Due date: November 11, 2015. 11:59 PM (Madison time). You have three late days — use it at as you wish. Once you run out of this quota, the penalty for late submission goes as follows for each subsequent 24 hour period: \{10\%, 25\%, 50\%\}. That is, if you submit this assignment on Nov. 13 at 4:00 AM in the morning, you lose two late days.

**What to turn in:**

1. Your submission should include your complete code base in an archive file (zip, tar.gz), example images you used to evaluate your implementation on (organized under different directories such as q1/, q2/, and so on), and a README describing how to run it.

2. A brief report (typed up, submit as a PDF file, NO handwritten scanned copies) describing what you implemented and known failure cases.

**Notes from instructor:**

- Start early!
- Stick with small images to start with to keep the running time manageable. There will be no penalty for (slow) running time. However, use this opportunity to learn about data structures that you think might be helpful. You may ask the TA or instructor for suggestions, and discuss the problem with others (minimally). But all parts of the submitted code must be your own. Matlab is slow with loops, and vectorized code is significantly faster, see [http://www.mathworks.com/support/tech-notes/1100/1109.html](http://www.mathworks.com/support/tech-notes/1100/1109.html).
- Submission instructions (for Moodle) and a number of images will be provided shortly. Remember that the performance of your submitted code on these ‘example images’ is not sufficient for full credit. Try thinking about and accounting for degenerate cases. You should generate new images to evaluate the robustness of your code. Data we use to grade your submitted code will be available to you after submission.

**Problem 1**

(Intelligent Scissors/Live-wires, 35pts) This question is based on the "Intelligent Scissors/Live-wire" discussion from class. Please write a function `imOut = scissors(imIn, seedRow, seedCol, destRow, destCol)` that computes a live-wire (shortest path) from a seed position (pixel) given by coordinates `[seedRow, seedCol]` to a destination (pixel) given by coordinates `[destRow, destCol]`. Feel free to use the `ginput()` functionality from Matlab to get this input in interactive mode. As discussed in class, the live-wire will prefer going through small cost (high edge magnitude) pixels rather than through smoothly varying regions in the image, this will yield a partial delineation of the object boundary. Your output will be 0’s everywhere in `imOut` and 1’s where the edge is selected as part of the live-wire. Helpful preliminary details to address this question are given in the following sections.

If you have not already, please read the paper, [Intelligent Scissors for Image Composition (Mortensen and Barrett, 1995)](http://www.mathworks.com/support/tech-notes/1100/1109.html). We are essentially asking you to implement as much functionality from this paper as you can.

It is useful to think of the image as a graph (see Fig. 1), and the pixels as nodes (or vertices) in the graph. Each edge in the graph has a cost, a function of the derivative of the image across the edge. Since we would want to solve a shortest path problem, a high image gradient will correspond to a small cost, smooth regions in the image will have large edge costs, discouraging the live-wire to select them.

A first attempt at calculating the edge costs may be done in the following manner (note that this is only a suggestion, please use ideas from the paper above and generally be creative!). To calculate the costs of
the edges, we may first estimate the intensity derivative, $D$ across each edge. For diagonal edges marked 3 between $(i, j)$ and $(i+1, j-1)$ in Fig. 1 above,

$$D_{(i,j),(i+1,j-1)} = \frac{||I(i+1,j) - I(i,j-1)||}{\sqrt{2}},$$  \hspace{0.5cm} (1)$$

where $I(u,v)$ is the image intensity value at $(u,v)$, and $\sqrt{2}$ is a normalization for the length of the edge. Similarly, for the horizontal edge marked 2 in Fig. 1 we may estimate the intensity derivative as

$$D_{(i,j),(i+1,j)} = \left(\frac{I(i,j-1) + I(i+1,j-1)}{2} - \frac{I(i,j+1) + I(i+1,j+1)}{2}\right)/2.$$ \hspace{0.5cm} (2)$$

Similarly, for the vertical links marked 1, we may estimate the magnitude of the derivative as

$$D_{(i,j),(i,j-1)} = \left(\frac{I(i-1,j) + I(i-1,j-1)}{2} - \frac{I(i+1,j) + I(i+1,j-1)}{2}\right)/2.$$ \hspace{0.5cm} (3)$$

If $u$ and $v$ denote pixels, and their corresponding edge strength is given as $D_{u,v}$, let $\Delta = \max_{u,v} D_{u,v}$. In other words, $\Delta$ is the maximum edge strength in the image. We may give the cost, $c$ for an edge between pixels $x$ and $y$ as

$$c_{x,y} = (\Delta - D_{x,y})\text{length}(x,y).$$ \hspace{0.5cm} (4)$$

Note that in an image graph length$(x,y)$ will be either 1 or $\sqrt{2}$. In (4), high $D_{x,y}$ values will map to small $c_{x,y}$ and small $D_{x,y}$ values will map to high $c_{x,y}$ values, giving us the necessary cost definition for a shortest path construction.

For your convenience, the shortest path algorithm is briefly summarized in the Appendix, where $x \sim y$ denotes that $x$ and $y$ are neighbors (adjacent).

**Problem 2**

(Hough Transform, **35pts**) The purpose of this component is to get familiar with the Hough transform for shape detection. You will need to implement the Hough transform on your own. The matlab built-in function `hough()` is not allowed.

---

Instructor: Vikas Singh (vsingh@biostat.wisc.edu), TA: Won Hwa Kim (wonhwa@cs.wisc.edu)  Page 2 of 4
1. **(10pts)** Write a program to automatically identify straight lines in an image: \( \text{lines} = \text{myHoughLine(imBW, n)} \), where \( \text{imBW} \) is a binary image and \( n \) is the desired number of lines (in order of voting importance). If there are not enough \( n \) lines in an image, return as many as your implementation detected. Test with \( n = 5 \), and plot your lines along with a few images in your report.

2. **(25pts)** If you haven’t already, read Generalizing the Hough transform to detect arbitrary shapes (Ballard, 1981) and review Lecture Note 5 from class. Write a program to find (roughly) circular objects of a specific radius in a binary image. You should write two functions, \( \text{yourcellvar} = \text{myHoughCircleTrain(imBW, c, ptlist)} \), and \( \text{myHoughCircleTest(imBWnew, yourcellvar)} \). The binary image \( \text{imBW} \) supplied to the first function will have a single circular object. Further, \( c \) will be the (given) reference point, and for convenience we will also provide you an ordered list of boundary points in \( \text{ptlist} \). Use this information to construct whatever data structure you want (and save all tabular data necessary for the subsequent step), and return it as a cell array variable, \( \text{yourcellvar} \). That is, \( \text{yourcellvar} \) is the object \( \text{myHoughCircleTrain} \) should return. Next, this will be passed directly to \( \text{myHoughCircleTest} \) where you will identify circular objects in a novel image, \( \text{imBWnew} \). Your function should report the reference points for the top two circles identified. You will receive full credit if your reference point is close enough.

Test with your own images and discuss your results with these images in your report.

**Problem 3**

(Snakes/Active contour models, **30pts**) In this problem, you are asked to implement the dynamic-programming snake algorithm as described in Using Dynamic Programming for Solving Variational Problems in Vision (Amini et al., 1990) and Lecture Note 6 from the class. You should provide one function \( \text{imOut} = \text{mySnake(imIn, imInitial, alpha, beta)} \), where \( \text{imIn} \) is a gray scale image, \( \text{imInitial} \) is a binary image for your initial curve, \( \alpha, \beta \) will be your parameters, \( \text{imOut} \) is your output image with a boundary on top of the input image (in fact, just the boundary will be OK as well).

Your function may include first-order and second-order derivative terms for the internal energy (i.e., \( \alpha, \beta \)), and allow each point to move to one of nine positions in its immediate vicinity. For the external energy, you may use the negative of the magnitude of the gradient. For simplicity, restrict your implementation to work only with closed curves. Start your snake from an initial curve that is larger than the object (you can capture this information via \( \text{ginput} \)) and display its evolution over time (say 10 steps).

Test your algorithm on the test images with different parameters for \( \alpha, \beta \), number and location of points. Discuss your results in your report.

Instructor: Vikas Singh (vsingh@biostat.wisc.edu), TA: Won Hwa Kim (wonhwa@cs.wisc.edu)   Page 3 of 4
Appendix

Algorithm 1 Shortest path

- Initialize the priority queue \( Q \) to be empty (we will maintain \( Q \) to keep track of nodes)
- Initialize each node to the UNVISITED state
- Set the total distance of the “seed” to be zero, every other node has a distance of \( \infty \).
- Insert seed into \( Q \)
  //Now find the shortest path
  \textbf{while} \( Q \) is NOT empty and \( q \neq dest \) \textbf{do}
  \hspace{1em} Extract a node \( q \in Q \), which has the minimum distance (for first operation, we will extract the seed node).
  \hspace{1em} //formally, extracted node corresponds to \( \min_{q \in Q} dist(q) \)
  \hspace{1em} mark \( q \) as VISITED
  \hspace{1em} for each \( r \) where \( r \sim q \) (i.e., in \( q \)’s neighborhood) \textbf{do}
    \hspace{2em} if \( r \) is UNSIGHTED (i.e., \( dist(r) = \infty \)) \textbf{then}
      \hspace{3em} mark \( r \) as SIGHTED
      \hspace{3em} Insert \( r \) in \( Q \) s.t. \( dist(r) = dist(q) + c_{r,q} \) (distance to \( q \) plus the cost of \( q \) to \( r \)).
      \hspace{3em} make an entry for \( r \) indicating that currently the best way to reach \( r \) is from \( q \).
    \hspace{2em} \textbf{else}
      \hspace{3em} if \( r \) is SIGHTED \textbf{then}
        \hspace{4em} Check if path through \( q \) to \( r \) is less than the current best known path to \( r \): \( dist(q) + c_{r,q} < dist(r) \)
        \hspace{4em} If yes, update \( r \)’s entry indicating that currently the best way to reach \( r \) is from \( q \).
        \hspace{4em} update \( dist(r) \)
      \hspace{2em} \textbf{end if}
    \hspace{2em} \textbf{end if}
  \hspace{1em} \textbf{end for}
\hspace{1em} Extract the node \( q \) with the minimum total cost in \( Q \)
\hspace{1em} \textbf{end while}