Lab 5 - Active Contouring

John Lawler

October 27, 2020

FUNCTIONALITY

There are two main algorithmic blocks in my program: the first part of the program calculates the sobel gradient, and the second part calculates and sums three different energy terms to produce the overall energy at each pixel. The energy on each pixel is used as a weight to pull a contour point towards its next best spot inside a given window of pixels.

The contour points are first stored from a text file for use later in calculating the internal energies. For calculating the sobel gradient, a 3 x 3 sobel gradient template matrix is convolved across the original input image (hawk.ppm). The convolved pixel values are normalized between 0 and 255 to output, and the output is shown in Figure 1. Next, The sobel gradient intensities (prior to the 0 to 255 normalization) are negated and normalized again to between 0 and 1. These values represent the external energies, and will be used later in the code for summation of all energies. The external energies are calculated one time and stored to their own array that is the size of the image; these remain static throughout the program and are not changed. The internal energies, however, are recalculated on each pass through of the contour points for determining their next best location.

The internal energies are calculated for each pixel inside a 7 by 7 window around every control point. This occurs on every control point (42 given points) for a total of 30 iterations. Later, I found that performing only 12 iterations provides the best fit results. The first internal energy is found via squaring the distance between the currently selected control point and the succeeding control point; i.e. every pixel's internal energy in the 7 by 7 window around the selected control point is found via its relative distance to the next control point. Once all the first internal energies are found, they are normalized to between 0 and 1. Next, the second internal energy is found on the selected control point via finding the deviation to the succeeding point. For this, the average distance between control points is calculated once for each overall iteration through the points. This distance is subtracted from the currently selected control point's distance to the succeeding point. Once subtracted, it is then squared, and this is performed on every pixel in a 7 by 7 window around every control point. Once all the second internal energies are found in a given window, they are normalized between 0 and 1. Now that we have two windows storing the normalized internal energies, the original external energy array is referenced at the corresponding control point's position to sum the two internal energies with the external energy. The resultant summed 7 by 7 window will not exceed energies of 3. The resulting lowest energy in the window is deemed the successor control point for the currently selected control point, and stored into a temporary array; once all control points have been passed through one time, then the resultant temporary array of new positional control points is set to equal the original control points, and the process is repeated once again.

Once the process has been performed for the set iterative count, the resulting control points are printed to the original image as shown in Figures 3 and 4.

DISCUSSION OF RESULTS

The result of the sobel gradient, prior to normalization and negation, is shown in Figure 1. I found using a sobel gradient template of a 3 by 3 worked nicely for generating distinct white outlines around the edges in the image. It also generated some slightly less strong edge cases within each object of the image; however, since the given contour points (shown in Figure 2) strictly outline the hawk alone, there is no worry of accidentally catching edge cases within the hawk or other objects. Figure 2 shows the initial contour points provided and their respective location on the original image given.

Results are shown in Figures 3 and 4. I found that iterating 30 times (as the lab requires) actually over-contoured the outline of the hawk, resulting in having multiple contour points on the same row and column of the image. The beginning contour point (at index of 0 in my contour point arrays for column and row) is saved to the end of the contour arrays to allow the ending contour point to loop back to the front again. Before completion, I forgot to reset the final contour points at the end of each contour array, and this caused the program to print out a plethora of repetitive contour points. This was fixed by resetting the end contour point (which should be the beginning point to maintain looping) to the beginning point at the end of every iteration through the contour points; so every new set of contour points also had the newly calculated initial contour point set the the end again (instead of maintaining the original initial contour point for all iterations which originally broke the contouring).

After reviewing the contours produced from the 30 iterations, I played around with the iteration count until I found a good point where all contours are still shown and not repeated. The iteration count for optimal contouring, in my scenario, was around 10 to 15 iterations. I show the results and contours of using an iterative count of 12 in Figure 3, which fits the edges of the hawk much better (with better uniformity) than an iterative count of 30.

DATA TABLES AND FIGURES

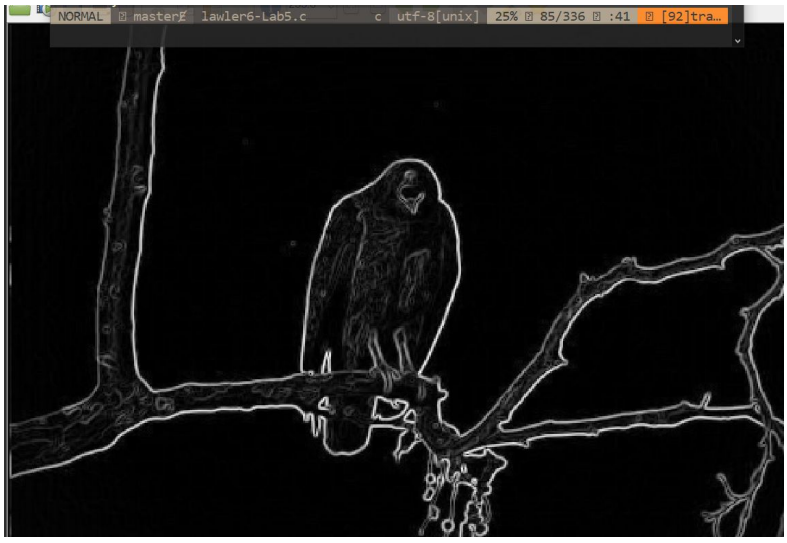

Figure 1: Sobel Gradient Output

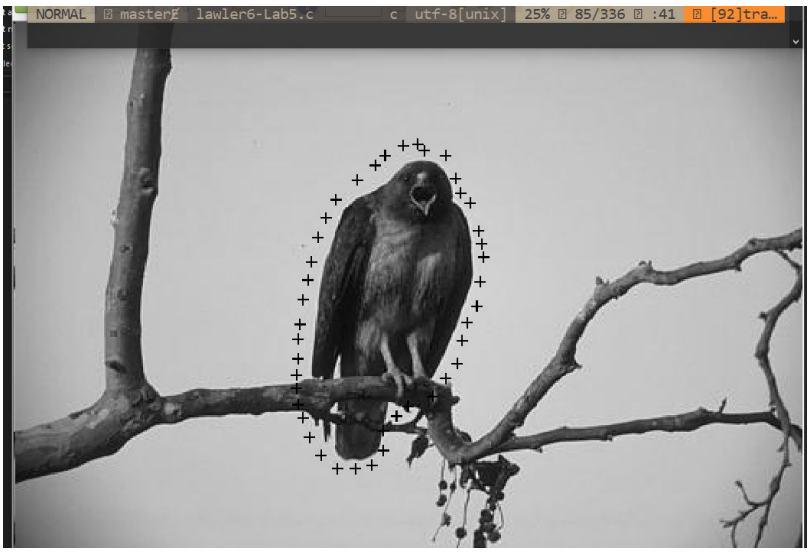

Figure 2: Initial Contours on the Image

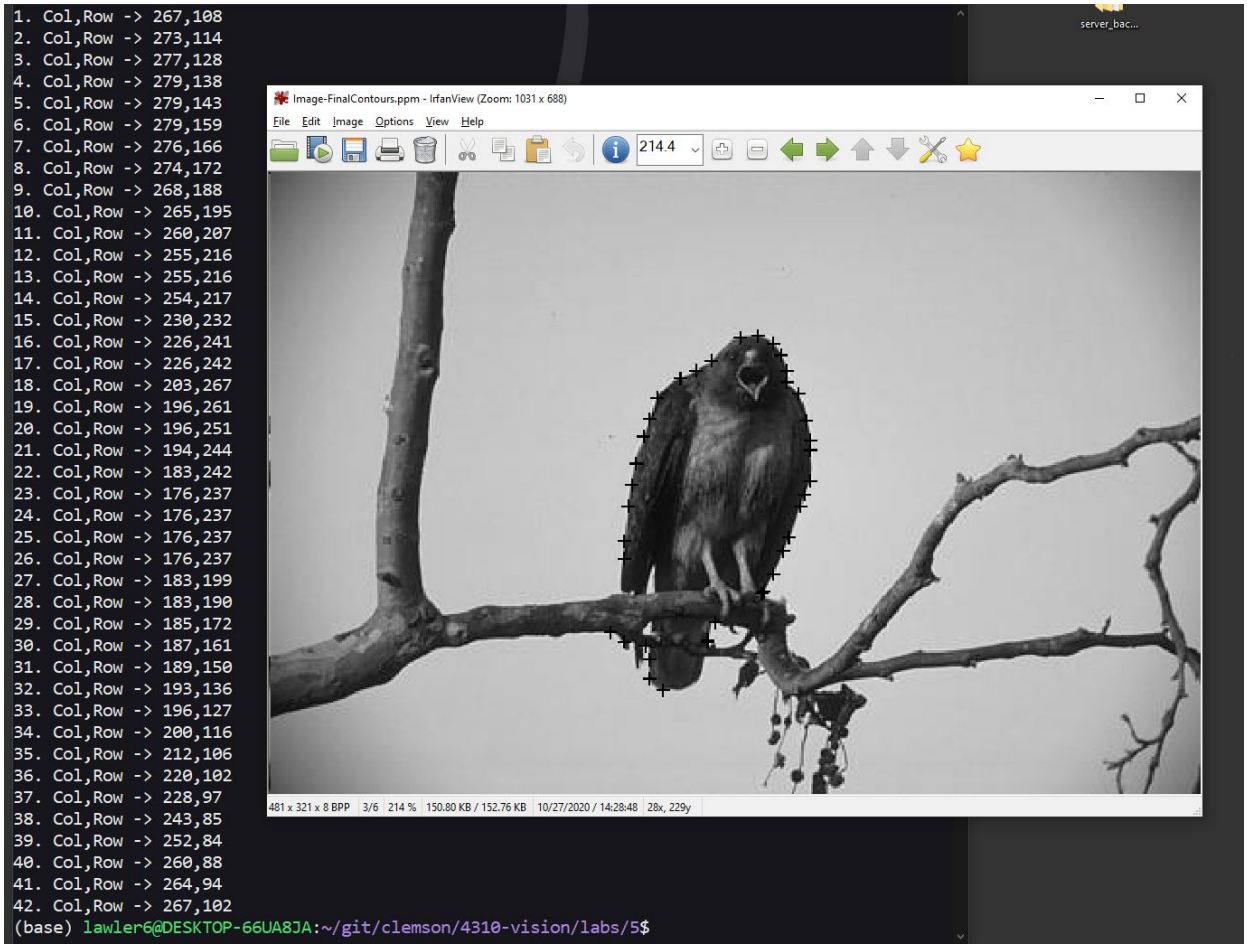

Figure 3: Contouring Final Result Contour points (with 12 iterations)

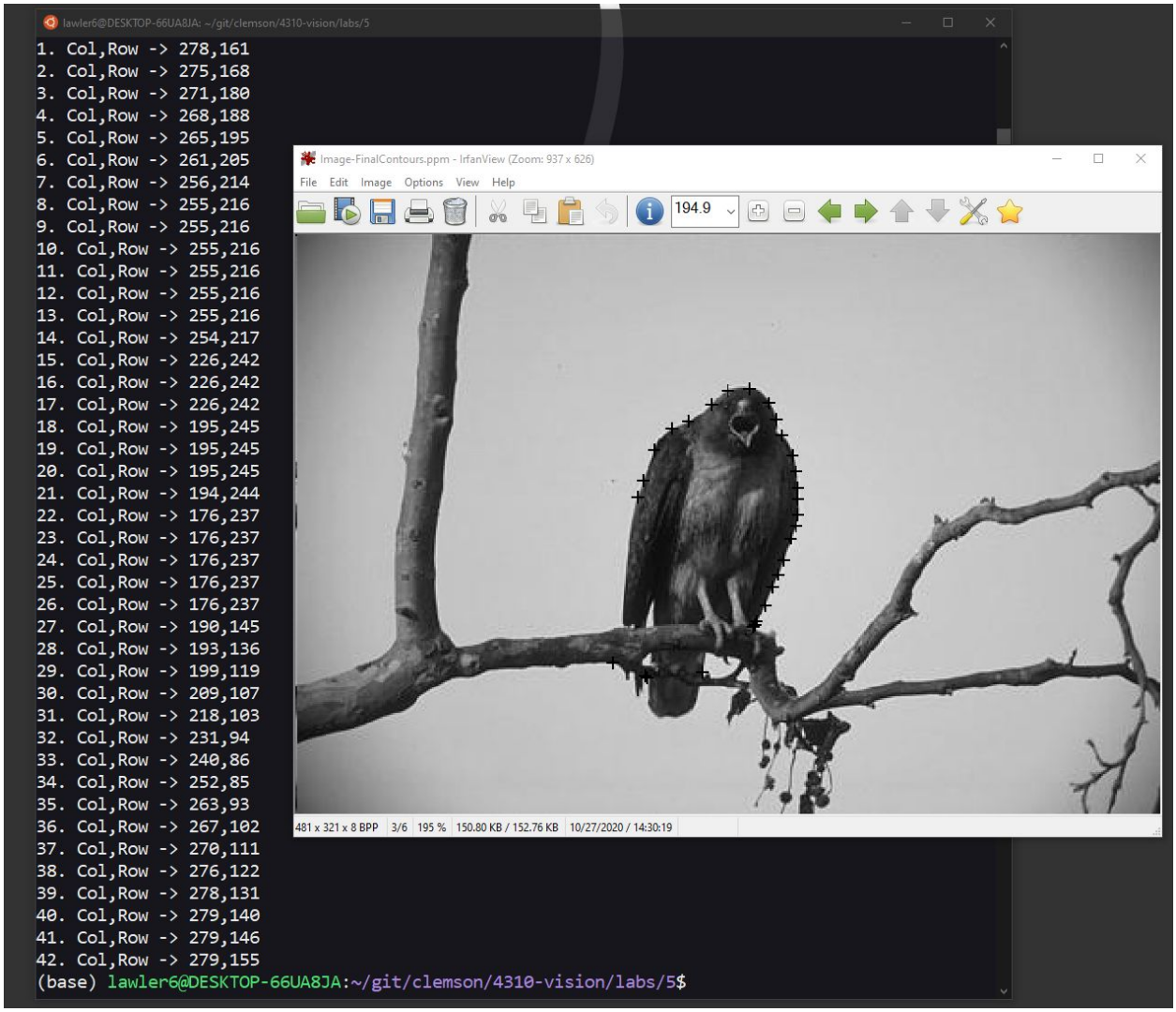

Figure 4: Final Contour Points (with 30 iterations)