Introduction to Computer Programming

Final Report

Dalia Sara Gala, DPhil in Biochemistry

Abstract

During the October 2019 DTC course called “Introduction to Programming”, students were assigned a task to choose one of four program writing options. I chose a task where the goal was to design and write an image analysis program capable of reading a DNA microarray image and returning a CSV file with signal strength values for “red” and “green” wells. I succeeded in writing the program which saves a CSV file, but this program has a limited functionality. The program can identify where the wells are quite well, and can return RGB pixel values for what should be the well centre; it can also compare the R value to G value, and return a value on a sliding scale, where -255 is a completely red pixel, and 255 is a completely green pixel. The program however can deal only with a single microarray plate, and not collection of plates arranged in one image. The program also reads the value of one pixel only, and the results which it gives are not very accurate. Considering that I learned programming in the past 3 weeks, I would say that the program does a decent job, yet it could be improved in many ways, something which has been done in commercial microarray plate analyser programs.

1. Background

A DNA microarray is a set of DNA oligonucleotides attached to a surface of a chip. Fragmented DNA of interest, which could for example come from a patient with a genetic disease, is labelled with one fluorophore, such as red, and comparatively hybridised to the oligonucleotides together with another fragmented DNA labelled green, which is the reference DNA. If an equal number of fragments attaches to the oligonucleotide of interest, yellow colour is visible in this well, and that means that the amount of DNA from the experimental sample and the reference sample is the same. If the colour is green, it means there is none or little of the experimental DNA and could indicate a deletion in the experimental sample. If red is the dominant colour, the experimental DNA dominates and that could mean a duplication. Hence, the technique of DNA microarray is a very useful one, used widely in research and diagnostics.

Microarray analysis is a bioinformatical task which particularly benefits from automation because of its high-throughput nature. Manually, it would take a large amount of time to go through each well in a single 96 well plate and assign a value to them. A program which analyses microarray plates saves both researchers and diagnosticians lots of precious time, and allows them to obtain results from their experiments faster, as well as should be less prone to the bias of human eye, as it analyses the pixel colour values rather than subjectively declaring the well “green” or “red”.

I chose this task because I will be working in a laboratory where we deal with a lot of image data, for which reason I wanted to hone my image manipulation skills. I wanted to challenge myself as well, because I felt like we did more work during the course with protein and DNA sequences, as well as ASCII art, but less with reading image files and manipulating them, especially ones which contain data and are to be analysed.

1. Introduction

The goal of the project was to write a program which would take an image of wells of a microarray and return a CSV output file with the intensity of the signal strength of the green and red wells. The image file provided at the start of the task had 24 plates arranged in 4 columns in 6 rows, and each plate had 96 wells arranged in 24 columns and 24 rows. To write the program, a way would have to be found to detect where the wells rows and columns are, and correctly read the pixel values at these points. Moreover, a way to compare the red and green value would be necessary in order to export a single “result” to a CSV file.

From the course, I was aware that the .jpg file is a binary file, so I knew I needed to make it human-readable. The plan was to use a python library to get that done. During the project introduction talk, it was mentioned that a good way to approach this task would be to get the average values of all rows and columns in the image as a way to find where the wells are. Having found these, one could get the average value for the well distance, and, possibly, the troughs distances if dealing with a multiplate image. Finally, the program would have to iterate over the coordinates of the “peaks” both for rows and columns to read individual values of the wells and append them as a CSV having compared them.

1. Methodology

I started working with the program by searching for image manipulation libraries which could be used with the file type provided (.jpg). I chose to use PIL to open and manipulate the image, as it is a comprehensive and well described library. I used NumPy to manipulate numbers, and MatPlotLib.Pyplot to plot necessary graphs to verify that the code is working as I expected it.

I built the code according to the plan described in the introduction. The code initially uses “try – except” to check if the file is in a .jpg or a .png format, and if the file exists. The code uses PIL to open and manipulate the image and convert it to greyscale. Still using PIL, the code extracts greyscale pixel values; after that, NumPy function mean is used to average the row an column pixel values. I struggled with finding a way to determine which values were the maxima for rows and columns and eventually found the SciPy function “find\_peaks” . This function is used to search for peaks in the given array of averaged values in rows and columns. Finally, the original image is being reopened as a colour image and PIL “image.convert” is used to get the RGB values from it. After that, the program iterates over given pixel coordinates and compares the R value to the G value, and appends the difference between them to the new array, which is exported to CSV using NumPy function “savetext”.

1. Results

A close up of a sign

Description automatically generated

Figure 1. A screenshot of the file prompt at the beginning of the program execution.

The program starts by asking for a file as input, which needs to be a .jpg or a .png file; if it is not, if no file is input, or the file does not exist, the program exits using the “sys” library.

A close up of a sign

Description automatically generated

A close up of a sign

Description automatically generatedA close up of a sign

Description automatically generated

Figure 2. A screenshot of the comments the program returns if there is no input file (top), if the file does not exist (middle) or the file is in the wrong format (bottom).

The program successfully opens the file using PIL Image.open, blurs it and converts it to a black-and-white image. Afterwards, the program uses np.mean inserted into two functions – GetAvgCols and GetAvgRows - to obtain the average row and column values. These can be plotted using MatPlotLib.PyPlot (Figure 3).

A picture containing object

Description automatically generated

Figure 3. A plot of the peaks found by SciPy find\_peaks in the average values of rows in a sample microarray. This plot is not being shown to the user during program execution.

The program then extracts the peak values from the averages from the black-and-white image; these can be plotted using PyPlot. The program uses SciPy function “find\_peaks” to achieve that. Once the peaks have been found, the program uses the NumPy function sort to keep the index values in the correct order. These peak values can then be plotted on top of the above plot to see where the peaks are being found among the average row and column values (Figure 5).

A picture containing boat

Description automatically generated

Figure 4. The plot from above with the found peak values superimposed as scatter plot. This plot is not being shown to the user during program execution.

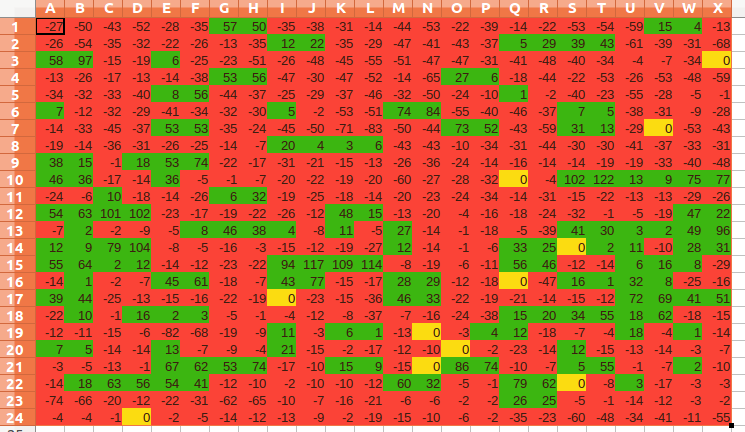
Next, the program produces a plot with a grid overlay. The well centres should correspond to the intersections between the grid lines of peak values of columns and rows. This image is saved for the user to verify how well the program found the well centres.

A picture containing object

Description automatically generated

Figure 5. An example of the grid showing predicted well centres generated by the program.

Finally, using PIL function image.open again, the program reopens the file as a colour file, and converts it to an RGB file with image.convert. It then iterates over row and column “peak” values using two nested “for” loops to get pixel values from these positions. Next, using another for loop, the program determines the difference between the value of the G and the R values. It appends the difference in the CSV file using np.savetxt, such that if G value is greater than R value, the difference will be a positive value, while if the R value is larger than the G value, the difference will be a negative number. Hence it creates a scale which presents red well as negative values, and green wells as positive values.

A close up of a light

Description automatically generated

Figure 6. An array generated by the program (left) together with the image which it analysed (right).

1. Discussion

The program which I wrote successfully analyses an image of a microarray. It is capable of determining where the wells are in this image quite well – despite the fact that often the well centre is shifted because of the approximate detection of the peak values. The program generates a CSV matrix which somehow resembles the image and calculates the difference between the red and the green values of the pixels which it considers the centres of the wells. The program works well both with symmetrical and unsymmetrical matrices, such that it can read both a 24x24 matrix and a 10x20 matrix.

The program, however, has several limitations. It cannot read a multiplate similar to the file which we were provided with for this task. I attempted to generate a code which could do that by calculating the average distance between wells and an average trough value – but it was not precise enough and the well centres were not aligned correctly with the grid points. Moreover, the program cannot account for the image being rotated, so only a straight array image can be analysed, and often that it not the case after the image was obtained.

The method by which the program calculates the well value could also be improved. Ideally, instead of reading just one-pixel value, it would read several around the pixel coordinates which it determines as the well centre. Moreover, it would be useful to assign ranges in which the values are considered “red” or “green”, because the colour human eye sees isn’t simply a result of “the R value is greater, so the well is red”.

1. Conclusions

The program which I wrote performs the function which it was designed to perform, despite its limitations. I wish I had more time to perfect it, and I am quite sure I could; I would like to add the functionality described above and make it more robust so it works better for an average everyday user. However, since I am new to programming, I am quite happy I finished it in the first place.

1. Appendix

#!/usr/bin/env python3

# Name: Microarray\_analysis\_program.py

# Created by: Dalia S. Gala, dalia.gala@merton.ox.ac.uk

# Date: 28 Oct 2019

# This program will accept a .jpg or .png file image containing multiple plates of DNA microarrays, and return a Comma Separated Variable (CSV) file listing the intensity of the signal strength for healthy (green) and unhealthy (red) test results.

import sys

import numpy as np

import matplotlib.pyplot as plt

import scipy

from scipy import signal

from scipy.signal import find\_peaks

import PIL

from PIL import Image

from PIL import ImageFilter

# OPEN FILE AND DEFINE SUITABILITY

MicroarrayJpg = input("\033[1;37;41mEnter filename in current directory, or the entire filepath starting at home; your file HAS TO be a .jpg or a .png image file of a single symmetrical plate of a DNA microarray; if it is not, the program won\'t run correctly:\033[0m")

try:

if len(MicroarrayJpg) == 0:

print("Proper usage of this program requires the input of a filename of a .jpg or .png file")

sys.exit()

elif ".jpg" in MicroarrayJpg or ".png" in MicroarrayJpg: #Input will require a .jpg file

im = Image.open(MicroarrayJpg, 'r') #If input is a .jpg, program will run

else:

print("Invalid file: must be a .jpg file!")

sys.exit() #If input not jpg, error and exit

except FileNotFoundError:

print("There is no such file - try again!")

sys.exit() #If input is a .jpg but not exist, error and exit"""

#CONVERT TO BLACK AND WHITE

img = im

img = img.filter(ImageFilter.GaussianBlur(radius=2))

#img.show() #DEBUGGING

img = img.convert('L') #makes it greyscale

#img.show() #DEBUGGING

img\_grey = np.asarray(img.getdata(),dtype=np.float64).reshape((img.size[1],img.size[0])) # get greyscale pixel values

#print(img\_grey.shape) #DEBUGGING

#GETTING AVERAGE VALUES FOR EACH COLUMN AND ROW; COMPARE UPON ROTATION TO GET MAXIMA

def GetAvgCols(imagefile):

return(imagefile.mean(axis = 0)) #returns the average values of each column; axis 0 = means which list it reads

def GetAvgRows(imagefile):

return(imagefile.mean(axis = 1)) # same as above, except for rows

GreyImgAvgRows = GetAvgRows(img\_grey)

GreyImgAvgCols = GetAvgCols(img\_grey)

#print(len(GreyImgAvgRows)) #DEBUGGING

#print(len(GreyImgAvgCols)) #DEBUGGING

#plt.plot(GreyImgAvgRows) #DEBUGGING

#plt.show() #DEBUGGING

# FIND PEAKS AND THEIR INDICES BLACK AND WHITE

# COLS

peaks\_values\_C = find\_peaks(GreyImgAvgCols, distance = 10, prominence = 0.2, width = 2.5)

# ROWS

peaks\_values\_R = find\_peaks(GreyImgAvgRows, distance = 10, prominence = 0.2, width = 2.5)

# SKIP THE 1ST ARRAY ELEMENT, ONLY TAKE 0TH

peaks\_values\_R = peaks\_values\_R[0]

peaks\_values\_C = peaks\_values\_C[0]

# print(peaks\_values\_R) #DEBUGGING

# print(peaks\_values\_C) #DEBUGGING

# plt.scatter(peaks\_values\_R, GreyImgAvgRows[peaks\_values\_R]) #DEBUGGING

# plt.show() #DEBUGGING

# SORT INDICES BY SIZE

grid\_rows\_sorted = np.sort(peaks\_values\_R)

grid\_cols\_sorted = np.sort(peaks\_values\_C)

# sorts these by size to have the correct row and column order

# print(len(grid\_rows\_sorted)) #DEBUGGING

# print(len(grid\_cols\_sorted)) #DEBUGGING

# MAKE A GRID IMAGE WHICH WILL BE OVERLAID ON THE MICROARRAY TO SEE HOW WELL IT SHOWS WELLS

# where the lines cross - that is where the well centre should be - this section just to see the grid

width, height = im.size

plt.plot([])

plt.imshow(im)

for y in peaks\_values\_C:

plt.axvline(width - y)

for x in peaks\_values\_R:

plt.axhline(height - x)

cur\_axes = plt.gca()

cur\_axes.axes.get\_xaxis().set\_visible(False)

cur\_axes.axes.get\_yaxis().set\_visible(False)

plt.savefig('GridforMicroarray.png')

# plt.show() #DEBUGGING

# OPEN IMAGE IN COLOUR AND REFER THE PROGRAM TO THE COORDINATES OBTAINED ABOVE

# TO GET IT TO READ RGB VALUES

image = Image.open(MicroarrayJpg, 'r')

img = image.filter(ImageFilter.GaussianBlur(radius=2))

# img.show() #DEBUGGING

rgb\_image = image.convert('RGB')

r, g, b = rgb\_image.getpixel((1, 1)) #converts the image to RGB and assigns the RGB values to r, g, b variables

# print(image) #DEBUGGING

pixel\_values = []

for coordR in range(len(peaks\_values\_R)):

row\_vec = []

for coordC in range(len(peaks\_values\_C)):

pixel = rgb\_image.getpixel((int(peaks\_values\_C[coordC]), int(peaks\_values\_R[coordR])))

# goes to the integer value of the pixel index in row and col and gets RGB values

row\_vec.append(pixel)

pixel\_values.append(row\_vec)

pixel\_values = np.array(pixel\_values)

# print(pixel\_values) #DEBUGGING

# print(pixel\_values.shape) #DEBUGGING

# COMPARE R AND G VALUES AND APPEND THE DIFFERENCE TO AN ARRAY WHICH WILL BE PRINTED TO A CSV FILE

rg\_values = []

for row in range(len(pixel\_values)):

rg\_row = []

for col in range(len(pixel\_values[row])):

if int(pixel\_values[row][col][1]) - int(pixel\_values[row][col][0]) > 0:

rg\_row.append(int(pixel\_values[row][col][0]) - int(pixel\_values[row][col][1]))

elif int(pixel\_values[row][col][0]) - int(pixel\_values[row][col][1]) == 0:

rg\_row.append(0)

else:

rg\_row.append(int(pixel\_values[row][col][0]) - int(pixel\_values[row][col][1]))

rg\_values.append(rg\_row)

# "if" takes difference of G - R, if it is bigger than 0 (means more green), appends the value to row

# "elif" appends 0 if they are equal, so that should be a yellow well

# "else" appends the difference to row, and this time it will be negative because G - R is now <0

rg\_values = np.array(rg\_values)

#print(rg\_values) #DEBUGGING

#print(rg\_values.shape) #DEBUGGING

np.savetxt("well\_values.csv", rg\_values, delimiter=', ', newline='\n', fmt='%d')

#saves file as csv, gives new lines to each row, fmt expects integer values, %s = string, %f = float