (x,y) has not yet been explored)
{
  mark (x,y) as having been explored;
  use a threshold to get the weight w of the pixel;
  if (the pixel at (x,y) is classified as object)
  {
    add (w,x,y) to the current blob result object;
    push (x,y) onto the stack;
  }
}
```

I have provided a `Threshold` interface and a few classes that implement the interface (two hard threshold classes and a fuzzy threshold class). The interface provides a method `weight` that converts a raw value (in our case a pixel gray value) into a weight between 0 and 1 that indicates relative confidence that a pixel is classified as *object*. For a hard threshold, only the values 0.0

and 1.0 are returned. For a fuzzy threshold, any value in the range can be returned. For the purpose of connectivity, any weight greater than zero is to be classified as *object*. The weight is used by the geometric analysis code in calculating various statistics.

Connectivity can be done like this, although you can do it however you like:

```
initialize mark array;
create a new blob result object to be the current one;
for (each pixel (x,y))
{
  explore(x, y);
  if (the pixel was pushed onto the stack)
  {
    do
    {
      pop an element p from the stack;
      for (each neighbor (u,v) of p)
        explore(u, v);
    }
    while (the stack is not empty);

    if (the result object is heavy enough)
      add the result object to a vector of results to return;

    create a new blob result object to be the current one;
  }
}
```

You can implement either 4-neighbor or 8-neighbor rules; with the images we will use, the results will be the same. With an elegant implementation, it's the same number of lines of code either way. See if you can avoid the sin of copy/paste programming.

Geometric Analysis

We would like to know various properties of each blob, particularly location, size, and orientation. We are primarily interested in geometric properties, which are ones that can be defined without reference to any coordinate system and which are therefore intrinsic to the blob.

The two simplest geometric properties are *area*, which can be used as a measurement of size, and *center of mass*, which can be used as a measure of location. Area is just total weight of the pixels in the blob—the area of an individual pixel is taken to be its weight, i.e. its relative confidence that the pixel is really *object* and not *background*. Pixels of low confidence have a small but important effect on area and center of mass, as you will see when you compare hard and fuzzy thresholds.

An example of a non-geometric blob property commonly used is the *bounding box*, the smallest rectangle oriented with the pixel grid that encloses the blob. It's very easy to compute—just the minimum and maximum *x* and *y* coordinates of the blob—but the resulting rectangle is defined relative to the pixel grid (i.e. coordinate system) of the camera and is therefore not an intrinsic property of the blob. For that reason I avoid using it and we will not calculate it in this assignment (although you can if you want to).

In addition to area and center of mass, we will calculate the principal moments and axes of inertia. The first principal axis of inertia is the line about which the blob has the smallest moment of inertia, and the moment about that line is the first principal moment. We will show that the first principal axis passes through the center of mass, and derive a formula for computing the moment and the axis.

The second principal axis of inertia is the line passing through the center of mass about which the blob has the largest moment of inertia, and the moment about that line is the second principal moment. We will show that the second principal axis of inertia is perpendicular to the first.

Notice that I've defined the principal axes and moments without any reference to a coordinate system, and so they are geometric, intrinsic properties. (For math majors, principal axes and moments are eigenvectors and eigenvalues of the blob's inertia matrix.)

The orientation of the first principal axis of inertia can be used as a measure of the orientation of a blob. The square roots of the first and second principal moments divided by area, which are in units of length, can be used as measures of the size of the object. You can imagine that they are the major and minor radii of an ellipse that best fits the blob. I call these values the first and second principal lengths, although I don't think that is a standard term. For example, if you multiply the principal lengths of a rectangle by $\sqrt{12}$ you will get the width and height. Unlike the bounding box, this width and height is independent of the rectangle's orientation relative to the pixel grid, and is therefore much more valuable.

Area, center of mass, and the principal moments are all computed from moments of pixel weight. In general, an n^{th} order moment of a set of pixels (w_i, x_i, y_i) is of the form

$$\sum_i w_i x_i^a y_i^b \tag{1}$$

where $a + b = n$, and a, b, and n are non-negative integers. It is easy to see that there are n+1 n^{th} order moments, and that the one 0th order moment is area. Center of mass is computed from 0th and 1st order moments in the obvious way:

$$\left(\frac{\sum_i w_i x_i}{\sum_i w_i}, \frac{\sum_i w_i y_i}{\sum_i w_i} \right) \tag{2}$$

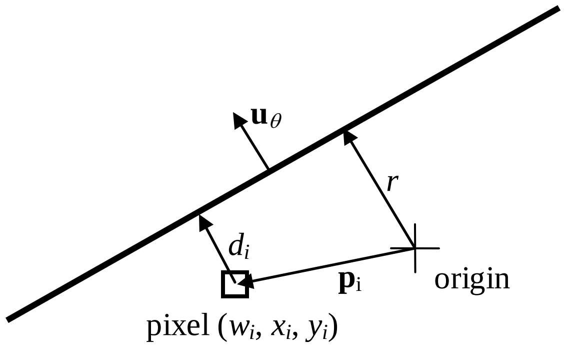
As we are about to show, the principal moments and axes are computed from the 0th, 1st, and 2nd order moments.

Note that for a circle, the moment of inertia about any line passing through the center is the same. There is no smallest value, and so the principal axes are undefined. This makes sense because a circle doesn't have an orientation. What is perhaps surprising is that the same holds for a square—the principal axes are undefined. A square does have an orientation, but it can only be revealed by appealing to the 3rd moments. When the principal axes are undefined, the two principal moments are equal.

(Note: You can complete the assignment using the formulas we are about to derive without actually understanding the derivation, but at least give it a try.)

In order to compute the principal axes, we need to use analytic geometry and represent lines with numbers. We reject the simple slope-intercept representation $y = ax + b$ because the slope becomes infinite for vertical lines and inaccurate for near-vertical lines. The two-intercept form is even worse—neither horizontal nor vertical lines can be represented. We reject two-point and point-direction forms because a line has only two degrees of freedom and these forms have three or four parameters, so an analytic solution would be underdetermined. We need a two-parameter representation that never blows up.

The solution is to represent a line in polar coordinates (r, θ), where r is the distance of the line from some arbitrary origin and θ is an angle that, for convenience, we make perpendicular to the line. Consider this picture:



This shows some line at distance r from the origin, with unit vector $\mathbf{u}_\theta$ perpendicular to the line. The components of $\mathbf{u}_\theta$ are $(\cos(\theta), \sin(\theta))$. Consider the pixel shown of weight w_i, at position defined by vector $\mathbf{p}_i$ of components (x_i, y_i), and at distance d_i from the line. Some simple vector algebra shows that

$$d_i = r - \mathbf{p}_i \bullet \mathbf{u}_\theta \tag{3}$$

The moment of inertia of a set of pixels is:

$$\begin{aligned} N &= \sum_i w_i d_i^2 \\ &= \sum w_i \left(r - \mathbf{p}_i \bullet \mathbf{u}_\theta\right)^2 \\ &= \sum w_i \left[r - (x_i \cos(\theta) + y_i \sin(\theta))\right]^2 \\ &= \sum w_i \left[r^2 - 2rx_i \cos(\theta) - 2ry_i \sin(\theta) + x_i^2 \cos^2(\theta) + 2x_i y_i \cos(\theta)\sin(\theta) + y_i^2 \sin^2(\theta)\right] \end{aligned} \tag{4}$$

Define the moments as follows:

$$
\begin{aligned}
M_1 &= \sum_i w_i \\
M_x &= \sum_i w_i x_i \\
M_y &= \sum_i w_i y_i \\
M_{xx} &= \sum_i w_i x_i^2 \\
M_{yy} &= \sum_i w_i y_i^2 \\
M_{xy} &= \sum_i w_i x_i y_i
\end{aligned}
\tag{5}
$$

Now we can write

$$
N = r^2 M_1 - 2rM_x \cos(\theta) - 2rM_y \sin(\theta) + M_{xx}\cos^2(\theta) + 2M_{xy}\cos(\theta)\sin(\theta) + M_{yy}\sin^2(\theta) \tag{6}
$$

To find the *r* that minimizes *N*:

$$
\begin{aligned}
&\frac{\partial N}{\partial r} = 2rM_1 - 2M_x \cos(\theta) - 2M_y \sin(\theta) = 0 \\
&rM_1 = M_x \cos(\theta) + M_y \sin(\theta)
\end{aligned}
$$

$$
r = C_x \cos(\theta) + C_y \sin(\theta) \tag{7}
$$

where from equation 2 (C_x, C_y) is the center of mass. It is not hard to see that equation 7 establishes that the line must pass through the center of mass.

To find the orientation of the line:

$$
\begin{aligned}
&\frac{\partial N}{\partial \theta} = 2rM_x \sin\theta - 2rM_y \cos\theta - 2M_{xx}\cos\theta\sin\theta - 2M_{xy}\sin^2\theta + 2M_{xy}\cos^2\theta + 2M_{yy}\sin\theta\cos\theta = 0 \\
&(C_x \cos\theta + C_y \sin\theta)(M_x \sin\theta - M_y \cos\theta) - (M_{xx} - M_{yy})\sin\theta\cos\theta + M_{xy}(\cos^2\theta - \sin^2\theta) = 0 \\
&(C_x M_x - C_y M_y - M_{xx} + M_{yy})\sin\theta\cos\theta + (M_{xy} - M_x M_y / M_1)(\cos^2\theta - \sin^2\theta) = 0 \\
&(C_x M_x - C_y M_y - M_{xx} + M_{yy})\sin(2\theta)/2 + (M_{xy} - M_x M_y / M_1)\cos(2\theta) = 0 \\
&\left[(M_1 M_{xx} - M_x^2) - (M_1 M_{yy} - M_y^2)\right]\sin(2\theta) = 2(M_1 M_{xy} - M_x M_y)\cos(2\theta)
\end{aligned}
\tag{8}
$$

Now define

$$
\begin{aligned}
a &= M_1 M_{xx} - M_x^2 \\
b &= M_1 M_{yy} - M_y^2 \\
c &= 2\left(M_1 M_{xy} - M_x M_y\right)
\end{aligned}
\tag{9}
$$

then, for any integer *k*,

$$\begin{aligned} \tan(2\theta) &= \frac{c}{a-b} \\ \theta &= \frac{\operatorname{atan2}(c, a-b) + k\pi}{2} \end{aligned} \tag{10}$$

There are two distinct solutions, corresponding to $k = 0$ and $k = 1$:

$$\begin{aligned} \theta_1 &= \frac{\operatorname{atan2}(c, a-b)}{2} \\ \theta_2 &= \theta_1 + \frac{\pi}{2} \end{aligned} \tag{11}$$

The two solutions correspond to the minimum and maximum moments of inertia, i.e. the angles of the two principal axes, and are obviously at right angles.

If your brain hasn't exploded yet, let's derive the principal moments. Starting with equation 6 and substituting for *r* using equation 7, and remembering center of mass from equation 2 and the definitions in equations 5 and 9, we get

$$\begin{aligned} N &= \frac{a\cos^2\theta + b\sin^2\theta + c\sin\theta\cos\theta}{M_1} \\ &= \frac{a(\cos 2\theta + 1) + b(1 - \cos 2\theta) + c\sin 2\theta}{2M_1} \\ &= \frac{a + b + (a-b)\cos 2\theta + c\sin 2\theta}{2M_1} \end{aligned} \tag{12}$$

From equation 10 we have

$$\begin{aligned} \cos 2\theta &= \frac{a-b}{\sqrt{c^2 + (a-b)^2}} \\ \sin 2\theta &= \frac{c}{\sqrt{c^2 + (a-b)^2}} \end{aligned} \tag{13}$$

So the principal moments are

$$N_{1,2} = \frac{a + b \pm \sqrt{c^2 + (a-b)^2}}{2M_1} \tag{14}$$

The $\pm$ comes because we can negate $\sin 2\theta$ and $\cos 2\theta$ simultaneously to switch between the two principal axes. The minus value gives the first principal moment (the smallest moment) and the plus value gives the second. Finally, the principal lengths are

$$L_{1,2} = \sqrt{\frac{N_{2,1}}{M_1}} \tag{15}$$

The first principal length comes from the second principal moment and is measured along the first principal axis, and vice versa.

Commentary

Blob analysis has a number of significant advantages, and a few severe disadvantages that make it of limited utility. It works well when two conditions are met:

- Pixels can be classified reliably as object or background from their gray level, color, or some local operator.
- The correspondence between blobs and physical objects can be determined based primarily on prior constraints (e.g., arranging for only one kind of object to appear in the field of view) and geometric properties of the blob, particularly area and principal lengths.

Unfortunately, these conditions are nearly impossible to satisfy in unconstrained 3D vision tasks facing biological organisms or autonomous machines. They can at times be satisfied in the more constrained tasks facing industrial vision systems, but far more typically one must employ sophisticated pattern recognition methods to locate and identify objects.

Connectivity analysis is the weak link. It requires the pixel classification condition to be satisfied, and it is also susceptible to wildly erratic behavior because a single pixel changing one gray level can connect two separate blobs into one, or break one blob in two. When this happens the object-blob correspondence is lost and the geometric quantities become meaningless as measurements of physical object properties.

However, if the classification and correspondence conditions are satisfied, blob analysis performs spectacularly. First, it is hugely faster than almost any other pattern recognition method, particularly when objects can rotate and/or change size. Blob analysis is roughly $O(n)$, where n is the number of pixels in the image. More sophisticated methods that can handle rotation and size changes can be $O(n^3)$ (and remember that n can exceed one million), and indeed what makes these methods sophisticated is the many tricks used to reduce the execution time without getting a wrong result.

Second, the geometric measurements are extremely accurate. Position can be measured to a small fraction of a pixel, for example, independent of the orientation of the blob. Accuracy holds up well when the image is contaminated by noise, as all real images are. It took around twenty years from the time that blob analysis made an appearance in commercial vision systems until methods were developed that could find patterns under rotation and size changes with high accuracy and acceptable speeds, and that are not subject to blob's classification and correspondence conditions.

Source Code

Assignment 2 includes the following Java source files:

`Camera.java`	A complete implementation of a simulated camera that can acquire gray-scale images of synthetically rendered shapes of precisely specified geometry. Noise and other simulated camera properties can be specified.
`Blob.java`	A skeleton implementation of blob analysis for you to complete. The skeleton includes the class structure, hard and fuzzy threshold classes, and a utility method to print results. You will implement the `Result` class and the `run` function, adding whatever other functions and classes that you want.
`Program.java`	A complete implementation of `main` that creates images from a camera, runs blob analysis, and prints results. The main program first creates and runs one image with two blobs to verify that your implementation is correct, and then a series of runs on a single blob translated and rotated for accuracy tests, with and without noise, and using a hard and a fuzzy threshold. Statistics of the runs are gathered and printed for your assessment. You may extend this code to explore other effects if you wish.

Procedure for Completing the Assignment

First read this assignment document. You should understand the section on connectivity analysis and the definitions in the section on geometric analysis. You need not follow the geometric derivations in detail, but at least be familiar with the outline. You will need to use equations 2, 5, 9, 11, 14, and 15.

The code provided should run as given. When you do it will print to the system console:

- The first image on which blob analysis will be run. This image has two blobs, one inside the other.

- A second image of a blob that will be rotated and translated for accuracy tests.

- Four result statistics tables, one for each combination of noise/no noise and hard/fuzzy threshold. The tables will be all zeros because blob is not yet implemented.

The simulated camera is set up by default to make pixels with gray levels in the range 64 – 192, simulating 8-bit pixels. The printed images are compressed to the gray level range 0 – 9 to keep the printout to a reasonable size. Keep in mind that the display shows gray levels, not thresholded weight.

For each of the four noise/threshold combinations, runs are made at 18 angles (0 – 170° in 10° steps) and 100 translations (offset 0.0 to 0.9 pixels in 0.1 pixel steps in *x* and *y*), for a total of 1800 runs. Statistics (mean, standard deviation, minimum, and maximum) are computed for the six geometric measurements (area, center of mass, angle of first principal axis, and principal lengths). For center of mass and angle, the true values are known and the statistics of the errors are computed. For area and the principal lengths the true values are not known (they could in principal be derived), and the statistics of the computed values are computed. In all cases the standard deviation value is a good assessment of accuracy—smaller is better. In all cases except angle, the statistics are in units of pixels. For angle, the units are degrees.

In `Camera.java` you should understand the `Image` class; you'll need to call the `getPixel` method. You do not need to study anything else in `Camera.java` unless you are curious, and I am happy to answer questions about it for those who are. `Program.java` is straightforward and you should understand what it's doing, at the very least to aid in debugging. Both files are good examples of programming style, except that I've been a little lazy in using public fields instead of creating getters and setters.

Study `Blob.java`. You must implement all seven members of the `Result` class, and add other fields and members to that class as appropriate to support the implementation. This is where all of the computations of geometric analysis are done.

Understand how thresholds work. They are very simple, and are implemented for you; you just have to understand how to use them.

Implement the `Run` function. This is where connectivity analysis is done. Results must be placed into the field `results`, which is of type `Vector<Result>`.

There is no point in running the accuracy tests (4 * 1800 = 7200 runs) until you believe that your implementation is working. Modify `main` in `Program.java` to skip those tests until you are getting reasonable results printed out after the first picture. There should be two blobs, and you can see what the center of mass and angle should be (approximately) from the code in `main` that creates the `RoundedRectangle` shapes.

Finally, run the accuracy tests and write a report concisely discussing these topics:

1. The accuracy of the geometric measurements. Why is it so good?
2. The effect of noise on the hard threshold results. Surprising?
3. The effect of noise on the fuzzy threshold results.
4. The relative accuracy of hard and fuzzy thresholds. What might explain this?

Upload to Blackboard a folder whose filename is your last name and contains:

- Your source code `Blob.java`, and the other files if you modified them.
- The system console output as a text file.
- A document (Word or PDF) with the above discussion.