ENG5027 Assignment 02 - FIR FIlters

- Team Members: Jinming Zhang (2639212z), Xiaohui Yu (2784582y), Jianyu Zhao (2721815z), Ziyuan Cheng (2786151c)

Task 1: Design Filters

According to the description, functions to design the filter need to be implemented and functions should automatically
decide the num of coefficients. For exmaple, high pass filter just need one cutoff frequency, however, band stop
frequency should have two cutoff frequency. Due to characteristics of python, **dynamic language**, at runtime the code
can change its structure based on certain conditions, meanwhile, the data type is not determined at the compile stage,
but the **type binding** is postponed to the **runtime stage**.Therefore, incoming parameters can be consistent. What is
more, considering **scalability**, in order to not need to refactor the entire structure when a new filter needs to be
designed, define a base class `Filter` and a **abstract** method `_design_filter` is necessary. And then, because the
process of high pass filter is same as band stop filter's, the method to filter can be defined in super class, when a
filter need a different function to filter, it is easy to **overload** it in specific derived class.

I. Define the super class, FirFilter

Steps:

1. Define a constructor, according to h(n) to init the buffer used in process of filter.

```
class FirFilter:
    def __init__(self, _h):
        self.h = _h
        self.M = len(_h)
        # Ring buffer
        self.buffer = np.zeros(self.M)
        self.offset = self.M - 1
```

2. Define a private abstract design function which need every derived class to implement since this function is not
   expected to be called by the outside.

```
# Abstract design method
@abc.abstractmethod
def _design_filter(self, w):
    pass
```

3. Define filter functions, including a real time filter function and entire filter function. In order to improve efficiency, to use a **loop array** and a **offset** to implement a buffer.

```
# Real time filter
def _do_filter(self, input):
    self.buffer[self.offset] = input
    # Move the offset
    self.offset -= 1
    if self.offset < 0:
        self.offset = self.M - 1
    # Calculate the output
    output = 0
    for i in range(self.M):
        output += self.h[i] * self.buffer[(i + self.offset) % self.M]
    return output

# Entire filter
def do_total_filter(self, input):
    n = len(input)
    output = np.zeros(n)
    for i in range(n):
        output[i] = self._do_filter(input[i])
    return output
```

II. Define the derived class, HighPassFilter

Steps:

1. Define a constructor, need to call the constructor of the super class.

```
class BandStopFilter(FirFilter):
  def __init__(self, _sample_rate, _w_1, _w_2, _freq_resolution=1):
    self.sample_rate = _sample_rate
    self.freq_resolution = _freq_resolution
    h = self._design_filter([_w_1, _w_2])
    super().__init__(h)
```

2. Implement the design function using IFFT and hamming window.

```
def _design_filter(self, w):
  fs = self.sample_rate
  M = int(fs / self.freq_resolution)
```

```
    w_1 = int(w[0] / fs * M)
    w_2 = int(w[1] / fs * M)
    # Define the ideal frequency response
    X = np.ones(M)
    X[w_1:w_2 + 1] = 0
    X[M - w_2:M - w_1 + 1] = 0
    # Compute h(n) using the inverse FFT
    x = np.fft.ifft(X)
    x = np.real(x)
    # Mirror h(n)
    h = np.zeros(M)
    h[0:int(M / 2)] = x[int(M / 2):M]
    h[int(M / 2):M] = x[0:int(M / 2)]
    # Window h(n) using a Hamming window
    h = h * np.hamming(M)

return h
```

III. Define the derived class, BandStopFilter

Steps:

1. Define a constructor, need to call the constructor of the super class.

```
class HighPassFilter(FirFilter):
    def __init__(self, _sample_rate, _w_c, _freq_resolution=1):
        self.sample_rate = _sample_rate
        self.freq_resolution = _freq_resolution
        h = self._design_filter(_w_c)
        super().__init__(h)
```

2. Implement the design function using IFFT and hamming window.

```
def _design_filter(self, w):
    fs = self.sample_rate
    M = int(fs / self.freq_resolution)
    w_c = int(w / fs * M)
    # Define the ideal frequency response
    X = np.ones(M)
    X[0:w_c + 1] = 0
    X[M - w_c:M + 1] = 0
    # Compute h(n) using the inverse FFT
    x = np.fft.ifft(X)
    x = np.real(x)
    # Mirror h(n)
    h = np.zeros(M)
    h[0:int(M / 2)] = x[int(M / 2):M]
    h[int(M / 2):M] = x[0:int(M / 2)]
```

```
    # Window h(n) using a Hamming window
    h = h * np.hamming(M)

    return h
```

IV. Define the derived class, LmsFilter

Steps:

1. Define a constructor, need to call the constructor of the super class.

```
class LmsFilter(FirFilter):
    def __init__(self, _sample_rate, _freq_resolution=1):
        self.sample_rate = _sample_rate
        self.freq_resolution = _freq_resolution
        h = self._design_filter(None)
        super().__init__(h)
```

2. Implement the design function using no other parameter but sampe rate.

```
def _design_filter(self, w):
    h = np.zeros(int(self.sample_rate / self.freq_resolution))
    return h
```

3. Overland the filter function, lms filter need calculate a error between input and noise. In addition, because of **
loop array** to implement the buffer, need to use **offset** to update the h(n)

```
def _do_filter(self, input, noise=None):
    # Calculate thr error
    canceller = super()._do_filter(noise)
    output = input - canceller
    # Update the h(n)
    for i in range(self.M):
        self.h[i] += output * self.learning_rate * self.buffer[(i + self.offset) %
self.M]
    return output

def do_total_filter(self, input):
    n = len(input)
    output = np.zeros(n)
    for i in range(n):
        noise = np.sin(2.0 * np.pi * self.noise_freq / self.sample_rate * i)
        output[i] = self._do_filter(input[i], noise)
    return output
```

Task 2: Implement band-stop filter and high-pass filter

Steps:

1. Creating band-stop filter and high-pass filter
   In Task1, the functions band-stop filter and high-pass filter have been created. Therefore, they can be called directly here.

   ```
   from firfilter import BandStopFilter, HighPassFilter
   band_stop_w_1 = 49
   band_stop_w_2 = 51
   high_pass_w_c = 2
   original_data = np.loadtxt(constant.file_path)
   n = len(original_data)
   # Create band-stop filter
   band_stop_filter = BandStopFilter(constant.sample_rate, band_stop_w_1,
   band_stop_w_2)
   # Create high-pass filter
   high_pass_filter = HighPassFilter(constant.sample_rate, high_pass_w_c)
   ```

2. Removing the 50Hz interference by the band-stop filter
   The topic requires using a band-stop filter to filter out 50hz of noise.

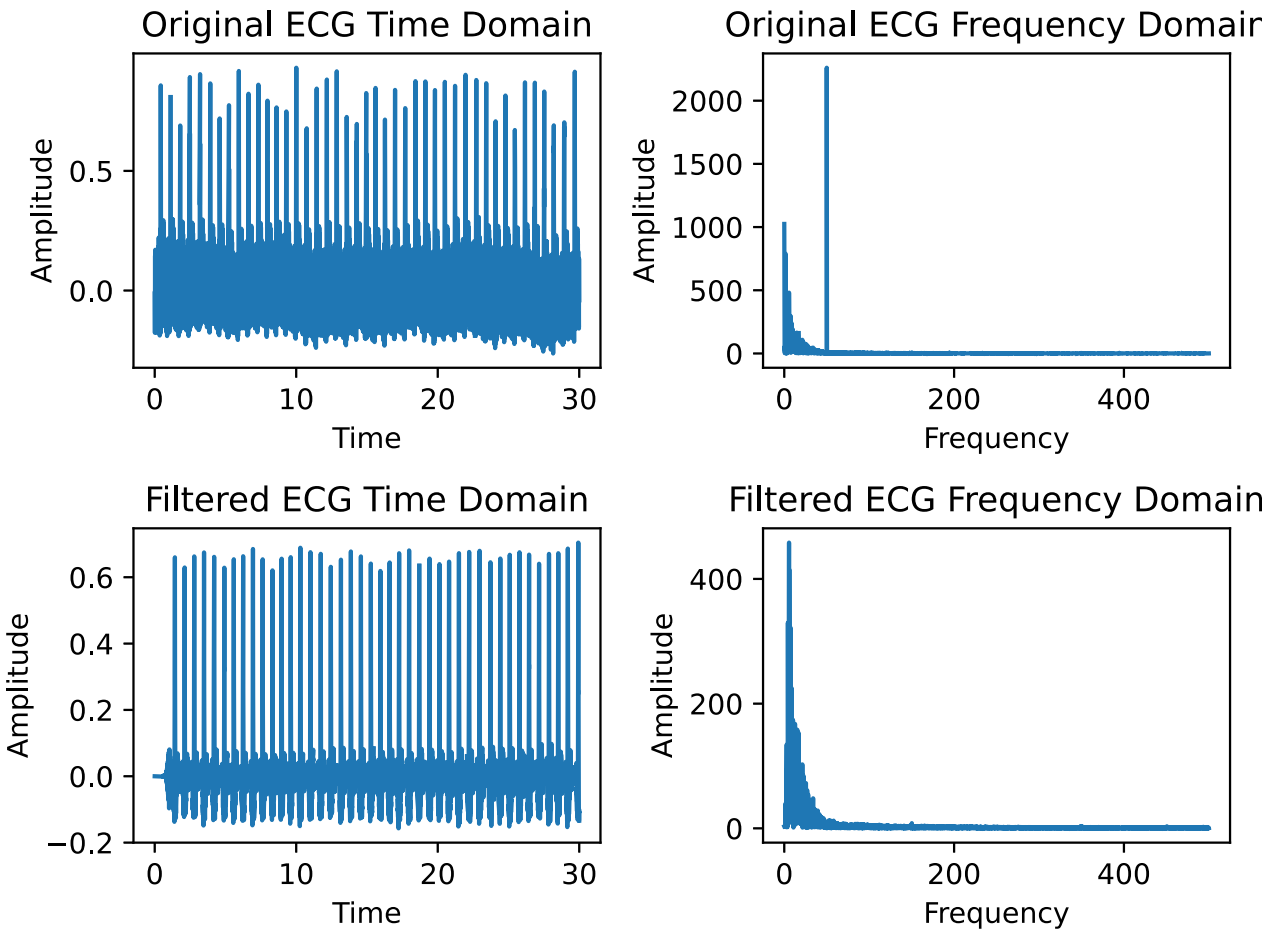
   ```
   # Remove the 50Hz interference by the band-stop filter
   filtered_data = band_stop_filter.do_total_filter(original_data)
   ```

3. Process the baseline wander by the high-pass filter
   Baseline wander is eliminated by utilizing a high-pass filter. Because if the baseline wander is not eliminated, then the trend term is treated as the original signal acquired, affecting the accuracy of the signal and the subsequent data processing results.

   ```
   # Process the baseline wander by the high-pass filter
   filtered_data = high_pass_filter.do_total_filter(filtered_data)
   ```

4. Ploting the time domain and frequency domain before and after filtering

5. Observation of the image shows that the 50hz is correctly filtered (as figures shown below)


Original ECG Time Domain
Amplitude
0.5
0.0
0
10
20
30
Time
Original ECG Frequency Domair
Amplitude
2000
1500
1000
500
0
0
200
400
Frequency
Filtered ECG Time Domain
Amplitude
0.6
0.4
0.2
0.0
−0.2
0
10
20
30
Time
Filtered ECG Frequency Domain
Amplitude
400
200
0
0
200
400
Frequency

Task 3: Implement LMS filter and high-pass filter

Steps:

1. Creating LMS filter and high-pass filter

```
noise_freq = 50
learning_rate = 0.001
high_pass_w_c = 2
original_data = np.loadtxt(constant.file_path)
n = len(original_data)
# Create LMS filter
lms_filter = LmsFilter(constant.sample_rate, noise_freq, learning_rate,
constant.freq_resolution)
# Create high-pass filter
high_pass_filter = HighPassFilter(constant.sample_rate, high_pass_w_c)
```

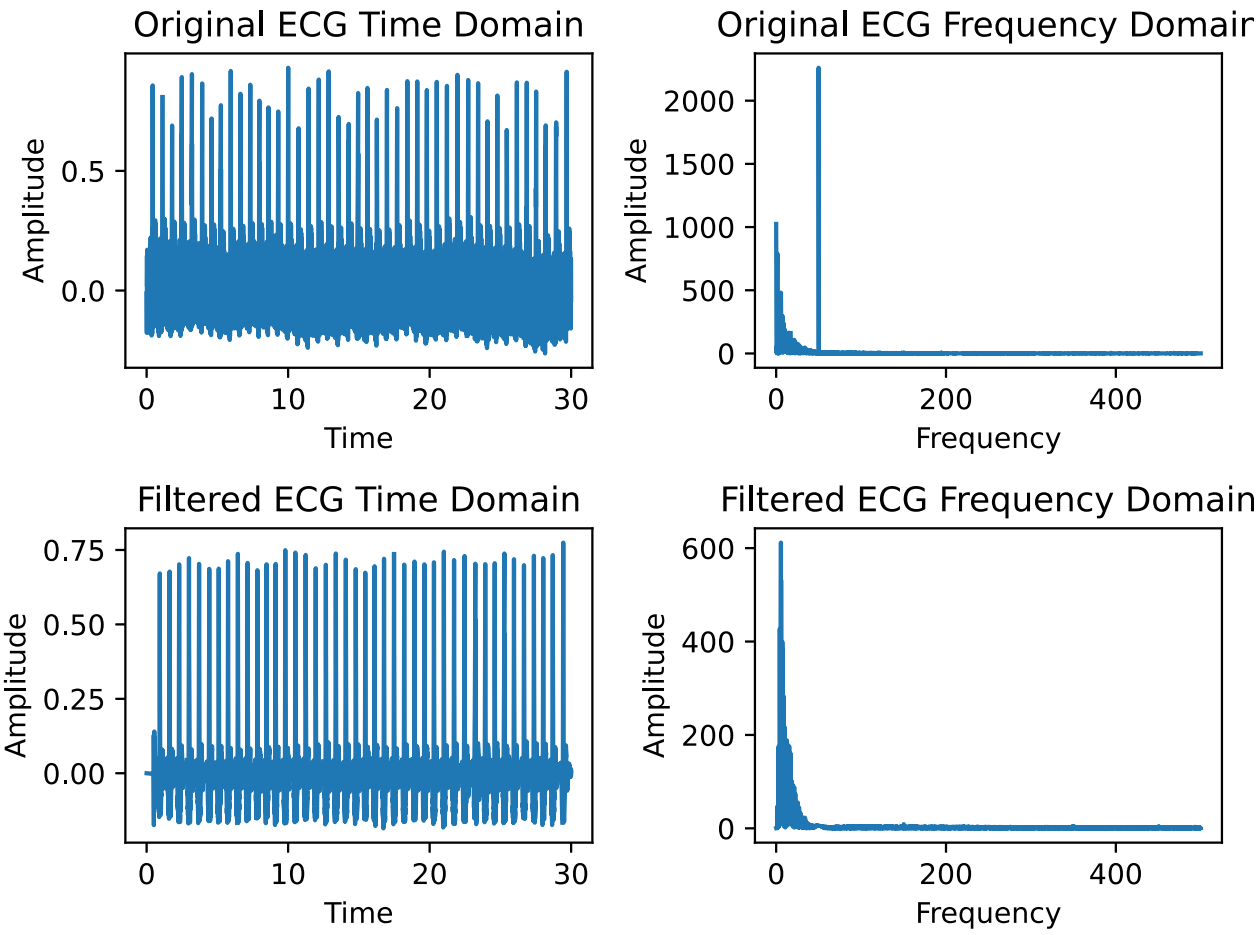
2. Similar to the Task 2, but at this time, we are using lms filter to remove the 50Hz interference.

```
# Remove the 50Hz interference by the lms filter
filtered_data = lms_filter.do_total_filter(original_data)
```

3. Process the baseline wander by the high-pass filter

```
# Process the baseline wander by the high-pass filter
filtered_data = high_pass_filter.do_total_filter(filtered_data)
```

4. Plot and comparing the time domain and frequency domain before and after the filtering processing (the figures shown below)


Original ECG Time Domain
Amplitude
0.5
0.0
0
10
20
30
Time
Original ECG Frequency Domair
Amplitude
2000
1500
1000
500
0
0
200
400
Frequency
Filtered ECG Time Domain
Amplitude
0.75
0.50
0.25
0.00
0
10
20
30
Time
Filtered ECG Frequency Domain
Amplitude
600
400
200
0
0
200
400
Frequency

Task 4: ECG heartbeat detection

Steps:

1. We've created a wavelet to simulate the shape of an R-peak. The angular frequency can be arbitrarily chosen as the appropriate value.

```
def createWavelet():
    t = np.arange(-0.4, 0.4, 1 / 1250)
    w = 250
    y = np.sin(w * t) / (w * t)
    return y
```

2. Filter out the 50hz frequency first.

```
    high_pass_filter = HighPassFilter(constant.sample_rate,high_pass_w_c)
    OutputAfterHighpassFilter = high_pass_filter.do_total_filter(data)
```

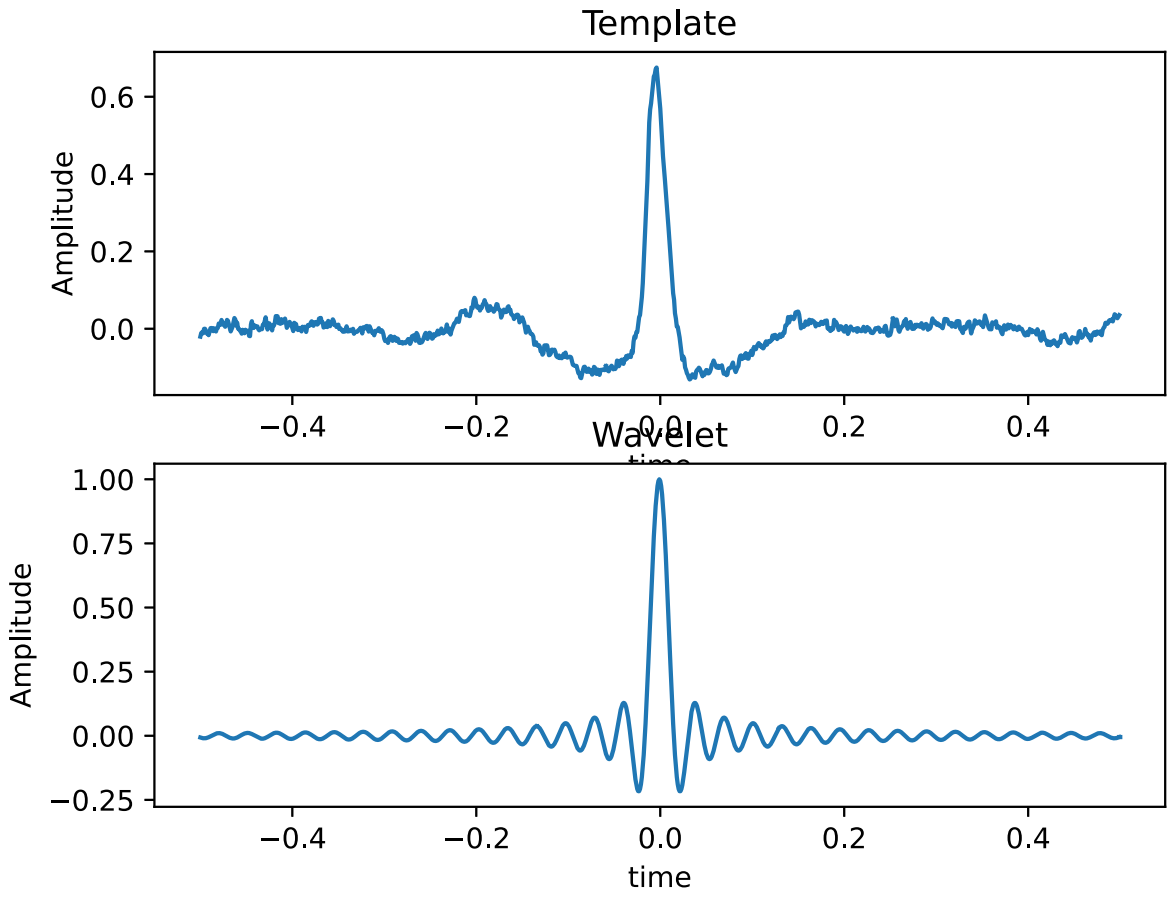
3. Then filter out the baseline wander.

```
    band_stop_filter = BandStopFilter(constant.sample_rate, band_stop_w_c[0],
band_stop_w_c[1])
    OutputAfterBandStopFilter =
band_stop_filter.do_total_filter(OutputAfterHighpassFilter)
```

4. We take one of these R peaks and compare it with a wavelet. And reverse time, so we can convolve it later in the time domain.

```
  template1 = OutputAfterBandStopFilter[10500:11500]  # create template
    template2 = createWavelet()
    fir_coeff1 = template1[::-1]  # reverse time
    fir_coeff2 = template2[::-1]

    template_t = np.arange(-500, 500)
    template_t = template_t / 1000
```

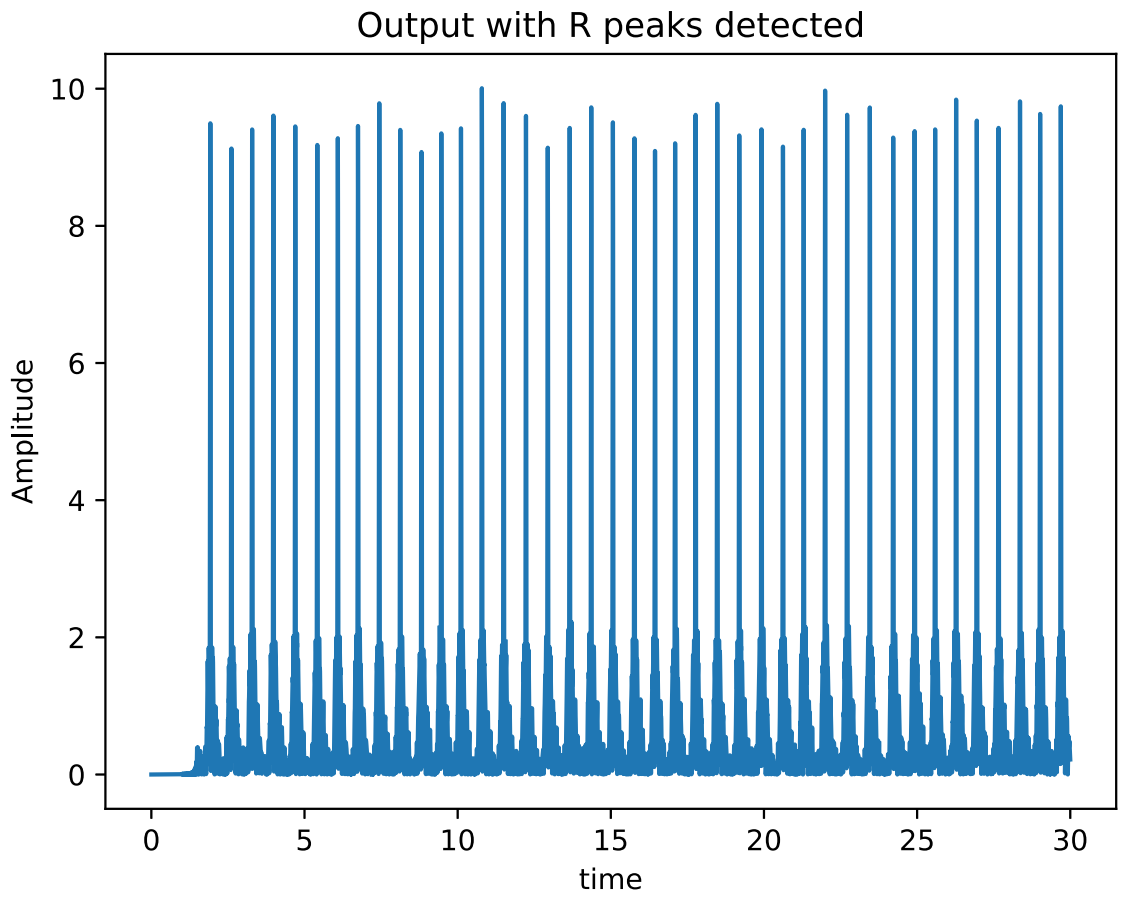


It is easy to see that R-peak is very similar to wavelet, so we will use it as a template to detect R-peak

5. Since the original data had positive and negative heart rate amplitude, we processed the amplitude with absolute value for the convenience of observation. So the R-peaks are all above the X-axis.

```
filter1 = FirFilter(fir_coeff1) # R-peaks filter
filter2 = FirFilter(fir_coeff2) # Wavelet filter

res2 = filter2.do_total_filter(OutputAfterBandStopFilter)
res2 = abs(res2)
```



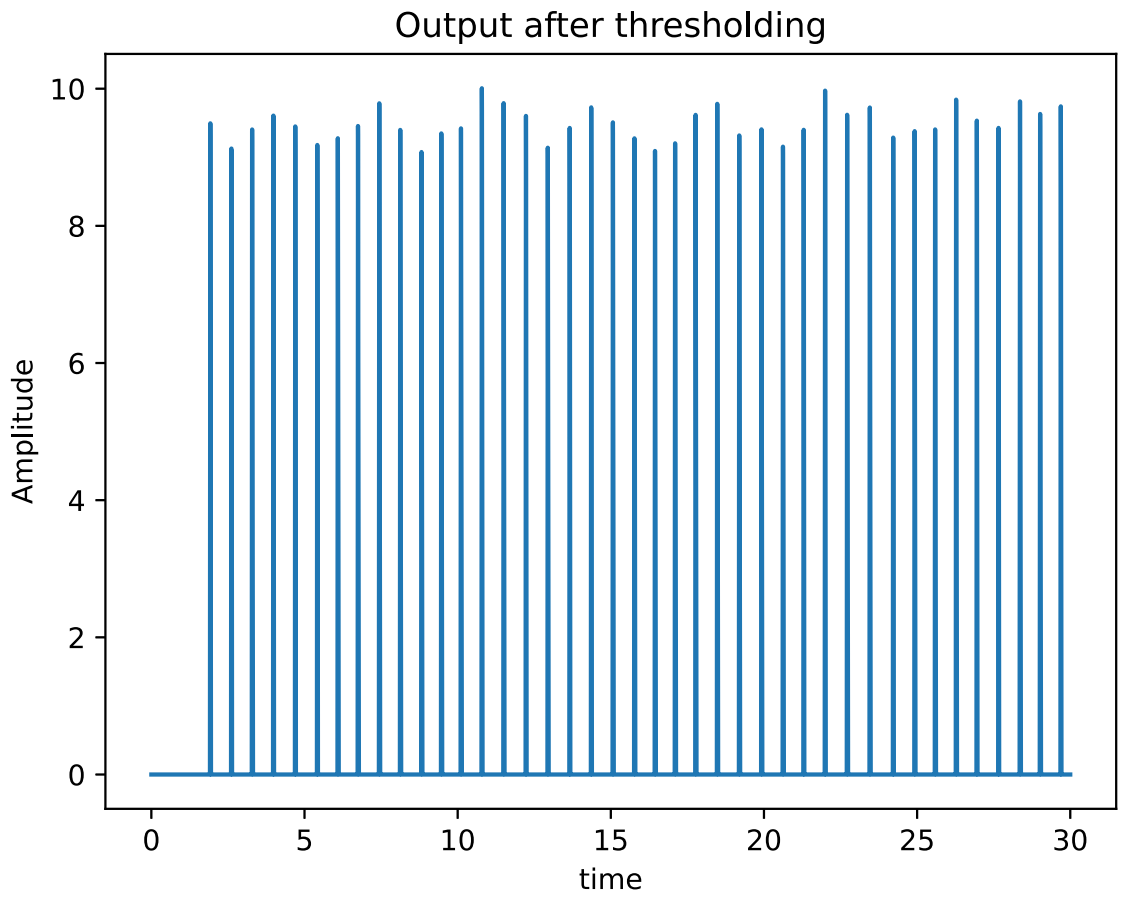
6. Now we are ready to leave the R-peak in place and set the peak below the threshold to 0. We can see from the original figure that the amplitude of peak R is mostly between 8 and 10, and other small peaks are around 2, so we set the threshold value to 5.

```
    for i in range(len(data)):
        if res2[i] <= 5:
            res2[i] = 0
        else:
            PeakTimeForECG[PeakNum] = i
            PeakNum += 1

    PeakTimeForECG = PeakTimeForECG[:PeakNum] # the place to time
    PeakTimeForECG = PeakTimeForECG / 1000
```

7. Further filtering is needed to prevent incorrect data from being persisted during processing. We know from life experience that humans can't beat their hearts twice in 0.3 seconds, so we filtered out the data.

```
    res3.append(PeakTimeForECG[0])
    for i in range(1, len(PeakTimeForECG)):
        if PeakTimeForECG[i] - PeakTimeForECG[i - 1] > 0.3: # remove the error, time
interval > 0.3
            res3.append(PeakTimeForECG[i])
```



So it looks very pure.

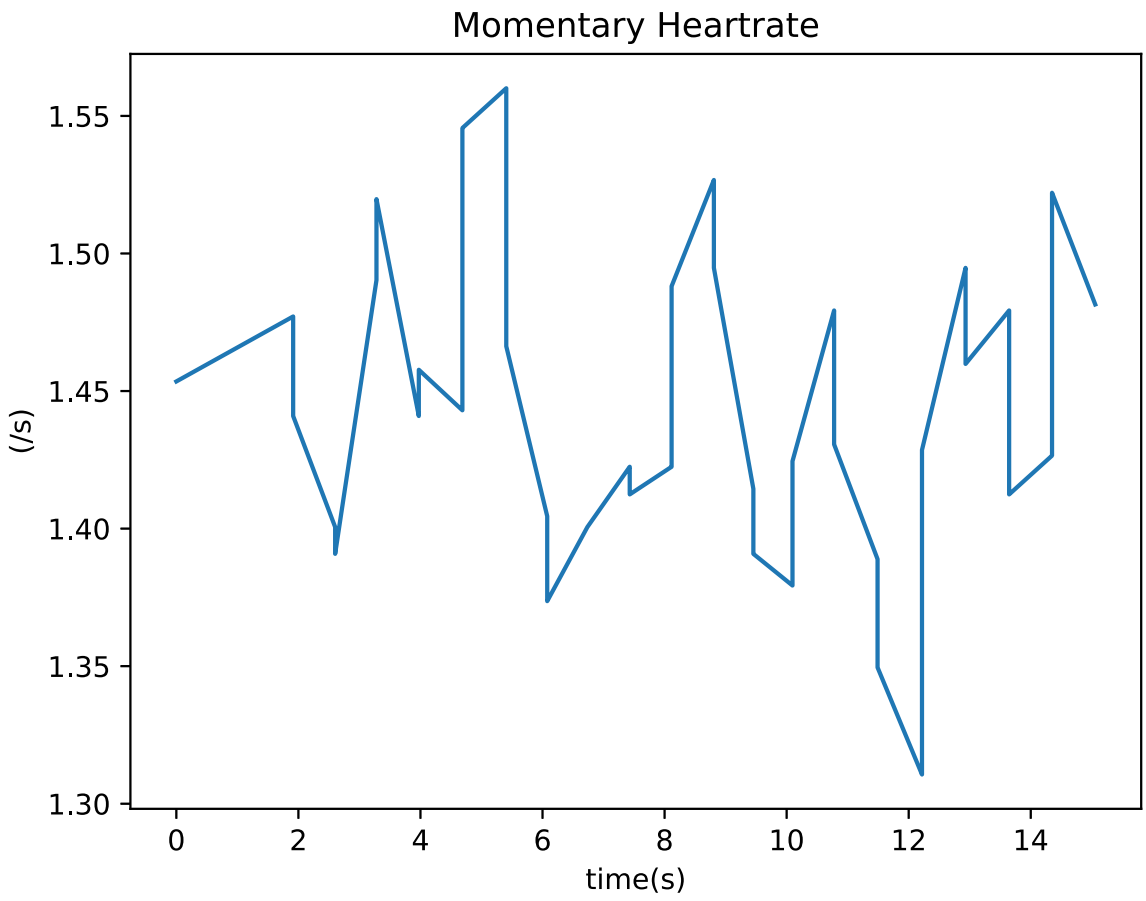
8. To plot heart rate over time, we calculate an instantaneous heart rate for each R peak.

```
InverseInterval = []
y_output = []
x_output = []
for i in range(1, len(res3)):
    InverseInterval.append((1 / (res3[i] - res3[i - 1]))) # Real heart rate
```

9. In order for x to match the length of y, let's append x twice

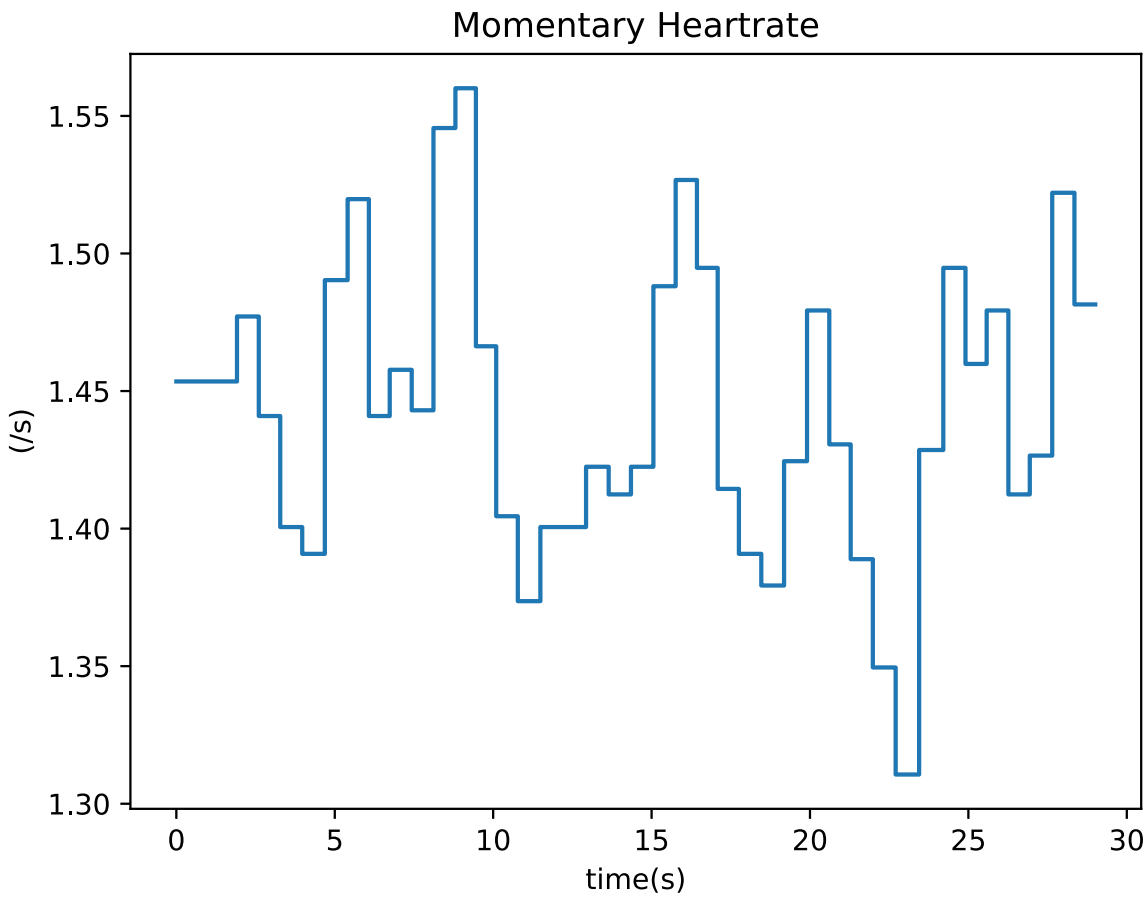
```
for i in range(len(InverseInterval)):
    y_output.append(InverseInterval[i])
x_output.append(0)
for i in range(1, len(y_output),2):
    x_output.append(res3[int(i / 2)])
    x_output.append(res3[int(i / 2)])

x_output = x_output[:-1]
```



But that doesn't seem intuitive, so let's add y twice as well. This will give us a smooth line, which is easier to see.

```
    for i in range(len(InverseInterval)):
        y_output.append(InverseInterval[i])
        # y_output.append(InverseInterval[i])
    x_output.append(0)
    for i in range(1, len(y_output),2):
        x_output.append(res3[int(i / 2)])
        x_output.append(res3[int(i / 2)])
```



So this person's heart rate is somewhere between 1.3/s and 1.6 /s.

Appendix:

Here is the GitHub repository link of this project: https://github.com/DSP-Lab-Group/Lab-02-FIR

firfilter.py:

```
import abc
import numpy as np


class FirFilter:
    def __init__(self, _h):
        self.h = _h
        self.M = len(_h)
        # Ring buffer
        self.buffer = np.zeros(self.M)
        self.offset = self.M - 1

    # Abstract design method
    @abc.abstractmethod
    def _design_filter(self, w):
        pass

    # Real time filter
    def _do_filter(self, input):
        self.buffer[self.offset] = input
        # Move the offset
        self.offset -= 1
        if self.offset < 0:
            self.offset = self.M - 1
        # Calculate the output
        output = 0
        for i in range(self.M):
            output += self.h[i] * self.buffer[(i + self.offset) % self.M]
        return output

    # Entire filter process
    def do_total_filter(self, input):
        n = len(input)
        output = np.zeros(n)
        for i in range(n):
            output[i] = self._do_filter(input[i])
        return output


class BandStopFilter(FirFilter):
    def __init__(self, _sample_rate, _w_1, _w_2, _freq_resolution=1):
        self.sample_rate = _sample_rate
```

```
        self.freq_resolution = _freq_resolution
        h = self._design_filter([_w_1, _w_2])
        super().__init__(h)

    def _design_filter(self, w):
        fs = self.sample_rate
        M = int(fs / self.freq_resolution)
        w_1 = int(w[0] / fs * M)
        w_2 = int(w[1] / fs * M)
        # Define the ideal frequency response
        X = np.ones(M)
        X[w_1:w_2 + 1] = 0
        X[M - w_2:M - w_1 + 1] = 0
        # Compute h(n) using the inverse FFT
        x = np.fft.ifft(X)
        x = np.real(x)
        # Mirror h(n)
        h = np.zeros(M)
        h[0:int(M / 2)] = x[int(M / 2):M]
        h[int(M / 2):M] = x[0:int(M / 2)]
        # Window h(n) using a Hamming window
        h = h * np.hamming(M)

        return h


class HighPassFilter(FirFilter):
    def __init__(self, _sample_rate, _w_c, _freq_resolution=1):
        self.sample_rate = _sample_rate
        self.freq_resolution = _freq_resolution
        h = self._design_filter(_w_c)
        super().__init__(h)

    def _design_filter(self, w):
        fs = self.sample_rate
        M = int(fs / self.freq_resolution)
        w_c = int(w / fs * M)
        # Define the ideal frequency response
        X = np.ones(M)
        X[0:w_c + 1] = 0
        X[M - w_c:M + 1] = 0
        # Compute h(n) using the inverse FFT
        x = np.fft.ifft(X)
        x = np.real(x)
        # Mirror h(n)
        h = np.zeros(M)
        h[0:int(M / 2)] = x[int(M / 2):M]
        h[int(M / 2):M] = x[0:int(M / 2)]
        # Window h(n) using a Hamming window
```

```
        h = h * np.hamming(M)

        return h


class LmsFilter(FirFilter):
    def __init__(self, _sample_rate, _noise_freq, _learning_rate, _freq_resolution=1):
        self.sample_rate = _sample_rate
        self.noise_freq = _noise_freq
        self.learning_rate = _learning_rate
        self.freq_resolution = _freq_resolution
        h = self._design_filter(None)
        super().__init__(h)

    def _design_filter(self, w):
        h = np.zeros(int(self.sample_rate / self.freq_resolution))
        return h

    def _do_filter(self, input, noise=None):
        # Calculate thr error
        canceller = super()._do_filter(noise)
        output = input - canceller
        # Update the h(n)
        for i in range(self.M):
            self.h[i] += output * self.learning_rate * self.buffer[(i + self.offset) %
self.M]
        return output

    def do_total_filter(self, input):
        n = len(input)
        output = np.zeros(n)
        for i in range(n):
            noise = np.sin(2.0 * np.pi * self.noise_freq / self.sample_rate * i)
            output[i] = self._do_filter(input[i], noise)
        return output
```

hpbsfilter.py:

```
import matplotlib.pyplot as plt
import numpy as np

import constant
import util
from firfilter import BandStopFilter, HighPassFilter

import matplotlib
matplotlib.use('TkAgg')
```

```
if __name__ == '__main__':
    band_stop_w_1 = 49
    band_stop_w_2 = 51
    high_pass_w_c = 2
    original_data = np.loadtxt(constant.file_path)
    n = len(original_data)
    # Create band-stop filter
    band_stop_filter = BandStopFilter(constant.sample_rate, band_stop_w_1,
band_stop_w_2)
    # Create high-pass filter
    high_pass_filter = HighPassFilter(constant.sample_rate, high_pass_w_c)
    # Remove the 50Hz interference by the band-stop filter
    filtered_data = band_stop_filter.do_total_filter(original_data)
    # Process the baseline wander by the high-pass filter
    filtered_data = high_pass_filter.do_total_filter(filtered_data)
    # Plot the original ECG time domain
    original_time, original_amplitude = util.cal_time_domain(original_data,
constant.sample_rate)
    plt.subplot(2, 2, 1)
    plt.plot(original_time, original_amplitude)
    plt.title('Original ECG Time Domain')
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    # Plot the original ECG frequency domain
    original_freq, original_amplitude = util.cal_frequency_domain(original_data,
constant.sample_rate)
    plt.subplot(2, 2, 2)
    plt.plot(original_freq, original_amplitude)
    plt.title('Original ECG Frequency Domain')
    plt.xlabel("Frequency")
    plt.ylabel("Amplitude")
    # Plot the filtered ECG time domain
    filtered_time, filtered_amplitude = util.cal_time_domain(filtered_data,
constant.sample_rate)
    plt.subplot(2, 2, 3)
    plt.plot(filtered_time, filtered_amplitude)
    plt.title('Filtered ECG Time Domain')
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    # Plot the filtered ECG frequency domain
    filtered_freq, filtered_amplitude = util.cal_frequency_domain(filtered_data,
constant.sample_rate)
    plt.subplot(2, 2, 4)
    plt.plot(filtered_freq, filtered_amplitude)
    plt.title('Filtered ECG Frequency Domain')
    plt.xlabel("Frequency")
    plt.ylabel("Amplitude")
    plt.tight_layout()
    plt.savefig("res/task_2.svg")
```

```
    plt.show()
```

lmsfilter.py:

```
import matplotlib.pyplot as plt
import numpy as np

import constant
import util
from firfilter import LmsFilter, HighPassFilter

import matplotlib
matplotlib.use('TkAgg')

if __name__ == '__main__':
    noise_freq = 50
    learning_rate = 0.001
    high_pass_w_c = 2
    original_data = np.loadtxt(constant.file_path)
    n = len(original_data)
    # Create LMS filter
    lms_filter = LmsFilter(constant.sample_rate, noise_freq, learning_rate,
constant.freq_resolution)
    # Create high-pass filter
    high_pass_filter = HighPassFilter(constant.sample_rate, high_pass_w_c)
    # Remove the 50Hz interference by the lms filter
    filtered_data = lms_filter.do_total_filter(original_data)
    # Process the baseline wander by the high-pass filter
    filtered_data = high_pass_filter.do_total_filter(filtered_data)
    # Plot the original ECG time domain
    original_time, original_amplitude = util.cal_time_domain(original_data,
constant.sample_rate)
    plt.subplot(2, 2, 1)
    plt.plot(original_time, original_amplitude)
    plt.title('Original ECG Time Domain')
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    # Plot the original ECG frequency domain
    original_freq, original_amplitude = util.cal_frequency_domain(original_data,
constant.sample_rate)
    plt.subplot(2, 2, 2)
    plt.plot(original_freq, original_amplitude)
    plt.title('Original ECG Frequency Domain')
    plt.xlabel("Frequency")
    plt.ylabel("Amplitude")
    # Plot the filtered ECG time domain
    filtered_time, filtered_amplitude = util.cal_time_domain(filtered_data,
constant.sample_rate)
```

```
    plt.subplot(2, 2, 3)
    plt.plot(filtered_time, filtered_amplitude)
    plt.title('Filtered ECG Time Domain')
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    # Plot the filtered ECG frequency domain
    filtered_freq, filtered_amplitude = util.cal_frequency_domain(filtered_data,
constant.sample_rate)
    plt.subplot(2, 2, 4)
    plt.plot(filtered_freq, filtered_amplitude)
    plt.title('Filtered ECG Frequency Domain')
    plt.xlabel("Frequency")
    plt.ylabel("Amplitude")
    plt.tight_layout()
    plt.savefig("res/task_3.svg")
    plt.show()
```

hrdetect.py:

```
import matplotlib
import numpy as np
import constant

import firdesign
from firfilter import FirFilter, BandStopFilter,HighPassFilter

matplotlib.use('TkAgg')
import matplotlib.pyplot as plt


def createWavelet():
    t = np.arange(-0.4, 0.4, 1 / 1250)
    w = 250
    y = np.sin(w * t) / (w * t)
    return y

if __name__ == '__main__':
    y1 = createWavelet()
    data = np.loadtxt('ECG_1000Hz.dat')
    PeakTimeForECG = np.zeros(30000)
    PeakNum = 0
    Frequency_Resolution = 1
    OutputAfterHighpassFilter = np.zeros(len(data))
    OutputAfterBandStopFilter = np.zeros(len(data))
    res1 = np.zeros(len(data))
    res2 = np.zeros(len(data))
    t = np.arange(0, 30000)
    t = t / 1000
```

```
    res3 = []

    high_pass_w_c = 2
    band_stop_w_c = [49, 51]
    sample_rate = 1000


    high_pass_filter = HighPassFilter(constant.sample_rate,high_pass_w_c)
    OutputAfterHighpassFilter = high_pass_filter.do_total_filter(data)

    band_stop_filter = BandStopFilter(constant.sample_rate, band_stop_w_c[0],
band_stop_w_c[1])
    OutputAfterBandStopFilter =
band_stop_filter.do_total_filter(OutputAfterHighpassFilter)

    template1 = OutputAfterBandStopFilter[10500:11500]  # create template
    template2 = createWavelet()
    fir_coeff1 = template1[::-1]  # reverse time
    fir_coeff2 = template2[::-1]

    template_t = np.arange(-500, 500)
    template_t = template_t / 1000

    plt.figure(1)

    plt.subplot(2, 1, 1)
    plt.xlabel("time")
    plt.ylabel("Amplitude")
    plt.title("Template")
    plt.plot(template_t, fir_coeff1)

    plt.subplot(2, 1, 2)
    plt.xlabel("time")
    plt.ylabel("Amplitude")
    plt.title("Wavelet")
    plt.plot(template_t, fir_coeff2)
    plt.savefig("res/task_4_(Template and Wavelet).svg")


    filter1 = FirFilter(fir_coeff1)
    filter2 = FirFilter(fir_coeff2)

    res2 = filter2.do_total_filter(OutputAfterBandStopFilter)
    res2 = abs(res2)


    plt.figure(2)
    plt.xlabel("time")
    plt.ylabel("Amplitude")
```

```
    plt.title("Output with R peaks detected")
    plt.plot(t, res2)
    plt.savefig("res/task_4_(R peaks detected).svg")

    for i in range(len(data)):
        if res2[i] <= 5:
            res2[i] = 0
        else:
            PeakTimeForECG[PeakNum] = i
            PeakNum += 1

    PeakTimeForECG = PeakTimeForECG[:PeakNum] # the place to time
    PeakTimeForECG = PeakTimeForECG / 1000

    res3.append(PeakTimeForECG[0])
    for i in range(1, len(PeakTimeForECG)):
        if PeakTimeForECG[i] - PeakTimeForECG[i - 1] > 0.3: # remove the error, time
interval > 0.3
            res3.append(PeakTimeForECG[i])

    InverseInterval = []
    y_output = []
    x_output = []
    for i in range(1, len(res3)):
        InverseInterval.append((1 / (res3[i] - res3[i - 1]))) # Real heart rate

    for i in range(len(InverseInterval)):
        y_output.append(InverseInterval[i])
        y_output.append(InverseInterval[i])
    x_output.append(0)
    for i in range(1, len(y_output),2):
        x_output.append(res3[int(i / 2)])
        x_output.append(res3[int(i / 2)])

    x_output = x_output[:-1]

    t1 = np.arange(0, 20)
    plt.figure(3)
    plt.xlabel("time")
    plt.ylabel("Amplitude")
    plt.title("Output after thresholding")
    plt.plot(t, res2)
    plt.savefig("res/task_4_(Output after thresholding).svg")

    plt.figure(4)
    plt.plot(x_output, y_output)
    plt.xlabel("time(s)")
    plt.ylabel("(/s)")
    plt.title("Momentary Heartrate")
```

```
    plt.savefig("res/task_4_(Momentary Heartrate).svg")
    plt.show()
```