

Watershed Algorithm

Ethan Triplett

March 1, 2020

1 Introduction

One approach to identifying items within an image is to segment the image based on the brightness of the image pixels. The watershed algorithm takes advantage of that approach by determining which pixels are the darkest relative to their surroundings and creating segmentations that are formed outwards from those points.

This report covers the implementation of the watershed algorithm in Python. Code for this project was written in Microsoft Visual Studio with Python 3.7. Libraries used to support the algorithm include cv2, random, imageio, numpy, and skimage. The goal of the project is to create colored segments that align with shapes in a source image. For simple images without noise the outcome is expected to be very accurate, with each shape fully identified by its segment. For noisy images the challenge will be limiting the number of segments enough to avoid splitting objects into multiple segments. Variations in pre-processing including Gaussian blur, morphological opening and closing, and distance transforms will attempt to remove noise before the watershed algorithm is applied to prevent over-segmentation.

2 Background

2.1 Watershed Concepts and Vocabulary

Before the algorithm can be implemented, it is important to understand the core concepts of the watershed process. An apt metaphor is the way water flows from the top of a mountain. Rain on a mountain starts over a wide area but funnels into streams as it descends. The streams flow downwards until they reach the lowest point in the area, which forms a lake.

In an image it is possible to follow the pixel intensities towards darker or brighter intensities by comparing with neighboring pixels. If one pixel is brighter than all of its neighbors, then it is locally maximum and its neighbors are “downhill” from it [1]. The lowest intensity neighbor is said to be “downstream” from the current pixel. When the minima representing the lowest intensity pixels is found, it is referred to as a “drain”, and every pixel that flows towards that drain is part of a “basin”. The pixels on the border between basins are referred to as “watershed” pixels, which is where the watershed algorithm gets its name.

2.2 Finding Minima

One of the foundational steps to the watershed algorithm is finding the local minima within the image. The process of finding these minima includes two main comparisons, seen in figure 1 [1]:

```
for (p = 0; p < NumberOfPixels)
    if (L(p) != NOTAMIN)
    {
        for q in the immediate neighborhood of p
        {
            if (f(p) > f(q) ) L(p) <-- NOTAMIN
            if (f(p) == f(q) && L(q) == NOTAMIN ) L(p) <-- NOTAMIN
        }
    }
```

Figure 1 - Finding Minima

The first is a check to see if the current pixel is brighter than any of its neighbors. Minima must be the darkest pixels in their immediate neighborhood, so pixels with darker neighbors can be removed from consideration immediately. The next consideration is whether the current pixel is equal in brightness to all of its neighbors, but one of its neighbors has already been determined to not be minimal. This case describes a plateau, where a collection of pixels all have the same brightness level. A plateau creates a challenge for the watershed algorithm because there are no pixels immediately “uphill” from the interior points. This is where the distance transform is useful.

2.3 The Distance Transform

The distance transform creates an image where each pixel value represents the distance of that pixel from a specific object in the image. Distances can be found relative to an edge when edge detection is applied, or in the case of the watershed algorithm the distance to a local minima can be found as well. To find Euclidean distances, the target image should have the minima identified with a number value and all other pixels set to infinity. A mask can then be applied to the image forwards and backwards, summing with a neighborhood of pixels and keeping the minimum for each pixel location until all pixels have a distance value. A before and after example of this process can be seen in figure 2:

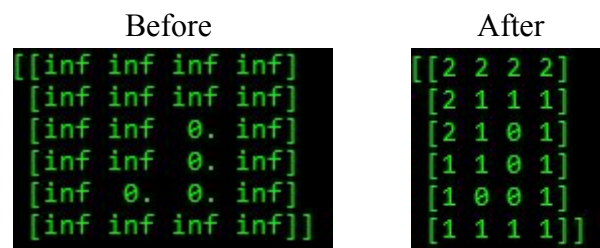


Figure 2 - Distance Transform

Finding these distances can break up plateaus in an image by replacing identical brightness pixels by the ascending distance values formed by moving further from the nearest brightness minimum. The watershed algorithm can immediately use these ascending values to follow uphill pixels.

3 The Watershed Algorithm

3.1 Algorithm Process

There are six main steps to segment an image using the watershed algorithm [1]:

- 1) Find all drains in the image, use them to create a label image in which every drain has a label unique from the other drains.
- 2) Create a set V that includes every pixel not interior to a drain.
- 3) Find the pixel in V with the lowest intensity. Transfer the corresponding label from the nearest drain in the label image to the current pixel from V.
- 4) Remove the pixel from the set V.
- 5) For every pixel adjacent to the current pixel,
 - a) If the neighbor pixel is uphill and unlabeled, give it the same label as the current pixel.
 - b) If the neighbor pixel is uphill but already has a label, change the label to indicate that it is a watershed pixel.
- 6) Repeat 3-5 until no pixels remain in V.

In the next section the code for each of these steps will be described in more detail.

3.2 Code Implementation

The watershed process begins by finding all of the drains within the image. A label image is created in which every pixel has a value of 255:

```
print("Finding Minima...")
mins = imageIn.astype(float)
minima = imageIn.astype(float)
mins = (mins * 0) + 255
minima = (minima * 0) + 255
```

Figure 3 - Creating Blank Label Image

Pixels in the original image are then compared with their neighbors. The pixels that could not be local minima are set to 0 to indicate that they could not be part of a drain:

```
for y in range(len(mins)-1):
    for x in range(len(mins[0])-1):
        if (imageIn[y-1][x-1] < imageIn[y][x] or
            imageIn[y-1][x] < imageIn[y][x] or
            imageIn[y-1][x+1] < imageIn[y][x] or
            imageIn[y][x-1] < imageIn[y][x] or
            imageIn[y][x+1] < imageIn[y][x] or
            imageIn[y+1][x-1] < imageIn[y][x] or
            imageIn[y+1][x] < imageIn[y][x] or
            imageIn[y+1][x+1] < imageIn[y][x]):
            mins[y][x] = np.inf
            minima[y][x] = 0

        if ((imageIn[y-1][x-1] == imageIn[y][x] and mins[y-1][x-1] == 0) or
            (imageIn[y-1][x] == imageIn[y][x] and mins[y-1][x] == 0) or
            (imageIn[y-1][x+1] == imageIn[y][x] and mins[y-1][x+1] == 0) or
            (imageIn[y][x-1] == imageIn[y][x] and mins[y][x-1] == 0) or
            (imageIn[y][x+1] == imageIn[y][x] and mins[y][x+1] == 0) or
            (imageIn[y+1][x-1] == imageIn[y][x] and mins[y+1][x-1] == 0) or
            (imageIn[y+1][x] == imageIn[y][x] and mins[y+1][x] == 0) or
            (imageIn[y+1][x+1] == imageIn[y][x]) and mins[y+1][x+1] == 0):
            mins[y][x] = np.inf
            minima[y][x] = 0
```

Figure 4 - Finding Minima

The resulting image forms a two dimensional array in which every index is either 0 or 255, where indices of 255 indicate drains. The above code does create a separate image in which the zeros are replaced by infinity. That image is used when the distance transform is to be implemented, which will be discussed later.

Once the drains have been identified, they must be assigned unique labels. A typical implementation of this process would iterate through the drains assigning labels to each pixel, with the same label for every pixel in the drain. While iterating through the drains row by row there are cases in which the shape of the drain causes more than one label to have been used by mistake. In these cases the lower value label is kept and the incorrectly labeled pixels must be iterated through to relabel them to the correct value. Because the code for this watershed implementation already incurred significant overhead, an efficient labeling library was used to reduce operating time:

```
def assignLabels(minima):
    print("Labeling...")
    unique = label(minima)
    return unique
```

Figure 5 - Labeling Minima

Now that each drain had a unique label, it was time to perform the main phase of the watershed algorithm in which the drains were expanded to fill each basin in the image. The collection of pixels to add labels to was assembled by iterating through the original image or the distance transform image and storing the pixel intensity and coordinates in an array. The array was then sorted by intensity to allow them to be processed in ascending order.

```
upstream = []

for y in range(len(drains)-1):
    for x in range(len(drains[0])-1):
        upstream.append([imageIn[y][x],y,x])

upstream = sorted(upstream,key=lambda x: x[0])
```

Figure 6 - Creating Upstream Pixel Array

Each pixel in the list then compared brightness and label values with each neighbor pixel. This process involves taking the lowest index pixel from the upstream list and pulling the coordinates that were stored so that the neighboring pixels can be examined.

```
while len(upstream) != 0:
    x = upstream[0][2] # Coordinates of the lowest brightness pixel
    y = upstream[0][1]
```

Figure 7 - The Loop For Filling Basins

Pixels that were already labeled but did not share the same value as the current pixel were given a value of -1 so that they could later be colored as a watershed pixel. Figure 7 shows a comparison with one of the pixel neighbors to determine if it is a watershed pixel.

```
# If neighbor pixel is already labeled differently, it is a watershed pixel:
if (drains[y-1][x-1] != 0) and (drains[y-1][x-1] != drains[y][x]):
    drains[y-1][x-1] = -1
```

Figure 8 - Checking For Watershed Pixel

If the neighboring pixels do not already have a label, then their brightness is checked to ensure they were upstream from the current pixel. Figure 8 shows the label of the current pixel being assigned to one of the adjacent upstream pixels:

```
elif (imageIn[y-1][x-1] > imageIn[y][x]) and (drains[y-1][x-1] != -1):  
    drains[y-1][x-1] = drains[y][x]
```

Figure 9 - Assigning Current Label To Upstream Neighbor

After all of the neighbors for a given pixel have labels the pixel is removed from the list of pixels that still require processing. When the entire list has been processed the labels are then converted into colored pixels. Each basin needs to be a unique color, so a random number generator is used to create a unique red green blue (RGB) triple.

```
def randColor(i):  
    return [randy.randint(1,254), randy.randint(1,254), randy.randint(1,254)]
```

Figure 10 - Generating A Random Colored Pixel

A list of colors is created that is long enough to provide a unique color for every drain as well as the watershed pixels. The highest drain label value is used to initialize a list of three value arrays. Index 0 of the list is assigned the value for a black pixel and is used to mark the watershed pixels. Index 1 is for white pixels, and is used to mark any pixels that were not successfully assigned labels during the watershed process.

```
colors = np.zeros((max(map(max, unique)) + 2, 3)) # Make a color list  
colors = list(map(randColor, colors))  
colors[0] = [0,0,0]  
colors[1] = [255,255,255]
```

Figure 11 - Creating The Color List

Once the list of colors has been created it is simple to color each basin in the image. Each pixel belonging to a given basin will have the same label value, so each pixel is colored with the color at the index of its label. The map function combined with lambda functions reduces the overhead created by using for loops for this simple exchange [2].

```
drains = list(map(lambda x: list(map(lambda y: colors[y+1], x)), unique))
```

Figure 12 - Coloring Pixels By Label Value

With every basin colored the core functionality of the watershed algorithm is complete. The effects of pre-processing the image will be discussed as part of the results and in depth analysis sections.

3.3 Results

Before and after results for three main images are presented here. Experimentation to attempt to improve one of the results will be provided in the in depth analysis section.



Image 1 - Before

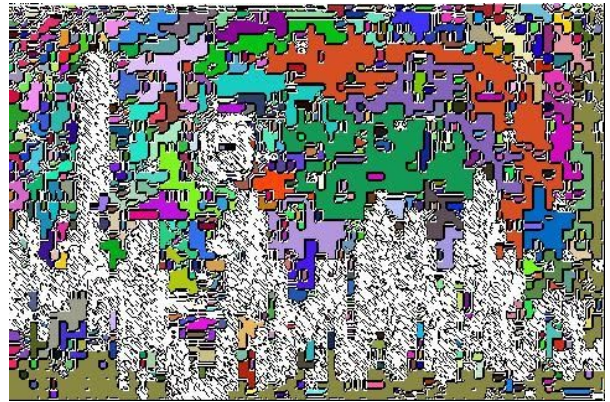


Image 1 - After



Image 2 - Before

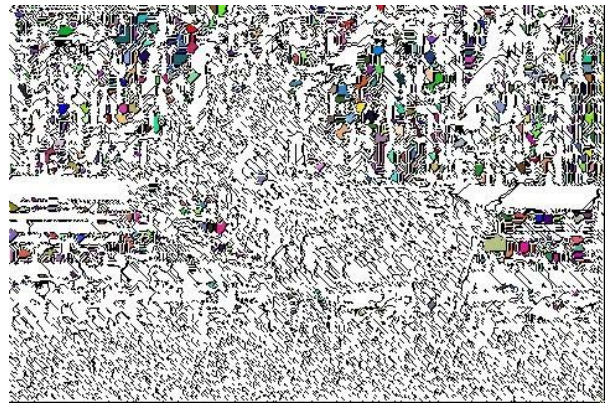


Image 2 - After

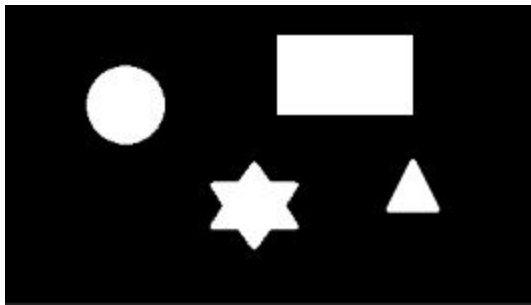


Image 3 - Before

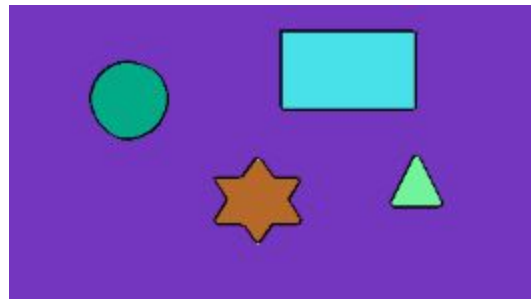


Image 3 - After

The three images had vastly different results due to differences in their content. Image 3 is a noise-free two color image created in Microsoft Paint to show how the algorithm works under ideal conditions. Each shape is colored with a well defined border, though the shape of the circle became slightly lopsided due to the border.

Image 1 and 2 are more realistic images featuring varied brightness levels and noise. Without any pre-processing these images produce over-segmented results. Thousands of drains are identified and expanded to fill basins. The algorithm had difficulty interpreting the rapid variations in brightness in the trees for image 1, and image 2 had difficulty defining any shape at all with most of the result image being marked as watershed pixels. Image 2 had 4710 drains in that result. Since image 1 has a vague outlining of the moon and the trees already, the in depth analysis section will attempt to clean up its result to have fewer segments in the sky and more clearly define the moon.

3.4 In Depth Analysis

In this section the effects of the distance transform, opening, closing, and Gaussian blur will be explored. The original image resulted in an over segmented watershed image, so reducing the number of minima in the image is likely to give a better final result. Figure 13 shows the 3,924 minima found in the original image. Many of these minima are dots found within the trees and moon, and the rest appear as continuous lines meandering around the image.

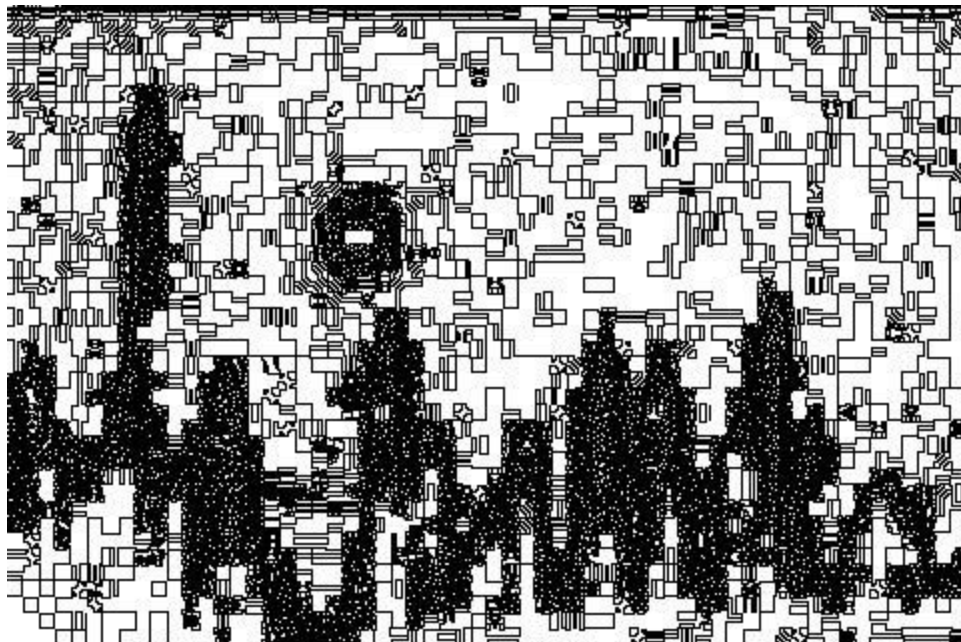
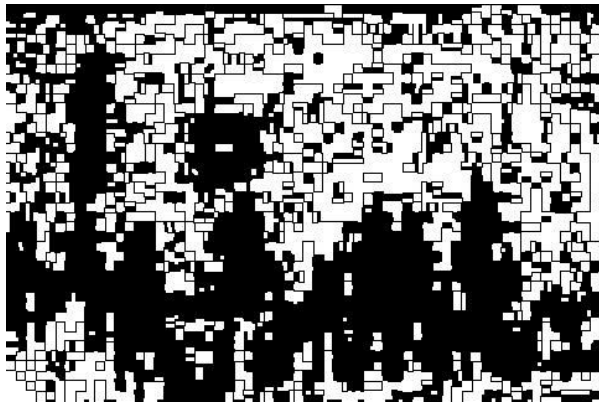
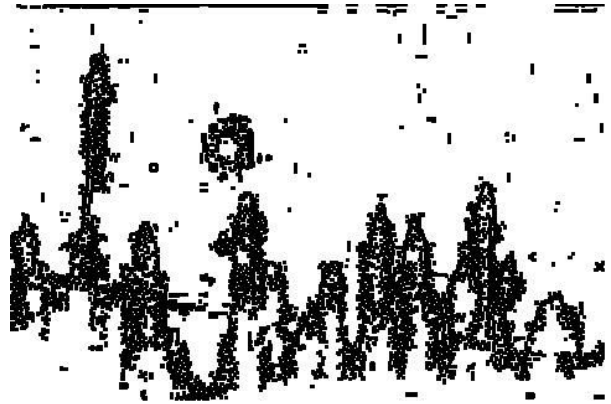


Figure 13 - Minima In Unprocessed Moon Image

The first goal will be the removal of the small dot minima within the tree sections. Opening and closing have the ability to remove small amounts of noise like that, so different structuring elements will be attempted to try to remove them. The order of operations also matters, so first the minima will be found, then opening and closing will be applied with 3x3 and 5x5 structuring elements (SE).



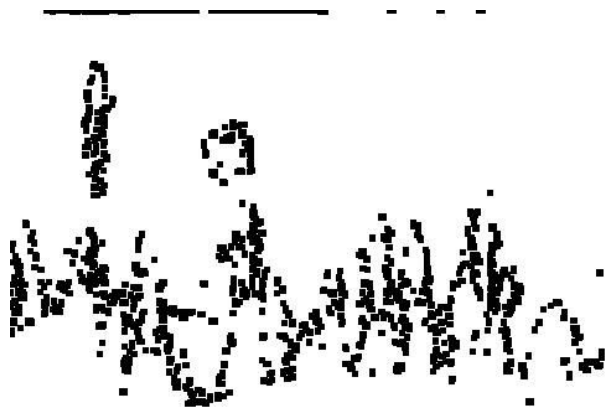
Opening with 3x3 Structuring Element After Finding Minima



Closing with 3x3 Structuring Element After Finding Minima



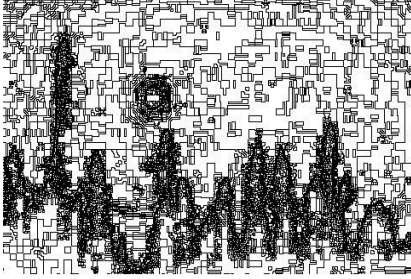
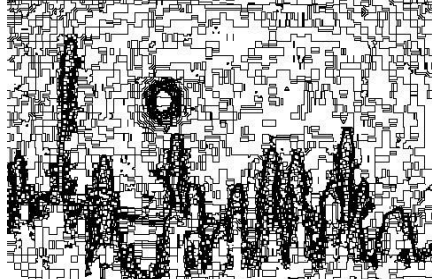
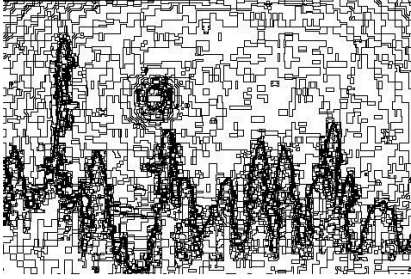
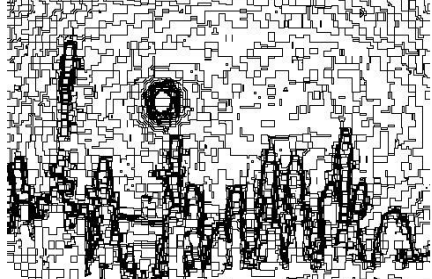
Opening with 5x5 Structuring Element After Finding Minima







Closing with 5x5 Structuring Element After Finding Minima

The results show that a smaller structuring element used for the closing operation reduces noise the most, so closing with a 3x3 SE will be the method of choice for removing noise after finding the minima.

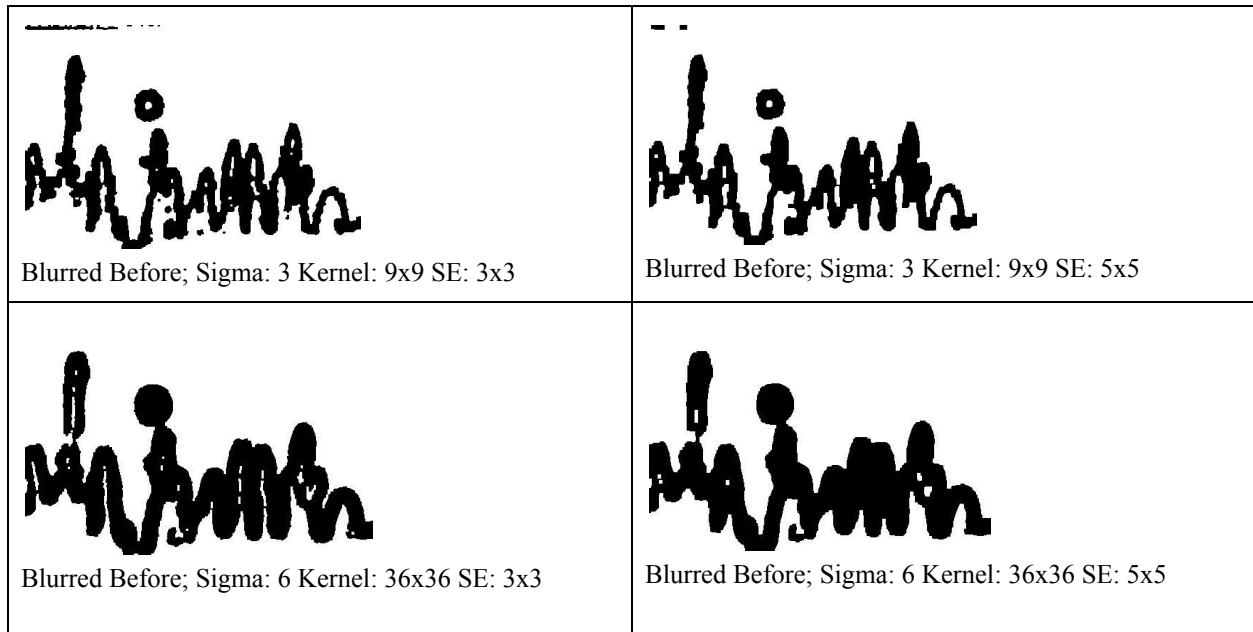
To determine just how important the order of operations for opening and closing, the same SEs were then applied before finding the minima of the image. The results are far less effective, so it is clear that the closing should be applied after the minima are found

 <p>Opening with 3x3 Structuring Element Before Finding Minima</p>	 <p>Closing with 3x3 Structuring Element Before Finding Minima</p>
 <p>Opening with 5x5 Structuring Element Before Finding Minima</p>	 <p>Closing with 5x5 Structuring Element Before Finding Minima</p>

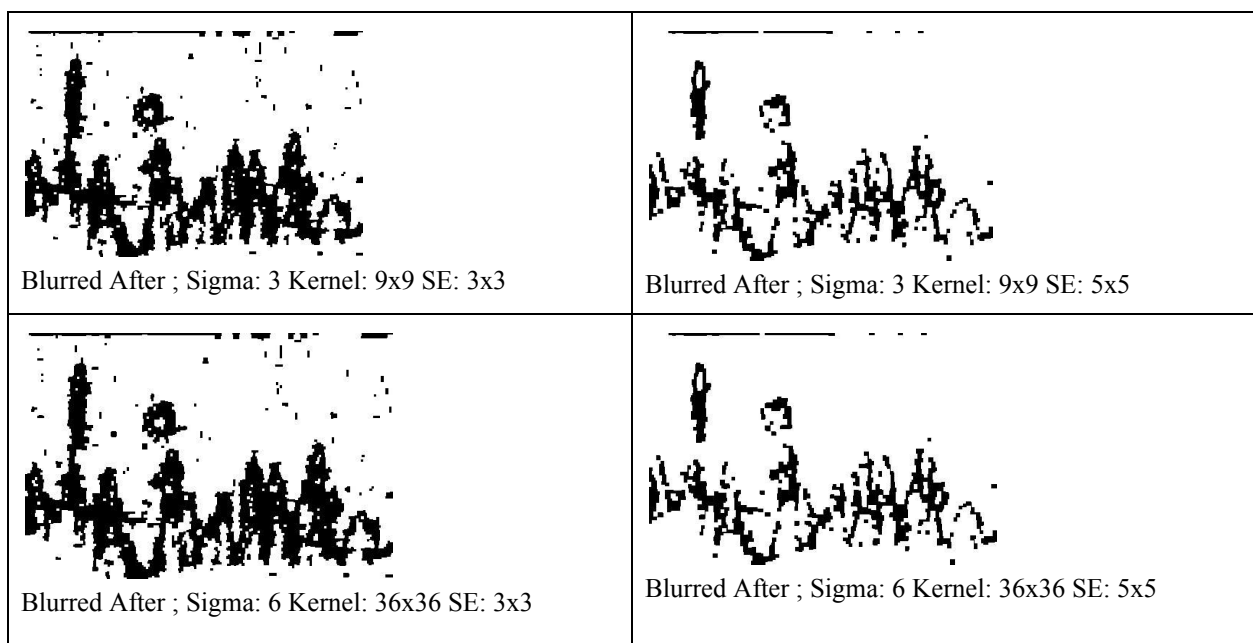
The combination of closing and opening can sometimes produce better results overall, so different combinations were also explored for the different SEs. These results can be seen in the following table. The best output came from opening the image closed by a 3x3 SE. Since closing with a 3x3 SE produced the best result earlier, this was not a surprise.

 <p>Opening Followed By Closing - 3x3 SE</p>	 <p>Closing Followed By Opening - 3x3 SE</p>
 <p>Opening Followed By Closing - 5x5 SE</p>	 <p>Closing Followed By Opening - 5x5 SE</p>

The next parameter tested was Gaussian blur. For the first test the blurring was applied to the original image before the minima were found. Two different levels of blurring were applied using the two different structuring elements, with closing then opening applied to the image.



Blurring before finding the minima clearly helps the final result. By blurring with a sigma of 3 and a 9x9 kernel, finding the minima, closing, then finally opening, a nearly noise free image is created.



Blurring afterwards does not produce significant results, so a pre blur with sigma 3 and 9x9 kernel will be used for the final result. With that image plugged into the watershed algorithm, the final result is:

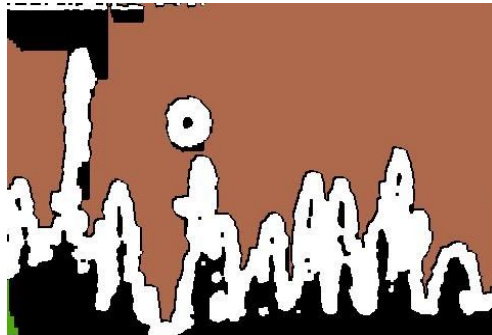


Figure 14 - Final Result With Pre-Processing

This image is not ideal, with some watershed lines found, but lots of junk data added. It's possible that a distance transform would help to isolate the objects in the image.

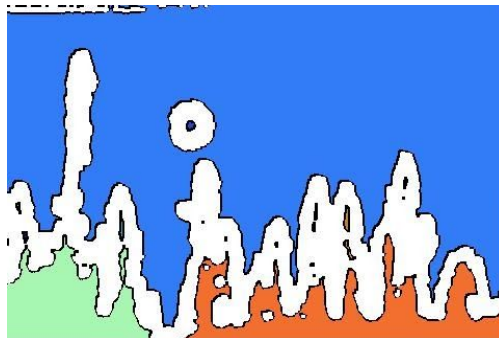


Figure 15 - Final Result With Distance Transform

The distance transform did end up helping significantly. The junk data that came from plateaus in the image have been removed and the outlines of the objects are more crisp.

4.0 Conclusion

This project used the watershed algorithm to segment the shapes within an image. By finding the local minima, labeling them with unique values, and applying those labels to all uphill neighbors a segmented image was created from three different sample images. For a noisy image the effect of pre-processing techniques like distance transforms and blurring resulted in fewer, clearer segments that more accurately represented the objects in the image. Ultimately it was determined that the best results come from images that have fewer minima. If the original image is noisy, then some noise can be removed by blurring the image, finding the minima, then closing and opening the image. Finally, a distance transform to remove plateaus in the image produces a clean result.

5.0 References

[1] Snyder, Wesley; Qi, Hairong. Fundamentals Of Computer Vision, 2017

[2] Python Documentation - <https://wiki.python.org/moin/PythonSpeed/PerformanceTips>.

[Online; Accessed 28-February-2020]

6.0 Appendix

```
import numpy as np
import imageio as io
import random as randy
import cv2
from skimage.measure import label, regionprops

def forwardT(image, mask):
    timg = image
    temp = np.zeros((len(image)+2, len(image[0])+2))

    temp[0][0] = np.inf
    temp[len(temp)-1][len(temp[0])-1] = np.inf
    temp[len(temp)-1][0] = np.inf
    temp[0][len(temp[0])-1] = np.inf

    for y in range(len(image[0])):
        temp[0][y+1] = np.inf
        temp[len(temp)-1][y+1] = np.inf
        for x in range(len(image)):
            temp[x+1][y+1] = image[x][y]
            temp[x+1][0] = np.inf
            temp[x+1][len(temp[0])-1] = np.inf

    for y in range(len(image)):
        for x in range(len(image[0])):
            data = np.array([mask[0][0]+temp[y][x],
                            mask[0][1]+temp[y][x+1],
                            mask[0][2]+temp[y][x+2],
                            mask[1][0]+temp[y+1][x],
                            mask[1][1]+temp[y+1][x+1]])
            data = np.sort(data)
            timg[y][x] = data[0]

    #print(timg)
    return timg

def backwardT(image, mask):
    timg = image
    temp = np.zeros((len(image)+2, len(image[0])+2))
```



```

temp[0][0] = np.inf
temp[len(temp)-1][len(temp[0])-1] = np.inf
temp[len(temp)-1][0] = np.inf
temp[0][len(temp[0])-1] = np.inf

for y in range(len(image[0])):
    temp[0][y+1] = np.inf
    temp[len(temp)-1][y+1] = np.inf
    for x in range(len(image)):
        temp[x+1][y+1] = image[x][y]
        temp[x+1][0] = np.inf
        temp[x+1][len(temp[0])-1] = np.inf

for y in range(len(image)):
    for x in range(len(image[0])):
        data = np.array([mask[0][1]+temp[y+1][x+1],
                        mask[0][2]+temp[y+1][x+2],
                        mask[1][0]+temp[y+2][x],
                        mask[1][1]+temp[y+2][x+1],
                        mask[1][2]+temp[y+2][x+2]])
        data = np.sort(data)
        timg[y][x] = data[0]

#print(timg)
return timg

def transform(image, mask, reverse):
    img = forwardT(image, mask)
    img = backwardT(img, reverse)
    return img

def distance(imageIn):
    mask = np.array([[4, 3, 4], [3, 0, np.inf]])
    rev_mask = np.array([[np.inf, 0, 3], [4, 3, 4]])
    print("Finding Minima...")
    mins = imageIn.astype(float)
    minima = imageIn.astype(float)
    mins = (mins * 0) + 255
    minima = (minima * 0) + 255

    for y in range(len(mins)-1):
        for x in range(len(mins[0])-1):
            if (imageIn[y-1][x-1] < imageIn[y][x] or
                imageIn[y-1][x] < imageIn[y][x] or
                imageIn[y-1][x+1] < imageIn[y][x] or
                imageIn[y][x-1] < imageIn[y][x] or
                imageIn[y][x+1] < imageIn[y][x] or
                imageIn[y+1][x-1] < imageIn[y][x] or
                imageIn[y+1][x] < imageIn[y][x] or
                imageIn[y+1][x+1] < imageIn[y][x]):

```

```

        mins[y][x] = np.inf
        minima[y][x] = 0

    if ((imageIn[y-1][x-1] == imageIn[y][x] and mins[y-1][x-1] == 0) or
        (imageIn[y-1][x] == imageIn[y][x] and mins[y-1][x] == 0) or
        (imageIn[y-1][x+1] == imageIn[y][x] and mins[y-1][x+1] == 0) or
        (imageIn[y][x-1] == imageIn[y][x] and mins[y][x-1] == 0) or
        (imageIn[y][x+1] == imageIn[y][x] and mins[y][x+1] == 0) or
        (imageIn[y+1][x-1] == imageIn[y][x] and mins[y+1][x-1] == 0) or
        (imageIn[y+1][x] == imageIn[y][x] and mins[y+1][x] == 0) or
        (imageIn[y+1][x+1] == imageIn[y][x]) and mins[y+1][x+1] == 0):
        mins[y][x] = np.inf
        minima[y][x] = 0

print("Finding Distances...")
i = 0
while (max(map(max, mins)) == np.inf):
    print(i)
    i += 1
    dist = transform(mins, mask, rev_mask)
    mins = dist
mins = mins / 3 # Normalizing by 3
return mins, minima

def assignLabels(minima):
    print("Labeling...")
    unique = label(minima)
    return unique

def poolParty(drains, imageIn):
    print("Watershedding...")
    upstream = []

    for y in range(len(drains)-1):
        for x in range(len(drains[0])-1):
            upstream.append([imageIn[y][x], y, x])

    upstream = sorted(upstream, key=lambda x: x[0])
    #while (0 in x for x in drains):
    while len(upstream) != 0:
        x = upstream[0][2] # Coordinates of the lowest brightness pixel
        y = upstream[0][1]
        print(len(upstream))

        # If neighbor pixel is already labeled differently, it is a watershed pixel:
        if (drains[y-1][x-1] != 0) and (drains[y-1][x-1] != drains[y][x]):
            drains[y-1][x-1] = -1

```

```

elif (drains[y-1][x] != 0) and (drains[y-1][x] != drains[y][x]):
    drains[y-1][x] = -1

elif (drains[y-1][x+1] != 0) and (drains[y-1][x+1] != drains[y][x]):
    drains[y-1][x+1] = -1

elif (drains[y][x-1] != 0) and (drains[y][x-1] != drains[y][x]):
    drains[y][x-1] = -1

elif (drains[y][x+1] != 0) and (drains[y][x+1] != drains[y][x]):
    drains[y][x+1] = -1

elif (drains[y+1][x-1] != 0) and (drains[y+1][x-1] != drains[y][x]):
    drains[y+1][x-1] = -1

elif (drains[y+1][x] != 0) and (drains[y+1][x] != drains[y][x]):
    drains[y+1][x] = -1

elif (drains[y+1][x+1] != 0) and (drains[y+1][x+1] != drains[y][x]):
    drains[y+1][x+1] = -1

# If neighbor pixel is unlabeled, label it with current label
elif (imageIn[y-1][x-1] > imageIn[y][x]) and (drains[y-1][x-1] != -1):
    drains[y-1][x-1] = drains[y][x]

elif (imageIn[y-1][x] > imageIn[y][x]) and (drains[y-1][x] != -1):
    drains[y-1][x] = drains[y][x]

elif (imageIn[y-1][x+1] > imageIn[y][x]) and (drains[y-1][x+1] != -1):
    drains[y-1][x+1] = drains[y][x]

elif (imageIn[y][x-1] > imageIn[y][x]) and (drains[y][x-1] != -1):
    drains[y][x-1] = drains[y][x]

elif (imageIn[y][x+1] > imageIn[y][x]) and (drains[y][x+1] != -1):
    drains[y][x+1] = drains[y][x]

elif (imageIn[y+1][x-1] > imageIn[y][x]) and (drains[y+1][x-1] != -1):
    drains[y+1][x-1] = drains[y][x]

elif (imageIn[y+1][x] > imageIn[y][x]) and (drains[y+1][x] != -1):
    drains[y+1][x] = drains[y][x]

elif (imageIn[y+1][x+1] > imageIn[y][x]) and (drains[y+1][x+1] != -1):
    drains[y+1][x+1] = drains[y][x]

else: # if none are greater...
    drains[y][x] = drains[y][x]
del(upstream[0])

```

```

        return drains

def randColor(i):
    return [randy.randint(1,254), randy.randint(1,254), randy.randint(1,254)]

def colorTime(unique):
    print("Colorizing...")
    drains = np.zeros((len(unique), len(unique[0]), 3)) # Make an RGB image
    colors = np.zeros((max(map(max, unique)) + 2, 3)) # Make a color list
    colors = list(map(randColor, colors))
    colors[0] = [0,0,0]
    colors[1] = [255,255,255]
    drains = list(map(lambda x: list(map(lambda y: colors[y+1], x)), unique)) # Color
every region based on label value
    return drains

##### Main #####

sigma = 1
kSize = 3 // 2
y, x = np.mgrid[-kSize:kSize+1, -kSize:kSize+1]
gausBase = 1 / (2 * np.pi * sigma**2)
gaus = (gausBase * np.exp(-((x**2) + (y**2)) / (2*sigma**2)))

pathIn = "C:\\Users\\a2tri\\source\\repos\\watershedMain\\"
imgName = "moon_segmentation.jpg"

kernel = np.ones((5,5), np.uint8)

imageIn = io.imread(pathIn + imgName)
#imageIn = cv2.morphologyEx(imageIn, cv2.MORPH_CLOSE, kernel)
imageOut = cv2.filter2D(imageIn, -1, gaus)
dist, minima = distance(imageOut)
labels = assignLabels(minima)
watershed = poolParty(labels, dist)
withColor = colorTime(watershed)

pathOut = pathIn + "NoD_" + imgName
io.imwrite(pathOut, withColor)
print(labels)

```

7.0 Bloopers

