Image Segmentation on Impacted Composite Materials, Undergraduate Research

Samuel Lee
Student
University of Texas at Arlington, Arlington, Texas 76109

Dr. Paul Davidson
Supervisor
University of Texas at Arlington, Arlington, Texas 76109

The purpose of this research was to learn the python programming language, and learn different image segmentation techniques in order to perform image segmentation on images of damaged composite materials. Composite materials suffer BVID (barely visible impact damage), and field technicians must know where damage may lie and the depth at which the damage has occurred. Rather than create a new method of image segmentation entirely, the goal of this research was to learn of pre-existing image segmentation methods and applying these methods to solve the problem statement.

Table of Contents

Problem Statement……1
Introduction……...1
Simple Thresholding Program……..……2
Second Image……....…5
Region-Growing……....…6
Watershed Method…….8
Canny Edge Detection…….10
Conclusion……...12
Bibliography……....14

Problem Statement

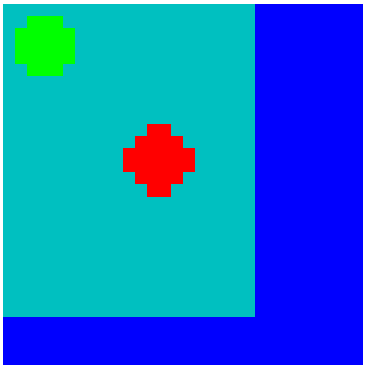
When an aircraft takes off, is in flight, or lands, damage occurs for a myriad of reasons. A stray rock might strike a vulnerable area of the aircraft, or small birds may hit the aircraft. In any case, what usually occurs is delamination of composite materials, where materials fracture due to the damage occurred. Not all of this damage is visible, as most damage is BVID (barely visible impact damage) that requires sophisticated scanning equipment to detect, and field technicians need to know exactly where this damage is. The purpose of this research is to find various ways to detect the damage, segment out areas of damage for field technicians to assess, and to do so as simply and quickly as possible. Time is a factor, as field technicians must use the fastest feasible equipment to carry out repairs, and simplicity of the user interface is desired to limit training required of field technicians. In this paper, c-scan images refers to images obtained through ultrasound imaging.

Introduction

Knowledge about the Python programming language was needed. October 2021 to December 2021 was used to learn about the Python programming language, as previous experience only included the JAVA programming language and the MATLAB programming language. Then, research into different image segmentation techniques began, with the first successful program created being a simple thresholding program. Spyder, a development environment licensed by the Massachusetts Institute of Technology, was used to code.

Simple Thresholding Program

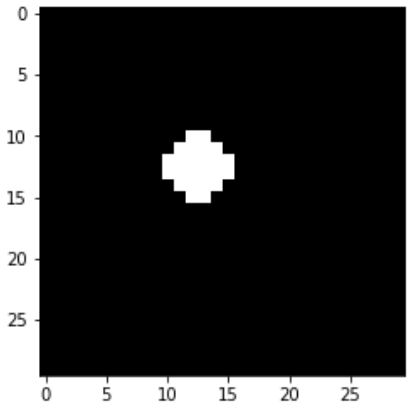
The first step was to create a program that created a simple image to test thresholding methods with. In a OneDrive folder shared with Dr. Davidson, there is a program titled "Create_Introductory_Image.py" which creates the image "task_1.png" (both the program and the image are in the folder). Running the program "Create_Introductory_Image.py" creates the image "task_1.png" which is a small, 30 pixel by 30 pixel image.



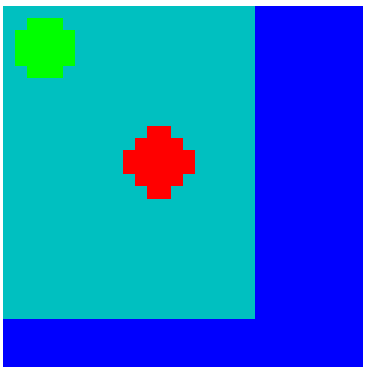
Creating a similar image is quite simple. This website allowed for an image to be created within 10-20 minutes (see footnote).[1]

Now that an image had been created with python, it was time to create a program to segment the image into the different shapes you see within the image. Now, this is why Dr. Davidson recommended starting with a small 30x30 image, as more difficulties occur with larger images.

In that OneDrive folder, open "First_Segmentation_Program.py" and run it. Assuming Spyder is downloaded properly and you have the proper packages downloaded, here is what you should see when the program runs:



Compare that with our original image.



From here on, "First_Segmentation_Program.py" will be referred to as First Program. You can see that the red ellipsoid shape is white, and everything else is black. This is the basic premise of Thresholding in image segmentation, where we select a range of pixel values to be white and

[1] Draw circle, rectangle, line, etc. with Python, Pillow. (n.d.). Retrieved January 13, 2021, from https://note.nkmk.me/en/python-pillow-imagedraw/

everything else in the image to be black. Now look at line 26 of the code, or run the program and look at the unique value array. You should see these numbers:

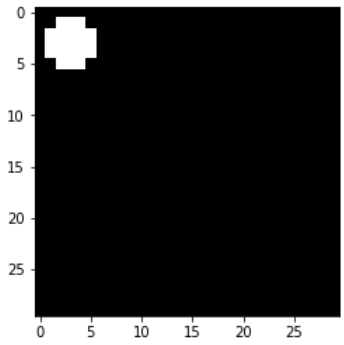
0.0721 0.59294116 0.2125 0.7154

and, look at line 32 of the code.

```
32          if (gray[i,j] <= 0.21 or gray[i,j] >= 0.22):
```

What we are doing is looping through the image, and setting everything not between 0.21 and 0.22 to black. Why these numbers specifically? Well looking at the numbers showed earlier, you'll notice the value 0.2125. This value corresponds to the red value used in the "task_1.png" image.

So, thresholding relies on a threshold value, in our case this is 0.2125. We filter anything that isn't this value to black. If you try the other numbers out you will get this for 0.0721:



So, in summary, those four values were the pixel values of our shapes. In order to get these four values, we need to find the unique values in the image. In lines 14-24, you will notice something quite different. Basically, this block of code was the first attempt to find the unique values and didn't work out so well. The premise was to use a histogram graph and find the numbers corresponding to the peaks, as the peaks should have given us the unique values of the image. Unfortunately, a feasible way to read the peak values was not found even after weeks of research. Rather than using a histogram, it is better to use np.unique as you see in line 44-46.

```
x, bin_edges = np.histogram(gray, bins=256, range=(0, 1))

plt.figure()
plt.title("Pixels per Grayscale Value")
plt.xlabel("grayscale value")
plt.ylabel("pixels")
plt.xlim([0.0, 1.0])
plt.plot(bin_edges[0:-1], x)
plt.show()

originalImage = cv2.imread('task_1.png')
```

This prints the histogram. Problem is finding the peaks of the histogram.

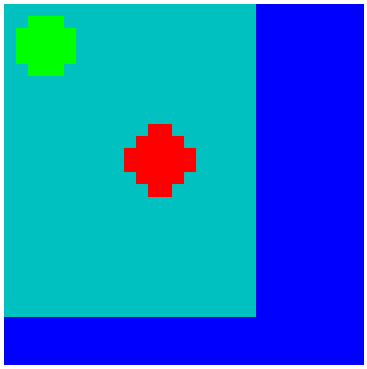
```
unique_array = np.unique(array)
print("\n These are your unique values: ")
print(unique_array)
```

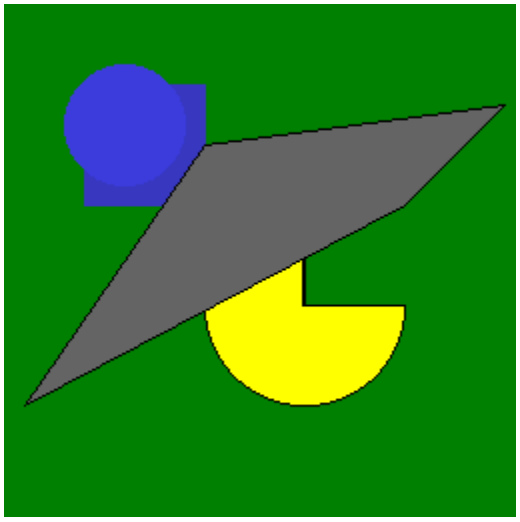
This prints the unique values of the array.

This concludes the first segmentation program created. Yet, for the purpose of the problem statement, this program was nowhere near enough. This program does not address the issue of gradients, the depth at which the image can be found, or a number of other issues common to the image segmentation field. Which brings us to the next program.

Second Image

First, open “Create_Second_Image.py” and “task_2.png” which creates a new image similar to the program discussed previously.

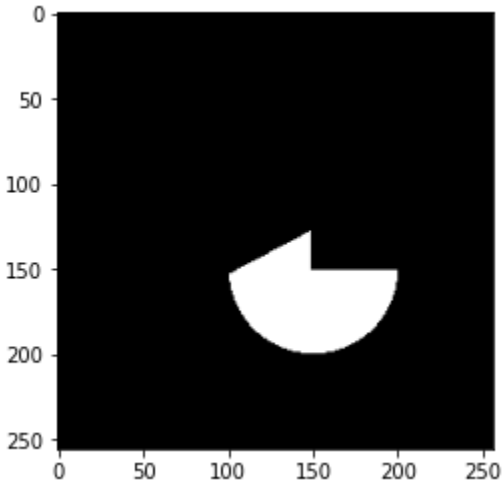




You can see that the second image is noticeably larger, with more complex shapes. More complex shapes create issues within the program, issues that do not arise with simpler shapes. Yet complex shapes are present within our problem statement, and addressing these issues brings

us closer to the problem statement. Thus, coding with these complex shapes brings us closer to the problem statement.

Our simple thresholding program still works even with this more complex image. “Second Image Segmentation - Threshold.py” is the program in the OneDrive folder that runs our simple thresholding on this new image.

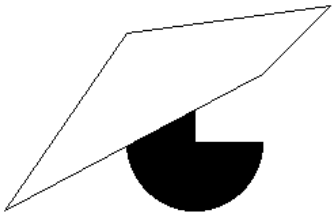


Above is the result when the program is run. Now, it was time to try out different image segmentation methods and see which ones suited our needs. The end product is to detect damaged areas of composite materials, and thresholding alone will not suffice.

Two that were tried were watershed, and region-growing.

Region-Growing

Open “region-growing method.py” and run the program. This should be your output.



```
17      def selectConnects(p):
18          if p != 0:
19              connects = [Point(-1, -1), Point(0, -1), Point(1, -1), Point(1, 0), Point
20                          Point(0, 1), Point(-1, 1), Point(-1, 0)]
21          else:
22              connects = [ Point(0, -1),  Point(1, 0),Point(0, 1), Point(-1, 0)]
23          return connects
```

Lines 17-23 are responsible for obtaining the pixels around the center pixel, comparing these surrounding pixels to the center pixel and determining if these pixels fit the region.

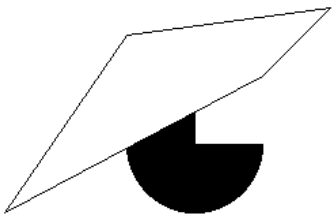
```
50      seeds = [Point(10,10),Point(82,150),Point(20,200)]
```

Line 50 is how we create our seeds, and in a region-growing method, the program determines the area around the initial seeds and groups pixels together based on the algorithm used. In this case pixels were chosen randomly, but for the problem statement, seeds should be chosen with more care.

```
    seedMark[currentPoint.x,currentPoint.y] = label
    for i in range(8):
        tmpX = currentPoint.x + connects[i].x
        tmpY = currentPoint.y + connects[i].y
        if tmpX < 0 or tmpY < 0 or tmpX >= height or tmpY >= weight:
            continue
        grayDiff = getGrayDiff(img,currentPoint,Point(tmpX,tmpY))
        if grayDiff < thresh and seedMark[tmpX,tmpY] == 0:
            seedMark[tmpX,tmpY] = label
            seedList.append(Point(tmpX,tmpY))
return seedMark
```

Line 36-46 is the meat of the algorithm, comparing the pixel values and seeing whether the pixels are part of the region or not.

So, here is a brief summary of region-growing image segmentation. You have initial starting points called seeds. You then compare the area around these seeds to the seed itself and determine whether or not these pixels belong to the same region or not. This area grows and grows until it hits another area. So in the picture below, one of the seeds was inside the ‘pie’ shape and the area grew to encompass all the pixels within the ‘pie’ shape.



Now, we will explore the watershed method.

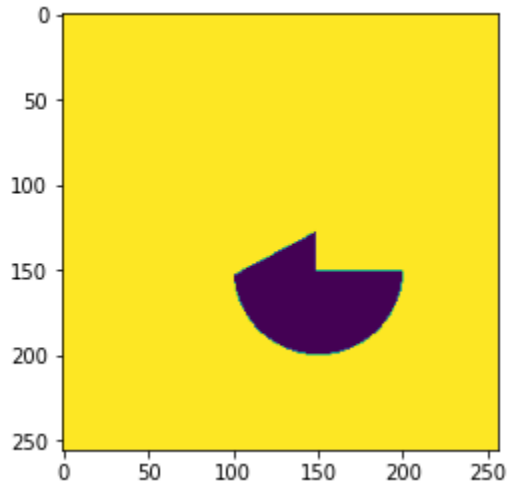
Watershed Method

Watershed is the most difficult of the methods explored, and as such, requires a lengthy explanation. Even after a considerable amount of research, a final watershed algorithm was not able to be coded. “watershed method.py” uses the CV2 watershed algorithm. Please familiarize yourself with the following:

- cv.THRESH_BINARY
- cv.THRESH_BINARY_INV
- cv.THRESH_TRUNC
- cv.THRESH_TOZERO
- cv.THRESH_TOZERO_INV

the OpenCV documentation is a great place to start learning about these algorithms (see footnote).[2]

Open “watershed method.py” and run it. This should be your result.

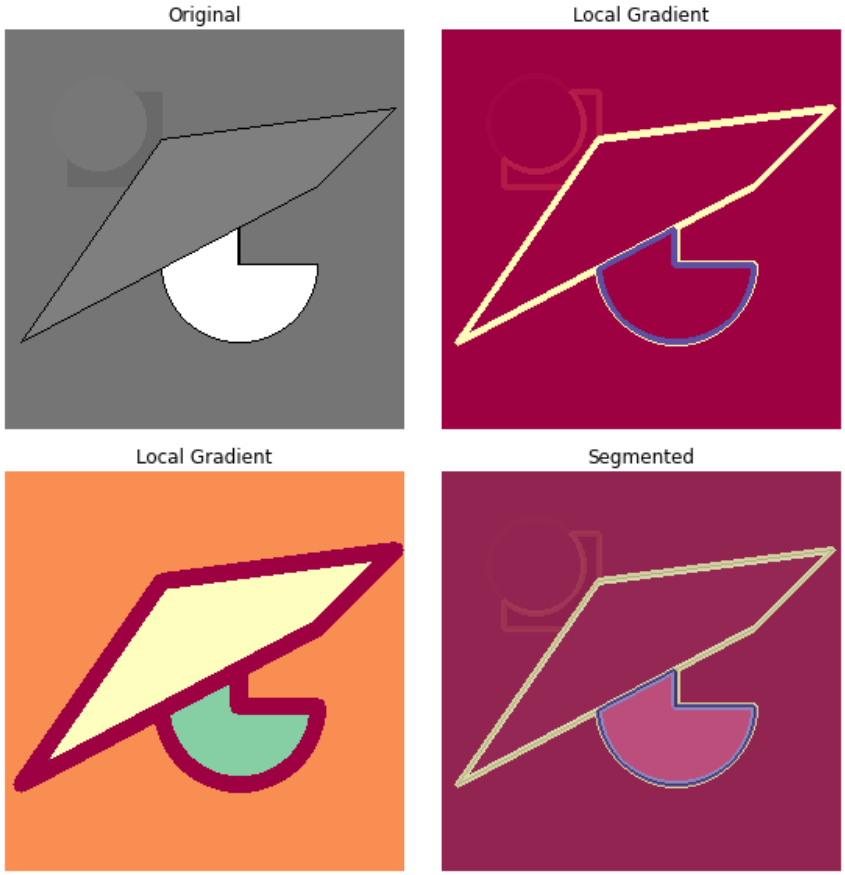


This is what happens when we use plt.imshow(thresh), which is cv2.morphologyEx() command.

[2] Image Thresholding. (n.d.). Retrieved March 15, 2021, from https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html

Clearly, the watershed algorithm was not performing the task it was supposed to. After some trial and error, another algorithm was found, one that was simpler and more intuitive to understand.

Please open up “watershed attempt 2.py” and if “task_2.png” is still in your downloads folder, this should be the image that is created. This program is not student code, it was provided by this youtube video (see footnote).[3]



We can see the semicircle, quadrilateral, circle, and square segmented from each other. The author heavily recommends that future students utilize this youtube video, as the markers are more accurate due to line 18 and 19, which allows us to choose what gradient we wish to have for our watershed algorithm.

```
18    markers = rank.gradient(image, disk(5)) < 20
19    markers = ndi.label(markers)[0]
```

[3] (2017, October 05). Retrieved May 13, 2021, from https://www.youtube.com/watch?v=FDMcroBeNCg

Unfortunately, that covered all knowledge of the watershed algorithm, as the research concluded before a working watershed algorithm was able to be coded. One important thing to note is that watershed-segmented images may be the key to recognizing at what depth damage lies, as the depth of the damage is determined by the color of the shape.

Canny Edge Detection

Edge Detection is simply an algorithm that detects the edges of an image. Canny Edge Detection allows us to perform more accurate edge detection. Here are the steps needed to perform Canny Edge Detection:

1. The image is filtered with the Gaussian filter.

A Gaussian filter uses a Gaussian matrix as its kernel in order to reduce noise.

2. Find the intensity gradient of an image.

Real-world shapes and images are not one static color. A piece of metal, for example, becomes rusted and tarnishes, such that there are very different colors all in one place, and the metal is not one solid, gray color, but rather a gradient of gray, brown rust, black tarnish, and other such imperfections. The intensity gradient is the change in the gradient of an image.

3. Non-maximum suppression.

In an image, some lines may be wider than others. Non-maximum suppression allows us to ensure that all lines are relatively equal, with thicker lines appearing thicker and thinner lines appearing thinner, with care placed to ensure neighboring pixels do not interfere with line generation.

4. Utilize double threshold.

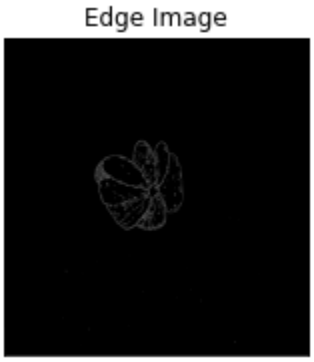
The final step I learned about was double threshold. We have a high and a low threshold, and anything outside of these thresholds do not make it into the image. This helps us deal with outlier pixels, and ensures that the edge detection is more accurate to the image. Unfortunately, research concluded at this step, and I was not able to research how we determine these high and low thresholds.

5. Edge Tracking by Hysteresis.

Hysteresis thresholding is when areas below the low threshold are considered above the low threshold because these areas connect to an area above the low threshold.

Opening Petal_Practice.jpg" shows us the following image:

This image was created with sharpies, and is not an image obtained from the internet. Now, cv2 has a program called cv2.Canny which performs canny edge detection. Open "Working Petal Practice Detection.py":

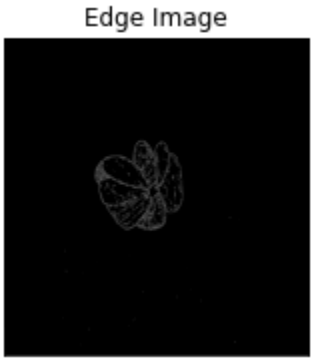



```
edges = cv2.Canny(img,50,100, L2gradient = True)
```

The first parameter in cv2.Canny() is our image, then the low gradient, then the high gradient, and then the L2gradient. L2gradient finds the magnitude of the gradient.

Here is what happens when L2gradient is turned off.

```
6     edges = cv2.Canny(img,50,100)#, L2gradient = True)
```



It is clear from the picture that l2gradient is effective at reducing the noise in the image. Note that even with l2gradient on, the image is not segmented perfectly into different shapes. For this, a more accurate low and high threshold is needed.

Conclusion

Canny edge detection and the watershed algorithm show the most promise for segmenting out the c-scan images of damaged composite materials. This is because Canny Edge Detection can accurately find the shape of the image, and then determine shapes in the image. These shapes correspond to damaged areas within the image.

After finding damaged areas, we can use the watershed algorithm to find the depth of the damaged areas, using a color reference chart. The segmented watershed allows us to see which shapes correspond to which colors, and colors in a c-scan correspond to depth. Therefore, both canny edge detection and watershed are needed to completely segment an image of damaged composite materials.

The next step for future research would be to complete the process of finding thresholds for canny edge detection, and to code a watershed algorithm that can read the results of canny edge detection.

Region-growing was not chosen as a final solution as region-growing requires seeds as initial guesses for the damaged areas. The problem statement was to allow as easy of an user interface

as possible, and if the field technician manually has to guess seeds for the region-growing algorithm, more training is required of the field technician.

Simple Thresholding was not chosen as a final solution, as thresholding requires a value. Rather, thresholding is best used in canny edge detection to reduce noise. Simple Thresholding is best used for simple objects, where the color of an image is static with no image gradient.

Unfortunately, some research had to be left out. If you go to this website, (https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html), you will see that the opencv readthedocs.io has been abandoned. This website was not cited in the bibliography as the information contained could no longer be reached. Furthermore, around 8-10 programs of failed attempts at image segmentation were not included, as these failed programs do not contribute to an understanding of the research conducted.

This concludes the research documentation, which summarizes research conducted from October 2020 to May 2021.

Bibliography

(2017, October 05). Retrieved May 13, 2021, from https://www.youtube.com/watch?v=FDMcroBeNCg
Better watershed algorithm than one previously used.

Draw circle, rectangle, line, etc. with Python, Pillow. (n.d.). Retrieved January 13, 2021, from https://note.nkmk.me/en/python-pillow-imagedraw/
Teaches how to draw images in python.

Canny Edge Detection. (n.d.). Retrieved April 15, 2021, from https://docs.opencv.org/3.4/d7/de1/tutorial_js_canny.html
Canny Edge Detection parameters.

Fan, Y., Beare, R., Matthews, H., Schneider, P., Kilpatrick, N., Clement, J., . . . Adamson, C. (n.d.). Marker-based watershed transform method for fully automatic mandibular segmentation from CBCT images. Retrieved March 27, 2021, from https://pubmed.ncbi.nlm.nih.gov/30379569/#:~:text=Methods:: The marker-based,rest of the CBCT image.
Defines marker-based watershed algorithm.

Image Module. (n.d.). Retrieved April 15, 2021, from https://pillow.readthedocs.io/en/stable/reference/Image.html
Information about PIL modules.

Image Segmentation with Distance Transform and Watershed Algorithm. (n.d.). Retrieved March 18, 2021, from https://docs.opencv.org/3.4/d2/dbd/tutorial_distance_transform.html
Distance Transform tutorial.

Image Thresholding. (n.d.). Retrieved March 15, 2021, from https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html
Information about CV watershed.

Kaur, D., & Kaur, Y. (2014). Various Image Segmentation Techniques: A Review. IJCSMC, 3(5), 809-814. Retrieved March 10, 2021, from https://ijcsmc.com/docs/papers/May2014/V3I5201499a84.pdf
Information on Threshold, Region based, edge based, clustering based, watershed, partial differential equations.

Medina-Carnicer, R., Muñoz-Salinas, R., Carmona-Poyato, A., & Madrid-Cuevas, F. J. (2010). Determining Hysteresis Thresholds for Edge Detection by Combining the Advantages and Disadvantages of Thresholding Methods. *IEEE Transactions on Image Processing*. Retrieved April 15, 2021, from https://www.researchgate.net/publication/224595829_Determining_Hysteresis_Thresholds_for_Edge_Detection_by_Combining_the_Advantages_and_Disadvantages_of_Thresholding_Methods
Learned about Hysteresis Thresholds.

Numpy.ones. (n.d.). Retrieved March 27, 2021, from https://numpy.org/doc/stable/reference/generated/numpy.ones.html
Used to learn about numpy.ones even with previous limited understanding.

Python implements region growing algorithm (regionGrow). (n.d.). Retrieved March 17, 2021, from https://www.programmersought.com/article/81151779785/
Great introduction to region-growing method.

Tay, Jian. (2020, October 11). Retrieved March 10, 2021, from https://www.youtube.com/watch?v=oxWfLTQoC5A
Teaches distance transform used in watershed algorithm.

Rai, A. (2020, December 23). Python: Cv2 Canny() Method. Retrieved April 15, 2021, from https://java2blog.com/cv2-canny-python/
Canny Edge Detection discussion.