1 Problem 1

Print out and colour the matrix `pictureMe.png`. *A fun exercise.*

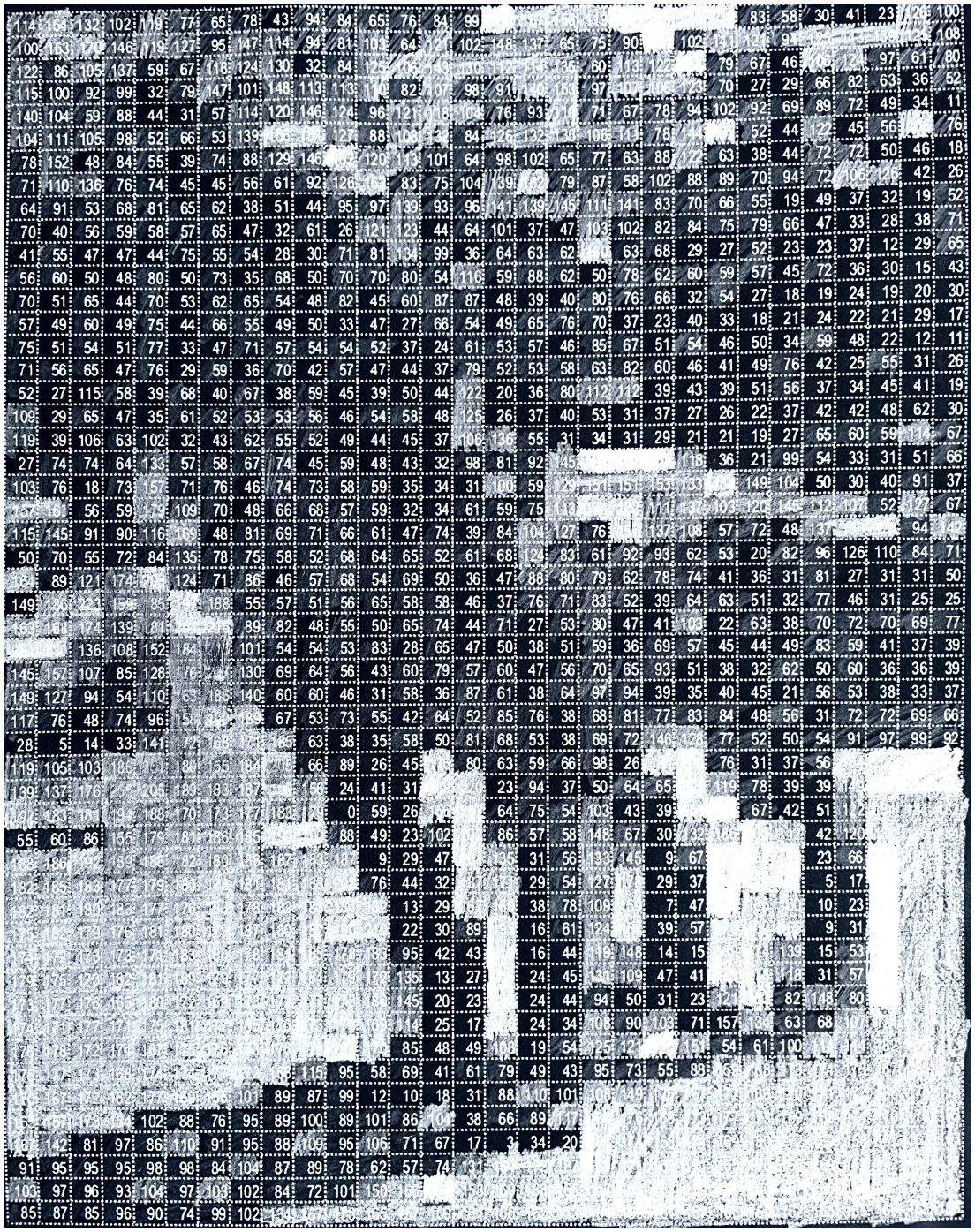

Figure 1: Coloured `pictureMe.png`

I used 4 different grade graphite pencils to colour the image. Included uncoloured squares (white) this made me split up the interval $[0, 255]$ into 5 linearly spaced sections, each corresponding to a white or the 4 shades of graphite.

I realised that most of the numbers were low, which correspond to darker pixels. Since I wanted to colour less, I corresponded high values with black and low values with white, and then inverted the final image (as you can see in Figure 1).

2 Problem 2

2. (a) *Read and show the image, and then capture the screenshot of the window showing the image. Save the captured screenshot as [your_unityid]_screenshot.png (e.g., twu19_screenshot.png)*

   Using the Python Image Library (PIL), I read in the image from its directory and displayed it using `matplotlib.pyplot.imshow()`. I used PIL instead of OpenCV or `matplotlib` to read in the file because OpenCV was not in RGB order and `matplotlib` was not in `uint8` format ($\in$ [0, 255]), whereas PIL did both without me having to reorder anything

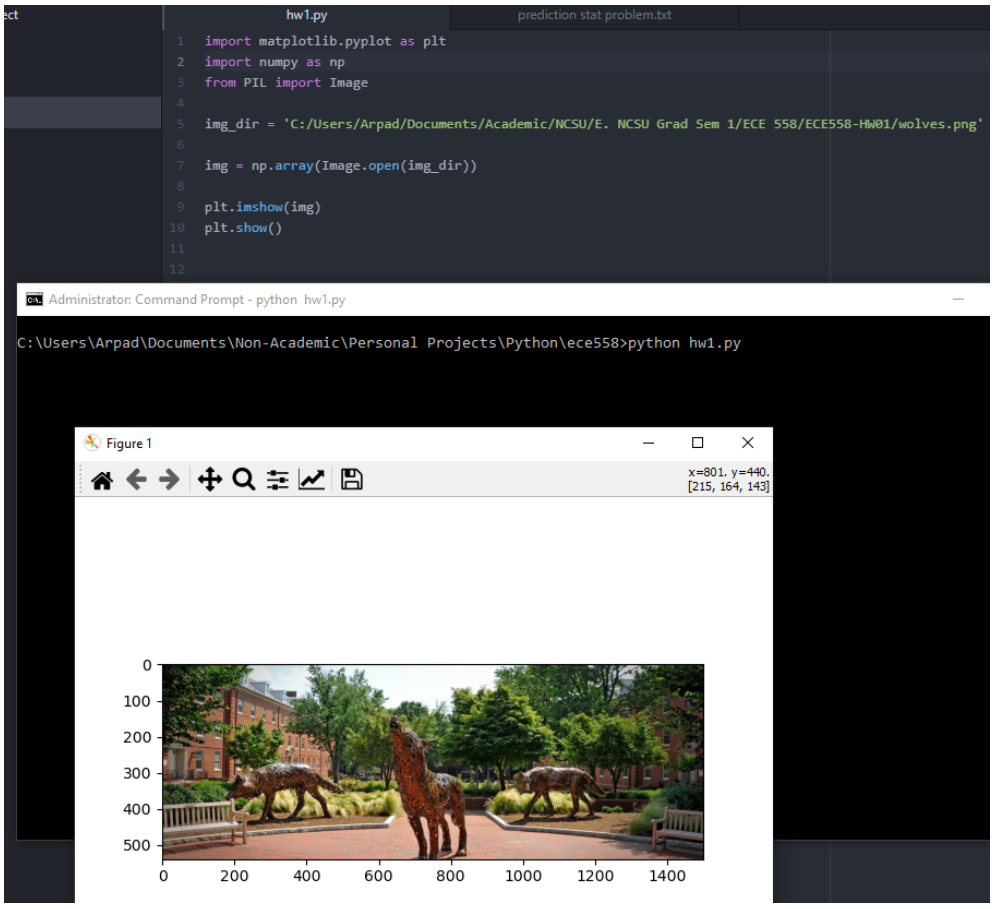

Figure 2: Screenshot of image being displayed

(b) i. *Find the digit signature of your unity id. First, convert each non-digit character in your unity id to ASCII value (e.g., 'twu19' corresponds to 116, 119, 117, 19).*

My Unity ID is `aavoros`, therefore it corresponds with ASCII numbers

$$\texttt{aavoros} = [97, 97, 118, 111, 114, 111, 115]$$

Regardless of having no number within my Unity ID, the program will still properly convert all characters to this 'digital signature' as given in the problem statement.

ii. *Second, count the number of occurrence of each of the digit number in each color channel of the image (the number of occurrence could be zero, but definitely less than the total number of pixels).*

The number of instances is given in Table 1 below. There is a redundancy in count for the characters `a` and `o` due to being duplicated.

Character	ASCII	# in R	# in G	# in B
a	97	3063	3472	2807
v	118	2506	3035	1910
o	111	2662	3149	2202
r	114	2639	3080	2057
s	115	2534	3170	2096

Table 1: Pixel counts for unique `aavoros` ASCII characters in `wolves.png`

iii. *Third, change to 255 the pixel values of the 5 by 5 sub-image (if valid) centered at each occurrence and then show the result image. After you changed all the occurrence, save the result image as [your_unityid]_signature.png (e.g., twu19_signature.png)*

Iterating through each channel, row, column, and ASCII value to change the value of a 5×5 box to 255 is susceptible to overwriting potential pixels which equal our ASCII values. Meaning, if there pixels with our ASCII values within 2-4 pixels of one another and in the same channel, whichever ASCII value is checked first will overwrite the other ASCII value to 255. This results in not every ASCII pixel their 5×5 square. A way around this would be to create a map of every pixel position where an ASCII value exists, and then proceed to colour a copy of the image w.r.t. the map, rather than colouring the image w.r.t. itself. However, I have been informed that w.r.t. grading, the order does not matter and overwriting is allowed. Therefore I have simply kept the old version of my program. The order of checking pixel values to colour pixel sub-images hierarchically follows the following

1. Loop through an ordered list of each ASCII value of all unique characters
2. Loop through each RGB channel
3. Loop through each row
4. Loop through each column
5. Check if coloured & if ASCII value equals pixel value. If so, colour the 5×5 box to 255

Here is the final digital signature for `aavoros`

Figure 3: 'Digital Signature' for my Unity ID: `aavoros`

Below is the listing for `hw1.py`. To summarize:

- PIL is used to read in the `wolves.png` image and is converted to a numpy array
- A character array holds a UnityID. This variable is fed through an algorithm which extracts the ASCII values for characters, and the integers remain
- All instances of the ASCII values within the `wolves.png` using binary operations
- The image in copied to output the Digital Signature. A mask of already altered values is created to check pixels faster. Each pixel in the (row, column, channel) position is compared to each ASCII value. If they equal, then the 5×5 box is recoloured and the mask is appropriately updated.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

# function to find bounds while painting. considers boundary conditions
# can change boundary size, but default is 5 (as said in hw1)
def get_bounds(r, c, num_r, num_c, box_l = 5):
    # 0 based indexing, this is max row/col value
    num_r -= 1
    num_c -= 1
    # init min/max values
    r_floor = 0
    c_floor = 0
    r_ceil = num_r
    c_ceil = num_c
    if not box_l % 2:
        # want box length to be odd, so the pixel value at r/c is perfectly in
                                                    the
        # center of the square
        raise ValueError('box length must be odd: box_l = %d' % (box_l))
    else:
        delta = int((box_l - 1) / 2)
        if r + delta < r_ceil:
            r_ceil = r + delta
        if r - delta > r_floor:
```

```
            r_floor = r - delta
        if c + delta < c_ceil:
            c_ceil = c + delta
        if c - delta > c_floor:
            c_floor = c - delta
    return np.array(range(r_floor, r_ceil)), np.array(range(c_floor, c_ceil))

# get the wolf image
folder_dir = 'C:/Users/Arpad/Documents/Academic/NCSU/E. NCSU Grad Sem 1/ECE 558/
                                        HW01/'
img_dir = folder_dir + 'wolves.png'

# display the wolf image
img = np.array(Image.open(img_dir))
plt.imshow(img)
plt.show()

# image characteristics
img_shape = np.shape(img)
num_r = img_shape[0]
num_c = img_shape[1]

# unity id, used in calculating the 'digital signature'
my_uid = 'aavoros'

# get the ASCII numbers of the unity id
ascii_uid = []
id_len = len(my_uid)
id_idx = 0
while id_idx < id_len:
    chr = my_uid[id_idx]
    if chr.isnumeric():
        offset = 1
        while my_uid[id_idx:id_idx + offset].isnumeric() and (id_idx + offset) <
                                                    = id_len:
            offset += 1
        ascii_uid.append(int(my_uid[id_idx:id_idx + offset - 1]))
        id_idx += offset - 2
    else:
        ascii_uid.append(ord(chr))
    id_idx += 1
ascii_uid = np.array(ascii_uid)

# print unity id with ASCII result
print(my_uid)
print(ascii_uid)

# find the pixel count of each ASCII code
num_red = np.zeros(np.shape(ascii_uid))
num_gre = np.zeros(np.shape(ascii_uid))
num_blu = np.zeros(np.shape(ascii_uid))
chr_idx = 0
for chr in ascii_uid:
    num_red[chr_idx] = int(np.sum(img[:, :, 0] == chr))
    num_gre[chr_idx] = int(np.sum(img[:, :, 1] == chr))
    num_blu[chr_idx] = int(np.sum(img[:, :, 2] == chr))
    chr_idx += 1

# print the counts
print("Number of red instances per ASCII value:")
print(num_red)
print("Number of green instances per ASCII value:")
```

```
print(num_gre)
print("Number of blue instances per ASCII value:")
print(num_blu)

# create a copy of the image to paint the digital signature
ds_img = np.copy(img)
# each channel, RGB
channel = [0, 1, 2]
# create a binary mask to skip already painted pixels
unchecked_mask = np.ones(img_shape, dtype = bool)
# find each pixel, paint the digital signature
for chr in np.unique(ascii_uid):
    for chnl in channel:
        for r in range(num_r):
            for c in range(num_c):
                if unchecked_mask[r, c, chnl] and ds_img[r, c, chnl] == chr:
                    r_range, c_range = get_bounds(r, c, num_r, num_c)
                    ds_img[r_range[:, None], c_range[None, :], chnl] = 255
                    unchecked_mask[r_range[:, None], c_range[None, :], chnl] =
                                                                    False

# save the image and display it
Image.fromarray(ds_img).save(folder_dir + my_uid + '_signature.png')
plt.imshow(ds_img)
plt.show()
```