Canny Edge Detection Report

Authors: Yunoo Kim (yk646), Corey Wang (clw233)

Introduction

In machine learning it is often beneficial to reduce the dimensionality of the data, and one of the ways of doing so is with edge detection, where only the edges of certain objects in the image are kept. The Canny algorithm is one way to detect edges, and involves 6 steps: grayscale conversion, blurring, gradient calculation, non-maximum suppression, double thresholding, and hysteresis.

Brief description of each implemented step

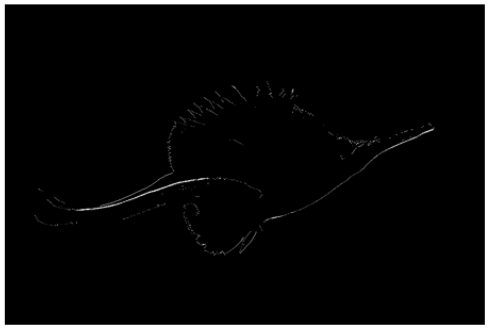
- **Step 1**: We read the image using matplotlib, exiting gracefully if the image file is not found. Then, we calculate the intensities of the pixels to produce a grayscale image.
- **Step 2**: We convert the image from a numpy array representation into a pillow image representation, which allows us to use pillow's filter method to apply the 3x3 gaussian kernel to the image, which creates a blurring effect.
- **Step 3**: We first initialize the Gx and Gy matrices provided in the slides, and then we used the filter method again to apply it to the image to get the image gradients. Finally, for each pixel, we take the maximum element of the Gx and Gy matrix to get the resulting gradient. We also calculate the gradient magnitude and direction, which is used in the next step.
- **Step 4**: We created a function 'nmsuppression' that took in x and y coordinates of a pixel as arguments. We checked the direction of the pixel by checking if its theta value (from Task 3) was greater or not than pi/4. If it was, then we checked if it was a local maximum in terms of magnitude to the pixels above and below it in the y direction; if it wasn't, then we checked if it was a local maximum to the pixels left and right of it in the x direction. If it was a local maximum, we did nothing. If it wasn't, we set its magnitude to 0 to discard it.
- **Step 5**: We created a function 'doublethreshold' that took in x and y coordinates of a pixel as arguments. We set the upper and lower thresholds to 100 and 50 respectively, as higher values often resulted in a pale, unclear image, and lower ones resulted in noisy images. We then compared the pixel's values to these thresholds and changed its value to 255, 150, or 0 depending on the results.

- **Step 6**: Using a similar structure to the previous tasks, we created a function ‘hysteresis’ that compared a pixel’s coordinates to its immediate neighbors. If it was neighbors with a strong edge pixel, we changed its intensity to 255. If it was not, we changed the intensity to 0, effectively discarding it. We then broadcasted the arrays for the x and y coordinates of weak pixels using ‘hysteresis’, creating a new image.

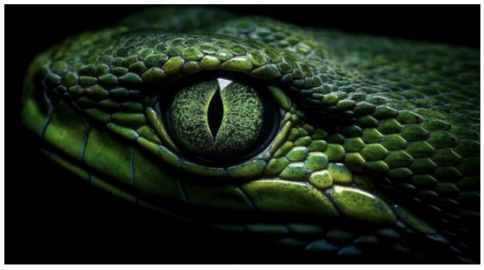
Evaluation with visual examples

Before

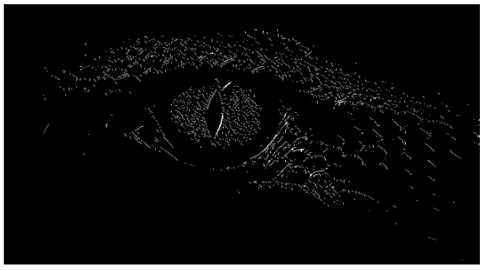
After



Before



After



Challenges

One of the challenges we faced was when we wanted to make a function to vectorize steps 4, 5, and 6. We tried making a 2d matrix where each element contained a tuple that represented its index, sothe index (3,4) would be in row 3 and column 4. However, running the vectorized function resulted in each individual index being put into the argument, so instead of the argument being (3,4), the function received 2 calls, the first being with argument 3 and the second being with argument 4. To work around this, we created 2 lists, one of them containing the x values and one of them containing the y values, and then called the vectorized function with 2 inputs: the x values and the y values. This allowed us to efficiently vectorize steps 4, 5, and 6, reducing the runtime drastically in comparison to nested for-loops.