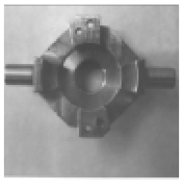
lab3

November 14, 2022

1 ECE 5470 Lab 3 Report

1.0.1 by Cynthia Li

[164]:
```
import numpy as np
from v4 import vx
from v4 import vd

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

vd.dispmvx("mp.vx", size=0.3, capt="im1.vx")
```



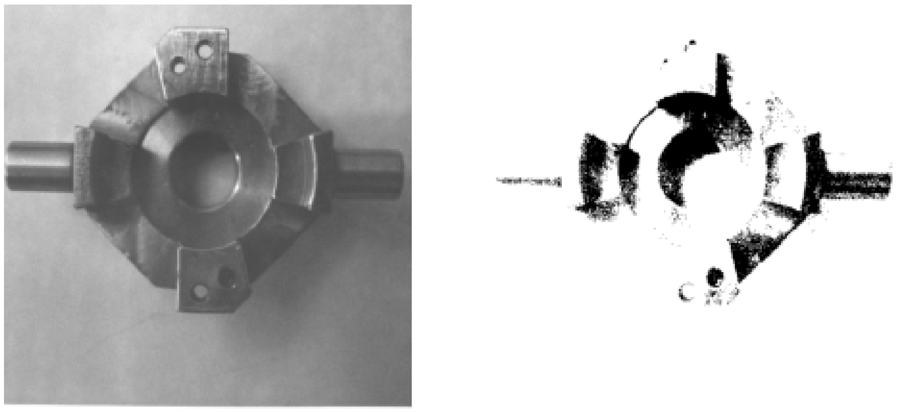
```
im1.vx
<scaled size: (512 x 512) >
```

1.1 Section 2

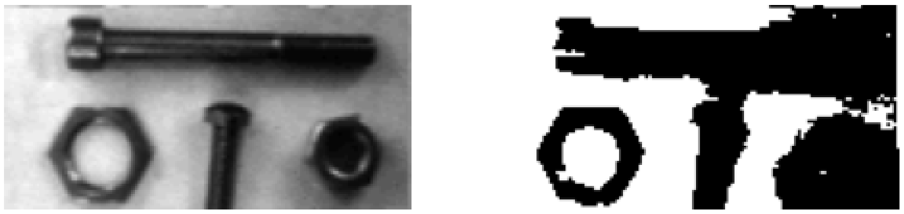
1.1.1 2.4.1 Display vtpeak Result Of Each Case

[165]:
```
vd.dispmvx('mp.vx','mp_pk.vx',size=0.75,  capt='result of vtpeakpy on mp.vx␣
 ↪with d=10')
vd.dispmvx('nb.vx','nb_pk.vx',size=0.75,  capt='result of vtpeakpy on nb.vx␣
 ↪with d=10')
vd.dispmvx('map','map_pk',size=0.75,  capt='result of vtpeakpy on map with␣
 ↪d=10')
vd.dispmvx('facsimile','facsimile_pk',size=0.75,  capt='result of vtpeakpy on␣
 ↪facsimile with d=10')
```

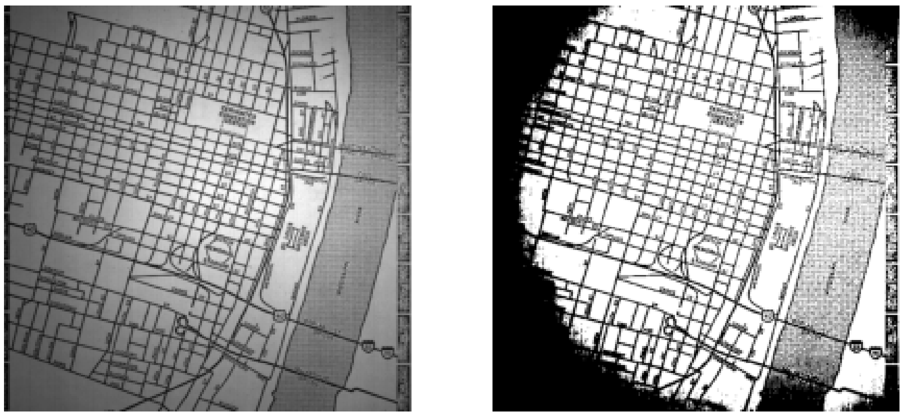
```python
vd.dispmvx('shtl.vx','shtl_pk.vx',size=0.75,  capt='result of vtpeakpy on facsimile with d=10')
```



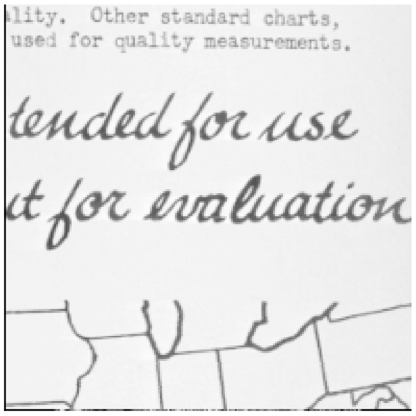
```
result of vtpeakpy on mp.vx with d=10
<scaled size: (512 x 512) (512 x 512) >
```

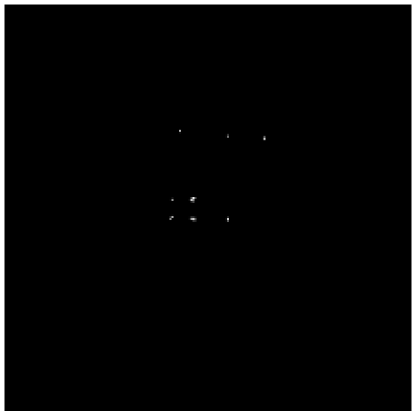


```
result of vtpeakpy on nb.vx with d=10
<scaled size: (64 x 128) (64 x 128) >
```

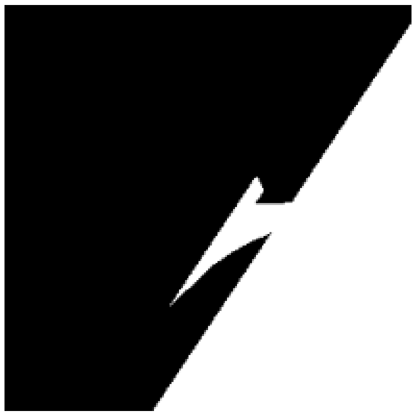


`result of vtpeakpy on map with d=10`
`<scaled size: (512 x 512) (512 x 512) >`





`result of vtpeakpy on facsimile with d=10`
`<scaled size: (256 x 256) (256 x 256) >`



`result of vtpeakpy on facsimile with d=10`
`<scaled size: (256 x 256) (256 x 256) >`

Observations on images after vtpeak From the results displayed in the above sets, We can see that vtpeak didn't have an ideal output when d is simply set to 10 in default for every image. Some observations of the image after vtpeak (compared to the original) from each case are listed as below:

In mp.vx, the contour of the image after vtpeak blends with the environment, and the shading on the object is strengthened.

In nb.vx's performance, the oject is roughly separated from the background, but the shading would cause overcovering.

In map, the majority part of the object is enhanced (the contrast between black as road and white blocks as land or house) but near the boundary, the shading makes the area all erased as black (undercovered).

In facsimile, the whole image is erased to black except several tiny points in the middle remained white.

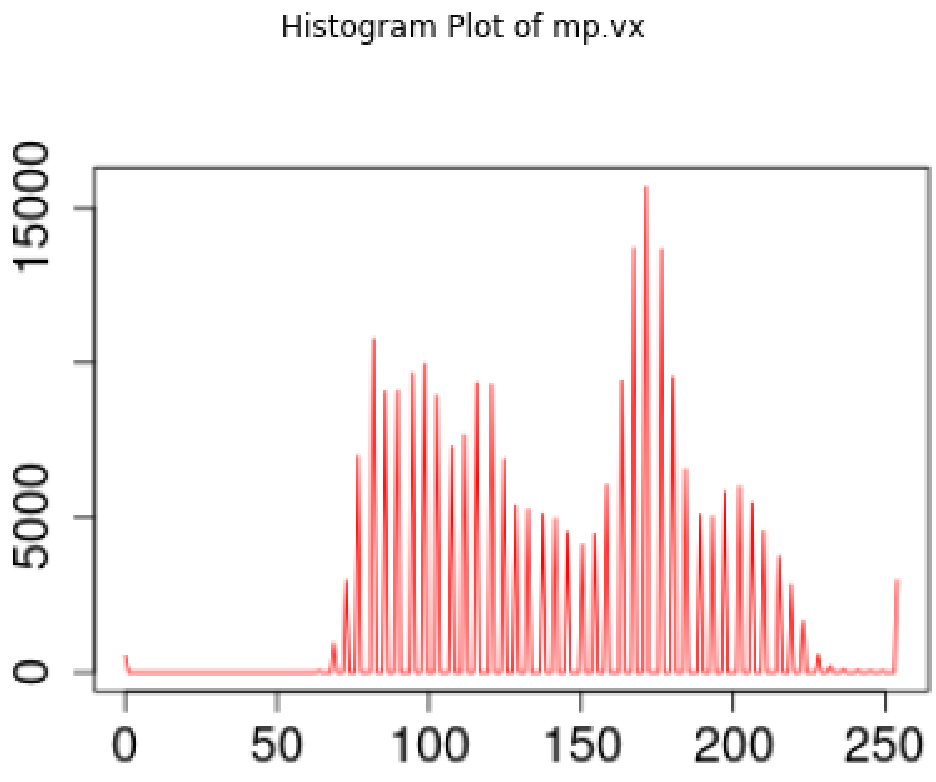
In shtl.vx, the image loses a lot of information. Only a small part of the shuttle is shown as white in the image, but all featuers regarding to its shape are lost.

1.1.2 2.4.2 Display Histogram And Threshold Of Each Case

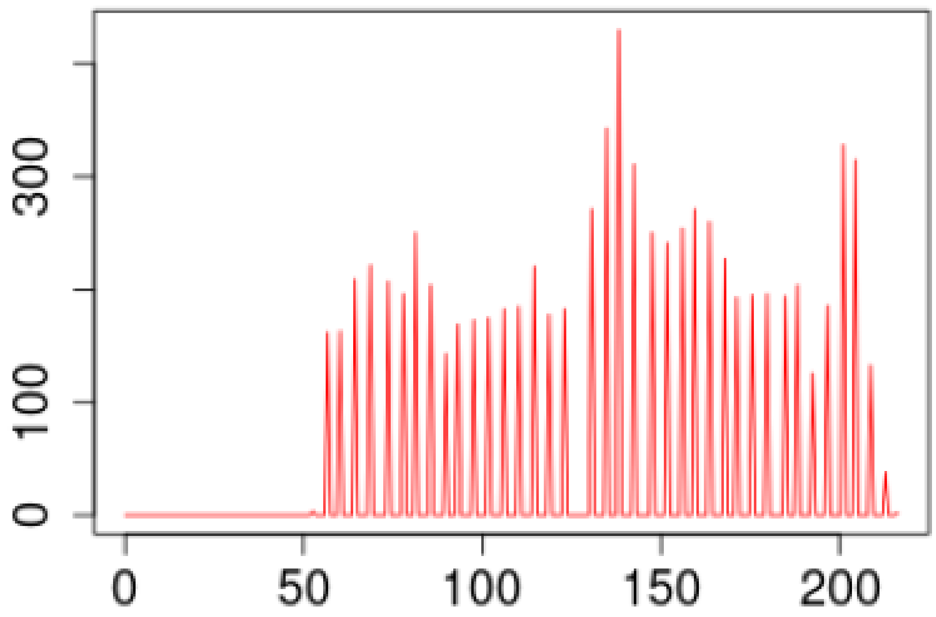
[181]:
```
# Display histogram of the original image for each case
# figure size is set to make the figure larger
# Axis are turned off because we are showing png/jpg image files
# Histograms images comes with their own axes, additional axes may cause␣
 ↪confusion
mp_plot = mpimg.imread('mp_plot.png')
plt.figure(figsize = (8, 6))
imgplot = plt.imshow(mp_plot)
plt.axis('off')
plt.title("Histogram Plot of mp.vx")

nb_plot = mpimg.imread('nb_plot.png')
plt.figure(figsize = (8, 6))
imgplot = plt.imshow(nb_plot)
plt.axis('off')
plt.title("Histogram Plot of nb.vx")

map_plot = mpimg.imread('map_plot.png')
plt.figure(figsize = (8, 6))
imgplot = plt.imshow(map_plot)
plt.axis('off')
plt.title("Histogram Plot of map")

facsimile_plot = mpimg.imread('facsimile_plot.png')
plt.figure(figsize = (8, 6))
imgplot = plt.imshow(facsimile_plot)
plt.axis('off')
plt.title("Histogram Plot of facsimile")

shtl_plot = mpimg.imread('shtl_plot.png')
plt.figure(figsize = (8, 6))
imgplot = plt.imshow(shtl_plot)
plt.axis('off')
plt.title("Histogram Plot of shtl.vx")
```

```
# Show the Threshold for each case
thresholds = mpimg.imread('threshold_vtpeakpy.jpg')
plt.figure(figsize = (10, 5)) ## make it larger to see (since original␣
 ↪screenshot is small)
imgplot = plt.imshow(thresholds)
plt.axis('off') ## turn off the axis because this is an image not a plot
plt.title("Check Automatic Thresholds for All The Images Above")
```

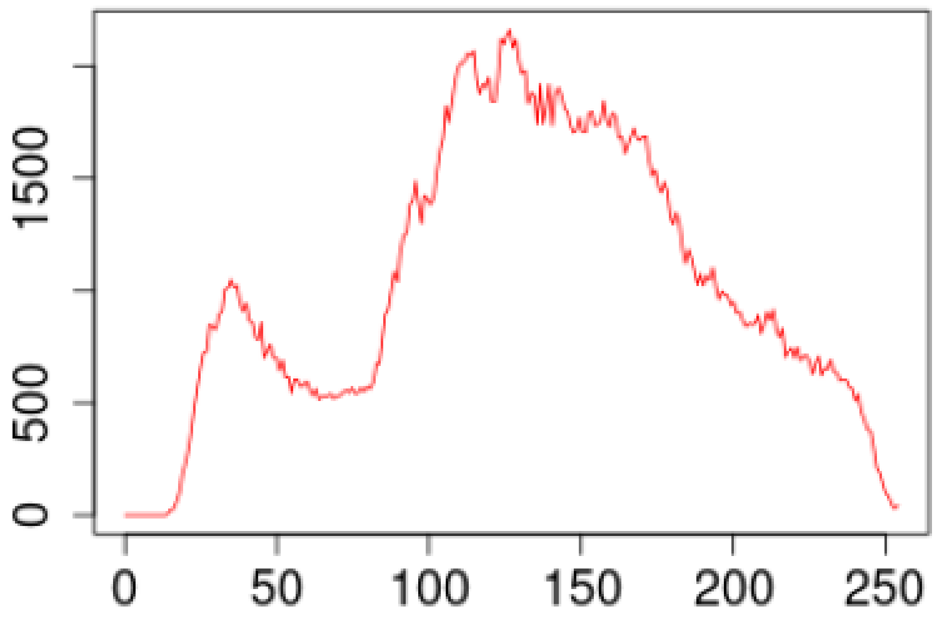
```
[181]: Text(0.5, 1.0, 'Check Automatic Thresholds for All The Images Above')
```



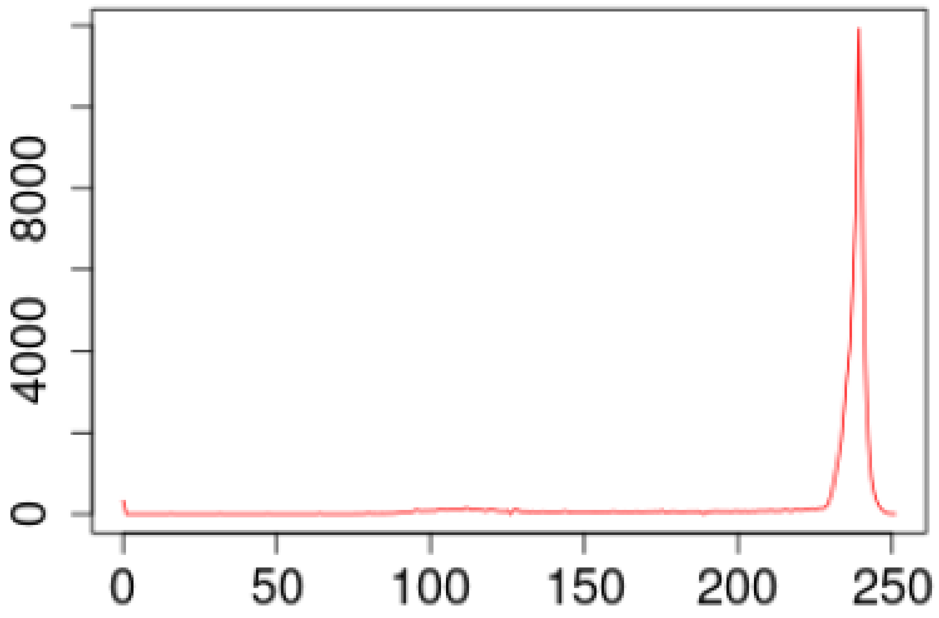
Histogram Plot of nb.vx



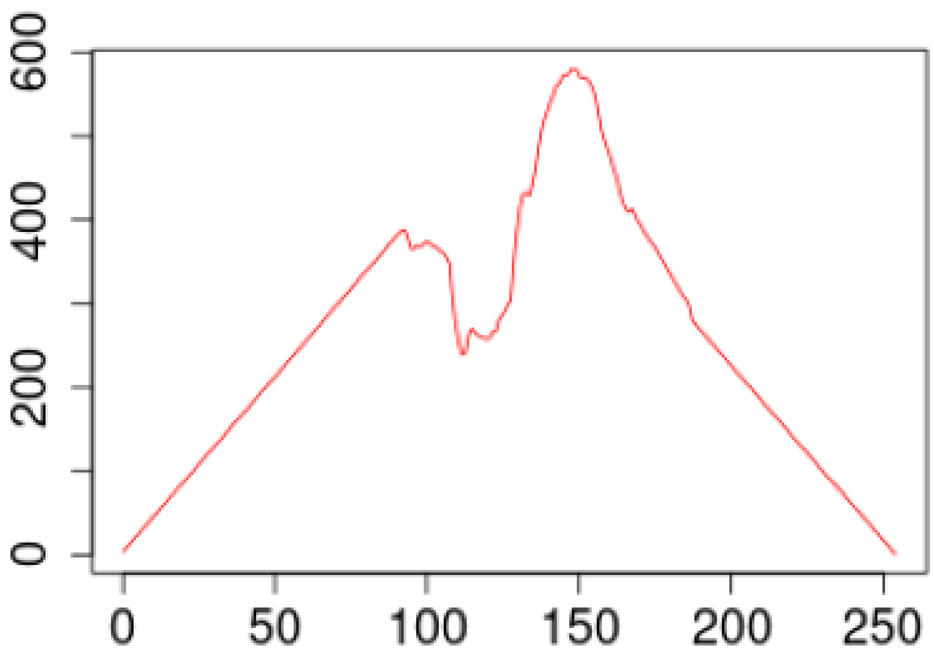
Histogram Plot of map



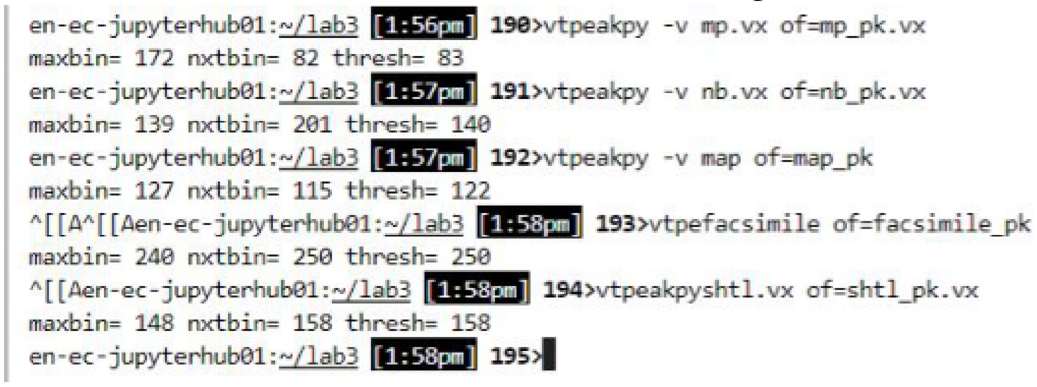
Histogram Plot of facsimile



Histogram Plot of shtl.vx



Check Automatic Thresholds for All The Images Above



```
en-ec-jupyterhub01:~/lab3 [1:56pm] 190>vtpeakpy -v mp.vx of=mp_pk.vx
maxbin= 172 nxtbin= 82 thresh= 83
en-ec-jupyterhub01:~/lab3 [1:57pm] 191>vtpeakpy -v nb.vx of=nb_pk.vx
maxbin= 139 nxtbin= 201 thresh= 140
en-ec-jupyterhub01:~/lab3 [1:57pm] 192>vtpeakpy -v map of=map_pk
maxbin= 127 nxtbin= 115 thresh= 122
^[[A^[[Aen-ec-jupyterhub01:~/lab3 [1:58pm] 193>vtpefacsimile of=facsimile_pk
maxbin= 240 nxtbin= 250 thresh= 250
^[[Aen-ec-jupyterhub01:~/lab3 [1:58pm] 194>vtpeakpyshtl.vx of=shtl_pk.vx
maxbin= 148 nxtbin= 158 thresh= 158
en-ec-jupyterhub01:~/lab3 [1:58pm] 195>
```

Discuss histogram and the threshold results of each case As we can see from the above histogram, there are mainly two problems with vtpeak. First is that the default d=10 for the vtpeak may not detects the peak in shape, the second is that it doesn't have a solution for images with only one peak.

In mp.vx, the highest spike and the second highest spikes (two on each side of the highest spike) are less than 10 unit apart on the histogram, so it will take the third highest spike at 82 as the second peak. This matches with what we would consider as a peak (shape in general, rather than spikes like this). And therefore, a d=10 works on this picture.

In nb.vx, it is pretty similar to mp.vx, the second highest spike was not counted as a peak due to d=10, and therefore, the peaks it recognized is the same as what we would consider as a peak in shape. So d = 10 should also works for this picture.

In map, however, problem occurs. From the histogram, we can see that between 115 and 127 there is a local convex part, and because 115 and 127 are 12 unit apart which is greater than d=10, they will be recognized as two peaks. However, if we look at the image in general, we may want to consider 115-127 as one peak, and take 35 (roughly estimated) as t second peak. In this case, d=10 is not a very nice parameter for vtpeak on this image.

In facsimile, there is a different problem. The information (pixel value) of the image is so concentrated at around 240, that there is just a single spike. Therefore, when the algorithm is using d=10 to find the second peak, there is no second peak, so it evaluates a point on the falling edge of the spike that is 10 unit apart from the peak as the second peak. In this case, the threshold is set at the peak, which filters out many pixels that contain information about the image. An noticing point here is if there is only a single spike, the second peak is on the falling edge rather than the rising edge. Therefore, d=10 is also not working well on this image.

In shtl.vx, the problem is the same as the map. d=10 would not detect a peak in the shape, but a high value on the falling edge of the highest peak that is still higher than the value of the second highest peak. In this case, we wouldn't expect d=10 would work very well for the image.

1.1.3 2.4.3 Discussion On vtpeak

After checking the vtpeakpy file, we can see that vtpeak is an algorithm that uses thresholding for image segmentation. How it operates is that it computes the histogram of the input image, first finds the highest peak(with index maxbin) in the whole histogram, then iterates to search for the second peak in region maxbin+d to 255 and 0 to maxbin-d, after deciding which is the second peak, it will find the index with minimum value between the two peaks as a threshold and conduct thresholding on the input image.

Overall, vtpeak, with a proper threshold, should output a binary image that segment the object from its background which is easier for analysis.

However, from the above 5 image tests, we can see that vtpeak does not work perfectly, especially without first peeking into the histogram of the input images. In many cases, it does require us to have knowledge on the histogram of the image so that we can manually set a parameter d to more accurately detects the peaks in the general shape rather than local fluctuation.

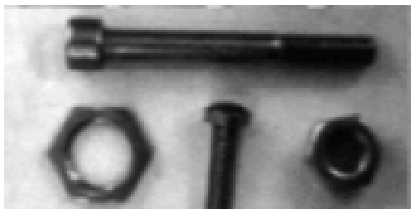
One defect with vtpeak is that the default parameter d=10 would likely to misrecognize a peak and result in a unhelpful thresholding in many cases (such as map, facsimile, and shuttle).

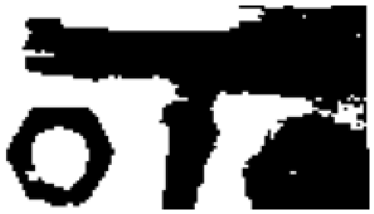
The second problem is that the algorithm doesn't do a very good job to threshold the images whose histogram has only one peak, or in terms of image, with little variation in color. It will basically choose an index (maxbin+d) on the falling edge of the spike and use it as the threshold, which is obviously not ideal because it would erase majority of the image's information.

A third issue with vtpeak is that how it finds the minimum between the peaks restricted the type of image it can be used. The algorithm simply looks for the first index with minimum value between second peak and maxbin (or maxbin and second peak if second peak has larger index). And the "first" becomes the problem. When there are multiple indexes with equal minimum, the algorithm will always take the smallest index, which may neglect important information contained by index with higher values. This is extremely obvious on image mp.vx and nb.vx where the algorithm simply takes the index of the first 0 next to the maxbin or the second peak (83 and 140), while the spikes should not be separated by that index (ideally maybe at 150 and 130).

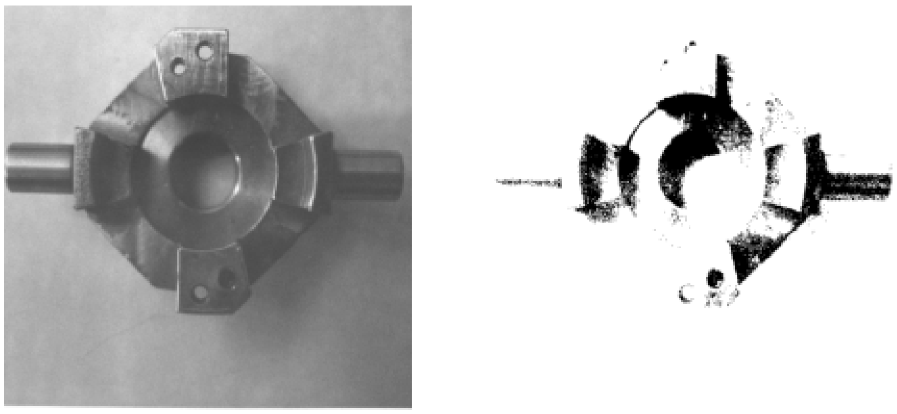
1.1.4 2.4.4 Testing Different Parameter d

[173]:
```
exec(vx.vxsh('vtpeakpy d=63 nb.vx of=nb_pk2.vx'))
vd.dispmvx('nb.vx','nb_pk2.vx',size=0.75,  capt='result of vtpeakpy on nb.vx␣
 ↪with d=63')

exec(vx.vxsh('vtpeakpy d=90 mp.vx of=mp_pk2.vx'))
vd.dispmvx('mp.vx','mp_pk2.vx',size=0.75,  capt='result of vtpeakpy on mp.vx␣
 ↪with d=90')

exec(vx.vxsh('vtpeakpy d=85 map of=map_pk2'))
vd.dispmvx('map','map_pk2',size=0.75,  capt='result of vtpeakpy on map with␣
 ↪d=85')

exec(vx.vxsh('vtpeakpy d=0 facsimile of=facsimile_pk2'))
vd.dispmvx('facsimile','facsimile_pk2',size=0.75,  capt='result of vtpeakpy on␣
 ↪facsimile with d=0')

exec(vx.vxsh('vtpeakpy d=50 shtl.vx of=shtl_pk2.vx'))
vd.dispmvx('shtl.vx','shtl_pk2.vx',size=0.75,  capt='result of vtpeakpy on shtl.
 ↪vx with d=50')
```

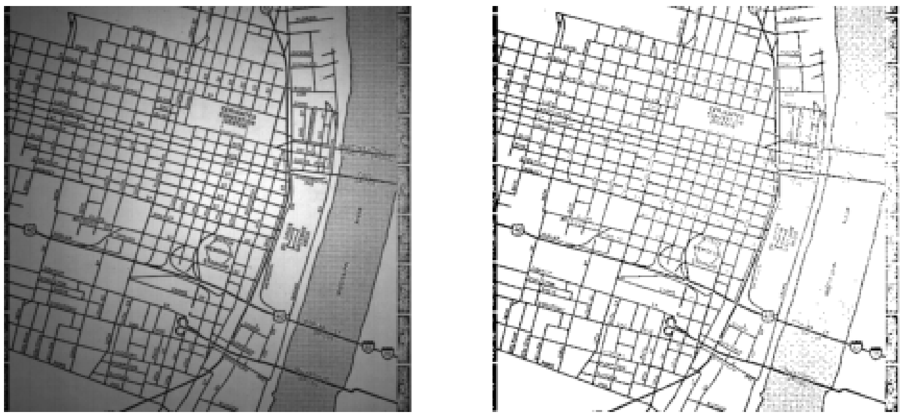




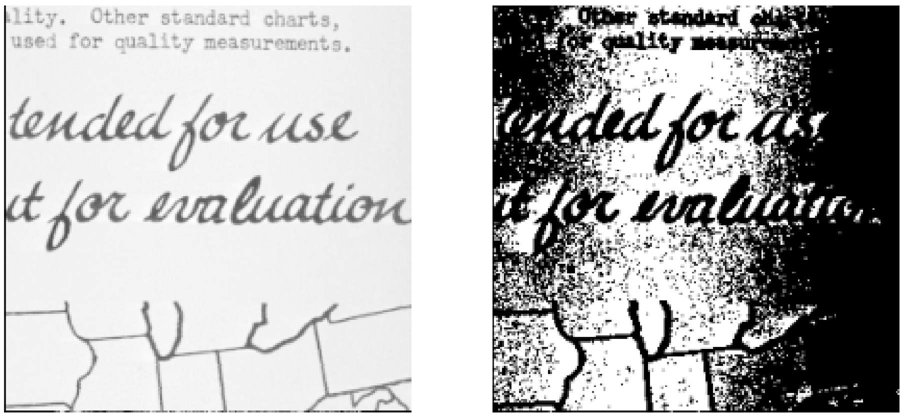
```
result of vtpeakpy on nb.vx with d=63
<scaled size: (64 x 128) (64 x 128) >
```



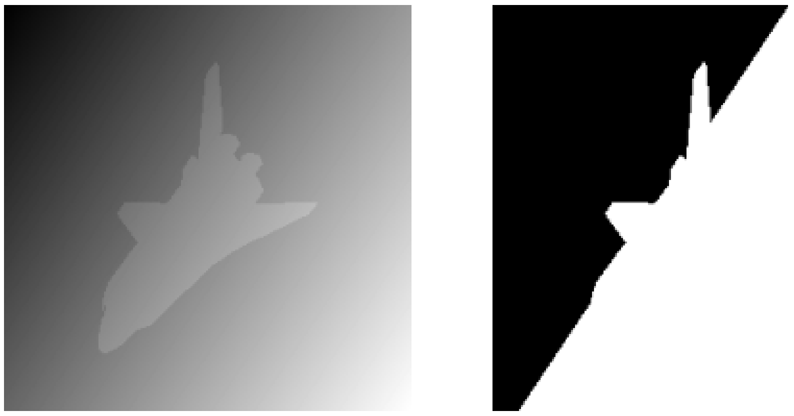
```
result of vtpeakpy on mp.vx with d=90
<scaled size: (512 x 512) (512 x 512) >
```



```
result of vtpeakpy on map with d=85
<scaled size: (512 x 512) (512 x 512) >
```



```
result of vtpeakpy on facsimile with d=0
<scaled size: (256 x 256) (256 x 256) >
```



```
result of vtpeakpy on shtl.vx with d=50
<scaled size: (256 x 256) (256 x 256) >
```

Improving Effect of vtpeak With Manually Input of d As discussed in 2.4.2, the default d would work for mp.vx and nb.vx since d=10 already allow the algorithm to find the actual peaks for these two images. So even when we change d = index of highest peak - index of second highest peak, it would gave the same result. And this result is the best output that the algorithm is able to give. This is why even when we change parameter d for mp.vx and nb.vx (as the first two sets of images above), we see no difference than the previous result in 2.4.1 with d=10.

However, for some other cases, changing d can improve the results a lot.

For map, we can see that the indexes for the two major peaks are approximately 130 and 40 but there is also a tiny peak at index=110 that we want to ignore and take it as the same peak as index=130. Therefore, to be safe, I selected d=85 (in actual testing, d from 68 to 101 gives the same result), and we can see that the output image basically eliminated the effect of weird lighting and shading in the original image and showed an intact object (though the gray band-shaped area in the original image would be lightened as well).

For facsimile, the best we can do is to use d=0 so that the threshold is set to the index of the peak, so we lose the least information under this algorithm. The output is now much better than the output in 2.4.1. Though it still have distracting black area around the corner, part of the drawing and text can be recognized, which is a huge improvement.

The improvement by changing d is limited though. There are certain types of image that this algorithm is not suitable to apply. For shtl, though we can change d to 50 (the peak is approximately at index 150 and 90, which gives a distance of 60) and the output improved from the results in 2.4.1 as it shows the upper boundary of the shuttle now, it is not very useful as the lower part of the object is completely invisible.

Final Discussion On vtpeak In conclusion, vtpeak does give a solution of thresholding and can provide a nice segmented binary images in some cases. However, it does requires the user to have knowledge of the image (histogram) and it does not work perfectly. In some cases, manually changing the parameter d after checking the image's histogram can improve the segmented output. But still, vtpeak are still pretty selective on the type of image that it would work well on. Image with slight shading or lighting may be well segmented, image with little color difference between the object and the background may have more trouble segmenting (likely to result in undercovering or overcovering). vtpeak doesn't have ideal output on images with large variance in the backgrounds, either.

1.2 Section 3 Iterative Threshold Selection (vits)

1.2.1 3.1 The vits program

[212]:
```
#!/usr/bin/env python

"""
vits:  Iterative Threshold Selection algorithm

       May uses input parameter th= as initial threshold, or defaultly uses the
       average of grayscale value of the image as the initial threshold. It
       iteratively takes the average (a1) of values above the threshold and the
       average (a2) of values below the threshold, computes new thresh=(a1+a2)/
 ↪2.
       The algorith determines the final thresh when a1 and a2 is not changing␣
 ↪or
       after 256 loops. It then perform thresholding to the input image and␣
 ↪output
       a new segmented binary image to the specified file.
"""

import numpy as np
import math
from v4 import vx
```

```python
vargs = vx.vaparse("if= of= th= -v -")

# Print argument info to stdout when prompted
if 'OPT' in locals():
    print ("vits: iterative threshold selection")
    print ("if= input file")
    print ("of= output file")
    print ("th=  initial threshold (default use average grayscale value)")
    print ("-v  (verbose) print threshold information")
    exit(0)

# Get input image and output image file name, print error message if missing
optv = '-v' in vargs
for arg in ('if', 'of'):
    if arg not in vargs:
        print ( 'vits error: missing required parameter %s' % arg)
        exit(1)

# read in input image
inimage = vx.Vx(vargs['if'])
im = inimage.i

# check if input image has correct datatype to apply operations
if im.dtype != 'uint8' :
    print ("error: image not byte type" )
    exit(1)

# Set an initial threshold
thresh = 0

# Set the thresh= th if parameter is specified
if 'th=' in vargs:
    th = int(vargs['th'])
    if th < 0 or th > 255: # print error message if the given th is invalid
        print("error: th= must be between 0 and 255" )
        exit(1)
    else:
        thresh = round(th)
# Set the thresh= average grayscale value of the image if not specified
else:
    thresh = round(np.average(im))

#print(thresh)

# Create a blank histogram of 256 bins
hist = np.zeros(256)
```

```
# Compute the histogram
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        hist[im[y,x]]+=1

# global values to update current average
avg1 = 0
avg2 = 0
# global values to store average from previous loop
avg1_pre = -1
avg2_pre = -1

count = 0

# Stop when the average stop changing or when the loop exceeds 256
while ((avg1 != avg1_pre or avg2 != avg2_pre) and (count <= 256)):
    # keep a copy of previous avg of R1 & R2 before updating
    avg1_pre = avg1
    avg2_pre = avg2

    # local values to store/update the total sum and number of pixels
    # in the region
    sum_r1 = 0
    sum_r2 = 0
    n_r1 = 0
    n_r2 = 0

    # compute average of R1 (region above thresh)
    for i in range (thresh+1, 256):
        sum_r1 = sum_r1 + hist[i] * i ## value * number of pixels that has this␣
 ↪value
        n_r1 = n_r1 + hist[i] ## update the number of pixels in R1
    if (n_r1 != 0):
        avg1 = round(sum_r1 / n_r1)
    else:
        avg1 = thresh

    # compute average of R2 (region below thresh)
    for i in range (0, thresh):
        sum_r2 = sum_r2 + hist[i] * i
        n_r2 = n_r2 + hist[i]
    if (n_r2 != 0):
        avg2 = round(sum_r2 / n_r2)
    else:
        avg2 = thresh
    # compute and update the new threshold
    if(avg1 == 0 and avg2 != 0):
        thresh = round((avg2 + thresh) / 2)
    elif(avg1 != 0 and avg2 == 0):
        thresh = round((avg1 + thresh) / 2)
    else: # this also include the case that avg1 and 2 are both 0 but it␣
↪shouldn't happen because this means the image is purely black
        thresh = round((avg1 + avg2) / 2)

    # print("avg1 = ", avg1, "avg2 = ", avg2, "thresh = ", thresh) # this line␣
↪is for testing
    # increment the count vairable to prevent infinite loop
    count+=1

# Apply thresholding to the input image
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        if (im[y,x] > thresh):
            im[y,x] = 255
        else:
            im[y,x] = 0


# Print output for verbose mode
if (optv):
    print("thresh=", thresh)

inimage.write(vargs['of'])
```

1.2.2 3.2 Program Discussion

In Section 3, our goal is to generate an iterative threshold selection algorithm in python, named as vits.py. In the coding process, we used vtpeakpy as a template. I will introduce the program (Checking Command arguments & Algorithm description section), discuss some considerations when writing the algorithm and the advantages of the method.

Checking Command arguments: 1. if= byte-type input binary image file name; 2. of= output binary image file name; 3. th= initial threshold: automatically rounded if input is not integer. Set to user input if the input is within 0-255, print error message to stdout if not within the range. If there is no input, defaultly set to the average pixel value of the image. 4. -v: user option to print the final threshold applied to the stdout.

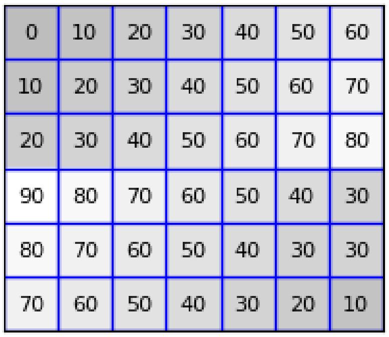
Algorithm description: 1. The program generates a histogram of the image. 2. The program divides the histogram into two regions: region R1 with index larger than the threshold, region R2 with index smaller than the threshold 3. The program computes the average of all pixels in R1 as avg1 and the average of all pixels in R2 as avg2. 4. The program update threshold to (avg1 + avg2)/2. 5. The program repeat step 2 to 4 until avg1 and avg2 stops changing. 6. Using the final threshold, the program search through each pixel, makes pixels with value greater than the threshold as 255 and others as 0. 7. The program writes the image after thresholding to the output file specified.

Things to consider for the program: 1. Computing the average pixel value of the image: 1) the easiest way to think about it is to iterate through every pixel, get a sum of pixel value as well as count the number of pixel, and then divide sum by count; 2) a leap from 1): use np.sum for the sum and im.shape[0]*im.shape[1] for count; 3) a leap from 2) use np.average to directly output the average of the image. 2. Checking th validity. We want th to be an integer (this is intuitively reasonable because our pixel value should be a byte, therefore an integer between 0-255 as the index of the histogram), but users may input a non-integer value and the average computation is likely to give a non-integer value as well. In this case, we want to convert the input/results to an integer. There are multiple functions we can use: 1) python's built-in int() casting, 2) math.trunc(), 3) python's built-in function round(). I chose to use round() because round() is the only method that actually round the input to the nearest integer while int() and math.trunc() would throw away the decimal parts. This may save some cycles of iteration to update the threshold sometime. And since each iteration search through almost all pixels in the image, if the image is very large, this can be slightly more efficient. 3. Dividing the regions: when dividing regions based on the threshold, we need to decided where to put pixels with values at threshold. We can put it in R1, R2, split it as 50:50, or just don't put it in any bin. Think about the algorithm and we can see that the averages are a little similar to the concept as "variances" from the "mean". We are trying to find a threshold so that the differences of R1 and R2 are about the same from the threshold. Therefore, if we consider it mathematically, putting the pixels at threshold into R1 or R2 are not that reasonable because they would skew the average towards the threshold rather than emphasize on the pixels far away from the threshold. And 50:50 may not be the most reasonable one because we are unsure the number of pixels in each region, if one region has few pixels and there are many pixels at threshold, a 50-50 split may result in over emphasizing the pixels at threshold when computing the average. However, all these methods will eventually let th converge to the actual threshold under the definition of this algorithm. I tested the algorithm using different grouping on pixel at threshold and they basically have same number of iterations. In this case, since leaving pixels=threshold out would require least iteration in the for loop computing averages, I decide to leave pixels=threshold out and strictly define R1 as greater than threshold and R2 as less than threshold. 4. Prevent infinite loops: I used a global variable to update the number of iteration in the while loop and set it as a condition in the while loop to force stop the program to prevent infinite looping. I set the force stop at 256 iterations which seems to be too large but also very safe for special cases.

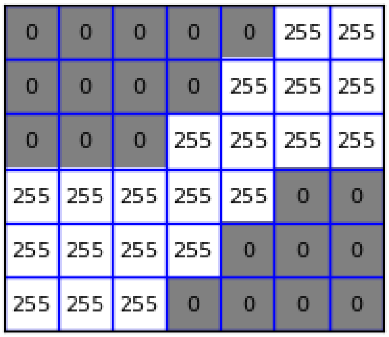
Advantages and Disadvantage of the algorithm vs. vtpeakpy: 1. One advantage of the algorithm is that unlike vtpeak, who only looks at peaks and minimums while may neglect the overall trend of the histogram, it determines the threshold based all pixels since it uses all pixels to compute the average. This may help it perform better on some images (such as mp.vx or nb.vx) whose information has been partially ignored by vtpeakpy. 2. The disadvantage of the algorithm is that it may require an even longer time to compute as it iterates through almost all pixels in every loop and it's likely that the while loop is entered for more than 3 times.

1.2.3 3.3 Small image tests

```
[219]: vd.dispsvx('testa.vx', 'testa_vits.vx', capt='results of vits on testa.vx')
```



0	10	20	30	40	50	60
10	20	30	40	50	60	70
20	30	40	50	60	70	80
90	80	70	60	50	40	30
80	70	60	50	40	30	30
70	60	50	40	30	20	10

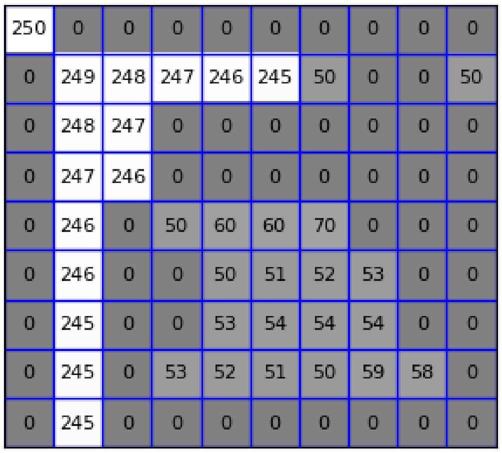


0	0	0	0	0	255	255
0	0	0	0	255	255	255
0	0	0	255	255	255	255
255	255	255	255	255	0	0
255	255	255	255	0	0	0
255	255	255	0	0	0	0

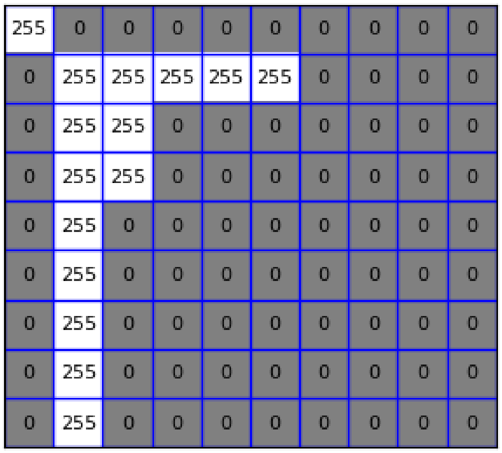
```
results of vits on testa.vx
<scaled size: (6 x 7) (6 x 7) >
```

This small image testa is just a general test to see whether the program can correctly find the average and use it as a threshold to clearly separate the pixels. The image used ascending numbers in the upper half and descending numbers in the lower half, the stepwise ascending/descending order can help us quickly locate an estimated average and see whether the separation is correct. As a result, we expect to see approximately the lower values on the upper left quadrant and the lower right quadrant as 0. After testing with the '-v' option, it returned th=46, and we can see that pixels with values higher than 46 are marked as 255, and pixels with values less than 46 are marked as 0, which is the same as our expectation.

```
[216]: vd.dispsvx('testb.vx', 'testb_vits.vx', capt='results of vits on testb.vx')
```



250	0	0	0	0	0	0	0	0	0
0	249	248	247	246	245	50	0	0	50
0	248	247	0	0	0	0	0	0	0
0	247	246	0	0	0	0	0	0	0
0	246	0	50	60	60	70	0	0	0
0	246	0	0	50	51	52	53	0	0
0	245	0	0	53	54	54	54	0	0
0	245	0	53	52	51	50	59	58	0
0	245	0	0	0	0	0	0	0	0

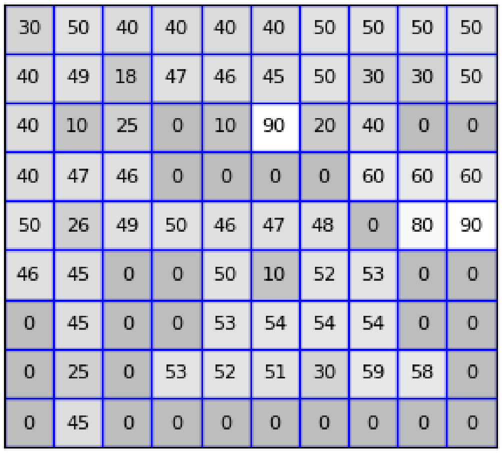


255	0	0	0	0	0	0	0	0	0
0	255	255	255	255	255	0	0	0	0
0	255	255	0	0	0	0	0	0	0
0	255	255	0	0	0	0	0	0	0
0	255	0	0	0	0	0	0	0	0
0	255	0	0	0	0	0	0	0	0
0	255	0	0	0	0	0	0	0	0
0	255	0	0	0	0	0	0	0	0
0	255	0	0	0	0	0	0	0	0

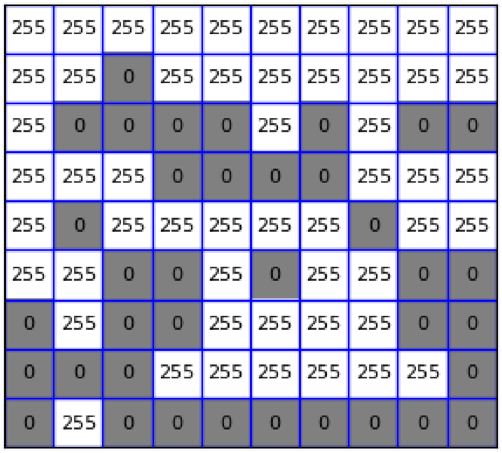
```
results of vits on testb.vx
<scaled size: (9 x 10) (9 x 10) >
```

This small image testb is used to test cases such as whether disconnected high/low value pixels (such as the 250 at [0, 0] and the 50 at [2, 9]) can be identified correctly and whether low/high value connect to a high/low region (such as the 50 at [2, 6]) can be identified correctly. From the results shown on the right, we can see that the disconnected pixels or pixels connected to a different region are correctly identified and thresholded, which matches our expectations as well.

```
[218]: vd.dispsvx('testc.vx', 'testc_vits.vx', capt='results of vits on testc.vx')
```



30	50	40	40	40	40	50	50	50	50
40	49	18	47	46	45	50	30	30	50
40	10	25	0	10	90	20	40	0	0
40	47	46	0	0	0	0	60	60	60
50	26	49	50	46	47	48	0	80	90
46	45	0	0	50	10	52	53	0	0
0	45	0	0	53	54	54	54	0	0
0	25	0	53	52	51	30	59	58	0
0	45	0	0	0	0	0	0	0	0



255	255	255	255	255	255	255	255	255	255
255	255	0	255	255	255	255	255	255	255
255	0	0	0	0	255	0	255	0	0
255	255	255	0	0	0	0	255	255	255
255	0	255	255	255	255	255	0	255	255
255	255	0	0	255	0	255	255	0	0
0	255	0	0	255	255	255	255	0	0
0	0	0	255	255	255	255	255	255	0
0	255	0	0	0	0	0	0	0	0

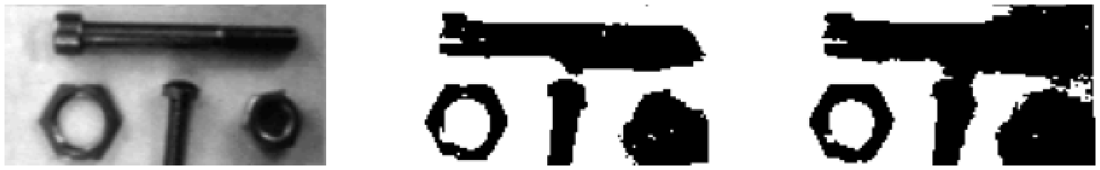
```
results of vits on testc.vx
<scaled size: (9 x 10) (9 x 10) >
```

Small image testc is to test whether pixels that should does not belong to the region but is wrapped (such as low value pixel surrounded by high value pixels) can be identified. Specifically, we used fully connected pixels such as 10 at [5, 6], 26 at [1, 5], partially connected pixels such as 18 at [2, 1], 10 at [1, 2], 20 at [6, 2]. From the result shown on the right, we can see these pixels are all successfully identified.

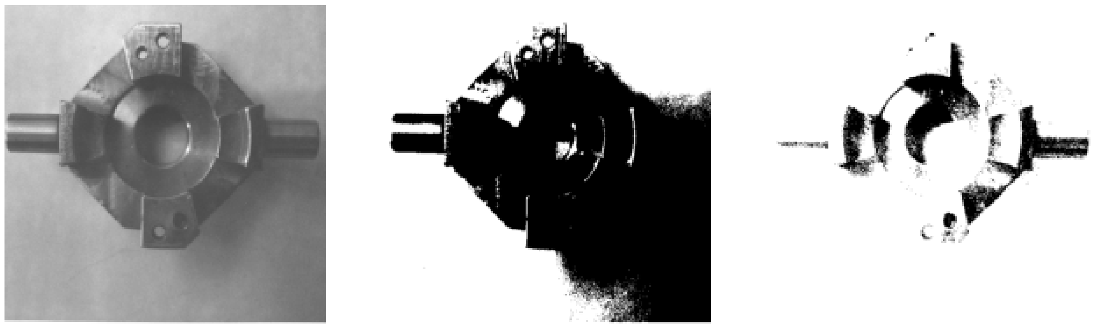
1.2.4 3.4 Large Image tests

```
[224]: exec(vx.vxsh('python3 vits.py if=nb.vx of=nb_vits.vx'))
       vd.dispmvx('nb.vx','nb_vits.vx', 'nb_pk2.vx', capt='result of vits on nb.vx␣
        ↪compared to results of vtpeakpy\n\n')

       exec(vx.vxsh('python3 vits.py if=mp.vx of=mp_vits.vx'))
       vd.dispmvx('mp.vx','mp_vits.vx', 'mp_pk2.vx',  capt='result of vits on mp.vx␣
        ↪compared to results of vtpeakpy\n\n')

       exec(vx.vxsh('python3 vits.py if=map of=map_vits'))
       vd.dispmvx('map','map_vits', 'map_pk2', capt='result of vits on map compared to␣
        ↪results of vtpeakpy\n\n')

       exec(vx.vxsh('python3 vits.py if=facsimile of=facsimile_vits'))
```

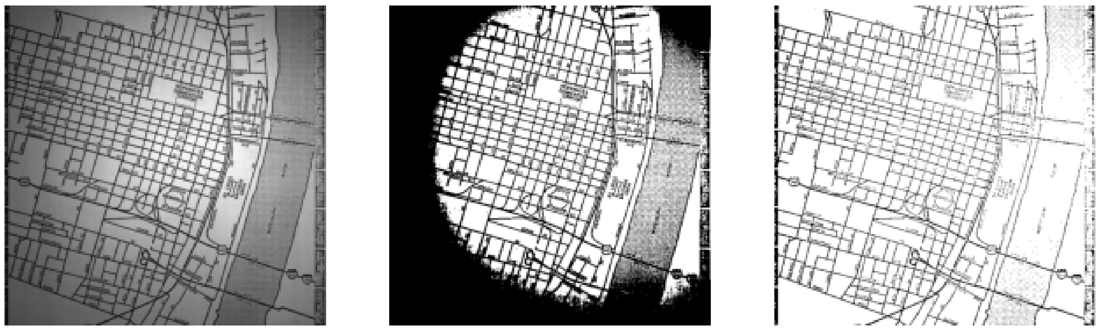
```
vd.dispmvx('facsimile','facsimile_vits', 'facsimile_pk2', capt='result of vits␣
  ↪on facsimile compared to results of vtpeakpy\n\n')

exec(vx.vxsh('python3 vits.py if=shtl.vx of=shtl_vits.vx'))
vd.dispmvx('shtl.vx','shtl_vits.vx', 'shtl_pk2.vx', capt='result of vits on␣
  ↪shtl.vx compared to results of vtpeakpy\n\n')
```



```
result of vits on nb.vx compared to results of vtpeakpy


<scaled size: (64 x 128) (64 x 128) (64 x 128) >
```

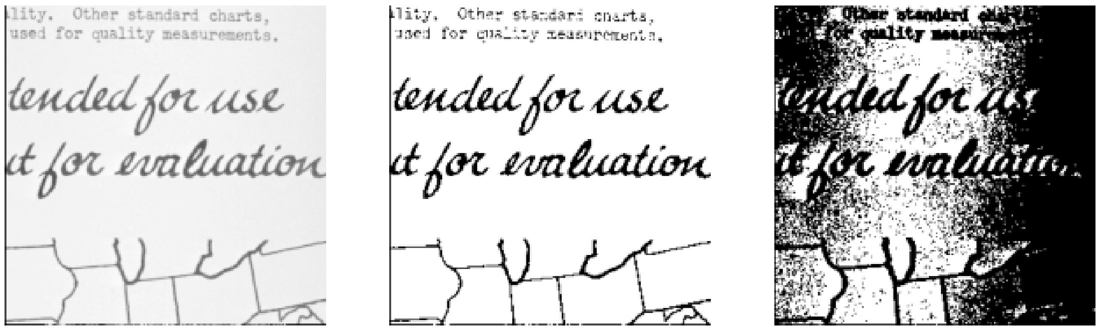


```
result of vits on mp.vx compared to results of vtpeakpy


<scaled size: (512 x 512) (512 x 512) (512 x 512) >
```



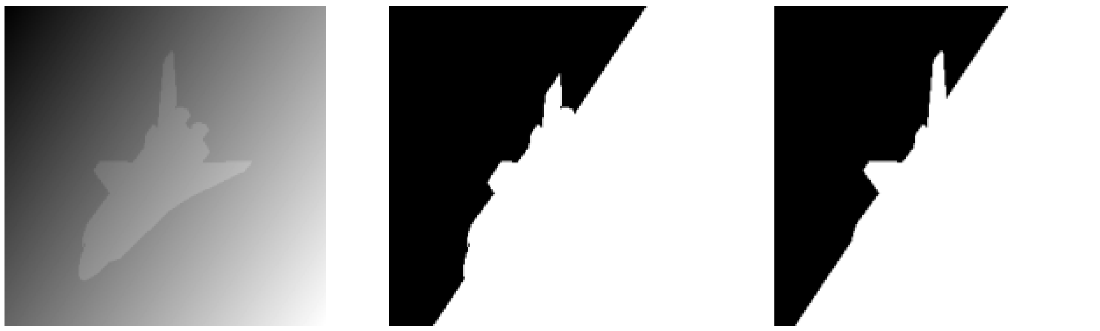
`result of vits on map compared to results of vtpeakpy`

`<scaled size: (512 x 512) (512 x 512) (512 x 512) >`



`result of vits on facsimile compared to results of vtpeakpy`

`<scaled size: (256 x 256) (256 x 256) (256 x 256) >`



```
result of vits on shtl.vx compared to results of vtpeakpy
```

```
<scaled size: (256 x 256) (256 x 256) (256 x 256) >
```

As the large image test results show above, vits can segment nb.vx and facsimile better than vtpeak. Vits can more clearly recognize the shadings around the objects in nb.vx and consider them not as part of the object as we expected. This is likely because vtpeak cannot deal with spike-shaped peaks in the histogram very well as we discussed in 2.4.3 and 2.4.4, the minimum it finds would just be the zeros between the spikes. Vits, on the other hand, takes the average, so it can give a better threshold to represent the "average"(not in the mathematical sense) of the whole image when the shading and the object has little color variance. In facsimile, vits has a nice and clean output compared to the best solution(with optimal d) that vtppeak can provide. Definitely as discussed before in 2.4.3, vtpeak cannot properly handle single-spike histogram, and vits wouldn't have this problem because it uses the average. Therefore, a better output from vits should be expected and is in fact achieved.

However, vits is not always better then vtpeak. In map, vits' result is basically the same as vtpeak's result using default d (not the optimal result of vtpeak). This is because the shading in the map image is heavy, in other words, the noise is heavy. Since the shaded area would generally be darker (with lower pixel values), vtpeak can successfully ignore it because it only cares about the peak whereas vits, who takes the average, may have put more emphasis on the noise than expected, thus providing an unreasonable threshold.

There are also certain images that both vits and vtpeak are not ideal to use. In mp.vx and shtl.vx, neither vits nor vtpeak provided an well segmented output. In mp.vx, the main problem is the metal reflection on the object and the shading within the object (like shadow in the hole in the middle) and the shading connected to the object (lower right corner of the image). Since the color difference between the shading and the object is similar, it is hard to be separated from the histogram since they are likely to have the same index. And the reflection has so high values and it is so different from the rest of the object that it would be considered as "noise" by both thresholding methods and be put to 255. In shtl.vx, we can see that the vits function has a slightly better result as it can show a little more upper half of the shuttle, however, the results are still not idea since vits also cuts out the lower half of the shuttle. It is to say, that in an image like shuttle, where the color variation in background is much higher than the color variation in the object, the background kind of "becomes the foreground" in these image processing methods. Or we can just say that the noise from the background is too heavy that has introduced large bias in these two algorithms.

Overall, we can say that vits may compensate the disadvantages of vtpeak as it can deal with images with spikes or single peak well. However, both have the problem dealing with heavy noise in the background or foreground such as shtl.vx and nb.vx.

1.3 4.Region Growing (vgrow)

1.3.1 4.1 The program vgrow

[5]:
```python
#!/usr/bin/env python
""" vgrow: a python region labeling function

"""
```

```python
import sys
import numpy as np
import math
from v4 import vx
from PIL import Image

# System's default recursion limitation is 1000,
# This is intended for some full-size images.
sys.setrecursionlimit(15000)

# parse command arguments
of=' '
vxif=' '
vargs = vx.vaparse( "if= of= r= -p -v -")

if '-' in vargs:
    print ("vgrow program")
    print ("if= input file")
    print ("of= output file")
    print ("r= integer that sets the range of the region pixels")
    print ("-p (pixelValue) set the label of the region to the first pivel␣
 ↪value")
    print ("-v  (verbose) print region range information")
    exit()

# set initial range value to default = 10
regionRange = 10

# Set range= r if parameter is specified
if 'r' in vargs:
    r_in = math.trunc(float(vargs['r']))
    if r_in <= 0 or r_in >= 128: # print error message if the given th is␣
 ↪invalid
        print("error: r= must be integer between 0 and 128" )
        exit(1)
    regionRange = round(r_in)

# check for input/output file name argument
for arg in ('if', 'of'):
    if arg not in vargs:
        print ( 'vgrow error: missing required parameter %s' % arg)
        exit(1)

# read in input image
inimage = vx.Vx(vargs['if'])
im = inimage.i
```

```
# check if input image has correct datatype to apply operations
if im.dtype != 'uint8' :
    print ("error: image not byte type" )
    exit(1)

# check for -p -v options
optp = '-p' in vargs
optv = '-v' in vargs

# Function setlabel: recursively called to label a unlabeled region whose␣
 ↪pixels are within the range of "range"
#                    of the first pixel value of the region
# input: index of current object pixel, x (for row), y (for column), and L for␣
 ↪label number
def setlabel (x, y, label):
    global im, tm, regionRange, first_pxl, color_label
    im[y,x] = label
    # weight red, green, and blue channels differently for color
    # label number "label" is used to change the color of different sections
    color_label[y, x] = [label*120%256, label*160%256, label*200%256]
    # label the neighbor pixel if it is non-zero, not labeled, and within the␣
 ↪range
    # 4 connected foreground 8 connected background
    if tm[y+2,x+1] > 0 and im[y+1,x] == 0 and (abs(int(tm[y+2,x+1]) -␣
 ↪first_pxl) < regionRange):
        setlabel(x, y+1, label)
    if tm[y,x+1] > 0 and im[y-1,x] == 0 and (abs(int(tm[y,x+1]) - first_pxl) <␣
 ↪regionRange):
        setlabel(x, y-1, label)
    if tm[y+1,x] > 0 and im[y,x-1] == 0 and (abs(int(tm[y+1,x]) - first_pxl) <␣
 ↪regionRange):
        setlabel(x-1, y, label)
    if tm[y+1,x+2] > 0 and im[y,x+1] == 0 and (abs(int(tm[y+1,x+2]) -␣
 ↪first_pxl) < regionRange):
        setlabel(x+1, y, label)

# create embedded copy of input image
tmimage = vx.Vx(inimage)
tmimage.embedim((1,1,1,1))
tm = tmimage.i

# generate a unique name for the colored image file
img_name = "colored_"
s_length = len(vargs['if'])
s_length -= 3
```

```
img_name += vargs['if'][:s_length] # get rid of the input file extenstion
img_name += "_vgrow.png"

# clear the output image
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        im[y, x] = 0



# Initialize a same-size array as im but with three channels filled with 0
# To store the RGB value for each pixel
color_label = np.zeros([im.shape[0], im.shape[1], 3], dtype=np.uint8)

# set first pixel value default = 0
first_pxl = 0
# set label to default = 1
label = 1
# examine through every pixel
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        if (im[y, x] == 0 and tm[y+1, x+1] != 0): ## check if the pixel is an␣
 ↪unlabeled object pixel
            # for range check
            first_pxl = round(tm[y+1, x+1])
            if (optp): ## label currect region as the regions's first pixel's␣
 ↪value
                label = tm[y+1, x+1]
                setlabel(x, y, first_pxl)
            else: ## label the current region using sequential numbering
                setlabel(x, y, label)
                # set label for the next objext (restart from 0 if all 255␣
 ↪numbering is used)
                if (label == 255):
                    label = 1
                else:
                    label = label + 1

# generate png RGB image from array 'color_label'
color = Image.fromarray(color_label)
color.save(img_name)

if optv:
    print(regionRange)

inimage.write(vargs['of'])
```

1.3.2 4.2 Program Discussion

In Section 3, our goal is to generate an region growing algorithm in python, named as vgrow.py. In the coding process, we used cclabel from lab2 as a template. I will introduce the program (Checking Command arguments & Algorithm description section), discuss some considerations when writing the algorithm and the advantages of the method.

Checking Command Argument: 1. if= byte-type input binary image file name; 2. of= output binary image file name; 3. r= range: defines the maximum absolute difference between the pixels and the first pixel value within a region. Automatically trimmed to integer if input is not integer. Set to user input if the input is within 0-128, print error message to stdout if not within the range. If there is no input, defaultly set to 10. 4. -p: SetToPixel: user option to set the region label to the value of the first pixel in the region. Defaultly using sequential numbering starting from 1. 5. -v: user option to print the range applied to the stdout.

Algorithm description: 1. The program clears the copy of input image (makes it zero so later we know if the pixel is labeled or not) 2. The program initialize two global variables first_pxl and label. first_pxl is used to store the value of the first pixel, label is used to store the current label used before updating in the for loop. 3. The program then enters the for loop to search through each pixel, check if the pixel in tm is non-zero (so it is an foreground pixel) and in im is zero (so it's non-labeled). It will move to the next pixel if the pixel is already labeled or the pixel is 0 (we consider 0 as a background pixel). Otherwise, it will move to step 4-5 4. When the pixel is an unlabeled foreground pixel (note that this is the first pixel in the current region), the program will update the the first_pxl to tm[y+1, x+1] which is the current region's first pixel's value. 5. The program then checks if the -p option is enabled by the user: if -p is asked, it will set label to the current pixel value (first pixel value), and call setlabel to label the current region; if not asked, it will use the default label (sequential numbering), call setlabel to label the current region, and check if sequential numbering has reached the edge (255) and start over from 1 if it has reached the edge. 6. In the setlabel function, the program set the pixel at the input indexes (im[y, x]) to the input label and checks if any 4-connected neighbors are 1) within non-zero 2) unlabeled 3) within the range. If the neighbor meets all three conditions above, the program will call setlabel on that neighbor (so this becomes recursive and ends when the region is completely marked) 7. Now the labeled image is completed, the program will write the new image to the output file specified.

Discussion On Program Implementation: 1. round & int: Unlike vits, whether to use int() casting or round matters in vgrow. The first problem is at reading in vargs['r']. The range is read in as a string, so we have to cast it to int or float first. However, since the user have a manual input, I want to assume that the user who put in a float value wants to set pixel range strictly less than the range. In this case, round(float(vars['r'])) would not satisfy the expectation. However, int() casting would report error if the input is not an integer, so it would prevent user from inputting an integer (we all know that integer is expected because pixel values are supposed to be integer and therefore the range should be as well, but just in case some users does that). Therefore, as a solution, I imported math package and used math.trunc(float(vargs['r'])) which should work the best here. Also, when checking if the pixel is within the range, we want to ensure the pixel difference to be less than the range. However, if we use round, and there is decimal places, there is a chance that the difference would be round up to equal to the set range, causing an error (and again, the pixel value itself shouldn't be float at all). Theoretically, the pixel difference should never be a float at this point. 2. We still need to reset the default recursion limit as because the default recursion depth (1000) is not enough for the full size image. 3. The sequence of abs(int(tm[] - first_pxl) matters because first_pxl is read out from the im which has data type of ‘uint8’. Without int() casting it would result in “fatal error: stack overflow”. 4. One thing I considered was where to check the optp. We can check it before the for loop to search through each pixel, or checking if the pixel is unlabeled foreground, or after updating the first-pxl. All three would work except for code redundancy, which is why I decided to keep current style. 5. One bug I encountered is that I could not read in argument “r=”, and in the end I realized it’s because in python’s implementation reading the argument, it would not consider ‘-’ as part of the argument. (This part will be further explained in small test c). 6. Another interesting thing is that colored image does not have much meaning in this case because the -p option to offer enough color variation in grayscale already.

Advantage And Disadvantage of the Program:

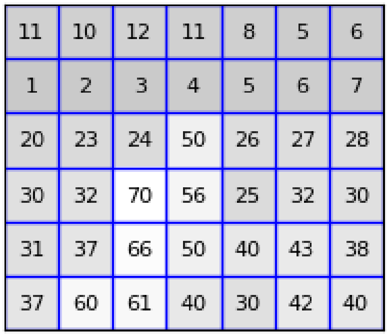
One advantage of the program is that it become less restricted on the input images compared to cclabel which required the user to set the background to black. vgrow is able to segment the regions even when the image background is not standardized. Another advantage is that vgrow can automatically produce color variation among regions which is useful in object segmentation whereas cclabel’s output is just an image with regions sequentially numbered and we have to write an extra part to produce a colored version to distinguish the regions.

One disadvantage of the program is that it requires the user to manually try different r for multiple times to get the optimal results. If the user has little knowledge of the image, the r chosen can produce terrible results.

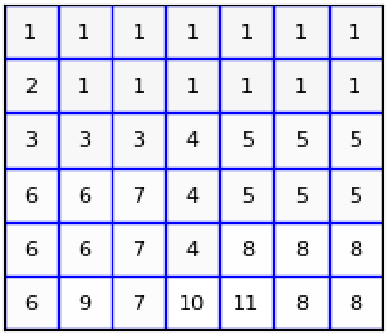
1.3.3 4.3 Small image tests

In small image tests, I mainly created three small images to test the program performance without r argument and -p options and then test the program function with r argument and -p option. Below are the results of the three small image tests.

[324]:
```
## Small Image Test A:
vd.dispsvx('test4a.vx', 'test4a_vgrow.vx')
```



11	10	12	11	8	5	6
1	2	3	4	5	6	7
20	23	24	50	26	27	28
30	32	70	56	25	32	30
31	37	66	50	40	43	38
37	60	61	40	30	42	40



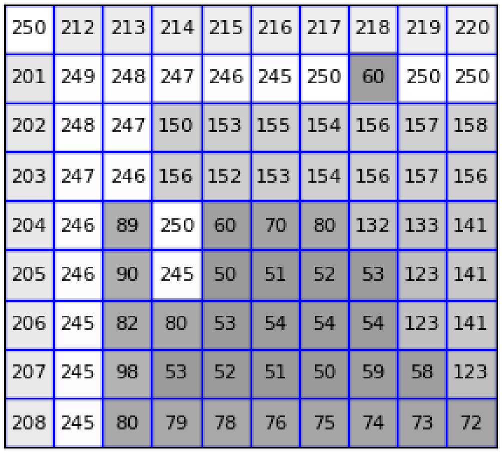
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	3	3	4	5	5	5
6	6	7	4	5	5	5
6	6	7	4	8	8	8
6	9	7	10	11	8	8

```
<scaled size: (6 x 7) (6 x 7) >
```

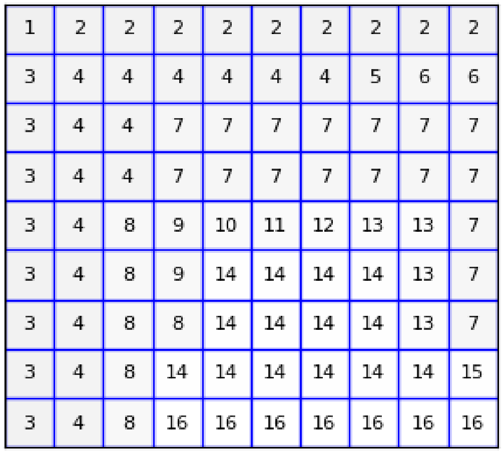
Small image test a aims to check: 1. Whether the labeling on connected pixels within the range of the first pixel is correct 2. Whether it properly excludes connected pixel at the boundary of the range of the first pixel, here boundary is 10 (such as the 50, 40 at [4, 3], [4, 4], 40, 30 at [5, 3], [5, 4], or 20, 30 at [0, 2], [0, 3], or 11, 1 at [0, 0], [0, 1]). 3. Whether it properly excludes connected pixel within range of neighbor pixel but not the first pixel (such as 60 and 61 at [5, 1], [5, 2]) 5. Whether sequential labeling is properly conducted.

From the result on the right, we can see that the sequential labeling is correct; pixel with differences 10 from the first pixel in the region is successfully excluded as a separate region; pixels within the range of neighbor but not the first pixel are excluded as a separate region as well. Therefore, the program can operate correctly when -v option is not enabled and r= is not specified.

```
[325]: # Small Image Test B:
       vd.dispsvx('test4b.vx', 'test4b_vgrow.vx')
```



250	212	213	214	215	216	217	218	219	220
201	249	248	247	246	245	250	60	250	250
202	248	247	150	153	155	154	156	157	158
203	247	246	156	152	153	154	156	157	156
204	246	89	250	60	70	80	132	133	141
205	246	90	245	50	51	52	53	123	141
206	245	82	80	53	54	54	54	123	141
207	245	98	53	52	51	50	59	58	123
208	245	80	79	78	76	75	74	73	72



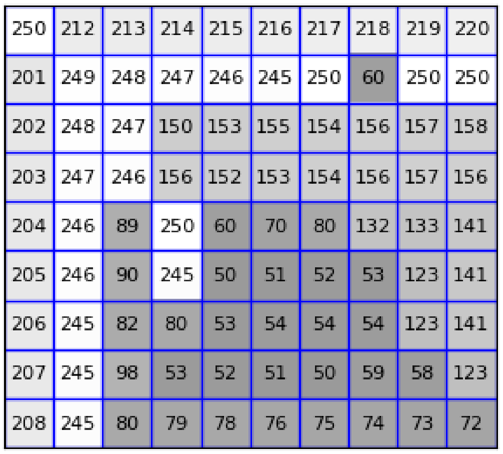
1	2	2	2	2	2	2	2	2	2
3	4	4	4	4	4	4	5	6	6
3	4	4	7	7	7	7	7	7	7
3	4	4	7	7	7	7	7	7	7
3	4	8	9	10	11	12	13	13	7
3	4	8	9	14	14	14	14	13	7
3	4	8	8	14	14	14	14	13	7
3	4	8	14	14	14	14	14	14	15
3	4	8	16	16	16	16	16	16	16

```
<scaled size: (9 x 10) (9 x 10) >
```

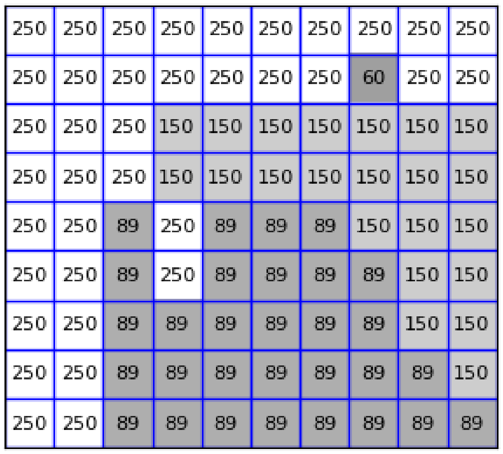
Small image test b aims to further test: 1. Whether the algorithm properly excludes the pixels within the range and that is 8-connected neighbor but not 4-connected neighbors (such as 250, 249 at [0, 0], [1, 1], 246, 250 at [3, 2], [4, 3]) 2. Whether it properly includes pixels within the range of the first pixel but not in the range of the neighboring pixel (such as 123, 141 at [6, 8], [6, 9] or at [7, 8], [7, 9] or at [7, 9], [8,9], or 82, 98 at [7, 2], [8, 2])

As a result from on the right, we can see that the program can correctly exclude the 8-connected neighbors and includes pixel within the range of the first pixel even when it's not within the range of neighboring pixels. So the programs operates correctly when -p and r= is not enabled.

```
[326]: # Small Image Test C:
       exec(vx.vxsh('python3 vgrow.py r=50 -p if=test4b.vx of=test4b_vgrow2.vx'))
       vd.dispsvx('test4b.vx', 'test4b_vgrow2.vx')
```



250	212	213	214	215	216	217	218	219	220
201	249	248	247	246	245	250	60	250	250
202	248	247	150	153	155	154	156	157	158
203	247	246	156	152	153	154	156	157	156
204	246	89	250	60	70	80	132	133	141
205	246	90	245	50	51	52	53	123	141
206	245	82	80	53	54	54	54	123	141
207	245	98	53	52	51	50	59	58	123
208	245	80	79	78	76	75	74	73	72



250	250	250	250	250	250	250	250	250	250
250	250	250	250	250	250	250	60	250	250
250	250	250	150	150	150	150	150	150	150
250	250	250	150	150	150	150	150	150	150
250	250	89	250	89	89	89	150	150	150
250	250	89	250	89	89	89	89	150	150
250	250	89	89	89	89	89	89	150	150
250	250	89	89	89	89	89	89	89	150
250	250	89	89	89	89	89	89	89	89

```
<scaled size: (9 x 10) (9 x 10) >
```

Small image test c aims to test: 1. Whether the -p option operates correctly; 2. Whether the r= parameter operates correctly.

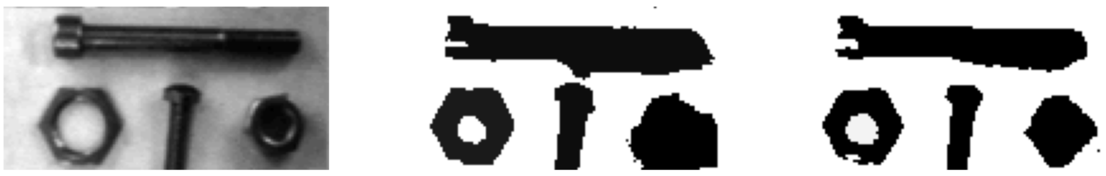
From the output on the right, we now see that when r is set to 50, many regions that are previously separated but within would be within the range of 50 now merged. For example, the first row and first column would not be considered as the same region as 250 at [0, 0] and 249 at [1, 1]. The labeling is also the same as expected output as each region is labeled by the first pixel's value (first: first to appear in the region when read from up to down, from left to right, while up and down has higher priority). As result we know that -p and r= is working correctly now.

This is actually where I tested and found that my r= argument wasn't read and started debug. Before this correct output, I got a output not as expectation, the first column and the first row are still separated regions than the 250 at [0, 0] when I set r=50. Though the -p worked as expectation because the labeling was using first pixel's value, the r argument was dysfunctional. I then went into the vgrow code and used print() method to print out regionRange at different steps (before reading in r, after reading in r) and also checked whether the r is found in the argument (optr= 'r=' in vargs, print(optr)) and I realized that the r= argument was not read at all. After a long time of debugging, I realized that the problem is that python only read 'r' as the argument rather than 'r='. So I revised this part and it operates correctly now as shown above.

1.3.4 4.4 Large Image tests

In large image tests, we mainly take a look at the nb.vx and the shtl.vx. Below, we will show the results on testing vgrow on the two full-size images with different r values and -p option enabled.

[328]:
```
# Large Image Test With nb.vx
exec(vx.vxsh('python3 vgrow.py r=67 -p if=nb.vx of=nb_vgrow.vx'))
exec(vx.vxsh('python3 vgrow.py r=83 -p if=nb.vx of=nb_vgrow_r2.vx'))
vd.dispmvx('nb.vx','nb_vgrow.vx', 'nb_vgrow_r2.vx', size = 2, capt='result of␣
↪vgrow on nb.vx at r=67, r=83')
```

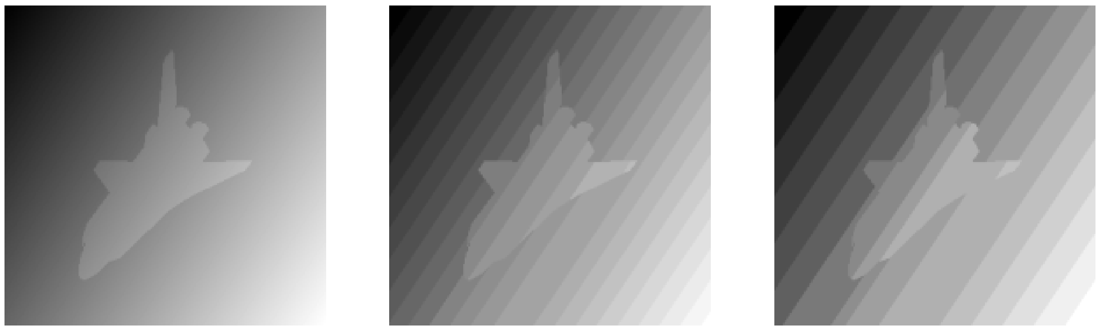


```
result of vgrow on nb.vx at r=67, r=83
<scaled size: (64 x 128) (64 x 128) (64 x 128) >
```

From the output image from vgrow as shown above, we can see that images within r=67 to r=83 all produce pretty nice segmentation that is even better than vits. It's a little hard to tell which r is the best though. In the range(67, 83), with r closer to 67, the algorithm will misrecognize part of the shading near the top object and the lower right object as part of the object as well (overcover). However, with r closer to 83, the algorithm will undercover the highlights on the top object, the lower left object and some edges of the lower right object. Both segmentation is not exactly perfect, but they are already give very nice outputs.

Personally, I am leaning towards that r=80 to 83 (there is no observable difference between the output with these r) is more ideal then r=67 since they have less area misidentified (a higher Jaccard index).

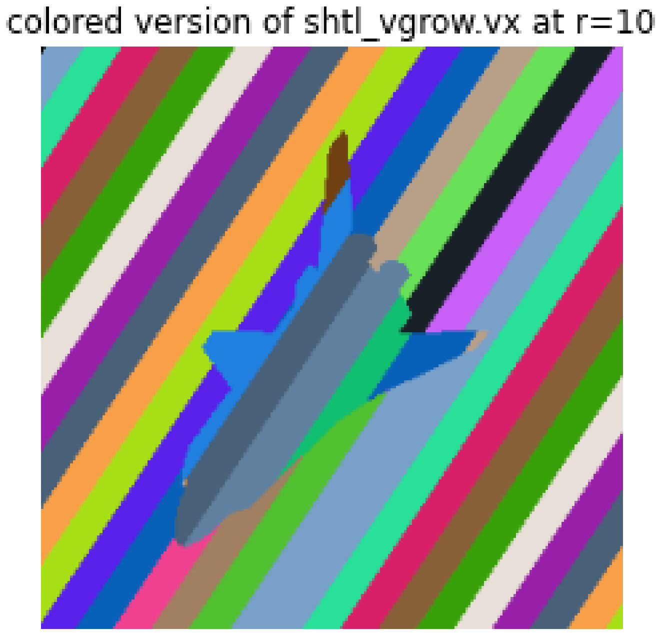
[329]:
```
# Large Image Test With shtl.vx
exec(vx.vxsh('python3 vgrow.py r=15 -p if=shtl.vx of=shtl_vgrow_r2.vx'))
exec(vx.vxsh('python3 vgrow.py r=10 -p if=shtl.vx of=shtl_vgrow.vx'))
vd.dispmvx('shtl.vx','shtl_vgrow.vx', 'shtl_vgrow_r2.vx', capt= 'result of␣
↪vgrow on shtl.vx at r=10, r=15\n\n')

color_shtl = mpimg.imread('colored_shtl_vgrow.png')
plt.figure(figsize = (4, 4))
imgplot = plt.imshow(color_shtl)
plt.axis('off')
plt.title("colored version of shtl_vgrow.vx at r=10")
```



```
result of vgrow on shtl.vx at r=10, r=15
```

```
<scaled size: (256 x 256) (256 x 256) (256 x 256) >
```

```
[329]: Text(0.5, 1.0, 'colored version of shtl_vgrow.vx at r=10')
```

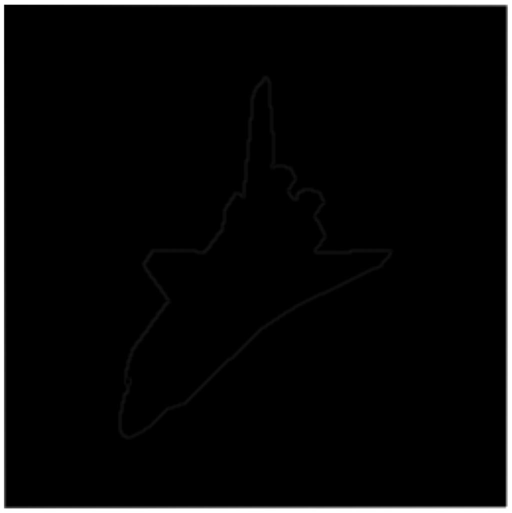
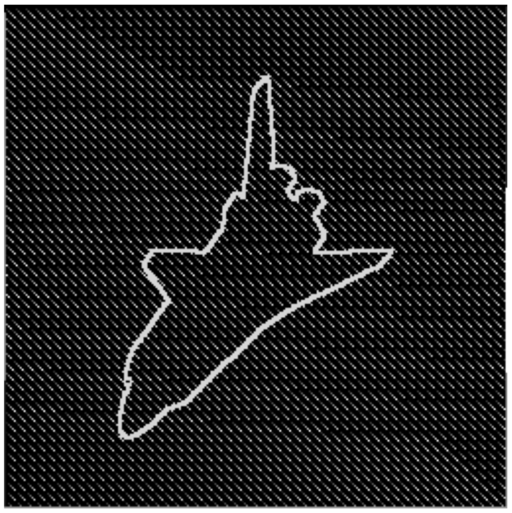


From the output image from vgrow on shtl as shown above, we find that when r is small, the closer it is to 0, the image is more like the original image, at approximately r = 10, the image is fully segmented (not merged with background), and for r > 10, such as r=15, the image would have part of the wing merged with the background. Therefore, the best r I used is r=10.

However, the fact is that vgrow does not segment the shuttle very ideally as well because the vgrow does not only segmented the object from the background, it also segmented the object itself, which is not desired.

4.5 Edge Image tests In this subsection, we used sobel, which is an edge operator, to detect the edge of shtl.vx first and then apply vgrow without -p on it. Below I will show the results of the edge detected image of shtl.vx and the result of vgrow on the edge detected image and discuss the results.

```
[323]: exec(vx.vxsh('python3 vgrow.py r=25 if=edge.vx of=edge_vgrow.vx'))
       vd.dispmvx('edge.vx','edge_vgrow.vx', size = 2, capt='result of vgrow on edge.
        ↪vx\n\n')
```

```
result of vgrow on edge.vx
```

```
<scaled size: (256 x 256) (256 x 256) >
```

[ ]:
```
# canny separable matrix
c = [1, 1]
c_p = [1, -1]

mask_x = np.empty((2, 2, 2), int)
mask_y = np.empty((2, 2, 2), int)
mask_z = np.empty((2, 2, 2), int)

for i in range(2):
    for j in range(2):
        for k in range(2):
            mask_x[i][j][k] = c_p[i] * c[j] * c[k]
            mask_y[i][j][k] = c[i] * c_p[j] * c[k]
            mask_z[i][j][k] = c[i] * c[j] * c_p[k]
print (mask_x)
print (mask_y)
print (mask_z)
```

From the result shown above, we can see that the original output of sobel is almost completely black with dark gray outline of the shuttle. However, this is a little hard for naked eyes to distinguish because the pixel value is too low. If we apply vgrow after the edge detection, it can now show clear outline of the object with white arrows in the background. Since we did not use -p option in the command, the algorithm will use sequential labeling, the entire outline is one region with the same pixel value, the white arrows in the background is created because the background of edge.vx is solely made up by zeros and ones, and therefore, they are detected as different regions.

As a conclusion, we can see that pre-processing the image may help improve the performance of the algorithm.