Problem 1: The Power of Matrix Programming

Introduction: This problem is to highlight how you can use matrix manipulations to solve problems and how they often can have certain advantages over traditional looping or recursive methods. One of the cornerstones of computer programming is the idea of recursion and one of the most frequently used examples to demonstrate recursion is the Fibonacci sequence. Here, we will explore how the recursive implementation of the Fibonacci sequence is significantly slower than a matrix implementation that you will develop.

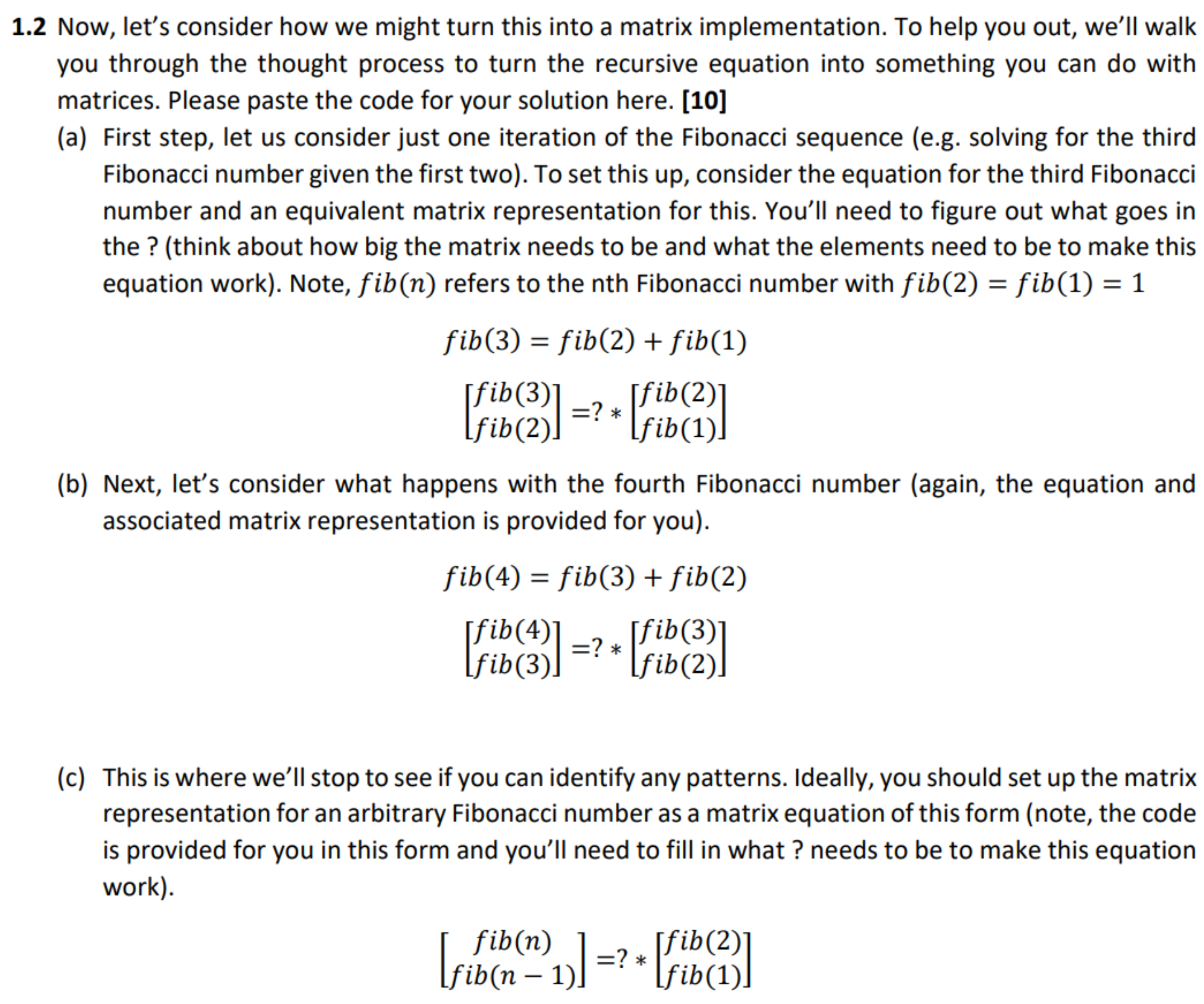
1.1 First things first, let's implement the recursive implementation of the Fibonacci sequence. We have set up the recursive function for you, and your job is simply to fill in a couple of lines in here that will perform the recursion itself. For those of you who need a reminder, the idea of recursion is based on the idea that functions can call themselves until you reach a base case. Please paste the code for your solution here. **[5]**

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 23 10:55:08 2023

File name: RecursionFibonacci.py
Author: Justin The
Date: 2/23/2023

TUse this script to develop the recursive implementation of the
Fibonacci
number equation. Only adjust code within the "TODO" brackets.

@author: Calvin
"""


# The main script
def main():

    n = 40 # Which fibonacci number do you want
    fib_n = fib_recurse(n) # Call the recursion function
    print( fib_n ) # Print out the nth Fibonacci number
```

```python
def fib_recurse(i):

    # Base cases
    if (i==1):
        fib_num = 1;
    elif (i==2):
        fib_num = 1;
    elif (i==0):
        fib_num = 0;
    else:
        fib_num = fib_recurse(i - 1) + fib_recurse(i-2);
#Recursive Case

    return fib_num

"""
The following code allows us to run this file as a script.
Note, this not the
only way to do this, but the benefit of using this method is
that all the
variables that are created as part of the script have local
scope.
"""
if __name__ == "__main__":
    main()
```

The output of the code above is:

```
102334155

#According to the List of Fibonacci Numbers: f(40) = 102334155
#_list of Fibonacci numbers_. (2013). Planetmath.org.
https://planetmath.org/listoffibonaccinumbers
```



1.2 Now, let's consider how we might turn this into a matrix implementation. To help you out, we'll walk you through the thought process to turn the recursive equation into something you can do with matrices. Please paste the code for your solution here. **[10]**

(a) First step, let us consider just one iteration of the Fibonacci sequence (e.g. solving for the third Fibonacci number given the first two). To set this up, consider the equation for the third Fibonacci number and an equivalent matrix representation for this. You'll need to figure out what goes in the ? (think about how big the matrix needs to be and what the elements need to be to make this equation work). Note, $fib(n)$ refers to the nth Fibonacci number with $fib(2) = fib(1) = 1$

$$fib(3) = fib(2) + fib(1)$$

$$\begin{bmatrix} fib(3) \\ fib(2) \end{bmatrix} = ? * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix}$$

(b) Next, let's consider what happens with the fourth Fibonacci number (again, the equation and associated matrix representation is provided for you).

$$fib(4) = fib(3) + fib(2)$$

$$\begin{bmatrix} fib(4) \\ fib(3) \end{bmatrix} = ? * \begin{bmatrix} fib(3) \\ fib(2) \end{bmatrix}$$

(c) This is where we'll stop to see if you can identify any patterns. Ideally, you should set up the matrix representation for an arbitrary Fibonacci number as a matrix equation of this form (note, the code is provided for you in this form and you'll need to fill in what ? needs to be to make this equation work).

$$\begin{bmatrix} fib(n) \\ fib(n-1) \end{bmatrix} = ? * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix}$$

$$fib(3) = fib(2) + fib(1)$$

Assuming that the ? is a 2x2 matrix that is multiplied by the initial matrix to get the final matrix:

$$\begin{bmatrix} fib(3) \\ fib(2) \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix}$$

$$\begin{bmatrix} fib(3) \\ fib(2) \end{bmatrix} \rightarrow \begin{matrix} fib(2) + fib(1) \\ fib(2) \end{matrix} = \begin{matrix} fib(3) \\ fib(2) \end{matrix}$$

Now consider the fourth Fibonacci number:

$$\begin{bmatrix} fib(4) \\ fib(3) \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{bmatrix} fib(3) \\ fib(2) \end{bmatrix}$$

Replace initial matrix with the equation above:

$$\begin{bmatrix} fib(4) \\ fib(3) \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix}$$

Now we can find the general pattern to find any arbitrary fibonacci number:

$$\begin{bmatrix} fib(n) \\ fib(n-1) \end{bmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} * \begin{bmatrix} fib(2) \\ fib(1) \end{bmatrix}$$

```
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 23 10:26:20 2023

File name: MatrixFibonacci.py
Author: Justin The
Date: 2/23/2023

TUse this script to develop the matrix implementation of the
Fibonacci
number equation. Only adjust code within the "TODO" brackets.

@author: Calvin
"""


# Imports - tell python to load in other libraries that will be
used
# You should not need to import anything else.
import numpy as np

# The main script
def main():
```

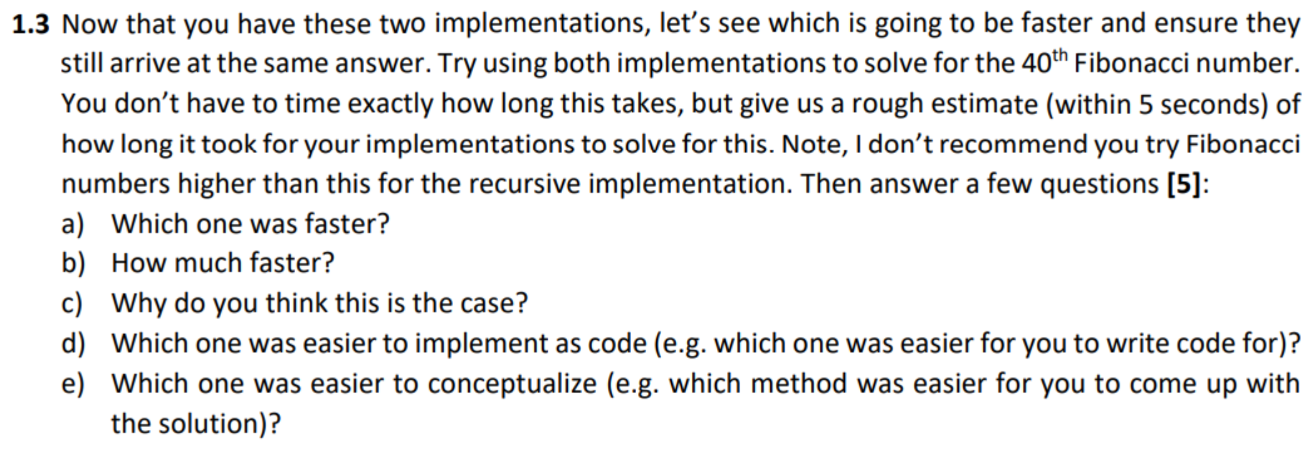
```
################################################################
###########
    # 1.2 - TODO

################################################################
###########
    # IMPLEMENT THE MATRIX VERSION OF FIBONACCI HERE. You just
need to fill in
    # the blank on this matrix. Some variables that are
provided:
    # n = the Nth fibonacci number that you want to solve for

    n = 40 # Which fibonacci number do you want
    mat_fib = np.array([[1,1],[1,0]])
    fib_base = np.array([[1],[1]]) # First and second fibonacci
numbers as a column vector

    fib_mul = np.linalg.matrix_power(mat_fib, n-2) #Multiply
mat_fib by itself n-2 times
    fib_n = np.matmul(fib_mul, fib_base) # Solve for the nth
and n-1th fibonacci numbers


################################################################
###########
    # 1.2 - END TODO

################################################################
###########

    print( fib_n[0] ) # Print out the nth Fibonacci number
```

```
"""
The following code allows us to run this file as a script.
Note, this not the
only way to do this, but the benefit of using this method is
that all the
variables that are created as part of the script have local
scope.
"""
if __name__ == "__main__":
    main()
```

The output of the code above is:

```
[102334155]

#According to the List of Fibonacci Numbers: f(40) = 102334155
#_list of Fibonacci numbers_. (2013). Planetmath.org.
https://planetmath.org/listoffibonaccinumbers
```

Both solutions have the same output.



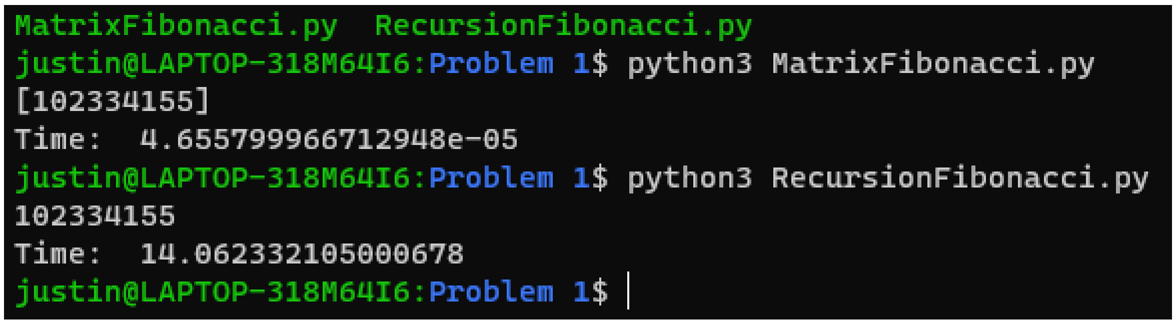
1.3 Now that you have these two implementations, let's see which is going to be faster and ensure they still arrive at the same answer. Try using both implementations to solve for the 40th Fibonacci number. You don't have to time exactly how long this takes, but give us a rough estimate (within 5 seconds) of how long it took for your implementations to solve for this. Note, I don't recommend you try Fibonacci numbers higher than this for the recursive implementation. Then answer a few questions **[5]**:

a) Which one was faster?
b) How much faster?
c) Why do you think this is the case?
d) Which one was easier to implement as code (e.g. which one was easier for you to write code for)?
e) Which one was easier to conceptualize (e.g. which method was easier for you to come up with the solution)?

The solution using matrix operations was faster than the Recursive solution. Using the "timeit" library, we can calculate the runtime for each solution. The following pieces of code were added to each solutions:

```python
import timeit

start = timeit.default_timer()

#Solution  here

stop = timeit.default_timer()
print('Time: ', stop - start)
```

The runtime for each solution as well as the output is listed below:



```
MatrixFibonacci.py  RecursionFibonacci.py
justin@LAPTOP-318M64I6:Problem 1$ python3 MatrixFibonacci.py
[102334155]
Time:  4.655799966712948e-05
justin@LAPTOP-318M64I6:Problem 1$ python3 RecursionFibonacci.py
102334155
Time:  14.062332105000678
justin@LAPTOP-318M64I6:Problem 1$ 
```

Runtime for 'RecursionFibonacci.py' = 14.0623s
Runtime for 'MatrixFibonacci.py' = 0.000046558s

$$'RecursionFibonacci'/'MatrixFibonacci' = 14.0623s/0.000046558s = 302\,038$$

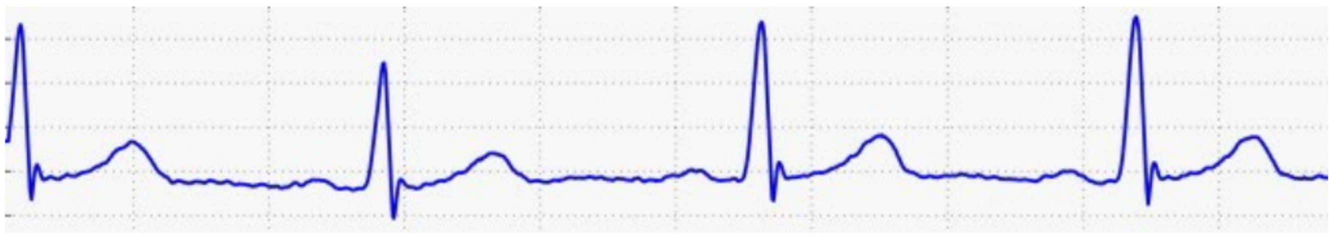
'MatrixFibonacci' is 300,000 times faster than 'RecursionFibonacci'. This is because matrix operations are much faster than looping recursively by calling the function over and over again. Moreover, the size of the matrix used in 'MatrixFibonacci' is very small (2x1 matrix), allowing for faster matrix operations and faster runtime.

In terms of which solution was easier to implement, 'MatrixFibonacci' was easier to implement as the general step was easier to find compared to the Recursive case in 'RecursionFibonacci'. 'MatrixFibonacci' did require the use of Numpy functions to calculate the matrix power but it was less complicated than calculating the Recursive case for 'RecursionFibonacci'.

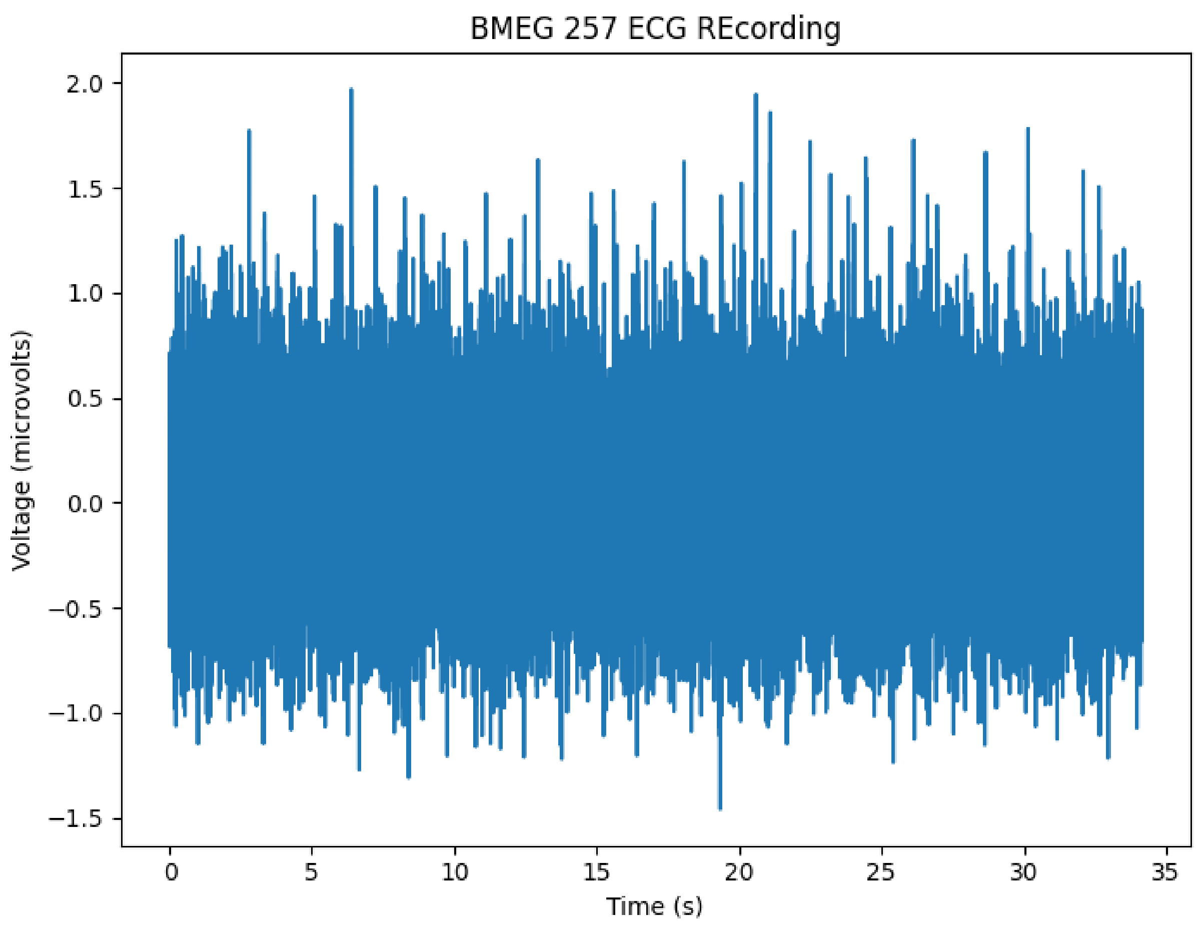
'MatrixFibonacci' was also easier to conceptualize as I had more experience with matrix operations compared to recursive operations.

Problem 2: Identifying Heart Beats using Cross Correlation.py

Introduction: Electrocardiography (ECG or EKG) sensors are often found in clinical settings and measure the electrical activity of the heart (in movies when things “flatline”, what you’re actually seeing is the cessation of electrical activity in the heart as measured by the ECG). ECGs are also becoming more commonplace in wearable devices as well, with many smart watches having a form of the single lead ECG implemented to monitor heart rate and heart rate variability. Here, we will explore how this is done using waveform template matching. Conceptually, these algorithms are shape-matching algorithms that identifies waveforms that “look” like a heartbeat. These algorithms tend to be relatively robust to noise and errors, which tend to be quite large when looking at wearable sensor data. This problem will introduce the simplest method for which this is done, known generally as cross-correlation, on a noisy ECG signal that might be collected from a wearable device. By the way here is what a “good” ECG looks like (with four heart beats).



2.1 First things first, let’s have a quick look at the data. The template for loading the data and plotting the data are already provided in the main script “HeartBeatDetectionScript” (.m or .py for matlab or python). This is the main script you will run throughout this program to track your progress, however much of the code will be placed in separate functions. You’ll notice that the plotting functions are commented out here (e.g. they will not be run). Uncomment the plotting lines and run the script to plot the ECG data. Copy the plot and show it here. What does the signal look like to you? Do you think you could figure out how many heartbeats are represented here? You can use this code for future reference on the syntax to plot things. **[2]**



```
ecg_data = LoadDataBinary("Data/ECG_Signal.bin")

fig1 = plt.figure(1)
fig1.clf()
ax = fig1.add_subplot(1,1,1)
ax.plot( ecg_data[:,0], ecg_data[:,1] )
ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
ax.set_title("BMEG 257 ECG REcording")
plt.show() #Added so that diagram popups
```

From the signal above, it is hard to make out the individual heartbeat waveforms, especially with all the noise. Also cannot figure out how many heartbeats there are represented in the signal just by looking at the signal above.

2.2 Now that you've plotted the data, let's take a look at the data itself. You'll notice that the data is housed as a matrix in a variable called "ecg_data". What are the dimensions of this matrix? What does each dimension represent? (hint: refer to the matrix programming lecture). **[3]**

Using the code snippet below from the numpy documentation,

```
print(ecg_data.size); #Prints the number of elements in the matrix
print(ecg_data.shape); #Prints the diemnsions (row, column) of the matrix
```

The output of the code snippet above is:

```
68282 #number of elements in matrix
(34141,2) #dimensions of matrix
```

This shows that the matrix 'ecg_data' , there are 68282 elements in the matrix and has 34141 rows and 2 columns. The first column represents time while the second column represents the voltage of the ECG signal in millivolts at a time specified in the first column.

$$\begin{bmatrix} Time(s) & Voltage(mV) \\ \dots & \dots \\ \dots & \dots \\ \dots & \dots \end{bmatrix}$$

$$\text{Total Elements} = 68282$$

2.3 The original file where this ECG data was stored is in binary form and has a size of 267kB (kiloBytes, in the Data folder as "ECG_Signal.bin"). Now we will have you save the data as a .csv file to demonstrate how the file size changes depending on how you represent the data. Before we do this, we will have you estimate how big the .csv file will be. To do this, assume each value in your "ecg_data" matrix will be stored using 6 characters in the final .csv file. Each character is represented using a single byte (1 byte = 8 bits). About how many bytes will your .csv file have (show your work)? Is this larger than the original binary file? (hint: refer to the data storage lecture) **[5]**

Original Binary File size: 267kB

$$\text{Total Elements} = 68282$$

Each element in the matrix will be stored using 6 characters in the .csv file,

$$68282 * 6 = 409\,692 \text{ charaters} = 409\,692 \text{ bytes}$$

The final .csv file will have an estimated size of $409\,692\ bytes \approx 410KB$ which is larger than the size of the binary form of 'ecg_data' since the numbers in the matrix are represented in binary in 'ECG_Signal.bin' while 'ECG_Signal.csv' is represented as a delimited text file.

2.4 Now, use matlab or python functions to save your data as a .csv file. How big is this file? Is it close to your estimate? Note, your estimate may be off because the code you use chose to represent data with more or fewer characters, check the file to see how many characters are being used to represent each data roughly. **[5]**

hint: you can see documentation here on how to save data as csv files for

matlab (writematrix or csvwrite): Write a matrix to a file - MATLAB writematrix (mathworks.com)

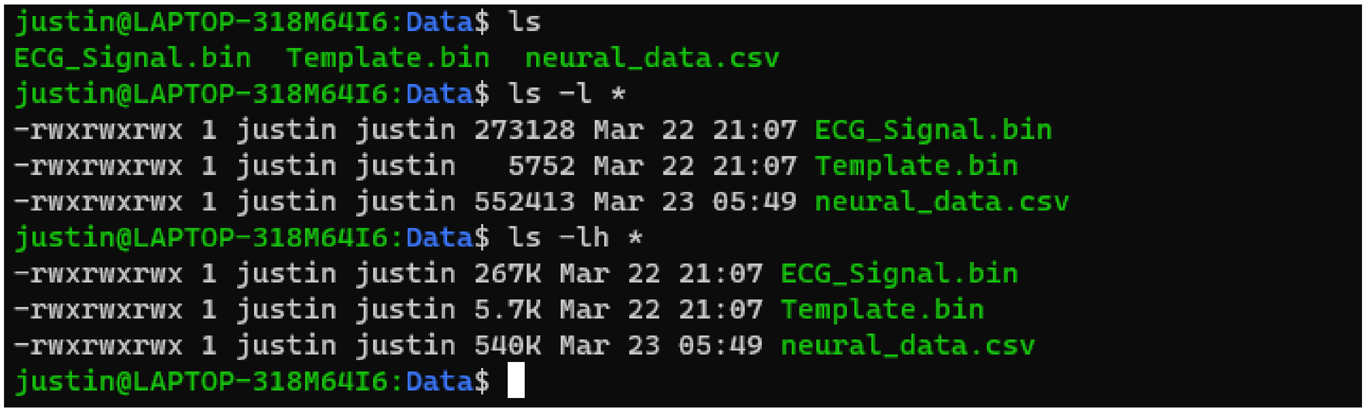
python (numpy's savetxt): numpy.savetxt — NumPy v1.19 Manual

Using the following code snippet from the numpy documentation:

```
filename - 'Data/neural_data.csv'
np.savetxt(filename, ecg_data, fmt='%.6g', delimiter=',')
```

The code snippet above could only limit the output data to be 6 significant figures since there is no way to limit the float value of the matrix to 6 characters (including negative sign and decimal point). The closest approximation to 6 characters for the float values is setting the value to 6 significant figures which is represented as $\%.6g$.

The 'neural_data.csv' file was then found under the 'Data' folder to which the corresponding file size for each file in the 'Data' folder was:



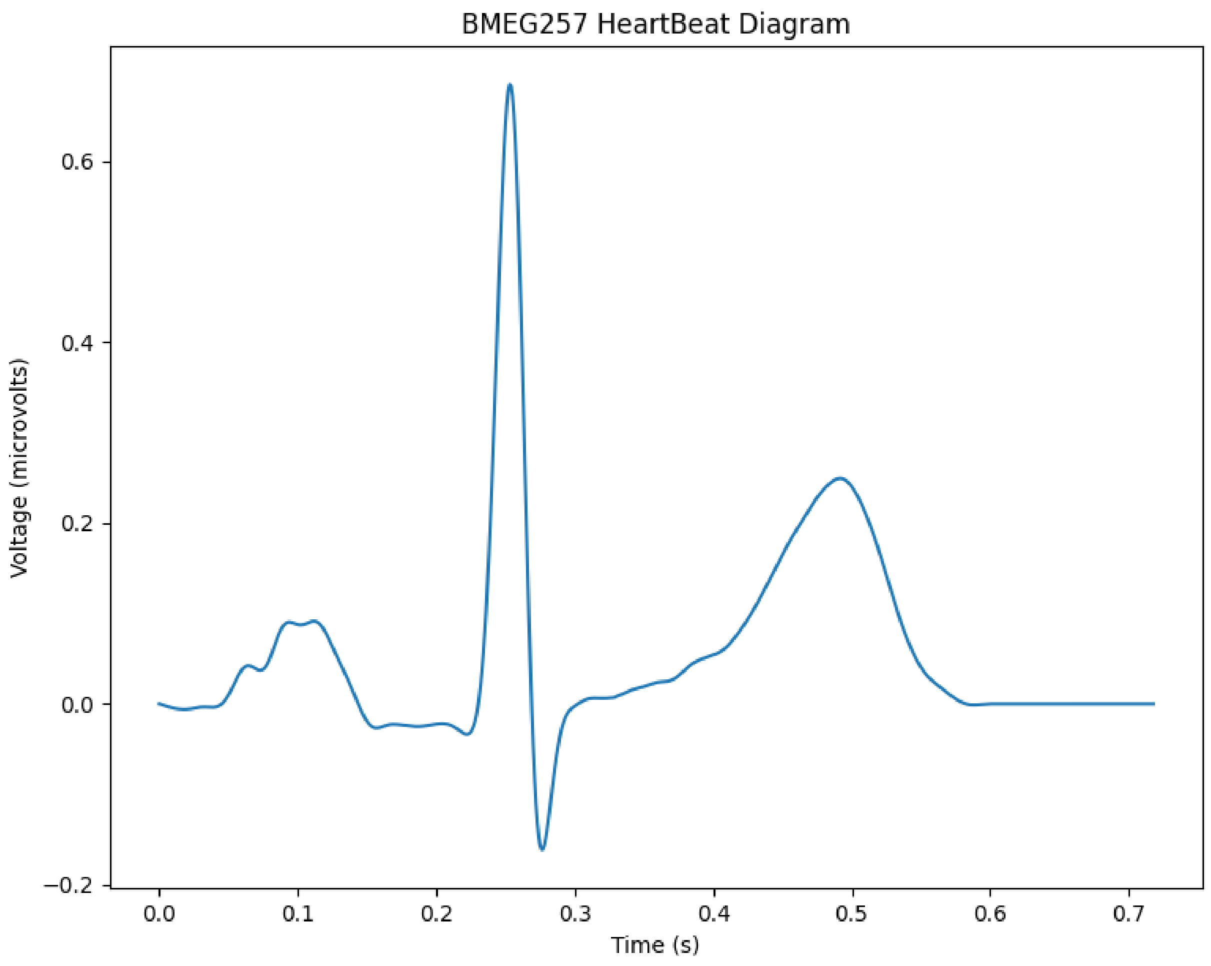
```
justin@LAPTOP-318M64I6:Data$ ls
ECG_Signal.bin  Template.bin  neural_data.csv
justin@LAPTOP-318M64I6:Data$ ls -l *
-rwxrwxrwx 1 justin justin 273128 Mar 22 21:07 ECG_Signal.bin
-rwxrwxrwx 1 justin justin   5752 Mar 22 21:07 Template.bin
-rwxrwxrwx 1 justin justin 552413 Mar 23 05:49 neural_data.csv
justin@LAPTOP-318M64I6:Data$ ls -lh *
-rwxrwxrwx 1 justin justin 267K Mar 22 21:07 ECG_Signal.bin
-rwxrwxrwx 1 justin justin 5.7K Mar 22 21:07 Template.bin
-rwxrwxrwx 1 justin justin 540K Mar 23 05:49 neural_data.csv
justin@LAPTOP-318M64I6:Data$
```

(Note: the top section is the file sizes represented in bytes; the bottom section are the file sizes scaled to KB, MB, GB or TB when the file size is larger than 1024 bytes)

$$\begin{matrix} ECG\_Signal.bin \\ neural\_data.csv \end{matrix} \rightarrow \begin{matrix} 273128\ bytes \\ 552413\ bytes \end{matrix} \approx \begin{matrix} 267\ kiloBytes \\ 540\ kiloBytes \end{matrix}$$

Compared to the previous approximation made for the .csv file, which was 410kB, the actual 'neural_data.csv' file is larger than the approximate csv file size by 130KB. However, it is a close approximation considering the fact that if there was no format specifier in 'np.savetxt', the size would be approximately 1.7MB.

2.5 Now let's start to work on the cross correlation. For this problem, we are providing you with the heartbeat shape that is recorded in the "ecg_data" signal. The data for the step is loaded into the script in as "heartbeat". You'll notice the that these data matrices have a similar organization as the "ecg_data" signal. First, let's plot this to see what a step looks like. Paste the plot here. **[5]**



Code snippet below graphs the Heartbeat diagram shown above

```
heartbeat = LoadDataBinary( 'Data/template.bin' );

#Plotting
fig2 = plt.figure(1)
fig2.clf()
ax = fig2.add_subplot(1,1,1)
ax.plot( heartbeat[:,0], heartbeat[:,1] )
ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
ax.set_title("BMEG257 HeartBeat Diagram")
plt.show() #Added for diagram popup
```

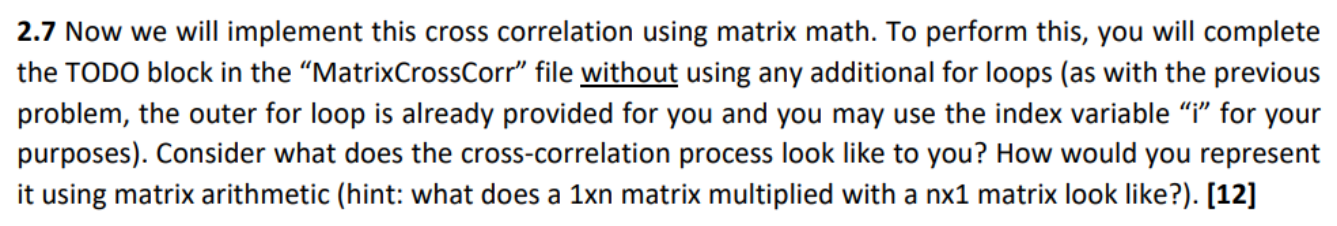
2.6 Now we will actually write an implementation of cross correlation. Cross correlation is a mathematical process that is useful for checking the similarity between two signals. In our case, we are trying to identify when the ECG data shows a pattern matching a heart beat. We can use cross correlation here to compare a what a heart beat looks like with a piece of the ECG signal, and when these match, we can claim that the heart likely beat at that point (see Figure 2). To compute the cross correlation, we need to take a section of the ECG recording that is the same length as the heart beat template. From there, we can perform the cross correlation, which is computed by taking each element in the heart beat template and multiplying it with the matched element in the ECG data segment, and then summing these products together. A perfect match would result in a large cross correlation, whereas a weak match would have a low cross correlation (Figure 2).

1. Take a section of the ECG recording that is the same length as the heartbeat template
2. Take each element in the heartbeat template and multiply with the matched element in the ECG data segment (dot product between heartbeat template and ECG data segment)
3. Sum the products together (dot product)

Large cross correlation -> perfect match
Low cross correlation -> weak match

```
import numpy as np

def ForLoopCrossCorr(ecg_data, heartbeat):
        # The outer for loop is written for you as it cycles
through the ecg data
        samples = ecg_data.shape[0];
        heartbeat_samples = heartbeat.shape[0];

        cross_data = np.zeros( [(samples - heartbeat_samples),
1] );
        for i in range( samples - heartbeat_samples ):

###############################################################
########
                # 2.6 - TODO
```

```
###############################################################
########
                # IMPLEMENT THE FOR LOOP CROSS CORRELATION
HERE.

                #only need to implement the inner part of the
cross correlation by finding
                #dot product between ecg_data and template ecg
matrix
                sum = 0;
                counter = 0;
                for j in range(i, i+heartbeat_samples):
                        sum += ecg_data[j]*heartbeat[counter]
                        counter+=1

                cross_correlation= sum;
        #######################################################
################
        # 2.6 - END TODO

###############################################################
########
        cross_data[i] = cross_correlation;
        return cross_data
```


2.7 Now we will implement this cross correlation using matrix math. To perform this, you will complete the TODO block in the "MatrixCrossCorr" file without using any additional for loops (as with the previous problem, the outer for loop is already provided for you and you may use the index variable "i" for your purposes). Consider what does the cross-correlation process look like to you? How would you represent it using matrix arithmetic (hint: what does a 1xn matrix multiplied with a nx1 matrix look like?). **[12]**

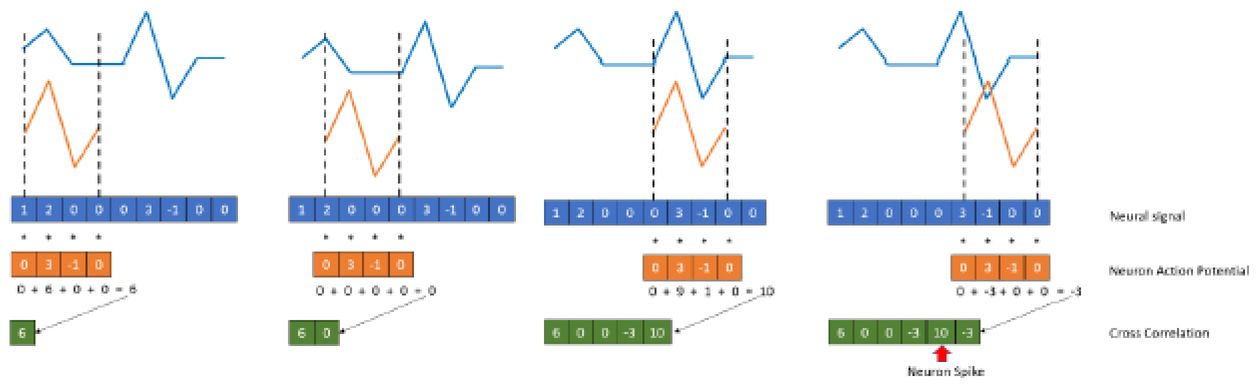
The matrix operation for each element in 'Cross-Data' is the dot product betweeen 'heartbeat' matrix and 'ecg_data' matrix (which has been cut to fit the dimensions of 'heartbeat' matrix).

$$cross[0] = \begin{pmatrix} heartbeat[0] \\ heartbeat[1] \\ \dots \\ heartbeat[n] \end{pmatrix} * \begin{pmatrix} ecg_d ata[0] \\ ecg_d ata[1] \\ \dots \\ ecg_d ata[n] \end{pmatrix}$$

The above matrix operattion is represented in the code snippet below:

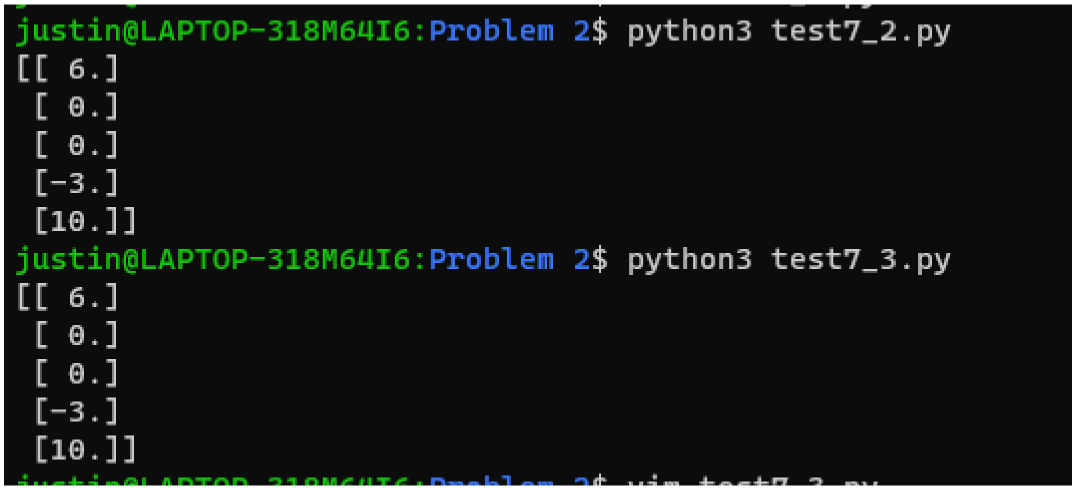
```
import numpy as np

def MatrixCrossCorr(ecg_data, heartbeat):
    samples = ecg_data.shape[0]; #Gets the size of ecg_data
matrix
    heartbeat_samples = heartbeat.shape[0]; #Gets the size of
heartbeat matrix
    cross_data = np.zeros( [(samples - heartbeat_samples), 1]
);

    for i in range( samples - heartbeat_samples ):
        #create a new array for ecg_data with the same size as
heartbeat array
        #calculate the dot product between ecg_data and
heartbeat array
        temp_matrix = ecg_data[i:i+heartbeat_samples]
        sum = 0
        sum = np.dot(temp_matrix, heartbeat)
        cross_data[i] = sum
    return cross_data
```

Given the test case from Q2.6 where the 'ecg_data' matrix and 'heartbeat' matrix is given as:



```python
def main():
        heartbeat = np.array([0, 3, -1, 0])
        ecg_data = np.array([1, 2, 0, 0, 0, 3, -1, 0, 0])
        print(MatrixCrossCorr(ecg_data, heartbeat));
```

The test case yielded the output 'cross-data matrix' to be:



```
justin@LAPTOP-318M64I6:Problem 2$ python3 test7_2.py
[[ 6.]
 [ 0.]
 [ 0.]
 [-3.]
 [10.]]
justin@LAPTOP-318M64I6:Problem 2$ python3 test7_3.py
[[ 6.]
 [ 0.]
 [ 0.]
 [-3.]
 [10.]]
```

$$cross\_data = \begin{bmatrix} 6 & 0 & 0 & -3 & 10 \end{bmatrix}^T$$

```python
import numpy as np

def main():
    temp1 = np.array([0, 3, -1, 0]) #transposing since matrix
in Heartbeat is column vector
    temp2 = np.array([1, 2, 0, 0, 0, 3, -1, 0, 0])
    heartbeat = np.transpose(temp1)
    ecg_data = np.transpose(temp2)
    #print(ecg_data.shape[0])
    #print(heartbeat.shape[0])
    print(MatrixCrossCorr(ecg_data, heartbeat));

def MatrixCrossCorr(ecg_data, heartbeat):
    samples = ecg_data.shape[0]; #Gets the size of ecg_data
matrix
    heartbeat_samples = heartbeat.shape[0]; #Gets the size of
heartbeat matrix
    cross_data = np.zeros( [(samples - heartbeat_samples), 1]
);
    for i in range( samples - heartbeat_samples ):
        #create a new array for ecg_data with the same size as
heartbeat array
        #calculate the dot product between ecg_data and
heartbeat array
        #TODO
        temp_matrix = ecg_data[i:i+heartbeat_samples]
        HBTransposed = np.transpose(heartbeat)
        #prod = temp_matrix @ HBTransposed #matrix multiply
with @
        prod = np.matmul(temp_matrix, HBTransposed)
        cross_correlation = np.sum(prod)
        cross_data[i] = cross_correlation
        #TODO
    return cross_data
```

2.8 Both the "ForLoopCrossCorr" and "MatrixCrossCorr" files generate a vector of cross correlation values (the green vector in Figure 2) representing comparisons of the heart beat template against segments of the ECG signal. But alone, those vectors do not really tell you where a heart beat is. Here, we will have you complete the TODO block in "ThresholdBeats" to identify when the cross correlation values exceed a threshold and can be considered a heart beat. There are several ways to implement this and this is where you can get a bit creative (simplest: you can say a heart beat occurs whenever the cross correlation is above a threshold, currently set at 7 because I know this works, but you are welcome to play around with this to see how it changes the results). What is important is that whenever you detect a heart beat, you set the appropriate index in the "beats" variable to one (by default, everything is set to 0 so no heart beats anywhere). When you complete this code and run the main "HeartBeatDetectionScript" file, it will automatically generate figures depicting these heart beats using both the for loop method (figure 10) and the matrix method (figure 11). Did both of your methods produce the same results according to the figures (they should!)? Based on the output figures, are the predicted heart beats the same? The ECG signal we used here is a simulated arrythmia which is usually identifiable by the fact that the heart rate is not consistent (e.g. the timing between heart beats is variable). Do you think the predicted heart beat patterns confirm this? Please copy both output figures here (figure 10 and figure 11) for full credit. [6]

The code for the above problem is shown below:

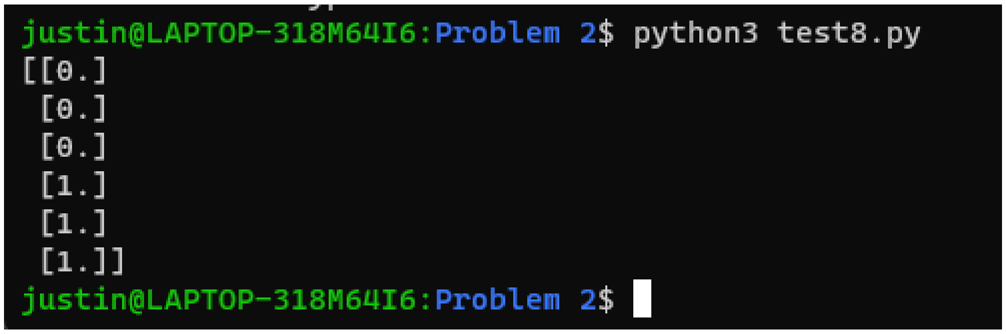
```
import numpy as np

def ThresholdBeats( cross_corr ):

    # Setup the output for spike train
    beats = np.zeros( [cross_corr.shape[0], 1] )

###############################################################
############
    # 2.8 - TODO

###############################################################
############
    # Implement your thresholding here. The beats vector above
    # is already set up as your output and is the same size as
the input
    # cross correlation. When you identify a heartbeat (e.g.
cross correlation
    # exceeds threshold), set the corresponding location in
beats to
```

```
    # 1.
    threshold = 7;
    beats = np.zeros( [cross_corr.shape[0], 1] )
    for i in range(beats.shape[0]):
        if cross_corr[i] >= threshold:
            beats[i] = 1;
    return beats


################################################################
############
    # 2.8 - END TODO

################################################################
############

    return beats
```

Testing the Code snippet above with a test matrix:

```
def main():
    test_matrix = np.array([6, 1, 5,7 ,8, 10])
    #cross_corr = np.transpose(temp_matrix)
    print(ThresholdBeats(test_matrix))
```

The output yields:

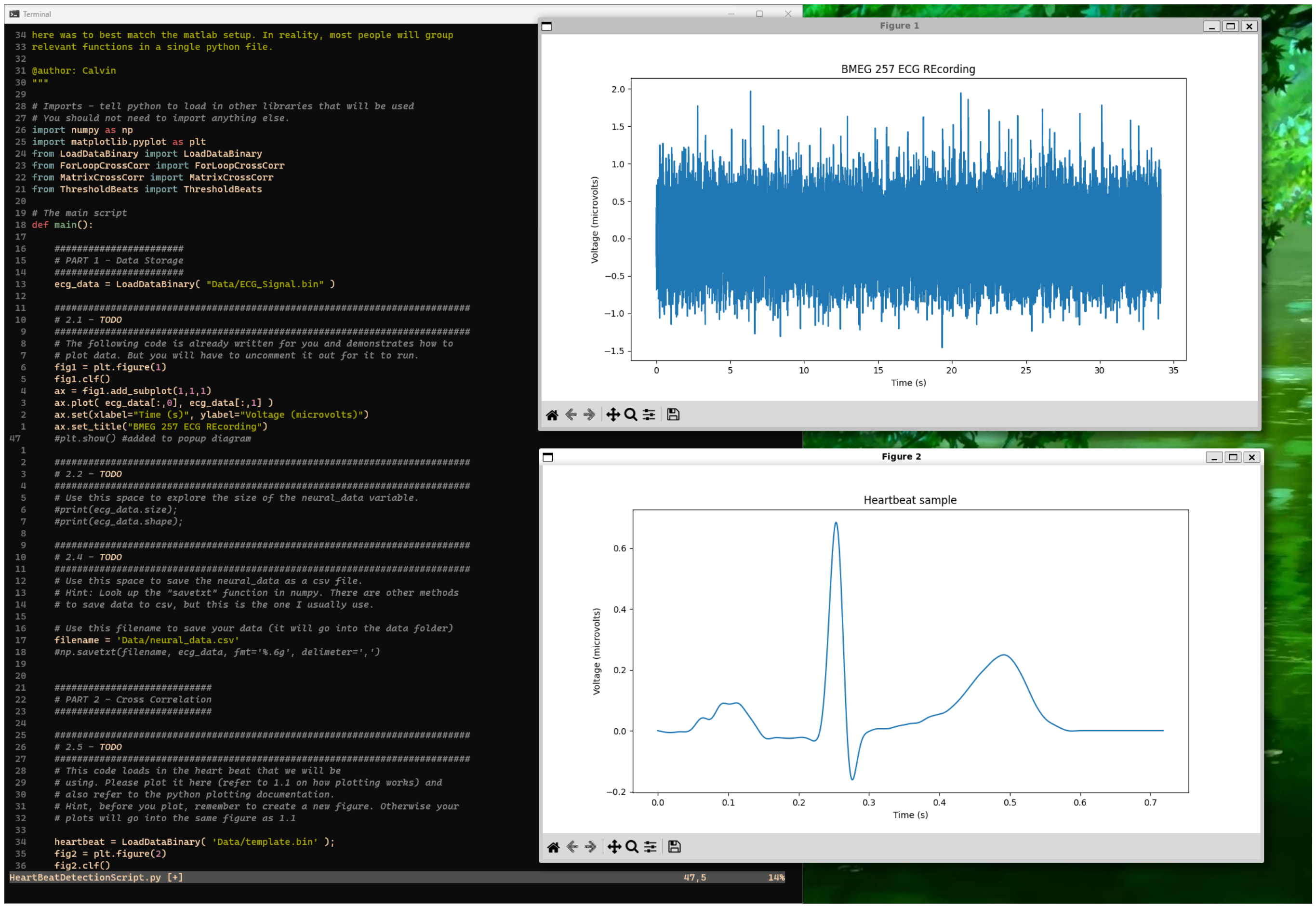


```
justin@LAPTOP-318M64I6:Problem 2$ python3 test8.py
[[0.]
 [0.]
 [0.]
 [1.]
 [1.]
 [1.]]
justin@LAPTOP-318M64I6:Problem 2$
```

Which is correct since we want all values that exceed the threshold (of 7) to correspond to a 1 in the beats matrix.

Final Heartbeats

The final and compiled diagrams are shown below after running 'HeartBeatDetectionScipt.py'.
Below is a code snippet as well as the raw ecg recording and a template for the heartbeat sample.



```
here was to best match the matlab setup. In reality, most people will group
relevant functions in a single python file.

@author: Calvin
"""

# Imports - tell python to load in other libraries that will be used
# You should not need to import anything else.
import numpy as np
import matplotlib.pyplot as plt
from LoadDataBinary import LoadDataBinary
from ForLoopCrossCorr import ForLoopCrossCorr
from MatrixCrossCorr import MatrixCrossCorr
from ThresholdBeats import ThresholdBeats

# The main script
def main():

    #######################
    # PART 1 - Data Storage
    #######################
    ecg_data = LoadDataBinary( "Data/ECG_Signal.bin" )

    ############################################################################
    # 2.1 - TODO
    ############################################################################
    # The following code is already written for you and demonstrates how to
    # plot data. But you will have to uncomment it out for it to run.
    fig1 = plt.figure(1)
    fig1.clf()
    ax = fig1.add_subplot(1,1,1)
    ax.plot( ecg_data[:,0], ecg_data[:,1] )
    ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
    ax.set_title("BMEG 257 ECG REcording")
    #plt.show() #added to popup diagram

    ############################################################################
    # 2.2 - TODO
    ############################################################################
    # Use this space to explore the size of the neural_data variable.
    #print(ecg_data.size);
    #print(ecg_data.shape);

    ############################################################################
    # 2.4 - TODO
    ############################################################################
    # Use this space to save the neural_data as a csv file.
    # Hint: Look up the "savetxt" function in numpy. There are other methods
    # to save data to csv, but this is the one I usually use.

    # Use this filename to save your data (it will go into the data folder)
    filename = 'Data/neural_data.csv'
    #np.savetxt(filename, ecg_data, fmt='%.6g', delimeter=',')


    ############################
    # PART 2 - Cross Correlation
    ############################

    ############################################################################
    # 2.5 - TODO
    ############################################################################
    # This code loads in the heart beat that we will be
    # using. Please plot it here (refer to 1.1 on how plotting works) and
    # also refer to the python plotting documentation.
    # Hint, before you plot, remember to create a new figure. Otherwise your
    # plots will go into the same figure as 1.1

    heartbeat = LoadDataBinary( 'Data/template.bin' );
    fig2 = plt.figure(2)
    fig2.clf()
HeartBeatDetectionScript.py [+]                                    47,5        14%
```

Figure 1 shows raw unfiltered ECG recording
Figure 2 shows a template for a heartbeat used for cross correlation

Figure 10: Number of Heartbeats counted using ForLoop Cross Correlation

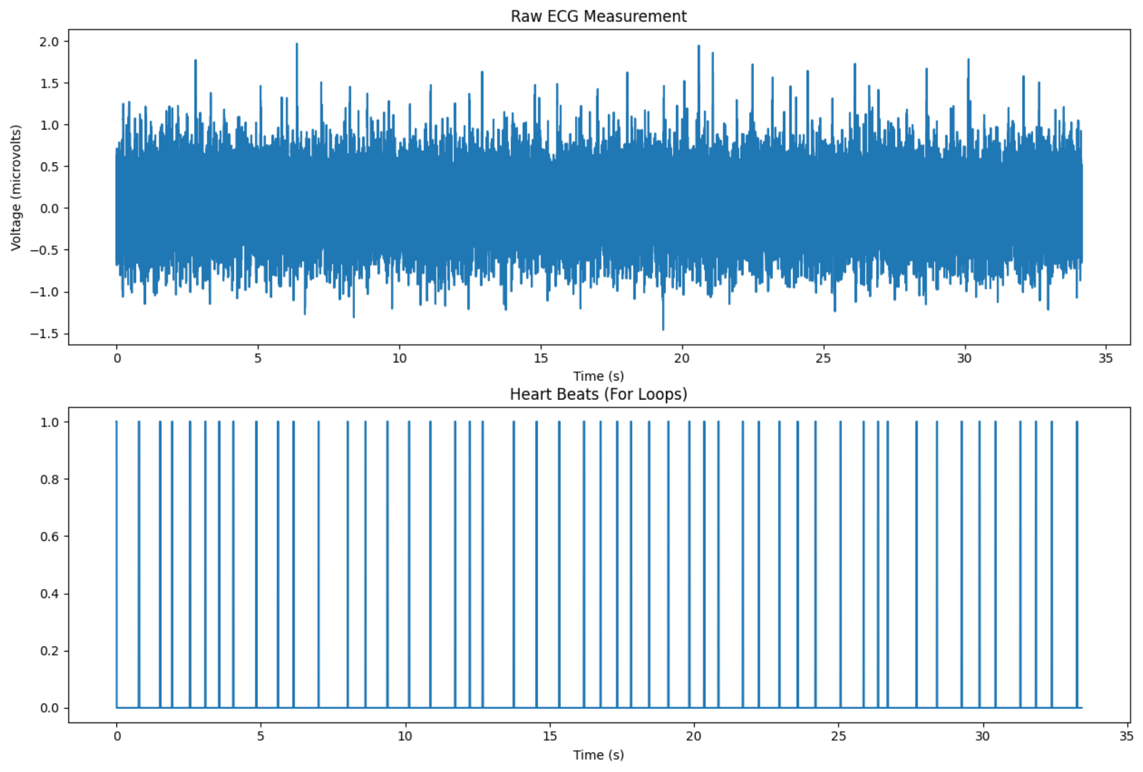
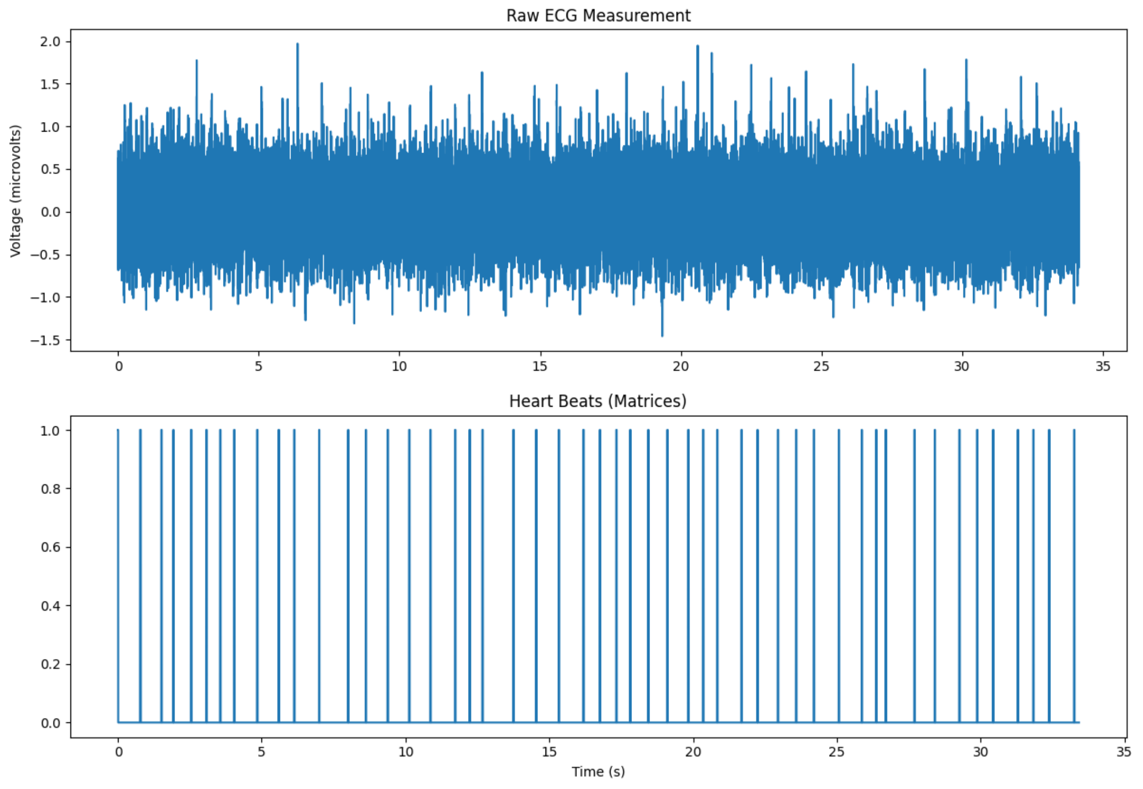
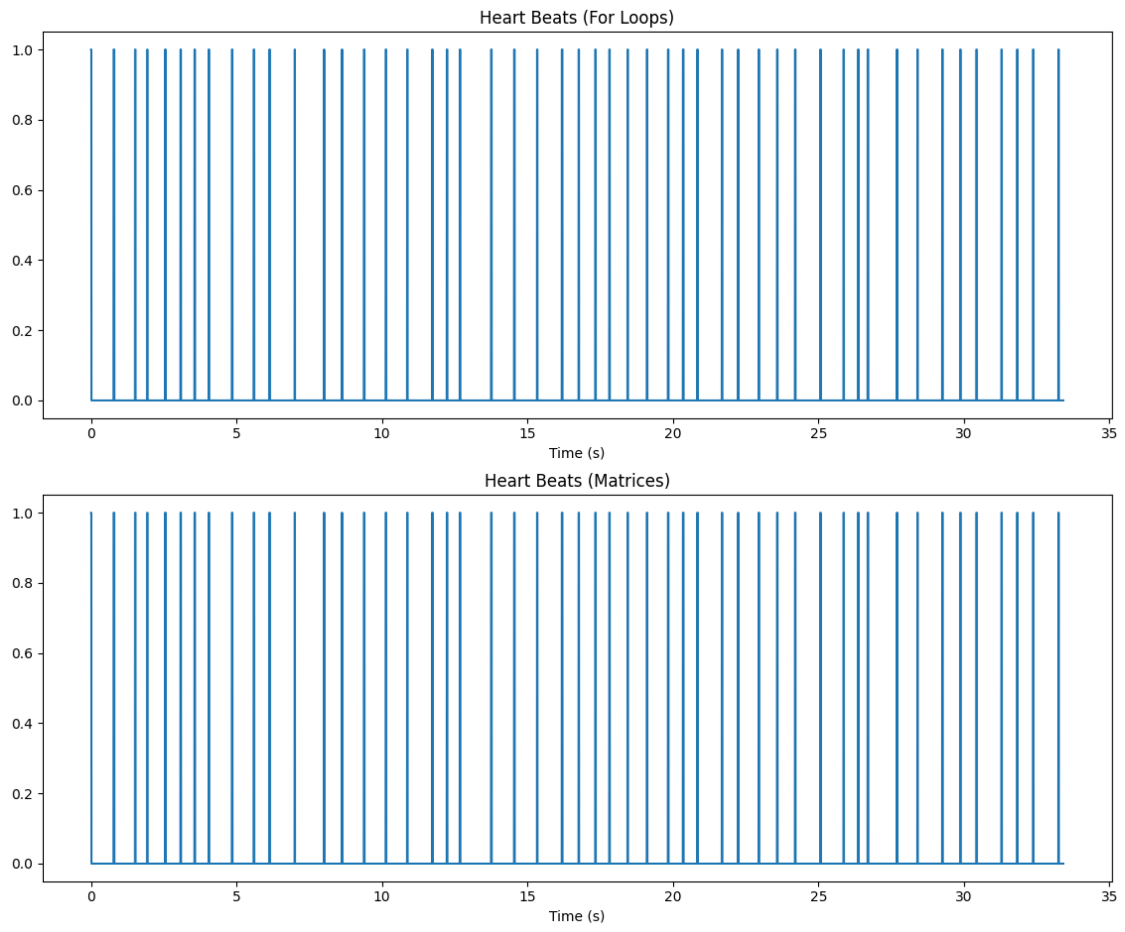


Figure 11: Number of Heartbeats counted using Matrix Cross Correlation



Raw ECG recording, shown above the number of heartbeats, does not predict the heartbeat patterns. Number of heartbeats is not consistent. According to Figure 10 and Figure 11, both methods produced the same results as the predicted heartbeats from both methods yielded the same output. To show further that both methods yielded the same output. Another graph was made to compare 'HeartBeat(For Loops)' and 'HeartBeat(Matrices)'.

Figure 12: Predicted Heartbeats from both methods; For Loop and Matrices



As noted by the graph above, the predicted number of heartbeats from both methods are the same, showing arrythmic (inconsistent) heart beating.

Full Code for HeartBeatDetectionScript.py

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 22 16:23:44 2021

File: HeartBeatDetectionScript.py
Author: You!
Date: 2/23/2023

This is the main script for Problem 2 of the programming
assignment in BMEG 257
This script has various "TODO" sections where you will be
required to fill
in code. This script also relies on a number of functions
located in other
files that you need to develop in order to run properly. Note,
the file setup
here was to best match the matlab setup. In reality, most
people will group
relevant functions in a single python file.

@author: Calvin
"""


# Imports - tell python to load in other libraries that will be
used
# You should not need to import anything else.
import numpy as np
import matplotlib.pyplot as plt
from LoadDataBinary import LoadDataBinary
from ForLoopCrossCorr import ForLoopCrossCorr
from MatrixCrossCorr import MatrixCrossCorr
from ThresholdBeats import ThresholdBeats
```

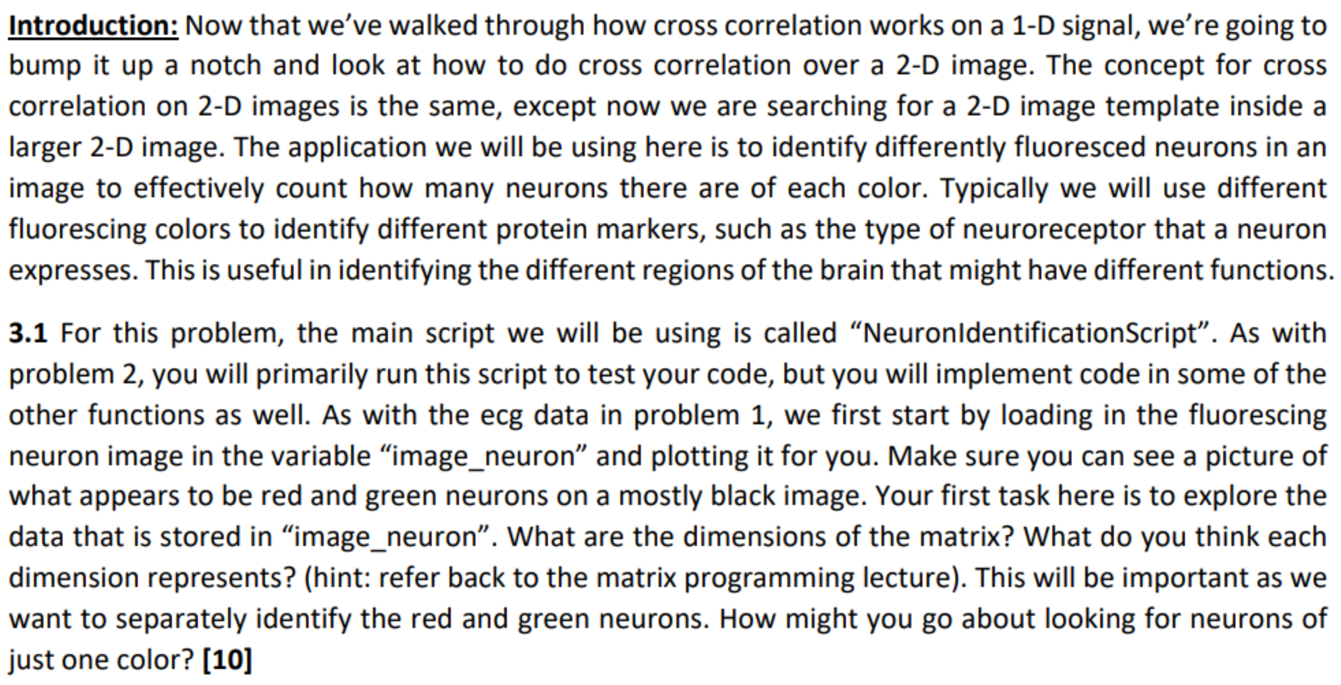
```python
# The main script
def main():

    ########################
    # PART 1 - Data Storage
    ########################
    ecg_data = LoadDataBinary( "Data/ECG_Signal.bin" )


##################################################################
###########
    # 2.1 - TODO

##################################################################
###########
    # The following code is already written for you and
demonstrates how to
    # plot data. But you will have to uncomment it out for it
to run.
    fig1 = plt.figure(1)
    fig1.clf()
    ax = fig1.add_subplot(1,1,1)
    ax.plot( ecg_data[:,0], ecg_data[:,1] )
    ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
    ax.set_title("BMEG 257 ECG REcording")
    #plt.show() #added to popup diagram


##################################################################
###########
    # 2.2 - TODO

##################################################################
###########
    # Use this space to explore the size of the neural_data
```

```
variable.
    #print(ecg_data.size);
    #print(ecg_data.shape);


##################################################################
###########
    # 2.4 - TODO

##################################################################
###########
    # Use this space to save the neural_data as a csv file.
    # Hint: Look up the "savetxt" function in numpy. There are
other methods
    # to save data to csv, but this is the one I usually use.

    # Use this filename to save your data (it will go into the
data folder)
    filename = 'Data/neural_data.csv'
    #np.savetxt(filename, ecg_data, fmt='%.6g', delimeter=',')


    ############################
    # PART 2 - Cross Correlation
    ############################


##################################################################
###########
    # 2.5 - TODO

##################################################################
###########
    # This code loads in the heart beat that we will be
    # using. Please plot it here (refer to 1.1 on how plotting
works) and
    # also refer to the python plotting documentation.
    # Hint, before you plot, remember to create a new figure.
Otherwise your
    # plots will go into the same figure as 1.1

    heartbeat = LoadDataBinary( 'Data/template.bin' );
    fig2 = plt.figure(2)
    fig2.clf()
    ax = fig2.add_subplot(1,1,1)
    ax.plot( heartbeat[:,0], heartbeat[:,1] )
    ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
    ax.set_title("Heartbeat sample")
    #plt.show() #Added for diagram popup


################################################################
###########
    # 2.6 - TODO

################################################################
###########
    # DO NOT ADJUST CODE HERE. OPEN THE FILE
"ForLoopCrossCorr.py" AND PUT YOUR
    # CODE THERE.

    for_cross_data = ForLoopCrossCorr( ecg_data[:,1],
heartbeat[:,1] );


################################################################
###########
    # 2.7 - TODO

################################################################
```

```
###########
    # DO NOT ADJUST CODE HERE. OPEN THE FILE
"MatrixCrossCorr.py" AND PUT YOUR
    # CODE THERE.

    mat_cross_data = MatrixCrossCorr( ecg_data[:,1],
heartbeat[:,1] );



###############################################################
###########
    # 2.8 - TODO

###############################################################
###########
    # DO NOT ADJUST CODE HERE. OPEN THE FILE
"ThresholdBeats.py" AND PUT YOUR
    # CODE THERE.

    for_beats = ThresholdBeats( for_cross_data );
    mat_beats = ThresholdBeats( mat_cross_data );



###############################################################
###########
    # 2.8 - END TODO

###############################################################
###########
    # THIS CODE WILL HELP VISUALIZE YOUR HEART BEATS. DO NOT
ADJUST.
    fig10 = plt.figure(10);
    fig10.clf()
    ax = fig10.add_subplot(2,1,1)
    ax.plot( ecg_data[:,0], ecg_data[:,1] )
    ax.set(xlabel="Time (s)", ylabel="Voltage (microvolts)")
    ax.set_title( "Raw ECG Measurement" );

    ax = fig10.add_subplot(2,1,2)
    ax.plot( ecg_data[:for_cross_data.shape[0],0],
for_beats[:,0] )
    ax.set(xlabel="Time (s)")
    ax.set_title( 'Heart Beats (For Loops)' );

    fig11 = plt.figure(11)
    fig11.clf()
    ax = fig11.add_subplot(2,1,1)
    ax.plot( ecg_data[:,0], ecg_data[:,1] )
    ax.set( ylabel = "Voltage (microvolts)");
    ax.set_title( "Raw ECG Measurement" );

    ax = fig11.add_subplot( 2,1,2 )
    ax.plot( ecg_data[:mat_cross_data.shape[0],0],
mat_beats[:,0] )
    ax.set( xlabel = "Time (s)");
    ax.set_title( 'Heart Beats (Matrices)' );

    fig12 = plt.figure(12)
    fig12.clf()
    ax = fig12.add_subplot(2,1,1)
    ax.plot( ecg_data[:for_cross_data.shape[0],0],
for_beats[:,0] )
    ax.set(xlabel="Time (s)")
    ax.set_title( 'Heart Beats (For Loops)' );

    ax = fig12.add_subplot( 2,1,2 )
    ax.plot( ecg_data[:mat_cross_data.shape[0],0],
mat_beats[:,0] )
```

```
    ax.set( xlabel = "Time (s)");
    ax.set_title( 'Heart Beats (Matrices)' );
    plt.show() #Added for display all figures



"""
The following code allows us to run this file as a script.
Note, this not the
only way to do this, but the benefit of using this method is
that all the
variables that are created as part of the script have local
scope.
"""
if __name__ == "__main__":
    main()
```

Problem 3: Finding Neurons in an Image

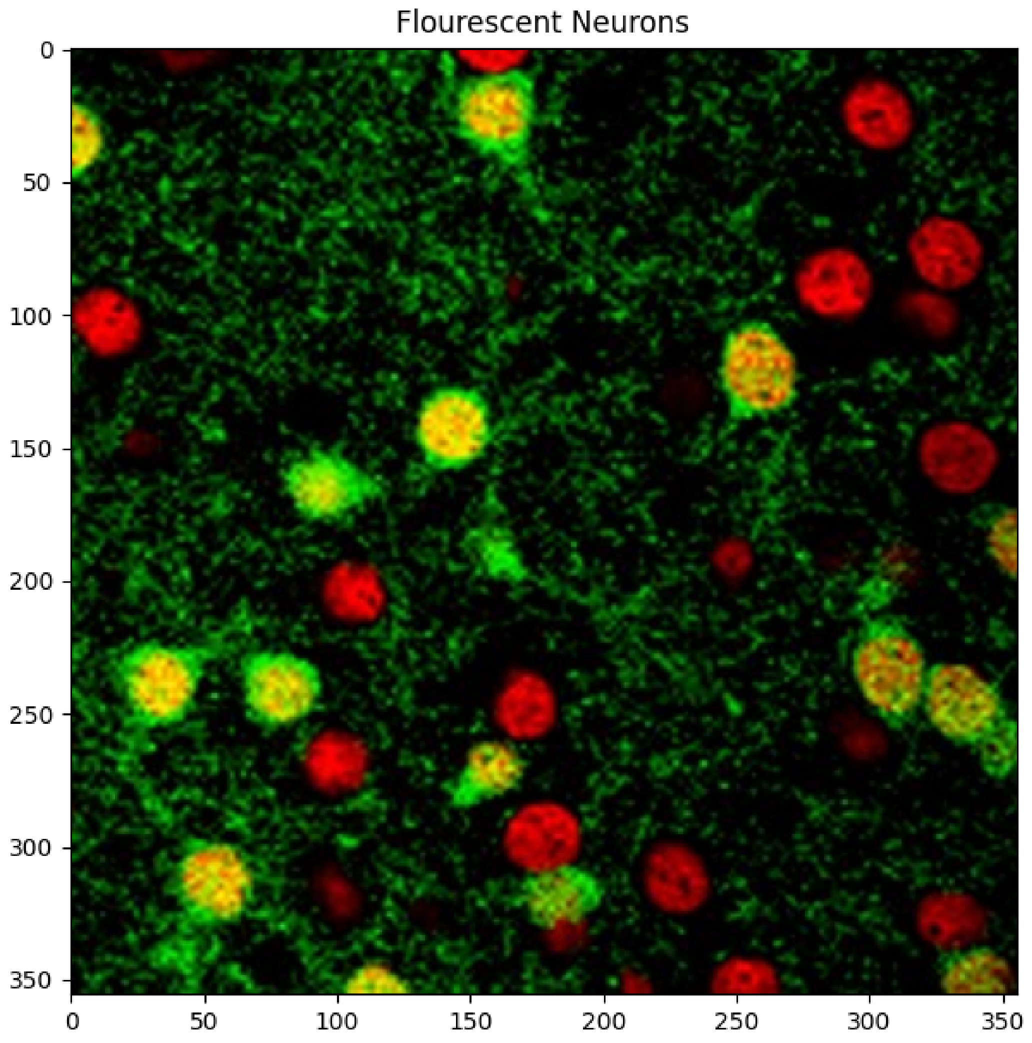


Introduction: Now that we've walked through how cross correlation works on a 1-D signal, we're going to bump it up a notch and look at how to do cross correlation over a 2-D image. The concept for cross correlation on 2-D images is the same, except now we are searching for a 2-D image template inside a larger 2-D image. The application we will be using here is to identify differently fluoresced neurons in an image to effectively count how many neurons there are of each color. Typically we will use different fluorescing colors to identify different protein markers, such as the type of neuroreceptor that a neuron expresses. This is useful in identifying the different regions of the brain that might have different functions.

3.1 For this problem, the main script we will be using is called "NeuronIdentificationScript". As with problem 2, you will primarily run this script to test your code, but you will implement code in some of the other functions as well. As with the ecg data in problem 1, we first start by loading in the fluorescing neuron image in the variable "image_neuron" and plotting it for you. Make sure you can see a picture of what appears to be red and green neurons on a mostly black image. Your first task here is to explore the data that is stored in "image_neuron". What are the dimensions of the matrix? What do you think each dimension represents? (hint: refer back to the matrix programming lecture). This will be important as we want to separately identify the red and green neurons. How might you go about looking for neurons of just one color? **[10]**

Running the code snippet below to show the full picture of red and green neurons on a mostly black image.

```
image_neuron = img.imread( 'Data/NeuronsImage.jpg' );

fig1 = plt.figure(1);
fig1.clf()
ax = fig1.add_subplot(1,1,1)
plt.imshow(image_neuron);
ax.set_title( 'Flourescent Neurons' );
plt.show() #Added to display all open figues
```

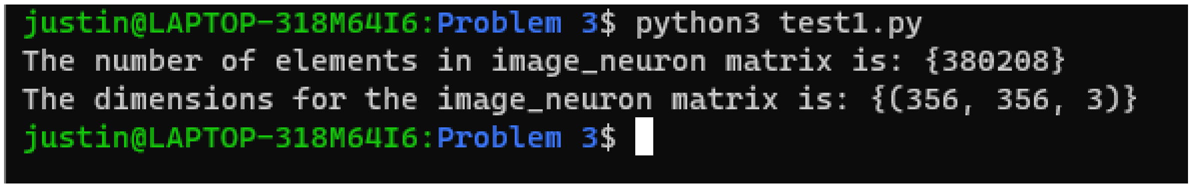


Above picture show red and green neurons on a mostly black background.
To find the size of the 'image_neuron' matrix, i added to the code snippet

below that prints the number of elements as well as the dimensions of 'image_neuron' matrix.

```
################################################################
############
# 3.1 - TODO
################################################################
############
# Use this space to explore the size of the image_neuron
variable.
print('The number of elements in image_neuron matrix is:',
{image_neuron.size})
print('The dimensions for the image_neuron matrix is:',
{image_neuron.shape})
```

The output of the print statements above is shown below:



```
justin@LAPTOP-318M64I6:Problem 3$ python3 test1.py
The number of elements in image_neuron matrix is: {380208}
The dimensions for the image_neuron matrix is: {(356, 356, 3)}
justin@LAPTOP-318M64I6:Problem 3$
```

```
The number of elements in image_neuron matrix is: {380208}
The dimensions for image_neuron matrix is {(356, 356, 3)}
```

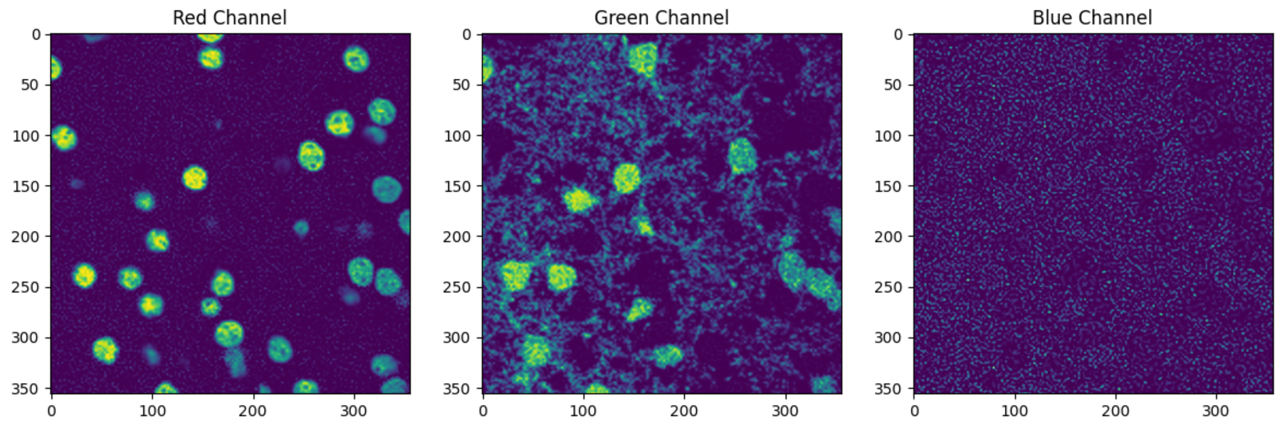
$$image\_neuron = (356 \text{ x } 356 \text{ x } 3) \text{ matrix}$$

Since it is a static image, there are three elements in the dimensions of 'image_neuron' that represent the resolution and number of colour channels.
The first two dimensions show the **resolution** of the colour channels. The size of one of the colour channels (red, green, blue) is 356 x 356. The last element shows the **number of colour channels** there are that represent the picture of neurons above, 3 colour channels show that there is a red, green and blue channel that make up the image.

When looking for neurons of a specific colour, we need to specify the colour channel to whether we want to find a neuron in the red, green or blue channel.
We can separate the colour channels from the image if we specify which colour channel we want. Using the code snippet below:

```
fig1 = plt.figure(1);
fig1.clf()
ax = fig1.add_subplot(1,3,1)
plt.imshow(image_neuron[:, :, 0]);
ax.set_title( 'Red Channel' );

ax = fig1.add_subplot(1,3,2)
plt.imshow(image_neuron[:, :, 1]);
ax.set_title( 'Green Channel' );

ax = fig1.add_subplot(1,3,3)
plt.imshow(image_neuron[:, :, 2]);
ax.set_title( 'Blue Channel' );
plt.show() #Added to display all open figues
```

We get the output image to be:



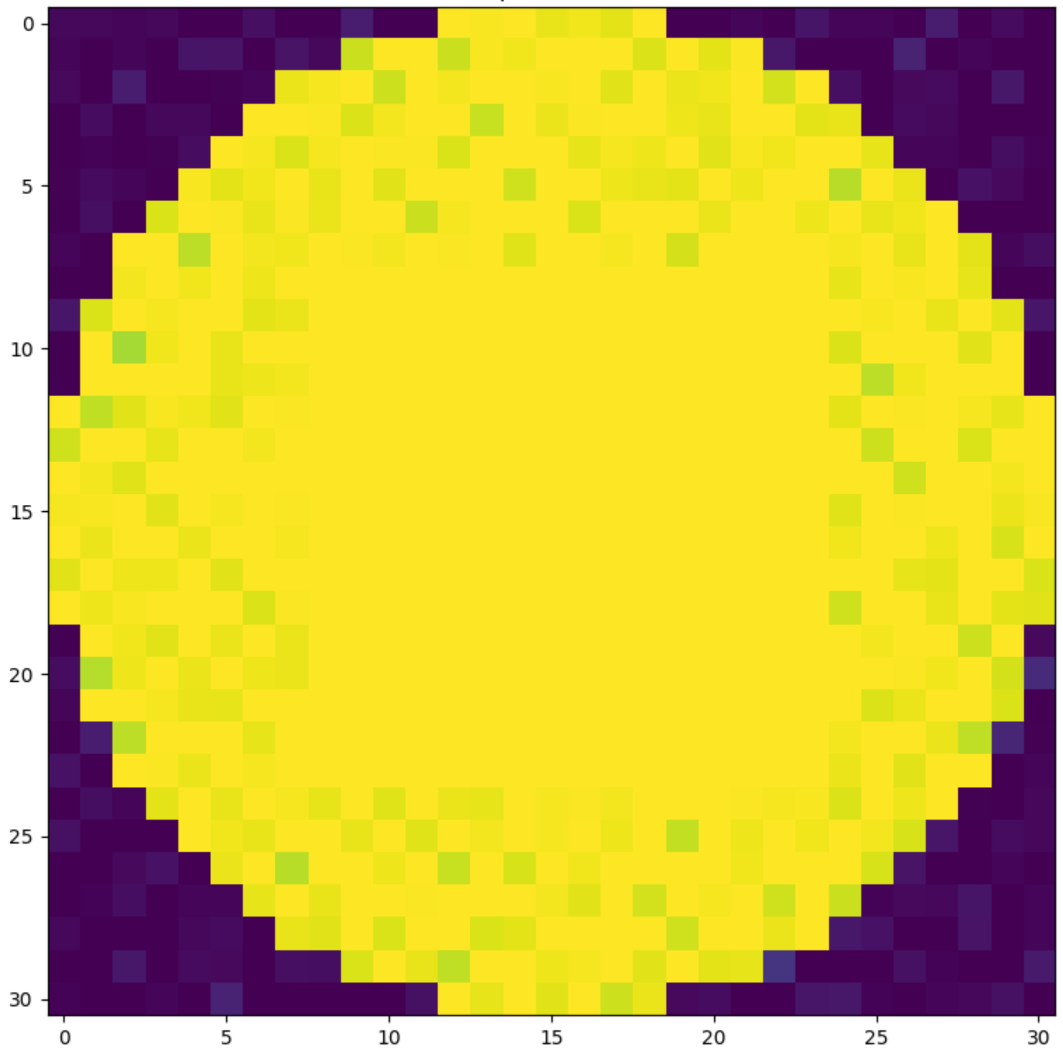
If we compare the separated colour channels to the image above, we can identify which channels represent which colours.

3.2 Before we start the cross correlation process, let's have a look at the template that you will be using to look for neurons in the image. The template is loaded in for you, so your job here is simply to plot the template (hint: use the same code that we used to plot the neuron image). What does this template look like to you? Post the template image here and let us know what it looks like to you (note, depending on what you are using, the image may be black and white or purple and yellow or some other combination of two colours). **[5]**

Running the code snippet below will yield the template neuron image:

```
###############################################################
###########
# 3.2 - TODO
###############################################################
###########
# This code loads in the template for what a "neuron" looks
like. Please
# use this space to plot the neuron template.
template_neuron = img.imread('Data/template.jpg');

fig2 = plt.figure(2);
fig2.clf()
ax = fig2.add_subplot(1,1,1)
plt.imshow(template_neuron);
ax.set_title( 'Template Neuron' );
plt.show() #Added to display all open figues
```

Template Neuron



3.3 Now let's get into the cross correlation process. Again, the cross correlation process is conceptually the same as with the 1-D ecg signal case, but now in 2-D. In the 2-D case, we are taking small sections of the larger image that match the size of the template and cross correlating the two. In this case, the cross correlation is computed by taking the product of each pixel in the template with the matching pixel in the image section, and then summing all of those products. As with the 1-D case, the larger the cross correlation value, the better the match. For this, we are allowing you to implement the cross correlation in whatever way you wish within the TODO block of the "ImageCorrelation" file. In problem 2, you will recall that we provided you with an outer loop that cycled through the larger neural signal and picked out sections to cross correlate. Here, we have left that process for you to figure out how to cycle through the image to pick out image sections to cross correlate (note, you will have to iterate left and right across the image as well as up and down over the image). I recommend referring to the code provided in problem 2 for inspiration. For this, also keep in mind that you will need to use this function to find both red and green neurons. Go back to the main script and fill in the inputs to your function that allow you to process just the red neurons and just the green neurons. **[12]**

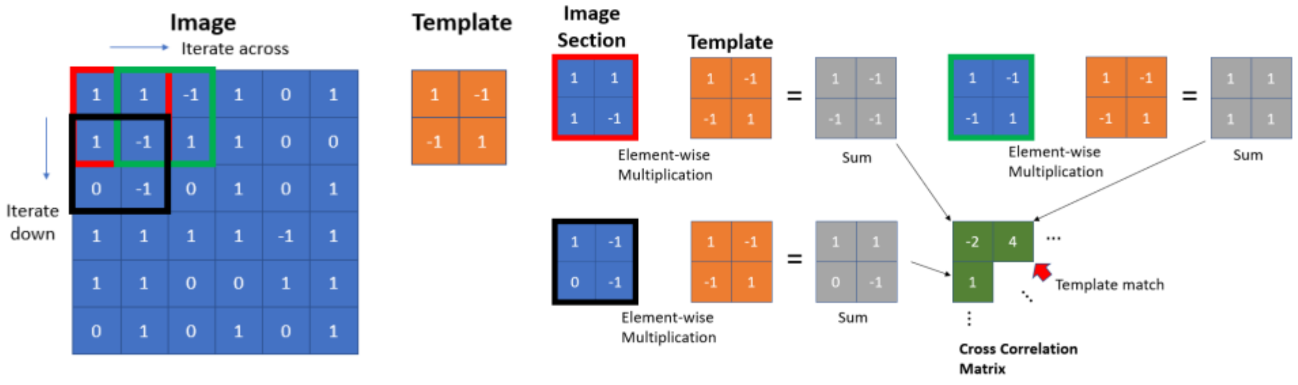


Figure 3: Graphical representation for how we compute 2-D cross correlations. We will compare the template against sections of the larger image. Once all sections are compared, the highest cross correlation values indicate a match.

```
def ImageCorrelation( template, image ):
    # Book-keeping - To do this type of image correlation, it
is common to
    # first subtract the mean value from the template.
    #template = template / 255.
    #template = template - np.mean( template );
    #image = image / 255.
    corr_matrix = np.zeros( [image.shape[0], image.shape[1]] );
    template_row_samples = template.shape[0]
    template_column_samples = template.shape[1]

    for j in range(image.shape[1] - template.shape[1]):
#iterate through columns
```

```
        for i in range(image.shape[0] - template.shape[0]):
#iterate through row
            temp_matrix = image[j:j+template_row_samples,
i:i+template_column_samples]
            #print(temp_matrix)
            product = np.multiply(template, temp_matrix)
            corr_matrix[j][i] = np.sum(product)
    #print(corr_matrix)
    return corr_matrix
```

Also need to fix the correlations for each colour channel in 'NeuronDetectionScript.py'

```
red_correlation = ImageCorrelation( template_neuron,
image_neuron[:, :, 0]);
green_correlation = ImageCorrelation( template_neuron,
image_neuron[:, :, 1]);
```

3.4 Now that you've completed the cross correlation, we have written the thresholding code here for you as it is a bit more complicated than in the 1-D neural data case. You are welcome to take a look at the thresholding function to see what's going on under the hood (it is commented to explain what is happening), but the idea is very much similar to the 1-D neural data case. What's important is that at the end of the script, a black and white image (or some other two colours) is produced that shows where the correlations were above threshold (white) and potentially represent neurons. Paste the plot here for both the red and green channels. Compare where the computer found red and green neurons with the original image. Do you see any similarities? Any differences? **[3]**

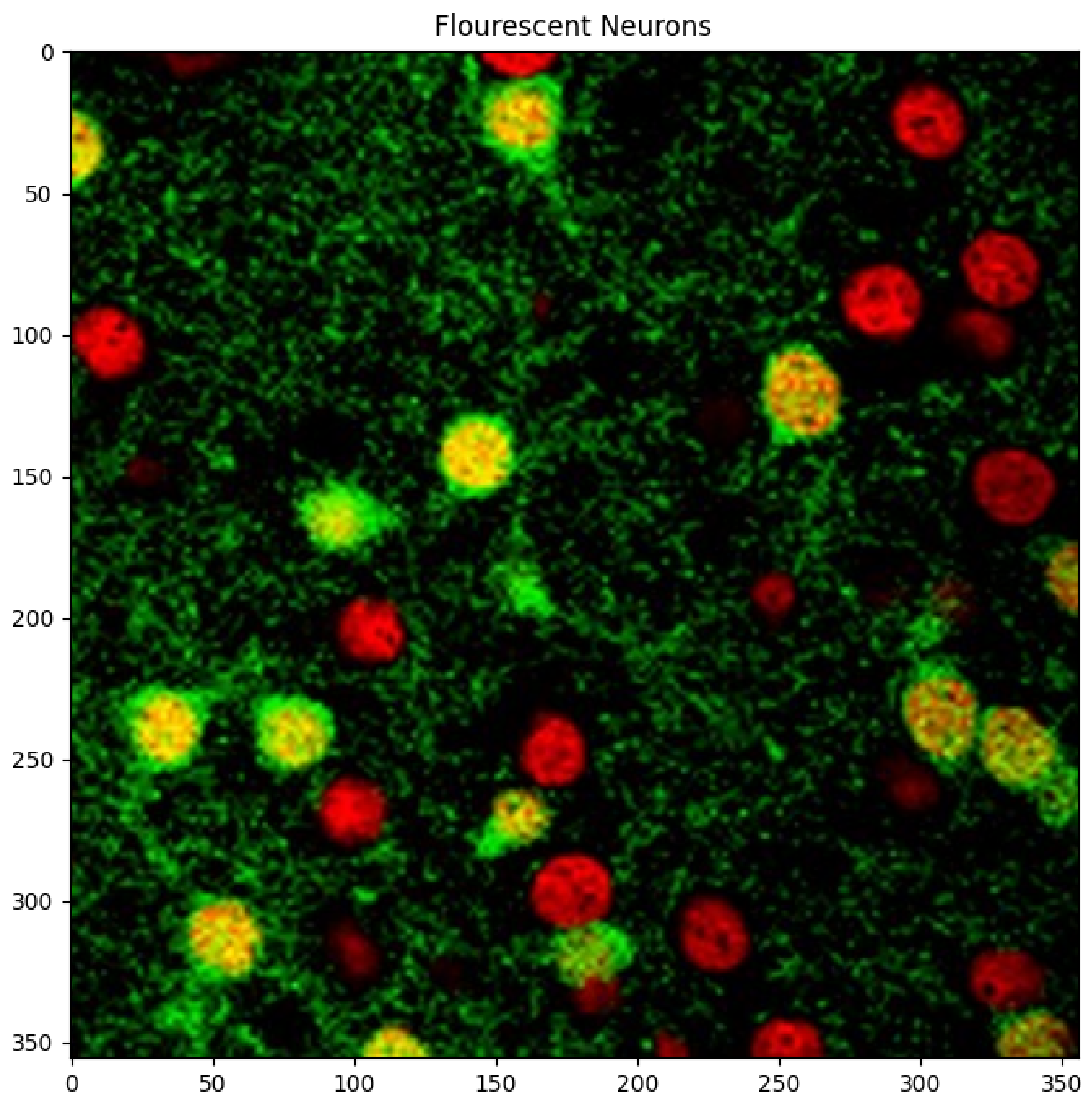

Flourescent Neurons
0
50
100
150
200
250
300
350
0
50
100
150
200
250
300
350

Figure 10: Red Mask; identify red neurons

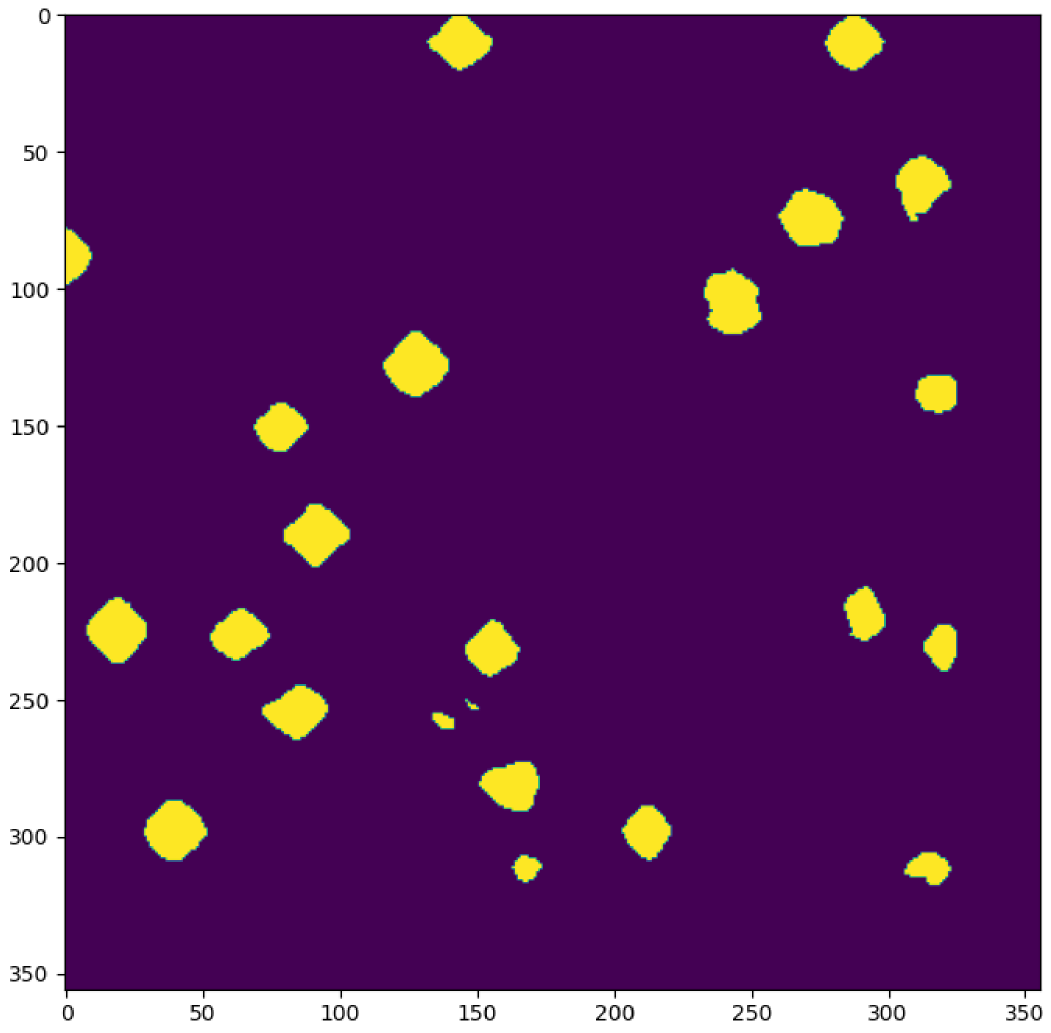
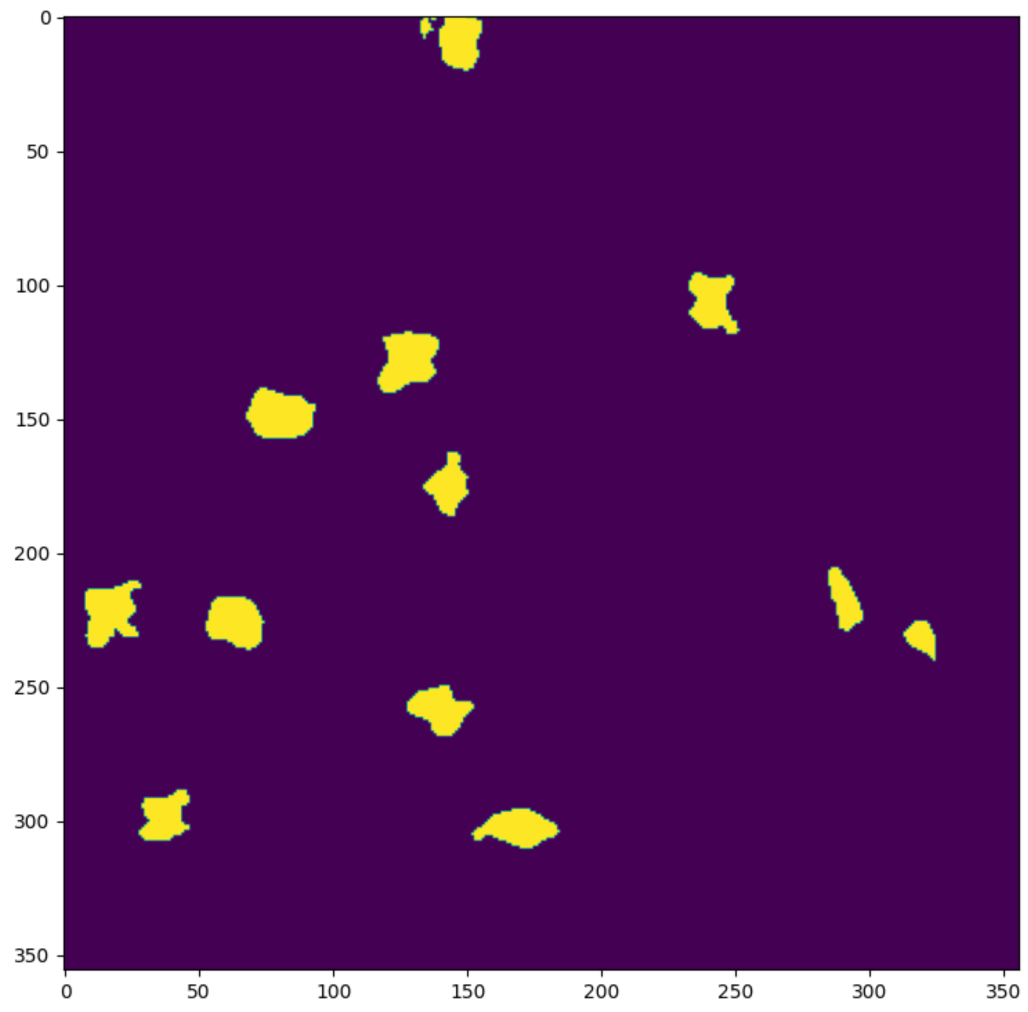


Figure 11: Green Mask; identify green neurons



There is a clear difference in between figure 10 and figure 11 as the computer found individual neurons that corresponds to the individual colour channels. Each neuron identified is specific to its colour channel. Thus, the figures show distinct neurons.