LABORATORY MANUAL

SC3102: Signals, Systems and Transforms
Hardware Lab 1 (Location: N4-01a-02)

Experiment 2: Linear time-invariant systems

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

LAB – 2

Linear time-invariant systems

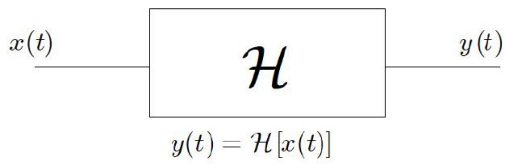
1. Objective

The objective of this laboratory is to learn about liner time-invariant systems and its applications in real world signals such as Functional magnetic resonance imaging (FMRI) and electrocardiogram (ECG) using python programming language. In this laboratory, we will

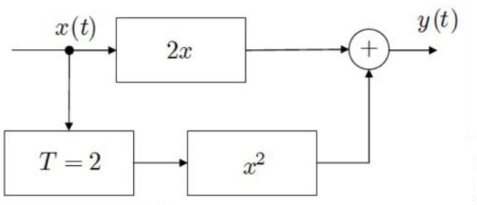
a) Understand the basics of linear time invariant systems
b) Learn the concepts and applications of convolution
c) Create hemodynamic model to predict profile of neural activity
d) Develop an algorithm for convolution and understand how it works

2. System

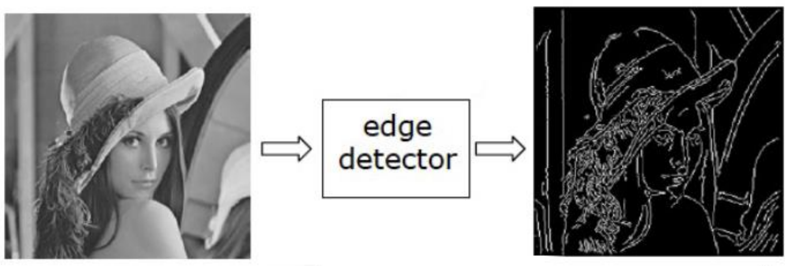
A system refers to any physical device that produces an output signal in response to an input signal, thus accomplishing a specific task.



Example of system: $y(t) = 2x(t) + x^2(t-2)$



System represents functions implemented by state machines, frequency responses, differential equations and so on. We are surrounded by systems, both natural and artificial (man-made). One typical example of a system is the image edge detection.



Run “Lab_2_Q2_EdgeDetection” file to see this in action. Run the program with a different test image to see the edge detection in action.
You’ll need to install opencv-python to run the code. Run the following code in the terminal to install

```
pip3 install --upgrade opencv-python
```

3. Linear time-invariant systems

A system is called linear time-invariant (LTI) if it is both linear and time-invariant. Refer lecture slides for more details on linear and time-invariant properties. Majority of naturally occurring and man-made systems can be modeled as LTI systems. It allows simple system analysis and system design. In order to do a detailed system analysis for LTI systems, we have to learn convolution operation in great details. The output of any LTI systems (whether it is discrete or continuous) can be modeled using convolution operation. Given the input $x(t)$, the output of a continuous-time LTI system $y(t)$ can be computed by **convolving** input $x(t)$ with system function $h(t)$. In the following section, we are learning more details of convolution operation.

3.1. State the convolution equation and discuss its application

As stated above, convolution is a mathematical way of representing the output of a linear system corresponding to an input. Mathematically, it is represented as follows:

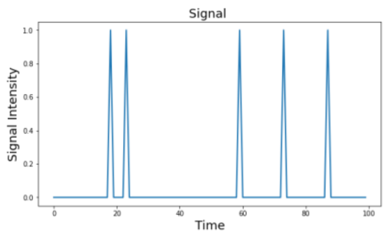
$$y(t) = h(t) * x(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau = \int_{-\infty}^{+\infty} h(\tau)x(t-\tau)d\tau$$

One way to think about it is that one signal is reversed and weighted at each time point of the other signal and then slides forward over time.

3.2. In this exercise we will create an algorithm for convolution to see how it works. To gain an intuition of how convolution works, we need two vectors. Let us call the first vector as the input signal and the other one as kernel. You might have come across the word kernel in image processing or neural networks. It can be simply defined as a filter that operates on the input signal to produce the output signal.

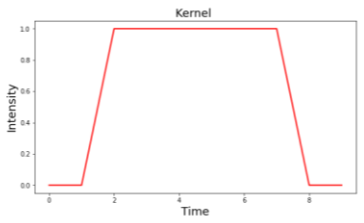
Input signal: Create a time series of spikes. Refer to the impulse signal in Lab 1. Create a signal with 100 samples, with 5-unit impulses at random locations.

```
n_samples =  #Fill the number of samples

#Create a zero vector and then fill spikes
signal = np.zeros(#Fill)   #zero vector
signal[np.random.randint(#Fill ,#Fill , 5)] = 1

#Plot the signal (sample output given)
```



3.3. Kernel: Create a boxcar kernel. Refer unit square wave in Lab 1. The kernel should be 10 samples long with boxcar width of about 6 samples.

```
kernel = np.zeros(#Fill)
kernel[2:8] = 1

#Plot the kernel (sample output given)
```



3.4. To understand convolution, we first need to familiarize with the dot product. The dot product is simply the sum of the elements of a vector weighted by the elements of another vector. This method is commonly used in signal processing, and in statistics as a measure of similarity between two vectors.

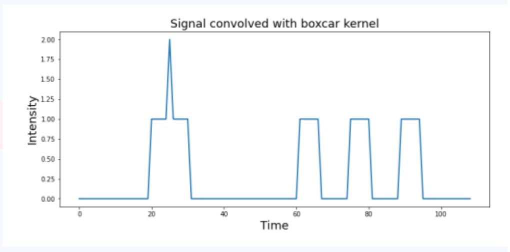
$$dotproduct_{ab} = \Sigma_{i=1}^{n} a_i b_i$$

Create some vectors of random numbers and see how the dot product works. The two vectors need to be of the same length.

```
a = np.random.randint(1,10,20)
b = np.random.randint(1,10,20)
print('Dot Product: %s' % np.dot(# Fill, # Fill))  #dot product
```

3.5. Now convolve the signal with the kernel by taking the dot product of the kernel with each time point of the signal. This can be done by creating a matrix of the kernel shifted each time point of the signal.

```
shifted_kernel = np.zeros((n_samples, n_samples+len(kernel) - 1))
rev_kernel = kernel[::-1] #Reversing the kernel
for i in range(n_samples):
    shifted_kernel[i, i:i+len(kernel)] = #
    reversed kernel
#we multiply the shifted kernal and the
signal to perform convolution
convolved_signal = np.dot(# signal, #
shifted signal)

plt.figure(figsize=(20,3))
plt.plot(convolved_signal, linewidth=2)
plt.ylabel('Intensity')
plt.xlabel('Time')
plt.title('Signal convolved with boxcar kernel')
plt.show()
```



Redo the exercise with more number of spikes in the signal

If you are unable to complete the question on time, modify the file "Lab_2_Q3_convolution.py" in the Q3 folder to plot all the sequences.

4. Introduction to FMRI

Functional magnetic resonance imaging or functional MRI (fMRI) measures brain activity by detecting changes associated with blood flow. In FMRI, we often have the subjects do a task while recording the signal. For example, we might have the subject looking at a fixation cross on the screen for most of the time, and momentarily show a very brief burst of visual stimulation, such as a flashing checkerboard.

We will call each such burst of stimulation an event.

The FMRI signal comes about first through changes in neuronal firing, and then by blood flow responses to the changes in neuronal firing. In order to predict the FMRI signal to an event, we first need a prediction (model) of the changes in neuronal firing, and second, we need a prediction (model) of how the blood flow will change in response to the neuronal firing.

Convolution is a simple way to create a hemodynamic model from a neuronal firing model

The neuronal firing model is our prediction of the profile of neural activity in response to the event. For example, with a single stimulation, we might predict that, as soon as the visual stimulation went on, the cells in the visual cortex instantly increased their firing and kept firing at the same rate while the stimulation was on.

In that case, our neural model of an event starting at 4 seconds, lasting 5 seconds, might look like this:

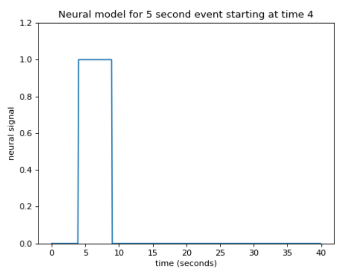


Figure 4.1: Example of a neuron firing model with visual stimuli starting at 4 seconds and lasting for 5 seconds

4.1. Now let us consider the stimulation happened 3 times. Reuse the code in question 1 from lab 1 and create an impulse response which is 40 seconds long of amplitude 2 at time = 4, then another of amplitude 1 at time = 10, and another of amplitude 3 at time = 20. This will be used as the input neural signal.

```
times = np.arange(0, 40, 0.1)
n_time_points = len(times)

# Fill your code here to plot
```

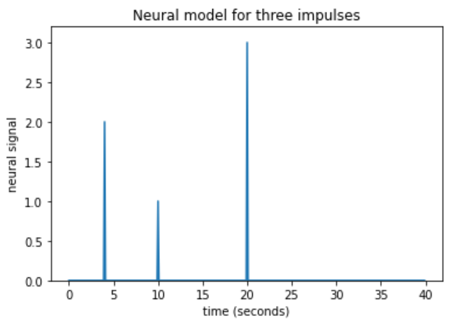


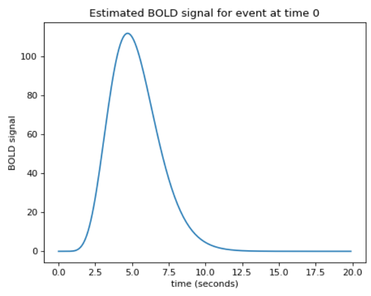
Figure 4.2: sample output for the neural input signal

Of course, we could have had another neural model, with the activity gradually increasing, or starting high and then dropping, but let us stick to this simple model for now.

4.2. Hemodynamic response

The hemodynamic response is a homeostatic process that replenishes the nutrients used by biological tissues by adjusting blood flow to areas of focal activity. Let us now assume that we know what the hemodynamic response is for a neural impulse without loss of generality. In real life, we could get this estimate from taking the FMRI signal following very brief events and averaging over many events. Here is one such estimate of the hemodynamic response to a very brief stimulus:

```
def hrf(t):
    "A hemodynamic response function"
    return t ** 8.6 * np.exp(-t / 0.547)

hrf_times = np.arange(0, 20, 0.1)
hrf_signal = hrf(#Fill) #hrf(t)
n_hrf_points = len(hrf_signal)

plt.plot(# Fill,# Fill)
plt.xlabel('time (seconds)')
plt.ylabel('BOLD signal')
plt.title('Estimated BOLD signal for event at
time 0')
```



This is the hemodynamic response to a neural impulse. In signal processing terms this is the hemodynamic impulse response function. It is usually called the hemodynamic response function (HRF) because it is a function that gives the predicted hemodynamic response at any given time following an impulse at time 0.

Now we need to predict our hemodynamic signal, given our prediction of neuronal firing.

4.3. Building the hemodynamic output from the neural input

We take the HRF (prediction for an impulse starting at time 0), and shift it by the position of the neural impulses to give our predicted output:

Here, the HRF is our kernel. Just like we did earlier in 3.5, we shift the HRF here. For the impulse amplitude 2 at time == 4, add the HRF shifted to start at time = 4, and scaled by 2. Then to that result, add the HRF shifted to time = 10 and scaled by 1. Finally, add the HRF shifted to time = 20 and scaled by 3:

```
bold_signal = np.zeros(n_time_points) #what is n_time_points? Check 4.1

bold_signal[i_time_4:i_time_4 + n_hrf_points] = hrf_signal * # Fill the amplitide
of the impulse
bold_signal[i_time_10:i_time_10 + n_hrf_points] += hrf_signal * # Fill
bold_signal[i_time_20:i_time_20 + n_hrf_points] += hrf_signal * # Fill

plt.plot(#Fill, #Fill)
plt.xlabel('time (seconds)')
plt.ylabel('bold signal')
plt.title('Output BOLD signal for three impulses')
```

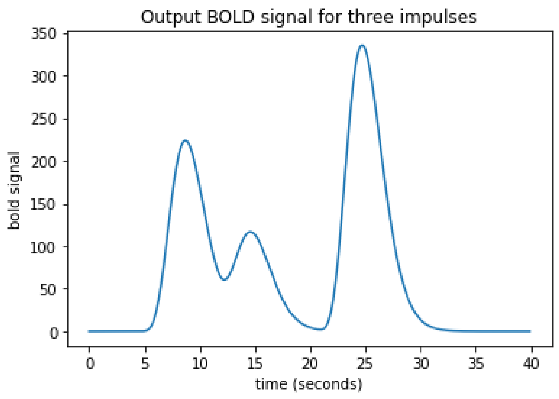


Figure: 4.4: Sample output of question 4.4

4.4. Create an algorithm for general case

Now we have a general algorithm for making our output hemodynamic signal from our input neural signal:

1. Start with an output vector, i.e., a vector of zeros
2. For each index i in the input vector (the neural signal), prepare a shifted copy of the HRF vector, starting at i. Call this the shifted HRF vector
3. Multiply the shifted HRF vector by the value in the input at index i, to give the shifted scaled HRF vector.
4. Add the shifted scaled HRF vector to the output.

There is a little problem with our algorithm – the length of the output vector.

Imagine that our input (neural) vector is N time points long. Say the original HRF vector is M time points long.

In our algorithm, when the iteration gets to the last index of the input vector (i = N−1), the shifted scaled HRF vector will, as ever, be M points long. If the output vector is the same length as the input vector, we can add only the first point of the new scaled HRF vector to the last point of the output vector, but all the subsequent values of the scaled HRF vector extend off the end of the output vector and have no corresponding index in the output. The way to solve this is to extend the output vector by the necessary M-1 points.

Create this algorithm using python code. Use the ideas from previous questions to easily code the algorithm. Plot the neural signal with atleast 3 spikes and the corresponding HRF response. (Use the same hrf signal from question 4.3 and neural signal from 4.2 to validate)

```
N = n_time_points
M = n_hrf_points
bold_signal = np.zeros(#Fill)  # extending the output vector by M-1 points
for i in range(N):
    input_value = neural_signal[i]
    # Adding the shifted, scaled HRF
    bold_signal[i : i + n_hrf_points] += hrf_signal * # neural signals

# We have to extend 'times' to deal with more points in 'bold_signal'
extra_times = np.arange(n_hrf_points - 1) * 0.1 + 40
times_and_tail = np.concatenate((times, extra_times))
plt.plot(times_and_tail, # Signal)
plt.xlabel('time (seconds)')
plt.ylabel('bold signal')
plt.title('Output BOLD signal using our algorithm')
```

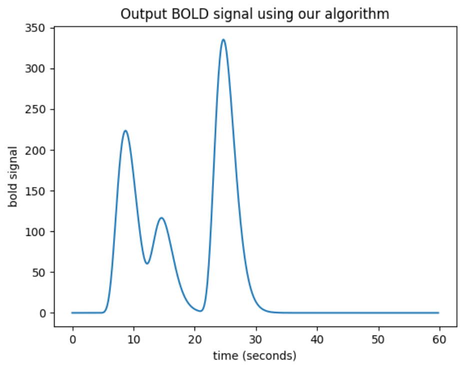


Figure: 4.5: Sample output of question 4.5

4.5. We have convolution

The algorithm we designed in the previous question is the convolution algorithm.

Using NumPy convolve function to convolve the neural signal and the HRF signal. The plot obtained will be the same as obtained in question 4.5 provided the same hrf signal from question 4.3 and neural signal from 4.2 is used.

```
bold_signal = np.convolve(# neural signal, # hrf signal)
plt.plot(times_and_tail, # Fill)
plt.xlabel('time (seconds)')
plt.ylabel('bold signal')
plt.title('Our algorithm is the same as convolution')
```

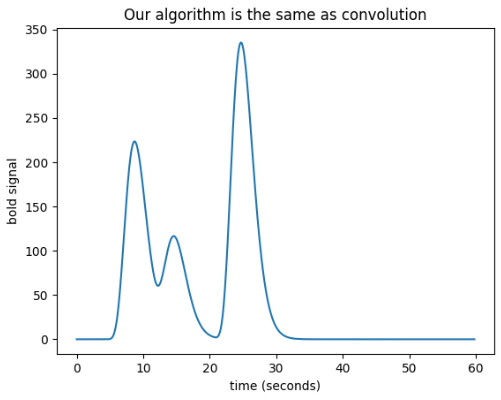


Figure: 4.6: Sample output of question 4.6

Redo questions 4 with different neural input signal (E.g., increase the number of spikes)

If you are unable to complete the question on time, modify the file in the Q4 folder to plot all the sequences.

5. LTI system

Given two discrete time systems (h1 and h2) with finite impulse response given by

h1[n] = [0.06523, 0.14936, 0.21529, 0.2402, 0.21529, 0.14936, 0.06523]
h2[n] = [-0.06523, -0.14936, -0.21529,0.7598, -0.21529, -0.14936, -0.06523]

5.1. Plot these two systems' impulse response. Also plot the response of $h3[n] = h1[n] * h2[n]$ You can compare your results using SciPy's lfilter module and numpy convolve function.

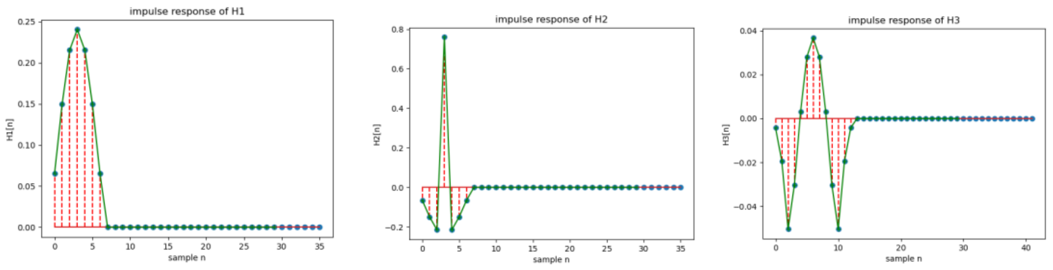


Figure: 5.1: Sample output of question 5.1

5.2. Implement your own code as a module to generate the output for the above system for input $x[n] = \delta[n] - 2\delta[n-15]$.

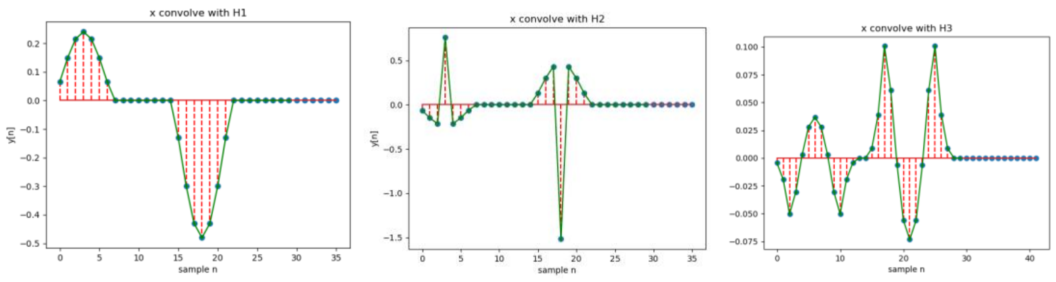


Figure: 5.2: Sample output of question 5.2

5.3. Let $x1[n] = \delta[n]$ and $x2[n] = -2\delta[n-15]$. This means that $x[n] = x1[n] + x2[n]$. Determine whether the system $y[n] = h3[n] * x[n]$ is a linear system.

To show that this is a linear system we need to show that

$$y[n] = h3[n] * x[n] = h3[n] * (x1[n] + x2[n]) = \text{y1[n]} + \text{y2[n]}$$

where y1[n] $= h3[n] * x1[n]$ and y2[n] $= h3[n] * x2[n]$.

Plot $h3[n] * (x1[n] + x2[n])$ and y1[n] + y2[n] separately and show that they are equal, hence proving that the system is linear.

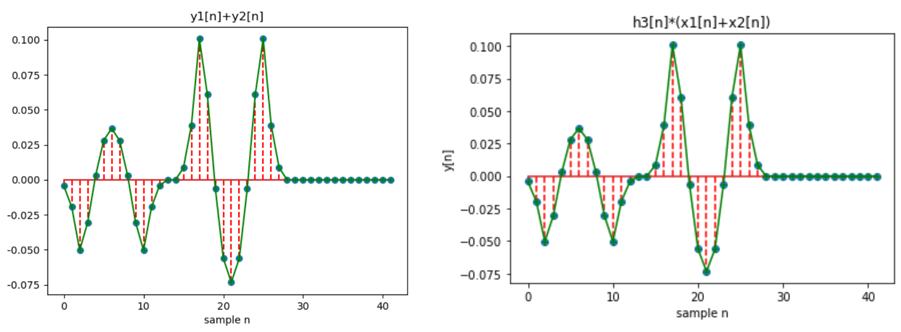


Figure: 5.3: Sample output of question 5.3

Sample code for this exercise is given below. You may change it according to the questions.

```
impulseH1 =  [0.06523, 0.14936,0.21529,0.2402,0.21529,0.14936,0.06523]
impulseH2 =  [-0.06523, -0.14936,-0.21529,0.7598,-0.21529,-0.14936,-0.06523]
impulseH3 = np.convolve(impulseH1,impulseH2)

x1 = np.zeros(30)
x1[0]  = 1

plt.figure()
y_float1  = np.convolve(x1,impulseH1)
y_float2  = signal.lfilter(impulseH1,[1], x1)
# showing that convolution can be realised by np.convolve and
scipy.signal.lfilter(BCooeff,[1],impulseX)
plt.stem(y_float1, linefmt='r--')
plt.plot(y_float2, 'g-+')
plt.title('impulse response of H1')
plt.xlabel('sample n'); plt.ylabel('H1[n]')

# Change the code according to the question
```

6. Filtering ECG signal (Optional)

In this exercise we explore a real-life application of convolution. We aim to filter out some of the noise from an ECG signal given in “ECG.csv” using convolution.

6.1. Write a python program to load the raw ECG signal. The signal is given as a csv file in the Q6 folder. Use pandas to open the csv file. Plot the ECG signal and identify the noise visually.

(If required: Install pandas from the terminal using pip install pandas)

```
import pandas as pd
import scipy.fftpack

dataset = pd.read_csv("ECG.csv")
y = [e for e in dataset.hart]

N = len(y) # Number of samplepoints
Fs = 1000 # sample spacing
T = 1.0 / Fs
x = np.linspace(0.0, N*T, N) #Compute x-axis

# Fill your code here for plotting
```

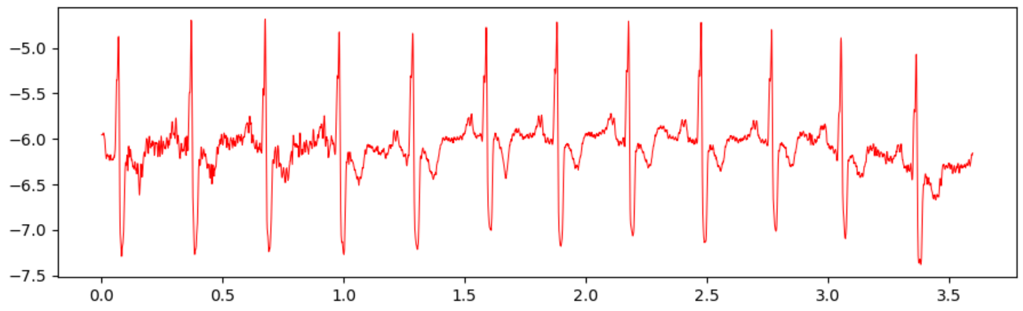


Figure 6.1: ECG signal

6.2. In this exercise we processes raw ECG signals to obtain a smoothened signal. Medical examinations can only be performed on a denised signal. Hence removing the noise is a cruicial step. Modern equipments have noise removing softowares inbuilt into the hardware. Here we attempt to remove the 50 Hz brum noise using a butterworth filter.

More about butterworth filter can be found here
https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html

Write a program to remove the 50 Hz noise from the given ECG signal.

```
b, a = signal.butter(4, 50/(Fs/2), 'low')
y_filt = signal.filtfilt(b,a, y)

# Fill your code here for plotting
```

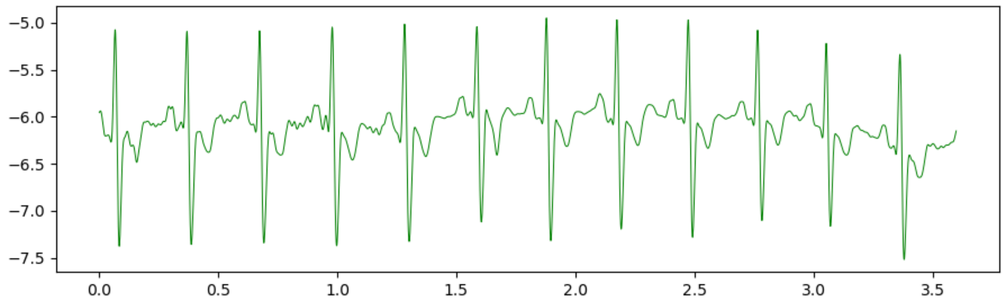


Figure 6.2: Sample output after filtering the 50 Hz noise.

6.3. Plot the ECG signals in the frequency spectrum before and after filtering to see whether the undesired frequencies have been filtered away. Use Fast Fourier Transform (FFT) to plot the signal in the frequency domain. You will learn more about Fourier transforms in the subsequent lectures.

More about FFT can be found here.
https://docs.scipy.org/doc/scipy/tutorial/fft.html

```
#Compute Fast Fourier Transform (fft)
yf = scipy.fftpack.fft(y)
yff = scipy.fftpack.fft(y_filt)

# Fill your code here for plotting
```

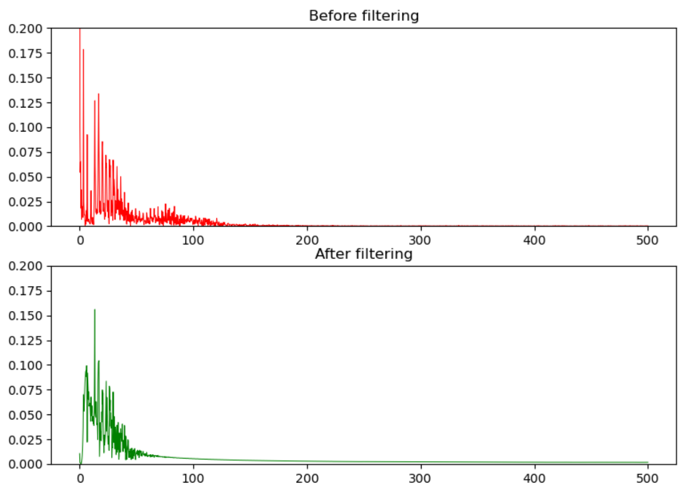


Figure 6.3: Sample output of the ECG signal in the frequency domain before and after filtering

If you are unable to complete the question on time, modify the file "Lab_2_Q6_ecg" in the Q6 folder to plot all the sequences.