Contour Detection

Method Description

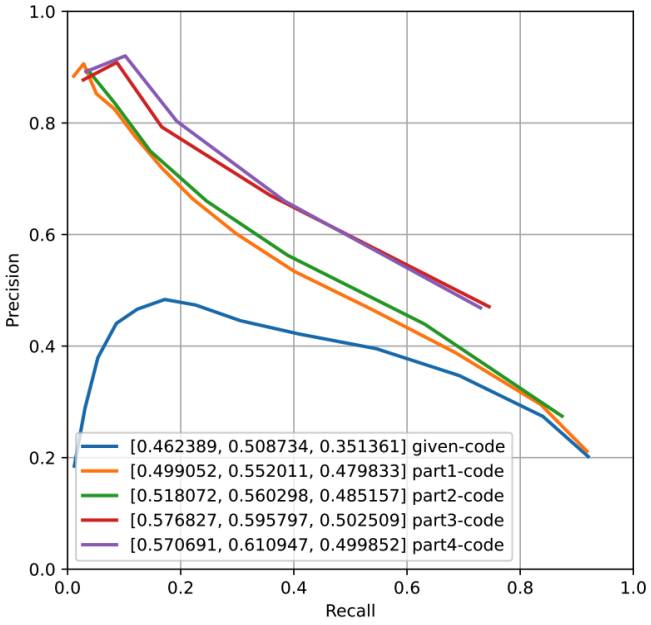
Warm-up
To minimize the artifact, we should change the mode in the convolve2d function. In the original function, we use boundary='fill', which pads a default value if a pixel is out of bound. To mitigate the artifact, we can make the padding more smoothly, such as boundary='symm'. If the pixel is out of bound, we pick the mirrored pixel (i.e. the pixel with the same distance to edge) as the out-of-bound pixel value.

Smoothing
I use scipy.ndimage.gaussian_gradient_magnitude to apply a derivative of the Gaussian filter on the image. The Gaussian filter removes the high frequency noise and smooths the frequency curve, making the edge detector produce better edges. In my experiment, sigma is set to 0.5, 1, 2, 3. It turns out setting sigma=1 produces the best result on both F1(avg=0.582, overall=0.537) and AP(0.503).

Non-maximum Suppression
According to lectures, I compute the gradient of each pixel and find the direction of each gradient. After that, I run a nested for loop. Each time, I compare mag[i][j] with their neighbors along the same direction. The direction will be calculated from the direction of the gradient (unit vector). Then, we use interpolation to find the magnitude of the neighbors, since their coordinates are floating points.

Bonus (scaling)
I combine results (into BIG mag) from different scaling of the image, with scale = [1, 1/2, 1/3, 1/4]. We need to rescale the image before and after the original calculation. After computing the rescaled mag, we combine it with the BIG mag. This method can capture both coarse and fine grain context, thus has a greater outcome.

Precision Recall Plot



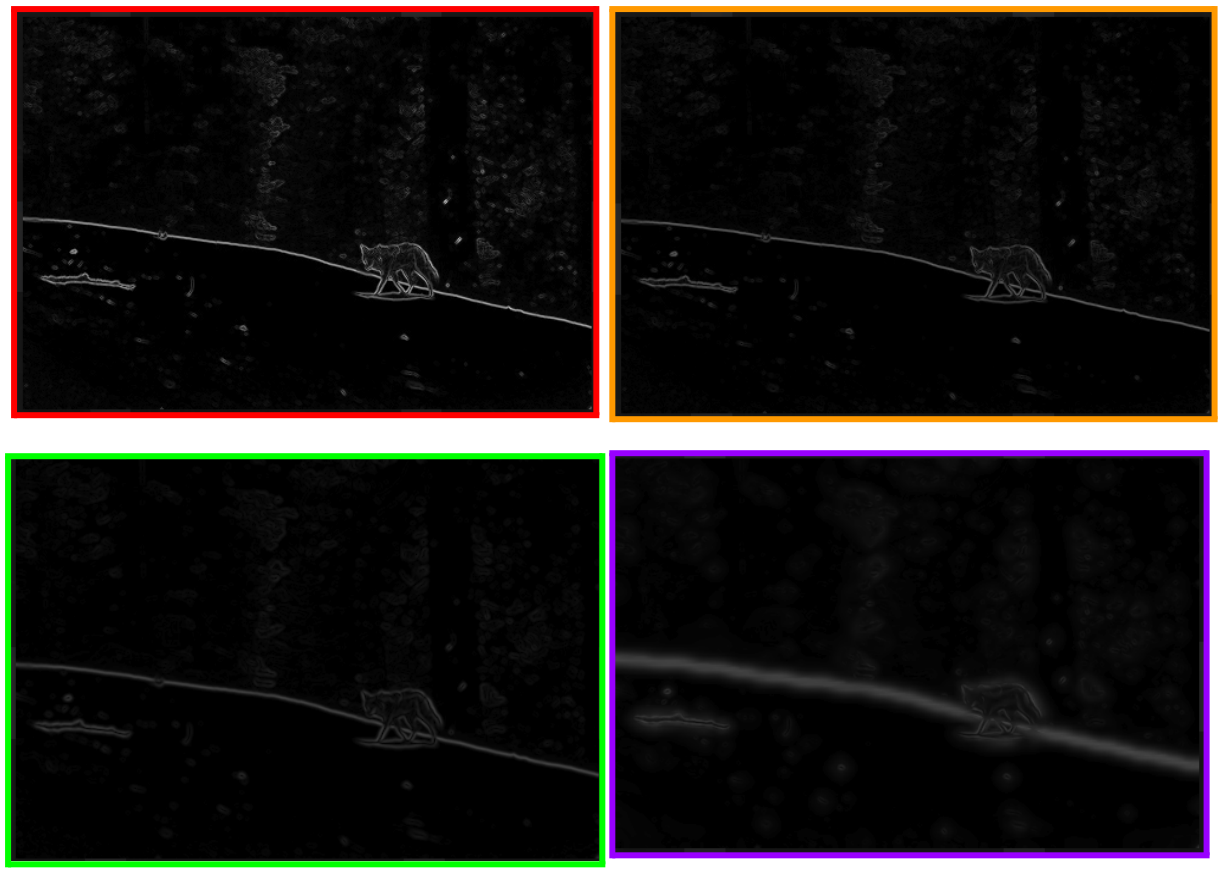
Results Table

Method	overall max F-score	average max F-score	AP	Runtime (seconds)
Initial implementation	0.450	0.516	0.385	0.010
Warm-up [remove boundary artifacts]	0.499	0.552	0.479	0.016
Smoothing	0.537	0.582	0.503	0.003563
Non-maximum Suppression	0.576	0.595	0.502	9.16
Val set numbers of best model [From gradescope]	0.571	0.611	0.500	13.46

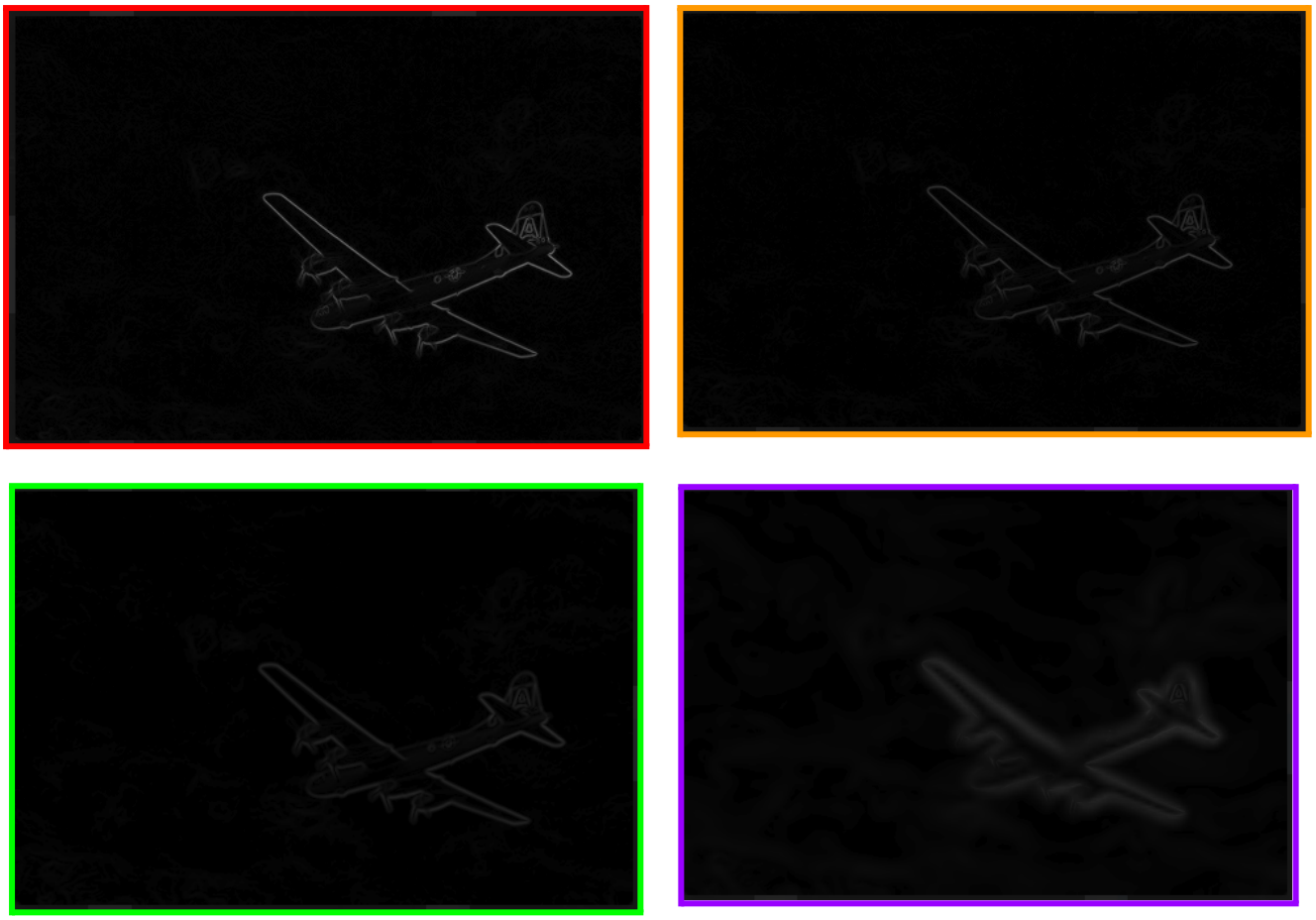
Visualizations

Describe the effect of your implementation for each part on these images by pointing to relevant image regions. Comment on where does your code work well, where it doesn't and why? You may also add visualizations of your own images. Here is how you can lay out images in markdown.

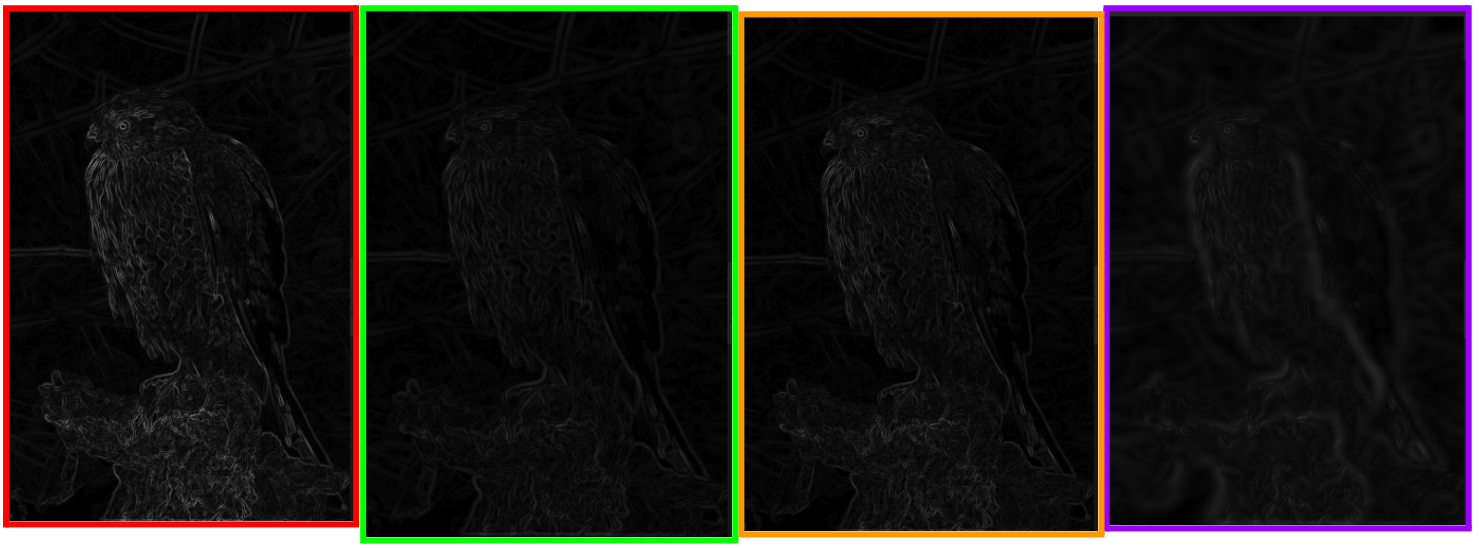
part1 = red, part2 = green, part3 = orange, part4 = purple



In part 1, the image contains a lot of noise. In part 2, after applying the Gaussian filter, much of the noise has been significantly reduced. In part 3, the lines have been substantially thinned. For the final part, we select the maximum parts to capture both coarse and fine-grain edges, enhancing the clarity and detail of the image's features.



Similar to the above examples (like a wolf walking beside a lake), for the airplane image, part 1 showcases a lot of details (not just edges), including things like the logo on the wing. Part 2 eliminates some of the thinner lines, yet some small residual lines remain, with the engine part still displaying small lines. Part 3 goes further in removing these redundant lines. Part 4 merges both coarse and fine-grain details, streamlining the image while retaining essential features.



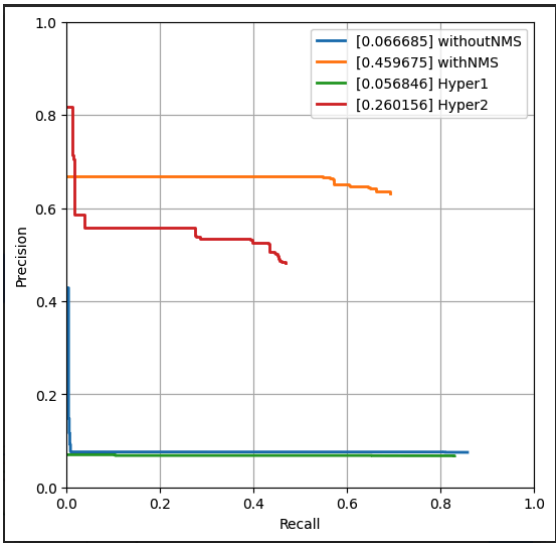
Similar to the above examples (such as a wolf walking beside a lake), for the airplane image, we can observe that part 1 contains a lot of details on the cloth and hair (not edges). Part 2 eliminates some thin lines, yet some small residual lines remain visible, especially around the engine part which still shows small lines. Part 3 further removes these redundant lines. Part 4 merges both coarse and fine-grain details. Notably, in part 4, the details have been removed and blurred, resulting in only the large outlines being extracted.

Corner Detection

Method Description

In this section, I implemented the Harris corner detector. First, I calculate the R matrix, then filter out some of the smaller values through a threshold. Next, I use a sliding window to select the most representative points (i.e., the points with the highest values) within a region and filter out the other points. This way, all the corners are selected.

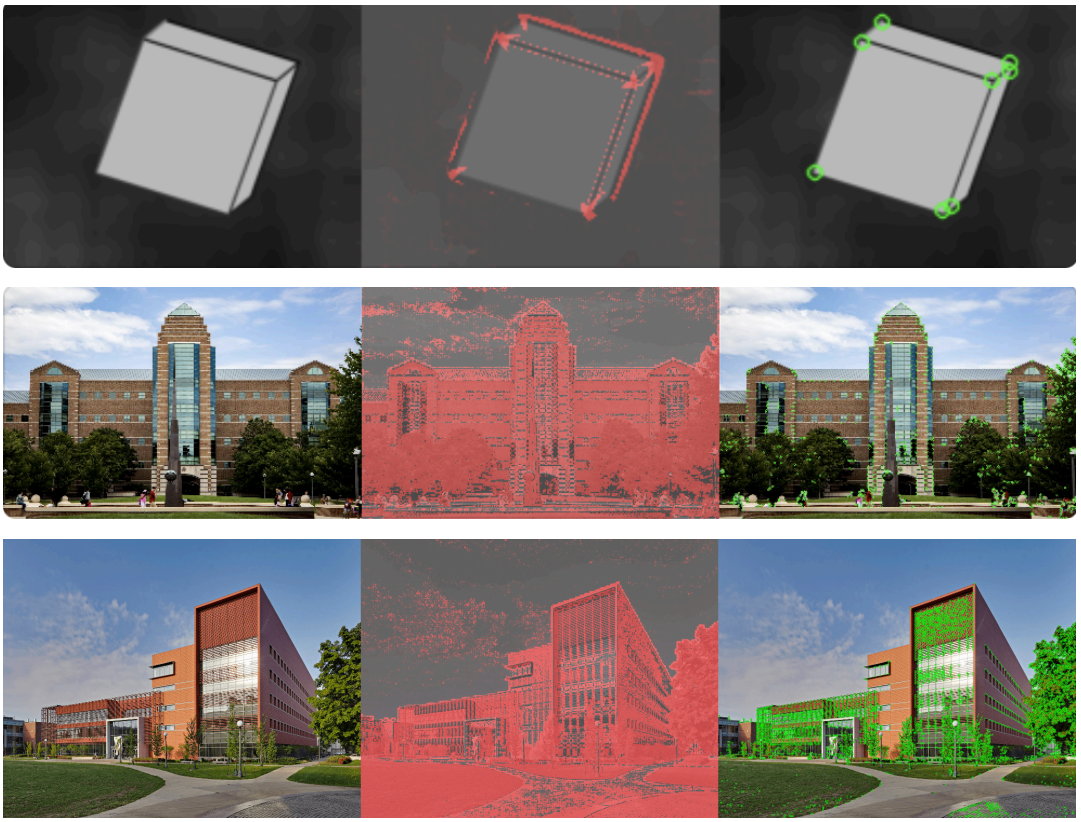
Precision Recall Plot



Results Table

Method	Average Precision	Runtime
Random	0.002	0.001
Harris w/o NMS	0.066	0.002
Harris w/ NMS	0.459	0.131
Hyper-parameters tried (1) (alpha=0.06, k=5, threshold = 0.02*max(R), sigma=1)	0.056846	0.159095
Hyper-parameters tried (2) (alpha=0.06, k=5, threshold = 0.015 * max(R), sigma=2)	0.26	0.113
Val set numbers of best model [From gradescope]	0.459	0.091

Visualizations



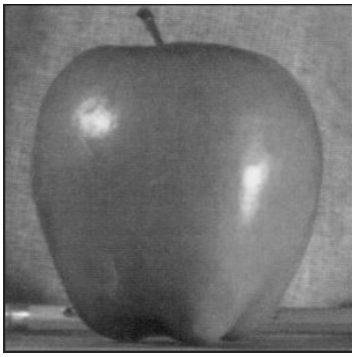
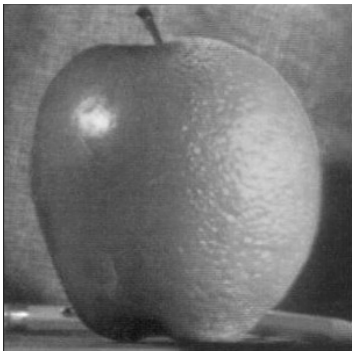
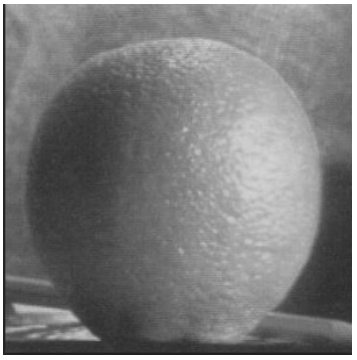
Among the three images, the cube performs the best, despite not identifying some backward points. The Beckman building's corners are mostly captured by the program, with no dots on the edges, indicating good performance. However, for the ECE building, the program captures too many points on trees and small windows, likely due to aliasing effects on the outer walls, making it easier for the program to mistakenly identify these areas as corners due to significant differences in neighboring pixels.

Multi-resolution Blending

Method Description

My approach involves using Gaussian difference to calculate the Laplacian pyramid, and then computing the matrix for each level of the pyramid. Each layer is then merged together to form a single matrix, then it will be normalized.

Oraple

Blends of your choice. *TODO:* Include visualizations of blends of your choice (include both original images and the blended image). Describe any modifications you made on top of what worked for the oraple.



In my image blending project, I selected two distinct faces for blending. I focused on choosing images with clear edges and similarities in boundaries (positions of hair and face) and adjusted the Gaussian kernel parameters to better capture image details. This adjustment resulted in a more natural and visually appealing blended image.