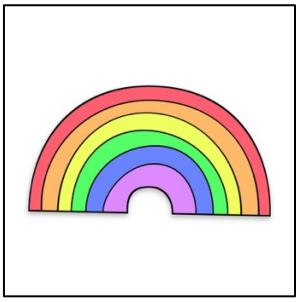
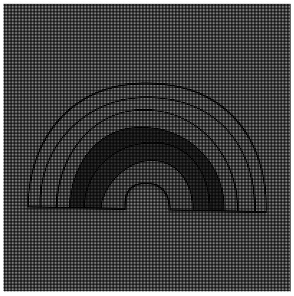
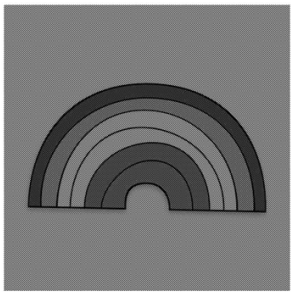
Daniel Korsunsky

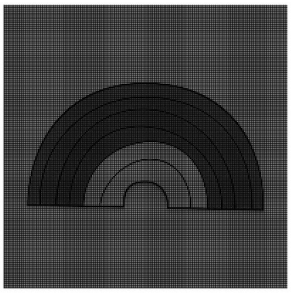
Question 1: Demosaicing

Part (a)

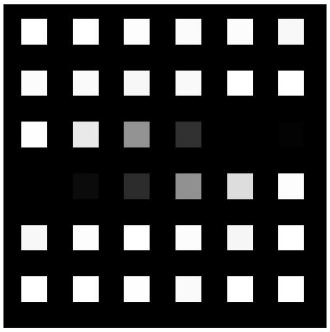

Original image

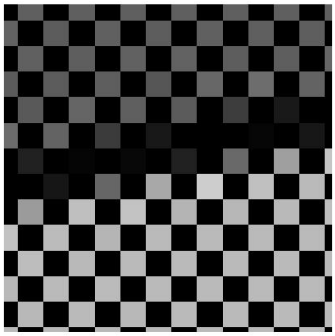

Red channel

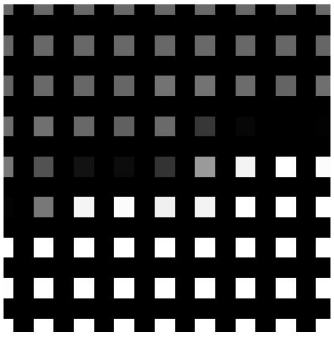

Green channel


Blue channel

Close-ups of channels:


Red channel


Green channel


Blue channel

The red channel has preserved R values in every even row and every odd column (zero values in ¾ of the pixels). The green channel has preserved G values in every odd column of every odd row and every even column of every even row (zero values in ½ of the pixels. The blue channel has preserved G values in every odd row and every even column (zero values in ¾ of the pixels).

Part (b)

Red and blue filter:

¼	½	¼
½	1	½
¼	½	¼

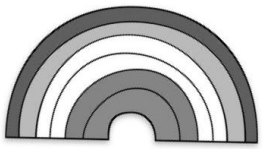
Green filter:

0	¼	0
¼	1	¼
0	¼	0

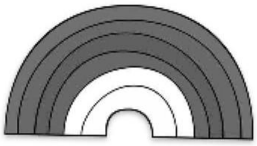
- *Red and blue filter*: The ¼ values are responsible for interpolating the missing R or B value at a B or R position, respectively. The ½ values are responsible for interpolating the missing R or B value at a G position: the left and right ½ values are responsible for a horizontal BGB or RGR sequence, while the top and bottom ½ values are responsible for a vertical BGB or RGR sequence. The central 1 value preserves the value of an existing (non-missing) R or B tile.
- *Green filter*: The ¼ values are responsible for interpolating the missing G value at a B or R position. The central 1 value preserves the value of an existing (non-missing) G tile.

Restored channels:

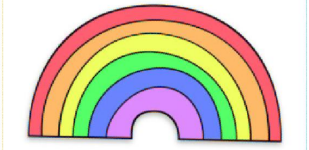
Red channel



Green channel



Blue channel



Restored image

Notice the artifact produced at the image borders. This is discussed in Part (c).

Part (c)

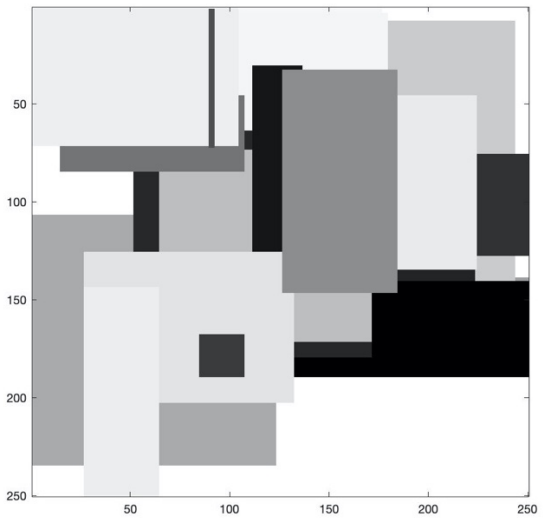
Flag of Ukraine (original)

Flag of Ukraine (post-demosaicing)

Artifacts: This simple demosaicing method produces dotted strips of pixels along sharp intensity edges (and image boundaries) that noticeably differ in color from either of the color regions. This is because the red, green, or blue values in pixels on opposite sides of the sharp intensity edge have very different values despite being in the same neighborhood. The filters from Part (b) attempt to restore these RGB values by averaging across the sharp intensity edge, which instead produces artifact values that differ significantly from either of the color regions.

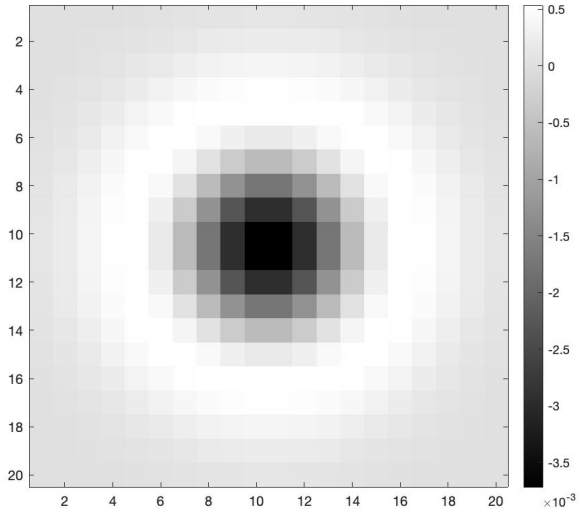
Question 2: Edge Detection

Part (a)



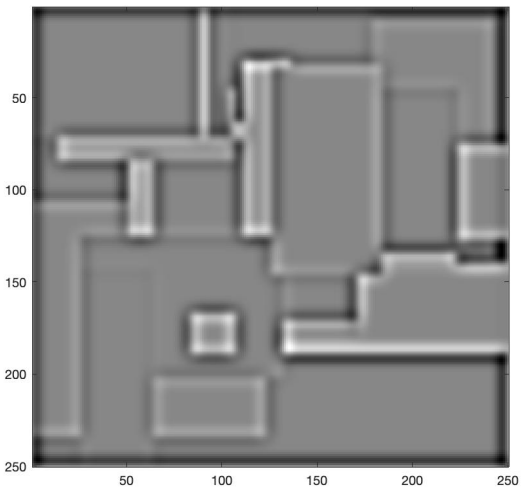
Original image (20 rectangles)

Part (b)

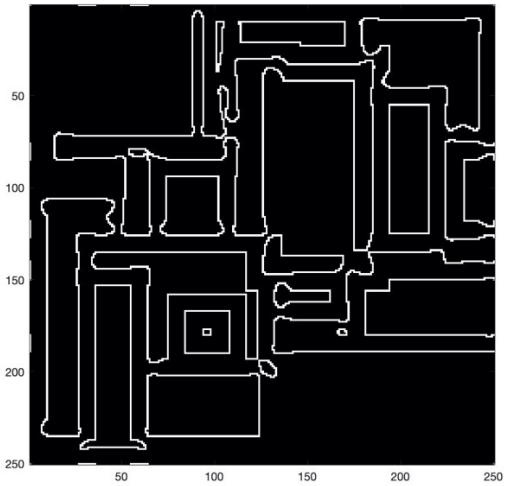


Laplacian of a Gaussian (20 x 20, σ = 3)

Part (c)

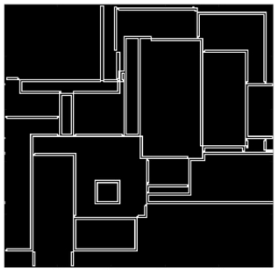


Original image convolved with LoG



Binary image of zero crossings

The existence of curvy edges and wide double edges in the binary image of zero crossings can be attributed to the significant blurring produced by convolving the image with the very large Laplacian of a Gaussian filter of size 20 and sigma 3. The zero crossings are much straighter and more precise when the image is convolved with a LoG filter of, say, size 5 and sigma 0.5 seen to the right:



Part (d)

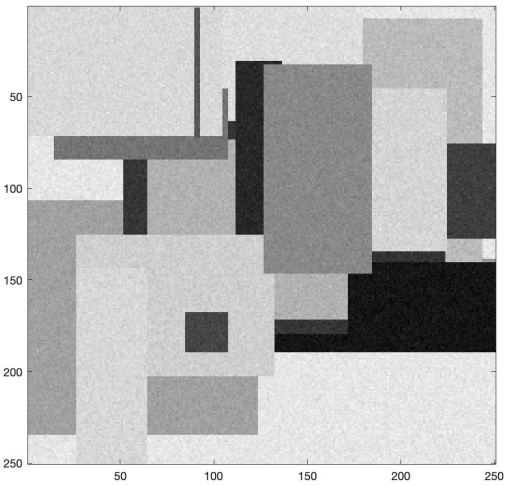
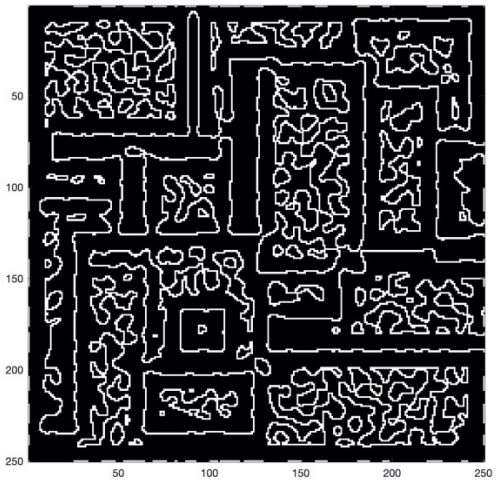


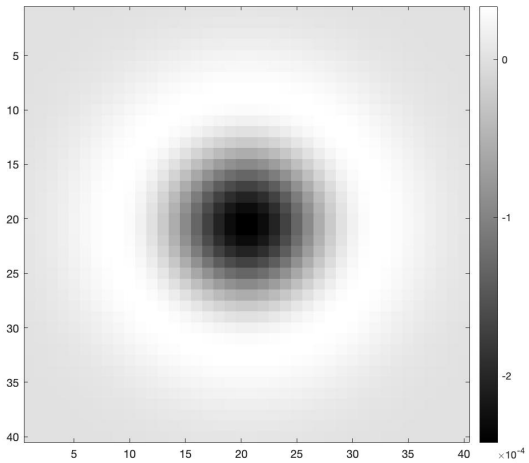
Image with added noise



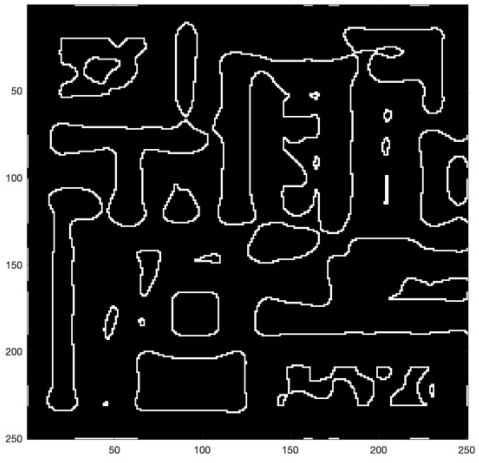
Binary image of zero crossings

The straight zero crossings that define rectangle boundaries are largely preserved along sharp intensity edges, where the added noise does not have as significant of an effect. The squiggly zero crossings largely appear within rectangles, where noise creates sharp differences across neighboring pixels that are interpreted as zero crossings.

Part (e)



Laplacian of a Gaussian (40 x 40, σ = 6)

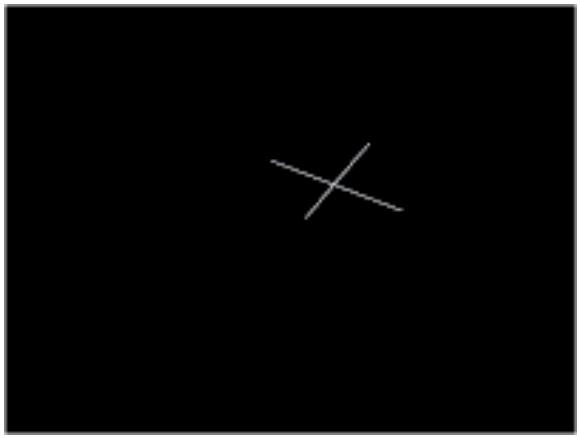


Binary image of zero crossings

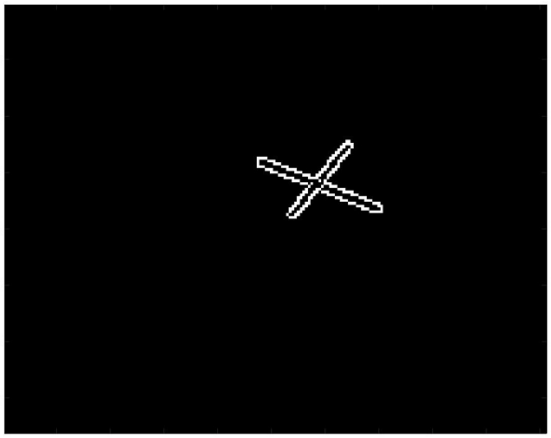
The zero crossings that correspond to the sharpest (most change in intensity) rectangle boundaries are somewhat conserved. Since the filtered image is more blurred by the larger LoG filter, the edges between rectangles become thicker and less pronounced, while the noise is smoothed over more significantly. Thus, the zero crossings become smoother and more rounded, as well as significantly less dense.

Question 3: Vanishing Points via Hough Transform

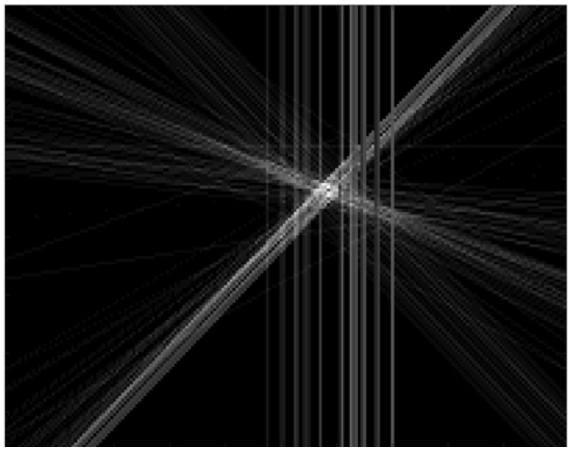
Part (a)



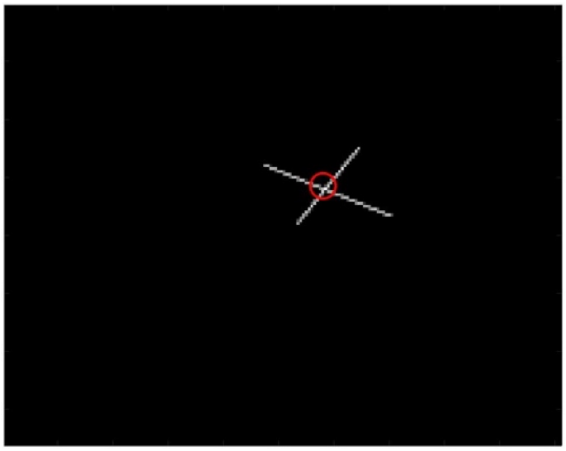
lines.jpg (made in Illustrator)



Edges computed by Canny method



Lines from Hough transform

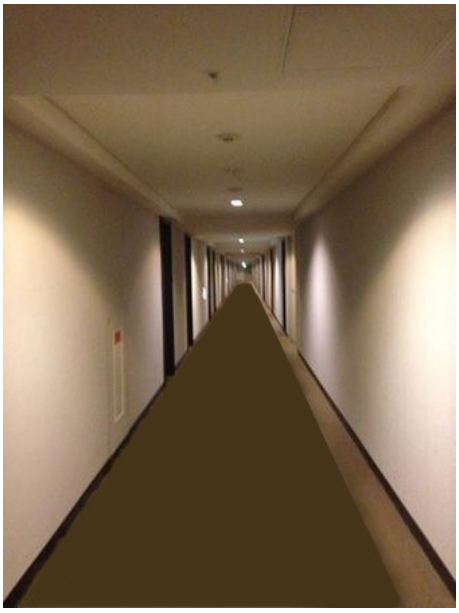


Peak of Hough transform in image

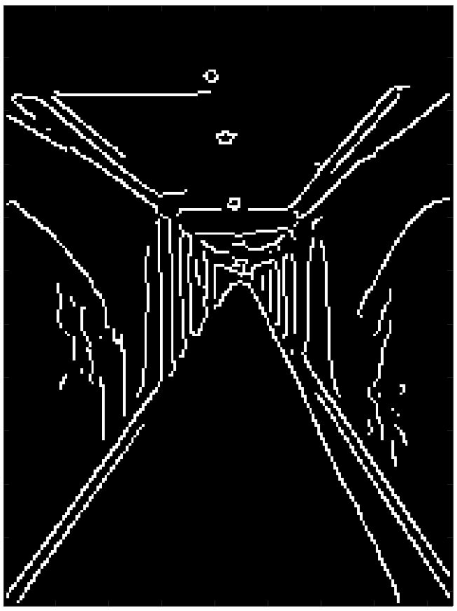
Note: All of the vertical lines correspond to script-computed edges with $\sin\theta_i = 0$.

In the artificially drawn **lines.jpg** image, the Hough transform algorithm accurately identifies the intersection point of the two lines. I argue that the slight error in precision in this example is due to the small image dimensions (150 x 200 pixels), so the gradients calculated by the Canny method are not precisely perpendicular to the lines. Thus, there is significant variability in the calculated slope of the line candidates from edge to edge, as can be seen in the bottom left image. This variability slightly decreases the precision of the vanishing point location.

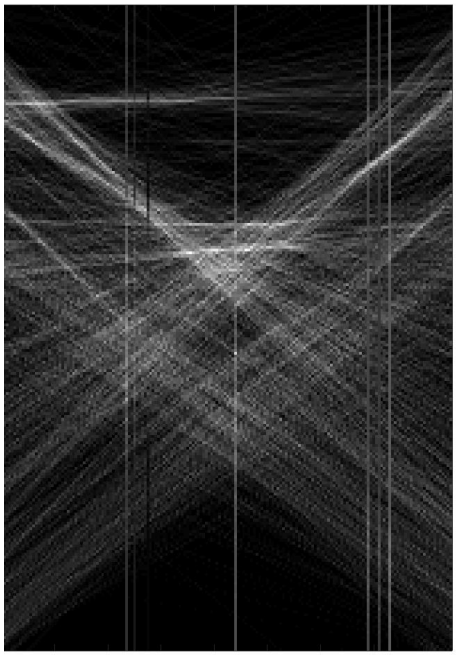
As can be seen in the lines calculated by the Hough algorithm,



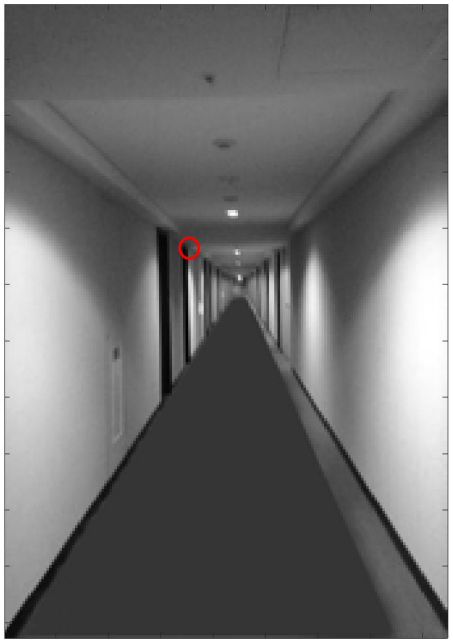
Original image (hallway.jpg)



Edges computed by Canny method



Lines computed by Hough transform



Peak of Hough transform in image

Discussion of these results is on the following page.

Given the **hallway.jpg** image, the Hough algorithm computes an incorrect vanishing point location. However, this is not because the algorithm calculates an edge's line candidate incorrectly. As you can see in the bottom left image, the algorithm does most frequently identify the four diagonal lines that converge at the end of the hallway, as required. However, the light fixtures on the ceiling create a significant number of horizontal edges that the algorithm also detects. Thus, the reason the algorithm calculates the vanishing point seen in the bottom right image is because these horizontal lines from the ceiling lighting intersect with the hallway diagonals. The main fault of the algorithm that I was not able to figure out is how to weigh the hallway lines more heavily than the ceiling light lines.

Part (b)

Part (a) method: **E * (√2)N**.

> The method votes for every point along the line for each of E edges. The worst case line is length (√2)N, which is the diagonal of the image. Thus, the computational complexity of this method is **E * (√2)N**.

Alternative method: $\mathbf{E^2}$

> This method calculates one point (at the intersection of two lines) for each pair of E edges. Since only one point is voted for, the computational complexity of this method is $E^2 * 1 = \mathbf{E^2}$. Note: If there was no repetition of edge pairs, however, the complexity would be **E!** instead of E^2.