Lab 2
Binary Image Processing

Jonathan Ananda Nusantara
jan265

Section 4:

In section 4, the task is to create a python program called boundpy. The objective is that the program will get a binary image as an input. This binary image will consist of the background, where the pixel value is zero, and a foreground, where the pixel value is between 1 to 255. In the output of the image, we would like to highlight the foreground in comparison to the background, and also clearly show the boundary.

In order achieve this goal of highlighting the object and showing the boundary of the foreground to the background, we would define three pixel classes or values. The first class is the background, where the pixel value is 0. The second class is the interior, where the pixel value is 128. The third and last class is the boundary, where the pixel value is 255. The program will iterate through each pixel of the image and determine which class each pixel belongs to. Then, it will change the value of the pixel at that location if needed. The final result and output of the program will be the image with the highlighted interior and a clearly visible boundary of the object.

This program is built from a template called vtempy. This template has the mechanism of how input and output files are handled when the command is called and flags are passed in Terminal. The template also provides the mechanism of how the input file is loaded into variable im. Then, it creates another image variable called tm, but also embed one pixel surrounding the current image. This is basically adding a row or column to the side of the original image. This is because this program requires viewing the pixel at each coordinate/location and comparing it with the surrounding. In order to simply compare with the neighboring pixel when we are on the edge of the image (min or max value of x or y), we added pixels around it that acts like a background. Finally, this template handles how the image is being output into the defined output file name.

A short algorithm is created to iterate through the pixels and categorizing them into the corresponding class and value. A for loop that iterates through the x axis range of input image is created inside another for loop that iterates through the y axis of input image (refer to Figure 1 below). This algorithm is designed using the 8-connected foreground and 4-connected background connectivity. In each index of the loop, we are checking the pixel value of our tm image, which is a copy of our original image with the added 1-pixel border. When checking the current index of the tm image, the index is converted from [y, x] to [y+1, x+1]. This is because when we embed the border in the tm image, the current origin [0, 0] pixel from the input image is now [1, 1], so there needs to be this conversion.

So in each [y+1, x+1] index of tm, we check if it the current pixel has a value higher than 0, which means that it is a foreground. If this is not satisfied, then the output im image at that pixel should stay 0 as background, and nothing should be changed. If it is a foreground, the next condition is to check if any of the top, bottom, left, or right neighbor pixel is a background (pixel value = 0). If this is satisfied, then this means that the current pixel is a boundary between foreground and background, and the current pixel at image im is changed to have value 255. If this is not satisfied, this pixel is an interior and the current pixel at image im is changed to have value 128. At the end of the code, the image im is being write to the output file name passed by the user.

Source code for Section 4:

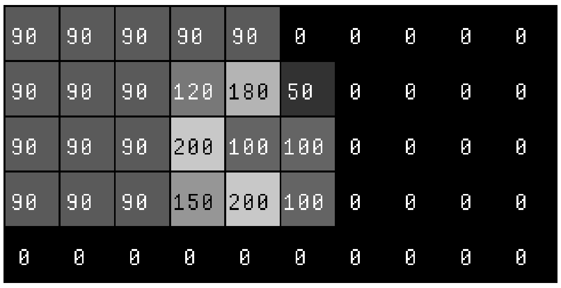
/home/jan265/lab2/boundpy
Page 1 of 1 **Wed 30 Sep 2020 12:07:36 PM EDT**

```python
#!/usr/bin/env python
""" boundy a python bound detection program
"""

import sys
from numpy  import *
from v4 import vx

of=' '
vxif=' '
clist = vx.vxparse(sys.argv,  "if= of= -v  - ")
exec (clist )

if 'OPT' in locals():
   print ("boundpy program")
   print ("if= input file")
   print ("of= output file")
   print ("[-v] verbose mode")
   exit(0)

if 'OPTv' in locals():
   optv=1
else:
   optv=0

inimage = vx.Vx( vxif )
im = inimage.i
tmimage = vx.Vx( inimage )
tmimage.embedim((1,1,1,1))
tm = tmimage.i

for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        # 8 connected foreground 4 connected background
        if tm[y+1,x+1] > 0: # If current pixel is foreground. +1 because of embed
            if tm[y+1,x+2] == 0 or tm[y+1,x] == 0 or tm[y,x+1] == 0 or tm[y+2,x+1] == 0: # If one of surrounding pixel is 0
                im[y,x] = 255 # Set pixel to 255 as it is boundary
            else: # If not a boundary pixel
                im[y,x] = 128 # Set pixel to 128 as it is interior

if optv:
   print (im)

inimage.write( of)
```

- 1 -

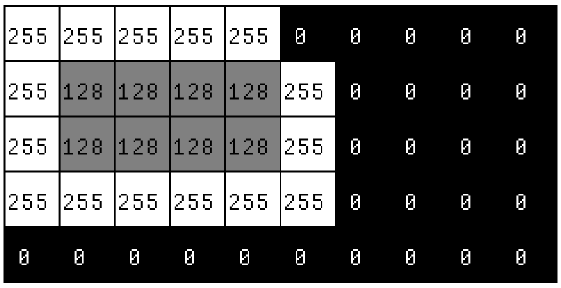
Figure 1 - The source code for Section 4.

This boundpy python program is first tested on a small image, named smallimedgesquare.vx, that was created to test the performance of boundpy. This small image consists of an object of various pixel values, that lies on the edge of the image (see Figure 2). It is also designed to have a a zero pixel value at the edge of object to show how this program has an 8-connected foreground and 4-connected background connectivity.



90	90	90	90	90	0	0	0	0	0
90	90	90	120	180	50	0	0	0	0
90	90	90	200	100	100	0	0	0	0
90	90	90	150	200	100	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 2 – input small image for the boundpy program.



255	255	255	255	255	0	0	0	0	0
255	128	128	128	128	255	0	0	0	0
255	128	128	128	128	255	0	0	0	0
255	255	255	255	255	255	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 3 – output of the small image from boundpy program.

The program will iterate starting from the top left corner, which is the [0, 0] pixel, then starts to move through the x axis (see Figure 2). It will find that the first pixel [0,0] detected is a has pixel value 90. It will also detect that this pixel is a boundary. This is because we are checking the neighboring pixels using our tm image, which has a 1-pixel border of background around it. So, this pixel would have a background pixel on its left and top, and this pixel is then a boundary. Thus, at the output image im, the index value is changed to 255 for boundary pixel (see Figure 3). On to the next pixel on its right, it will also find that the top neighbor is a background, so it is also a boundary. As it iterates, it will find that all foreground in this row is boundary. Half-way through the row, it will only detect the background with pixel value 0 and no change is made on the output image.

Going down to the next row, it will find that the first pixel detected on that row is a boundary. On to the next pixel on its right, the first interior pixel is detected, as the current pixel is a foreground with no background neighboring pixels. So, the pixel value in the output image im is changed to 128. As we iterate through the row of y = 1, at x = 4 when the pixel value of input image is 180, we can see that the top right corner is a background. So a question may appear as to is this a boundary or interior pixel. However, we defined our algorithm to have 8-connected foreground. So, the pixel on the right and top of the current pixel is connected on its corners, which will act as the boundary of the object. Thus, the current pixel is an interior with pixel value 128.

The program will continue iterating through the entire image, changing the pixel value of the foregrounds. In the end, the output image will show how the object now has a boundary of pixel value 255 and interior pixel value of 128 (see Figure 3).

After testing our boundpy program on a small image, it is then tested on a large image. The large image chosen for this program is the shuttle.vx image shown below (see Figure 4). It is a figure of a space shuttle, in which the shuttle is the only object in this image. The goal of the program would be to detect the edges of this shuttle, where the boundary will be marked with white color (pixel value 255) and the interior will be gray (pixel value 128).

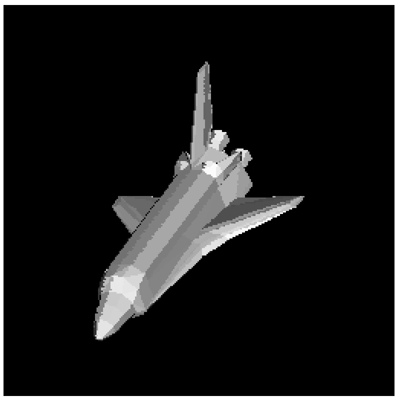


Figure 4 – Original shuttle.vx image for input to boundpy.

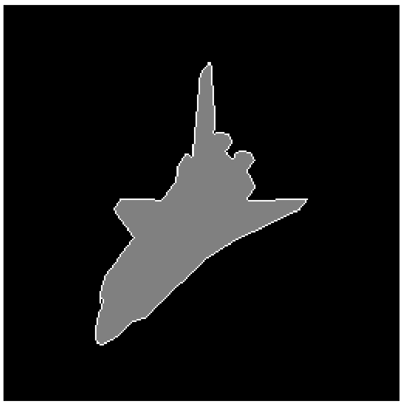


Figure 5 – Output of the shuttle.vx image from boundpy program.

Similar to when using a small image as input, this program will iterate through each pixel of the shuttle.vx image to detect for the foreground and determine whether this current pixel is an interior or a boundary. It will first scan through the x axis from left to right on the topmost row, then iterate to go down the row. The first foreground pixel that it detects would be a boundary pixel, which would be the tip of the back of the shuttle rudder on the top-mid section of the image. When this is detected on the tm image of the input, the pixel value for the output will be changed. This program will continue scanning the whole image, marking and detecting the boundaries and interiors, and the result of the output is a shuttle, which has its boundaries marked white and the interior colored gray (see Figure 5). A clear marking of where the foreground is compared to the background can be clearly seen.

One thing to note is on how the edges on the boundary is not smooth (see Figure 5). We can see how in some spot, the boundary pixel is connected with each other only on its corners. This again shows the 8-connected foreground that this program adopts.

Section 5:

In Section 5, the main goal is to create a python program called cclabelpy that would output an image where each object or component in the image is labeled. Each of the connected component in labelled uniquely in the sequence of n = 1, 2, 3… depending on when it is detected and n being the total number of components in the image. This python program is built using the template program vtemp2, which is similar to the template vtemp used for Section 4 program boundpy. This template provides how this program will accept the input and output file name, getting the input image from the input file name, creating an embed image tm based on input image, then writing the output result to the output file name.

In this labeling program, we will be using the recursive method of connected component labeling. The way this works is that it will iterate through the pixels to find a foreground (pixel with value > 0) that has not been labeled yet. When it finds it, it will label it with the current label number n, where n is 1, 2, 3… up to the number of total components in the image. After it detect this unlabeled foreground, it will recursively try to find foregrounds connected to this pixel on the top, bottom, left, and right and label everything connected to it with the same label. After labeling everything connected, it will go back to the first pixel of that label, then continue to look for other unlabeled foreground pixel. We are also using the 4 connected foreground and 8 connected background connectivity. When we are looking for connected components, we are only looking to 4 sides: top, bottom, left, and right, and we are not looking into the corner edges of the pixel.

In our code, an algorithm is created to apply the recursive method explained above. First, the output image im is cleared by iterating through each pixel and setting its value to 0. The goal is that im will be used to mark each label and we should have a clean canvas for this. Then, a for loop to iterate the pixels of the image starting from the top-most row through the x axis from left to right, then going down the rows to iterate each pixel. In this for loop, the goal is to look for an unlabeled foreground. It will do this by checking the pixel value at tm[y+1, x+1] and label value at im[y,x]. We are adding 1 to each pixel due to using the embed tm image. When an unlabeled foreground is detected, it will call the setlabel function and pass the input (x, y, and n). This setlabel function will be discussed next. This function will basically label that pixel at the current label, then recursively try to find all connected components to it and give the same label. When it is done, it will go back to this primary loop, increase the current label number n, then continue looking for unlabeled foreground until it covers the whole image.

The function definition called setlabel has 3 input parameters: x, y, and n. X and y is the is the current x and y index that will that will be used to find the connected pixels on the neighbors. N is the current label digit, which will be used to label all connected components. The way this function works is that it will first label this current pixel at output image as n. Then, it will check if the pixel above is an unlabeled foreground, and if it is, will call the setlabel function with the updated pixel value (in this case we increment y value by 1). Then, it will perform the same thing for the pixel below, on its left, and on its right in this order. This will recursively label all connected components on all sides. Again, we are using 4 connected foreground connectivity, so we are only checking 4 sides. Once it is done, it will go back to the main for loop and increase the label n value. This means that we have found all pixel for that component or label.

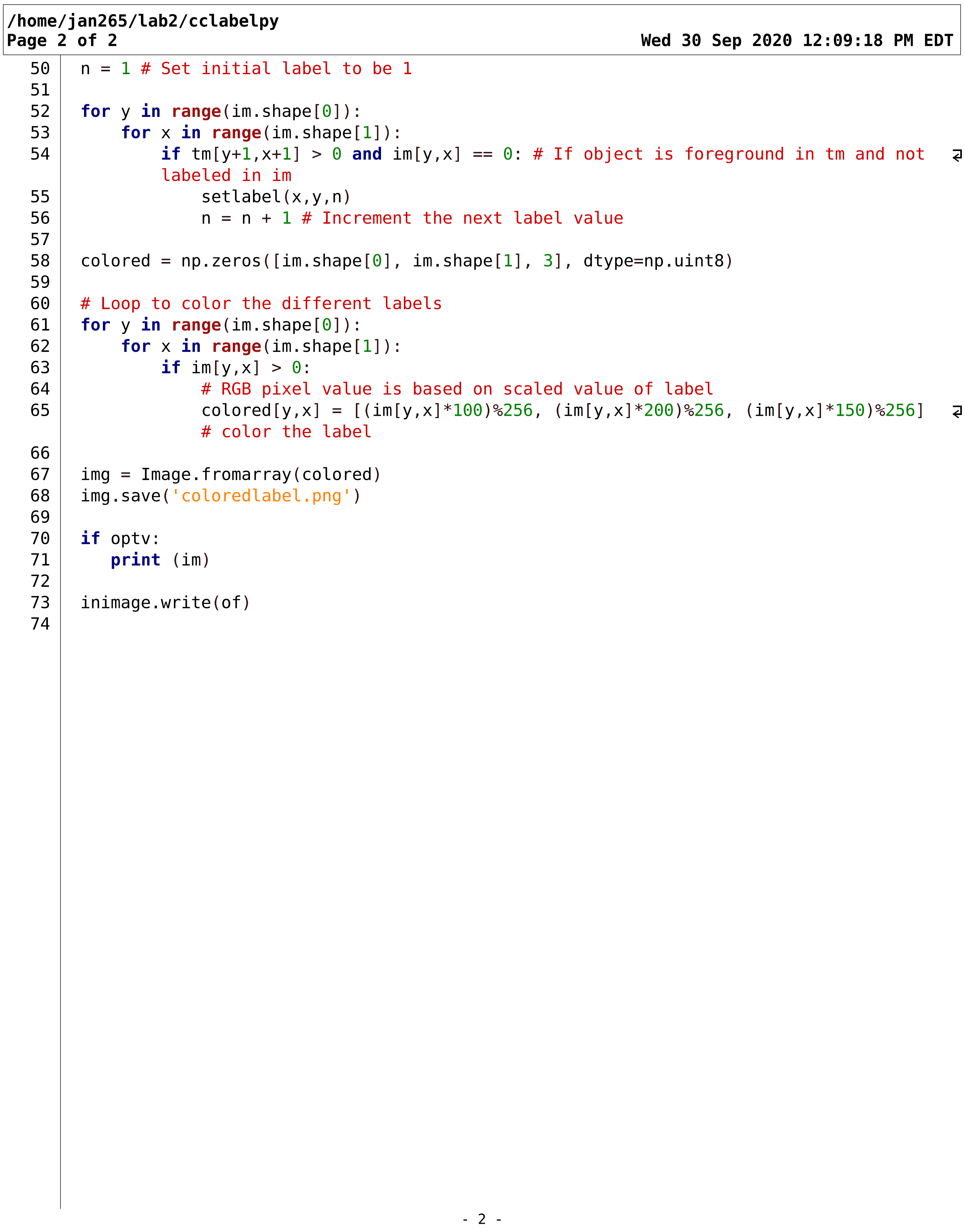
After iterating through all pixels in the main for loop, all the connected components on the image will now be labeled, and there is a total of n components labeled consecutively. The output image is written to the output file stated by the user.

For the extra credit portion, the goal to have RGB color differentiation of each label in the output of the large image. So, the numpy library is imported for multidimensional array capabilities, and the Image library from PIL library is imported for image processing capabilities. Then, an array is made of a similar dimension of the input image, however in each index, it will consist of 3 values: Red, Green, Blue, instead of the regular greyscale image that only has the greyscale pixel value. Then, a for loop is made to iterate the whole output image, which has the labels of the connected component. For each labeled pixel that was found in the output image, a value is also given to the RGB array that was created in the same coordinate. However, the value that is given consist of RGB components, where for each R, G, and B component, it is based on a constant that is multiplied by the label value (then divided by 256 using % modulo function to give remainder and to never get above 255). So, each label will have its own RGB combination, creating different colors for each label. This colored label image is then output to be a .png image filw with name 'coloredlabel.png'.

Source code for Section 5:

/home/jan265/lab2/cclabelpy
Page 1 of 2 **Wed 30 Sep 2020 12:09:18 PM EDT**

```python
#!/usr/bin/env python

""" cclabelpy a python recursive labeling program
"""

import sys
from numpy  import *
from v4 import vx
import numpy as np
from PIL import Image

of=' '
vxif=' '
clist = vx.vxparse(sys.argv,  "if= of= -v  - ")
exec (clist )

if 'OPT' in locals():
   print ("cclabelpy program")
   print ("if= input file")
   print ("of= output file")
   print ("[-v] verbose mode")
   exit(0)

optv = 'OPTv' in locals()

def setlabel (x, y, n): # n is current label
    global im, tm
    im[y,x] = n
    # 4 connected foreground 8 connected background
    if tm[y+2,x+1] > 0 and im[y+1,x] == 0: # If pixel above current tm is foreground and not labeled in im
        setlabel(x, y+1, n)
    if tm[y,x+1] > 0 and im[y-1,x] == 0: # If pixel below current tm is foreground and not labeled in im
        setlabel(x, y-1, n)
    if tm[y+1,x] > 0 and im[y,x-1] == 0: # If pixel left of current tm is foreground and not labeled in im
        setlabel(x-1, y, n)
    if tm[y+1,x+2] > 0 and im[y,x+1] == 0: # If pixel right current tm is foreground and not labeled in im
        setlabel(x+1, y, n)

inimage = vx.Vx( vxif )
im = inimage.i
tmimage = vx.Vx( inimage )
tmimage.embedim((1,1,1,1))
tm = tmimage.i

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

```



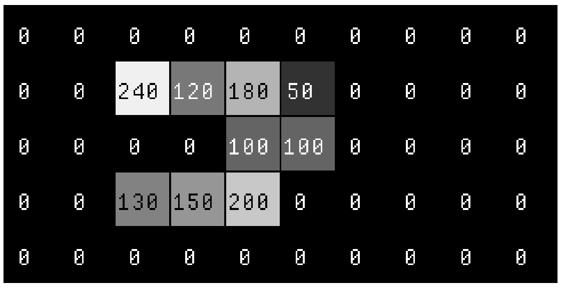
/home/jan265/lab2/cclabelpy
Page 2 of 2 Wed 30 Sep 2020 12:09:18 PM EDT

```python
n = 1 # Set initial label to be 1

for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        if tm[y+1,x+1] > 0 and im[y,x] == 0: # If object is foreground in tm and not labeled in im
            setlabel(x,y,n)
            n = n + 1 # Increment the next label value

colored = np.zeros([im.shape[0], im.shape[1], 3], dtype=np.uint8)

# Loop to color the different labels
for y in range(im.shape[0]):
    for x in range(im.shape[1]):
        if im[y,x] > 0:
            # RGB pixel value is based on scaled value of label
            colored[y,x] = [(im[y,x]*100)%256, (im[y,x]*200)%256, (im[y,x]*150)%256] # color the label

img = Image.fromarray(colored)
img.save('coloredlabel.png')

if optv:
   print (im)

inimage.write(of)
```

- 2 -

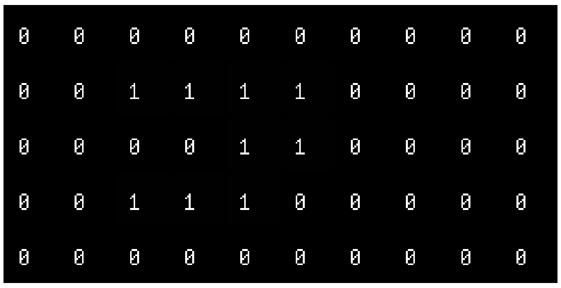
Figure 6 – Source code of section 5.

The cclabelpy program is first tested with small image to test if it actually works for different scenarios. The small image is more effective in testing, because we can easily notice each pixel values, unlike large image where we will actually have to zoom in and pan around, making it difficult to see errors. Our first test image is one that has 9 connected components (see Figure 7). As can be seen below, there are 9 components or pixels connected together to form one big component.



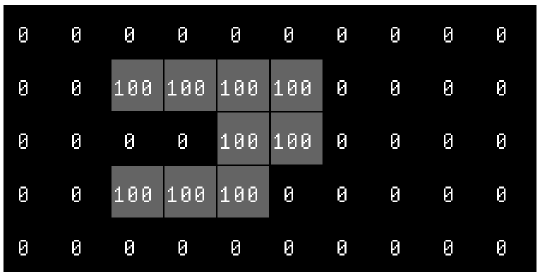
0	0	0	0	0	0	0	0	0	0
0	0	240	120	180	50	0	0	0	0
0	0	0	0	100	100	0	0	0	0
0	0	130	150	200	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 7 – Input small image a for cclabelpy program.



0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0	0	0
0	0	0	0	1	1	0	0	0	0
0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 8 - Output of the small image a from cclabelpy.



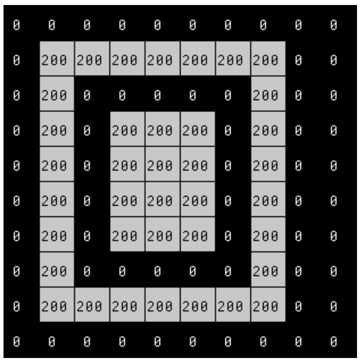
0	0	0	0	0	0	0	0	0	0
0	0	100	100	100	100	0	0	0	0
0	0	0	0	100	100	0	0	0	0
0	0	100	100	100	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figure 9 – Scaled pixel value of output small image a from cclabelpy.

The small image at Figure 7 is used as an input to the program. It will start iterating to find the unlabeled foreground. In the first row (y = 0), there will be no unlabeled foreground, then it will go to the row below (y=1). At the third pixel of this row (y=1), which has the value of 240, is the first unlabeled foreground to be detected. That pixel location is labeled as 1 in the output image (see Figure 8). Then, the algorithm will now search for other foregrounds that are connected to it, recursively searching the top, bottom, left, and right of that pixel. After searching for the top, bottom, and left, which are all backgrounds, it will find an unlabeled foreground on its right. The pixel on its right with the value 120 is then labeled as 1 in the output image. It will then recursively check for neighboring connected component using the same algorithm. In the end, since all components in this image are connected, they will be all labeled in this recursive call with label 1 in the output image. The sequence of the pixel getting detected is in this order based on the pixel value (240, 120, 180, left 100, 200, 150, 130, right 100, 50). After labeling all connected component, the algorithm goes back to the main for loop at location of pixel value 240. It will continue iterating through the image, but will found no more unlabeled foreground, as all foreground has been labeled with the same label. The program will end and the final output is a single component with label 1 (see Figure 8).

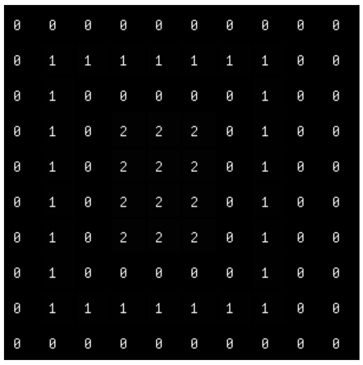
For added visibility, the output image from Figure 8 is modified to have each pixel scaled by 100 using the vpix command. The output is Figure 9, where the labeled component and the background is visible from each other.

Next, we tested the cclabelpy program using a small test image, where there is a component inside another component (see Figure 10). We can see how there is a component that is bordered around a background, then bordered with another component. The goal of this is to see how this program will detect these two components and on which is labeled first. It will also show if the foreground pixels on the top of the border will be labeled first, before the component in the middle if the image, despite being on top of it.



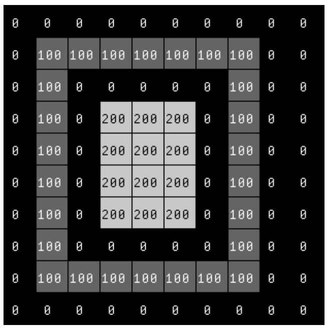
0	0	0	0	0	0	0	0	0	0
0	200	200	200	200	200	200	200	0	0
0	200	0	0	0	0	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	0	0	0	0	200	0	0
0	200	200	200	200	200	200	200	0	0
0	0	0	0	0	0	0	0	0	0

Figure 10 – Input small image b for cclabelpy.



0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0
0	1	0	0	0	0	0	1	0	0
0	1	0	2	2	2	0	1	0	0
0	1	0	2	2	2	0	1	0	0
0	1	0	2	2	2	0	1	0	0
0	1	0	2	2	2	0	1	0	0
0	1	0	0	0	0	0	1	0	0
0	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0

Figure 11 – Output of small image b from cclabelpy.



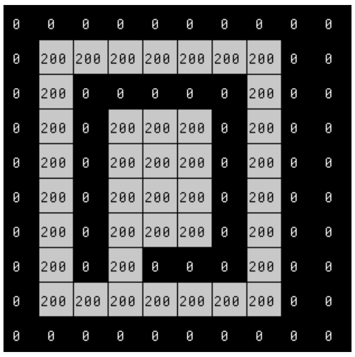
0	0	0	0	0	0	0	0	0	0
0	100	100	100	100	100	100	100	0	0
0	100	0	0	0	0	0	100	0	0
0	100	0	200	200	200	0	100	0	0
0	100	0	200	200	200	0	100	0	0
0	100	0	200	200	200	0	100	0	0
0	100	0	200	200	200	0	100	0	0
0	100	0	0	0	0	0	100	0	0
0	100	100	100	100	100	100	100	0	0
0	0	0	0	0	0	0	0	0	0

Figure 12 – Scaled pixel value of output small image b from cclabelpy.

The iteration starts at the top left corner, and the algorithm quickly found that there is no foreground on that row (see Figure 10). On the next row, in the second pixel, it will find the first unlabeled foreground, and label it in the output image. The recursive algorithm will now search for other neighboring unlabeled foreground, in the order of top, bottom, left, and right. From this algorithm, the next unlabeled foreground detected is the pixel below it. The algorithm will quickly find of the connected components of this first label, starting from top left, going down to the most bottom foreground, then going to the right, then to the top, then to the left up to the point there are no more neighboring unlabeled foreground. After all connected components have been labeled, the main loop continues to find other unlabeled foreground, and this time going to label it as 2. After iterating, it will then find the foreground located around the middle of the image, and the recursive labeling algorithm will label all of the connected component. There are only two components in this input, and the output is as shown in Figure 11.

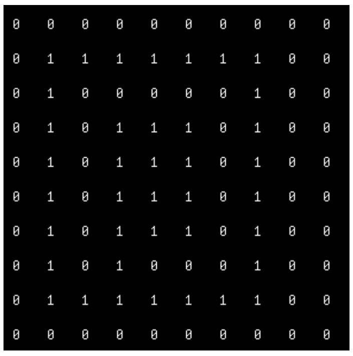
For added visibility, the output image from Figure 11 is modified to have each pixel scaled by 100 using the vpix command. The output is Figure 12, where the labeled component and the background is visible from each other.

Next, we tested the cclabelpy program using a small test image, where there is a component inside another component, but the two are connected via a single foreground pixel. We can see how there is a component that is bordered around a background, then bordered with another component, but the two are connected via a single pixel. The goal of this is to see how this program will label the components, which at first may seem to be two different components, to be as one label.



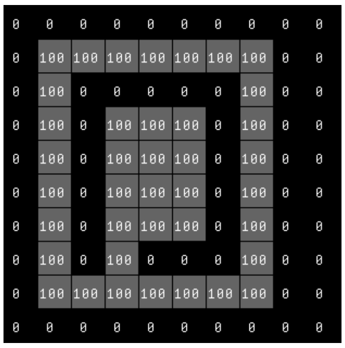
0	0	0	0	0	0	0	0	0	0
0	200	200	200	200	200	200	200	0	0
0	200	0	0	0	0	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	200	200	0	200	0	0
0	200	0	200	0	0	0	200	0	0
0	200	200	200	200	200	200	200	0	0
0	0	0	0	0	0	0	0	0	0

Figure 13 - Input small image c for cclabelpy.



0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0
0	1	0	0	0	0	0	1	0	0
0	1	0	1	1	1	0	1	0	0
0	1	0	1	1	1	0	1	0	0
0	1	0	1	1	1	0	1	0	0
0	1	0	1	1	1	0	1	0	0
0	1	0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0

Figure 14 - Output of small image c from cclabelpy.



0	0	0	0	0	0	0	0	0	0
0	100	100	100	100	100	100	100	0	0
0	100	0	0	0	0	0	100	0	0
0	100	0	100	100	100	0	100	0	0
0	100	0	100	100	100	0	100	0	0
0	100	0	100	100	100	0	100	0	0
0	100	0	100	100	100	0	100	0	0
0	100	0	100	0	0	0	100	0	0
0	100	100	100	100	100	100	100	0	0
0	0	0	0	0	0	0	0	0	0

Figure 15 - Scaled pixel value of output small image c from cclabelpy.

The iteration starts at the top left corner, and the algorithm quickly found that there is no foreground on that row (see Figure 13). On the next row, in the second pixel, it will find the first unlabeled foreground, and label it in the output image. The recursive algorithm will now search for other neighboring unlabeled foreground, in the order of top, bottom, left, and right. From this algorithm, the next unlabeled foreground detected is the pixel below it. The algorithm will quickly find of the connected components of this first label, starting from top left, going down to the most bottom foreground. As it goes to the right, after labeling third pixel on row y=8, it will detect the neighbor unlabeled foreground on top, then recursively labeling all of the connected components in the middle of the image with the same algorithm (top, bottom, left, then right). After labeling all foregrounds in the middle of the image, it goes back to the foreground pixel around the border and continues going to the right, then to the top, then to the left up to the point there are no more neighboring unlabeled foreground. After all connected components have been labeled, the main loop continues to find other unlabeled foreground, and this time going to label it as 2. However, it will iterate and find no more unlabeled foreground. There are only 1 component in this input, and the output is as shown in Figure 14.

For added visibility, the output image from Figure 14 is modified to have each pixel scaled by 100 using the vpix command. The output is Figure 15, where the labeled component and the background is visible from each other.

Lastly, the cclabelpy program is tested on a large image. The large image chosen to test is the im1.vx, which is provided for the lab. This image is of the phrase “tended for use”, where some letters are connected, while others are disconnected from each other. The goal is to see how many different components are there in this image.

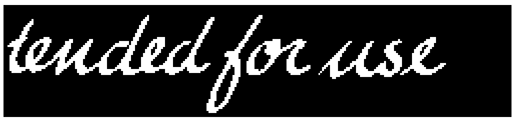


Figure 16 – Original image of im1.vx.

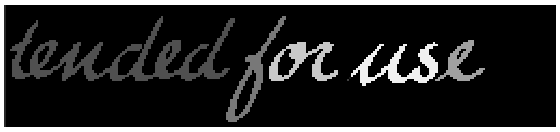


Figure 17 – The image im1.png as output of the cclabelpy program with grayscale label. The pixel values are scaled for visibility purposes.

This algorithm will start from the top-left pixel (see Figure 16). As it checks all pixels through the x axis and from the top going down the rows, the first unlabeled foreground detected is the ‘d’ letter on ‘tended’. The recursive algorithm will then label all connected components to this pixel, thus labeling the word ‘tended’ with the same label 1. In figure 17 above, we can see how the word ‘tended’ has the same label color. This figure has a scaled label value by 40 to allow visibility between labels.

The program will then continue iterating the image and found that the top tip of letter ‘f’ is the detected unlabeled foreground pixel. The recursive algorithm then labels the letter f as the second label. Next, the program continues doing this and then labeled ‘e’, ‘or’, and ‘us’ in that order. We can see the different greyscale intensities in Figure 17, showing the different label values that are scaled by 40.

For added visibility, the output image of the input Figure 16 is modified to have each pixel scaled by 40 using the vpix command. The output is Figure 17, where the labeled component and the background is visible from each other. This is performed because the difference in 1 pixel value, which is represented as the label, is just not visible.

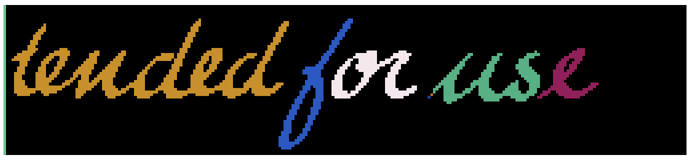


Figure 18 – The image im1.png as output of cclabelpy with RGB label. The pixel values are each object is adjusted to have different colors.

For extra credit, the goal is to have each label display different RGB colors. An array of the size of the input image is created, but having 3 values to each index for RGB values. Then it iterates the output from Figure 16 input to find labeled component, then calculates RGB value based on a multiplied constant with the label value from the output, then use the RGB value to the same coordinate in the colored array. A .png image is generated based on this colored array, and the output is Figure 18. This coloring algorithm can be used for other outputs of cclabelpy for coloring the different labels for easier visualization.