PS #3 Report

Problem 1: Image Improvement

Introduction

Pixel-to-area operations are very useful for improving image quality in noisy or blurry images. Using the correct sequence of filters can allow for the repair of images with multiple blemishes.

Methods

To improve the quality of the provided images, a combination of sharpening filters and median blur filters were applied. The sharpening filter used is a Laplacian filter with the following kernel:

-1	-1	-1
-1	9	-1
-1	-1	-1

This kernel emphasizes the edges while maintaining the brightness of the original image since the pixel weights sum to unity.

Results

For the circuit board image, the aforementioned sharpening filter was used to reduce the blurring of the image and restore some of the sharp edges on the PCB.

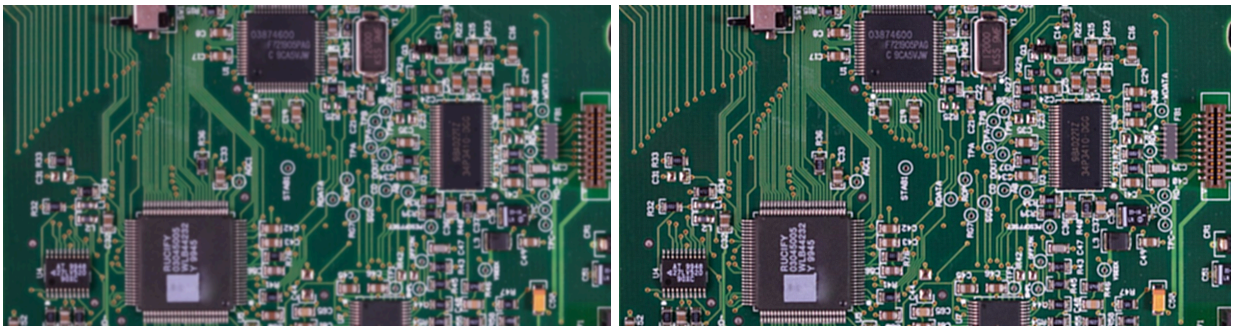


Fig 1: Original circuitboard.png (left) compared to the image sharpened by the filter in the above section (right)

The filter was implemented using the `cv.filter2D()` method and a kernel defined as follows:

```
'''
A sharpening filter reduces the image blur, but cannot fully recapture the detail of the ground truth image.
'''
circuitboard = cv.imread("circuitboard.png", cv.IMREAD_COLOR)
# Define a sharpening filter
n = -1
kernel = np.array([[n, n, n],
                   [n, 8*abs(n)+1, n],
                   [n, n, n]])
circuitboard_improved = cv.filter2D(circuitboard, -1, kernel)
cv.imshow("circuit filtered", circuitboard_improved)
cv.imwrite("circuitboard-improved.png", circuitboard_improved)
```

For the wedding image, there were lots of specks that needed to be removed. To do this, a median blur filter was applied. Median blur filters excel at removing “spike noise”, or outlier pixels in the image, such as the dark specks on a light background in the wedding photo.

Fig 2: The original wedding.png image with black specks (left) and the improved image with median blur applied (right)

Median blur was applied to the image using the `cv.medianBlur()` method. An aperture size of 3x3 was used.

```
'''
A median filter removes the black speckle noise from the image without reducing the overall fidelity of the image
'''
wedding = cv.imread("wedding.png", cv.IMREAD_COLOR)
wedding_filtered = cv.medianBlur(wedding, 3)
cv.imshow("wedding filtered", wedding_filtered)
cv.imwrite("wedding-improved.png", wedding_filtered)
```

The PCB and dog images each combine the issues of the previous 2 images - it has speckled white spots across a blurred image. To rectify the image, a combination of both filters was applied. First, a median blur filter was used to remove the speckles, and then a sharpening filter was used to unblur the resulting image. The order of these filters was important, since a

sharpening filter applied before the noise is removed would amplify the speckles as they have a high contrast with their background.

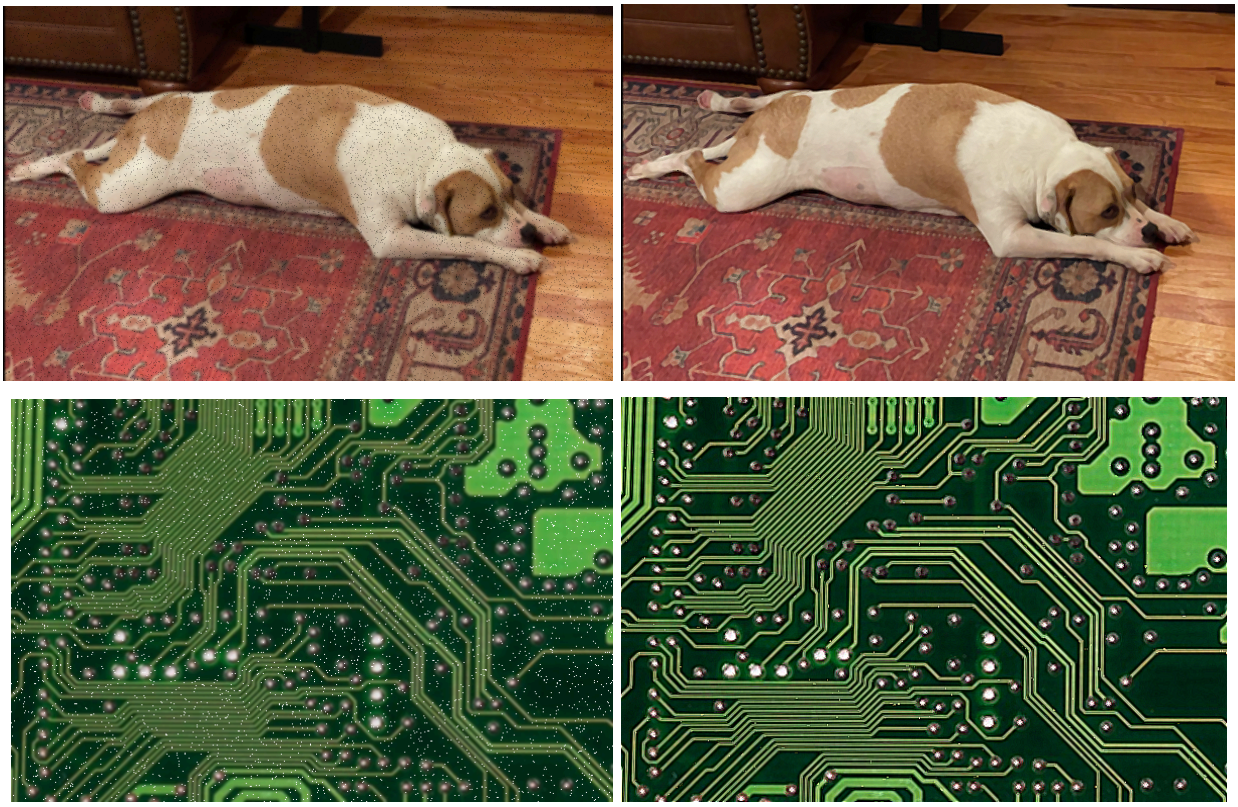


Fig 3: The original dog and PCB images (left) and the denoised + sharpened images (right)

Problem #2: Edge Detection

Introduction

Edge detection is an important step in being able to detect objects in images. There are many different methods to detect edges in images. In this assignment, a bidirectional Sobel filter and Canny edge detection are used and compared.

Methods

A unidirectional Sobel filter combines unidirectional derivative and Gaussian filters in perpendicular directions. Below shows the horizontal derivative filter and the vertical Gaussian filter combining to make the horizontal Sobel filter.

					1			-1		1
-1		1	&		2		=	-2		2
					1			-1		1

To apply a bidirectional Sobel filter, simply apply both the horizontal and vertical filters to the original image and sum the results. In code, these filters are applied using the `cv.filter2D()` method and the results are summed elementwise.

```
# Function that applies a Sobel filter to an image both horizontally and vertically
def sobel(img):
    h_kernel = (1/8) * np.array([[-1, 0, 1],
                                 [-2, 0, 2],
                                 [-1, 0, 1]])
    v_kernel = (1/8) * np.array([[1, 2, 1],
                                 [0, 0, 0],
                                 [-1, -2, -1]])
    return cv.filter2D(img, -1, v_kernel) + cv.filter2D(img, -1, h_kernel)
```

Canny edge detection is a more complex method for identifying edges in images. The basic process is as follows:

1. Reduce noise and differentiate
2. Detect max-slope-magnitude locations
3. Apply a threshold to detect edges

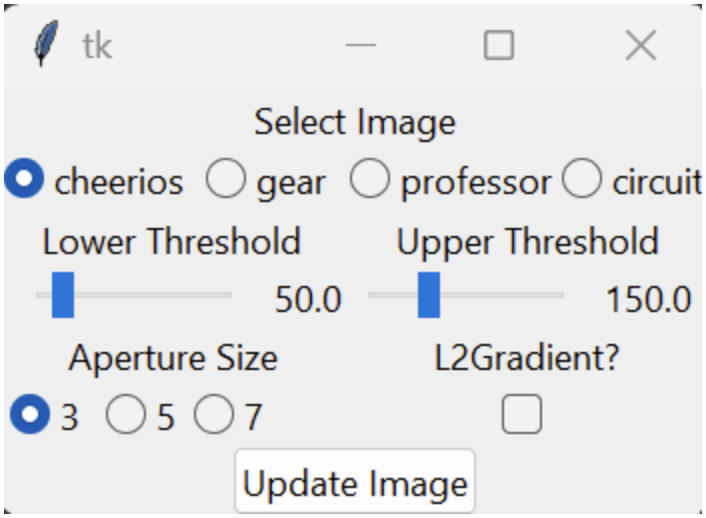
Thresholding is done using a method called “hysteresis thresholding”. In this technique, a lower threshold and an upper threshold are used. Any point with a derivative below the lower threshold is not an edge, and any point above the upper threshold is an edge. For all the points in between, they are characterized as edges only if they are adjacent to another point characterized as an edge.

Canny edge detection is implemented in this assignment using the `cv.Canny()` method. A `tkinter` GUI was developed and is used to tune the parameters of the Canny edge detection.

```
'''
Function that applies Canny edge detection to an image with the following params:
upper - Upper bound of the derivative threshold
lower - Lower bound of the derivative threshold
ap - Aperature size, can be 3, 5, or 7
l2 - Boolean indicating whether or not to use L2Gradient magnitude
'''
def canny(file, upper, lower, ap, l2):
    img = cv.imread(file, cv.IMREAD_COLOR)
    img_canny = cv.Canny(img, lower, upper, apertureSize=ap, L2gradient=l2)
    cv.imshow(file + " canny", img_canny)
    filename_canny = file.split('.')[0] + "-canny." + file.split('.')[1]
    cv.imwrite(filename_canny, img_canny)
    cv.waitKey(0)
    cv.destroyAllWindows()
```

```
# Set up a GUI to select an image and adjust parameters for Canny edge detection
root = Tk()
frm = ttk.Frame(root)
frm.grid()

# Default parameters
ap = IntVar(value = 3)
l2 = BooleanVar(value = 0)
img = StringVar(value = "cheerios.png")
upper = IntVar(value = 150)
lower = IntVar(value = 50)

# Image selection radio buttons
ttk.Label(frm, text="Select Image").grid(column=0, row=0, columnspan=8)
ttk.Radiobutton(frm, text="cheerios", variable=img, value="cheerios.png").grid(column=0, row=1, columnspan=2)
ttk.Radiobutton(frm, text="gear", variable=img, value="gear.png").grid(column=2, row=1, columnspan=2)
ttk.Radiobutton(frm, text="professor", variable=img, value="professor.png").grid(column=4, row=1, columnspan=2)
ttk.Radiobutton(frm, text="circuit", variable=img, value="circuit.png").grid(column=6, row=1, columnspan=2)

# Upper and lower threshold sliders
ttk.Label(frm, text="Lower Threshold").grid(column=0, row=2, columnspan=4)
lower_scale = ttk.Scale(frm, from_=0, to=500, orient=HORIZONTAL, variable=lower)
lower_scale.grid(column=0, row=3, columnspan=3)
lower_scale.set(lower.get())
ttk.Label(frm, textvariable=lower).grid(column=3, row=3)
ttk.Label(frm, text="Upper Threshold").grid(column=4, row=2, columnspan=4)
upper_scale = ttk.Scale(frm, from_=0, to=500, orient=HORIZONTAL, variable=upper)
upper_scale.grid(column=4, row=3, columnspan=3)
upper_scale.set(upper.get())
ttk.Label(frm, textvariable=upper).grid(column=7, row=3)

# Aperature size radio buttons
ttk.Label(frm, text="Aperture Size").grid(column=0, row=4, columnspan=4)
ttk.Radiobutton(frm, text="3", variable=ap, value=3).grid(column=0, row=5)
ttk.Radiobutton(frm, text="5", variable=ap, value=5).grid(column=1, row=5)
ttk.Radiobutton(frm, text="7", variable=ap, value=7).grid(column=2, row=5)

# L2Gradient check box
ttk.Label(frm, text="L2Gradient?").grid(column=4, row=4, columnspan=4)
ttk.Checkbutton(frm, variable = l2, onvalue=1, offvalue=0).grid(column=4, row=5, columnspan=4)

# Button to display and save result
ttk.Button(frm, text="Update Image", command=lambda: canny(img.get(), int(upper.get()), int(lower.get()), int(ap.get()), l2.get())).grid(column=0, row=6, columnspan=8)
root.mainloop()
```

The GUI contains radio buttons to select which image to use, sliders for setting the lower and upper thresholds, radio buttons for the aperture size, and a checkbox for setting the gradient magnitude calculation method.



Results

After tuning, the following Canny edge detection parameters yielded the best results for each input image.

Image	**Lower Threshold**	**Upper Threshold**	**Aperture Size**	**L2Gradient?**
cheerios.png	222	255	3	Yes
gear.png	151	395	3	Yes
professor.png	87	127	3	Yes
circuit.png	75	186	3	Yes

Table 1: Canny edge detection parameters that yielded the best results for each image.

The Canny edge detection performed slightly better for the cheerios image. The Sobel filter missed the edges of the cheerios that were facing away from the shadows (towards the light source) due to there being less contrast between the cheerios and the table than between the cheerios and the table. However, the Canny edge detector did pick up more noise than the Sobel filter.

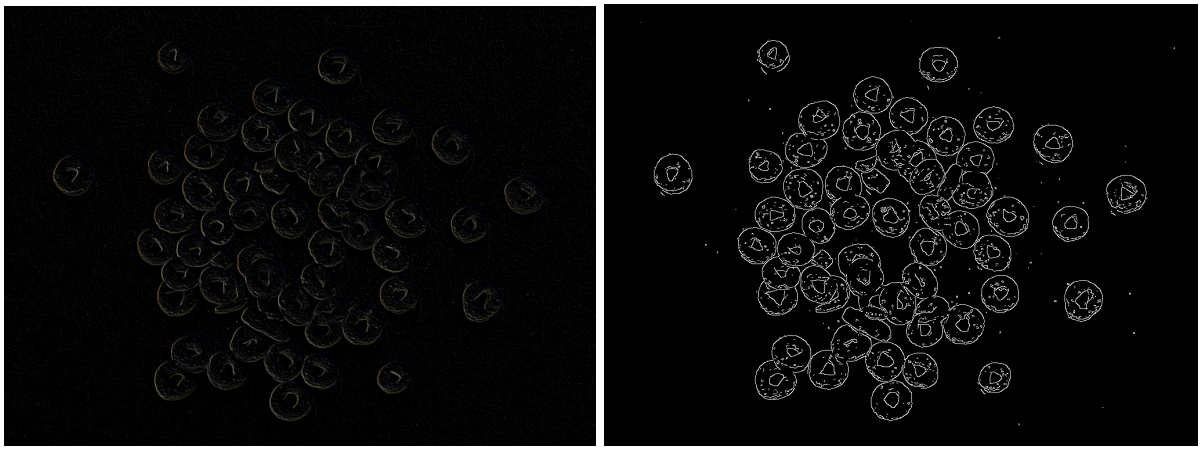


Fig 4: Sobel edge detection (left) and Canny edge detection (right) results for cheerios.png

For the image of the gear, Canny edge detection performed significantly better than the Sobel filter. The Sobel filter, similar to the cheerios image, missed some of the edges leading to an incomplete outline of the main feature of the image. The Canny edge detector easily detected the main features of the shape of the gear.

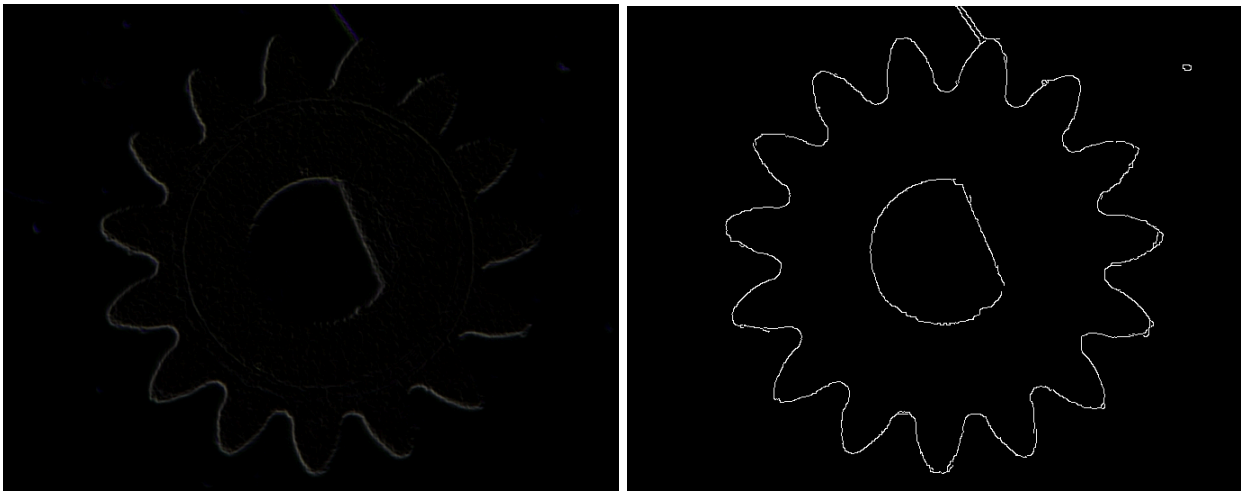


Fig 5: Sobel edge detection (left) and Canny edge detection (right) results for gear.png

The professor image was by far the most difficult to tune parameters for. The stitching on the jacket caused a lot of noise and was often detected before other features that would be considered more important/more distinct. For this image, it is very important to decide in advance what features are important and what edges are important to detect. For example, if the application doesn't require the trusses in the bridge to be fully outlined, different parameters can be selected to avoid noise in the jacket. To address noise in the jacket if smaller details in other parts of the image are important, a smoothing filter could be applied before edge detection is done.

Fig 6: Sobel edge detection (left) and Canny edge detection (right) results for professor.png. The Sobel filtered image has been recolored for improved contrast.

For the circuit image, the Sobel edge detection was actually fairly effective, but did fail to pick up on a few of the more subtle edges, such as the lighter green wires. The Canny edge detection was able to detect some of these wires, but still missed some of them, such as part of the wiring between the ports in the “555” area. It was difficult to find parameters that would detect these edges without detecting noise in the solder pads themselves from lighting/reflection. It’s possible that some mild blurring applied before edge detection would improve performance.

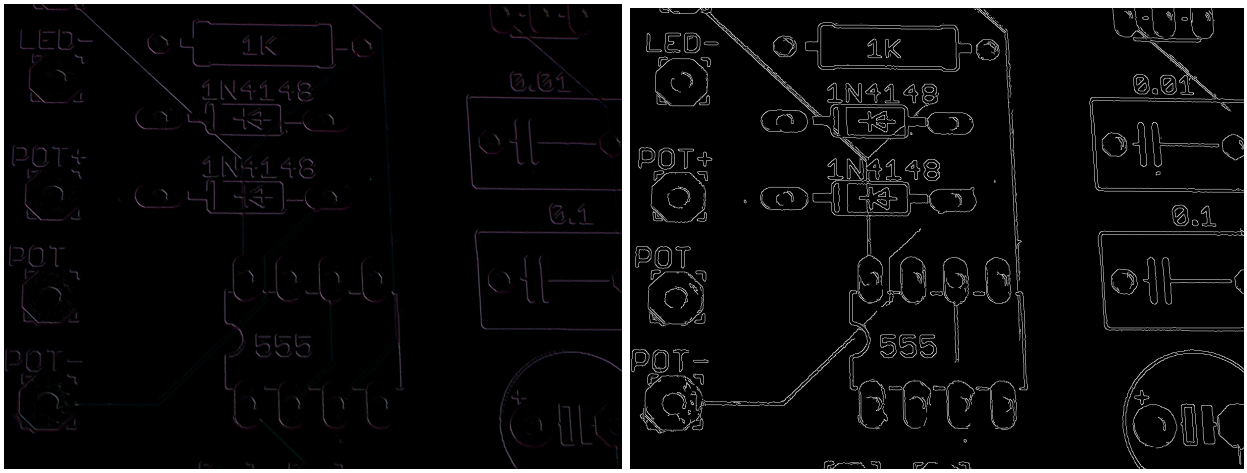


Fig 7: Sobel edge detection (left) and Canny edge detection (right) results for circuit.png