

# unit\_test

December 6, 2020

## 1 Test Your Algorithm

### 1.1 Instructions

1. From the **Pulse Rate Algorithm** Notebook you can do one of the following:
  - Copy over all the **Code** section to the following Code block.
  - Download as a Python (.py) and copy the code to the following Code block.
2. In the bottom right, click the Test Run button.

#### 1.1.1 Didn't Pass

If your code didn't pass the test, go back to the previous Concept or to your local setup and continue iterating on your algorithm and try to bring your training error down before testing again.

#### 1.1.2 Pass

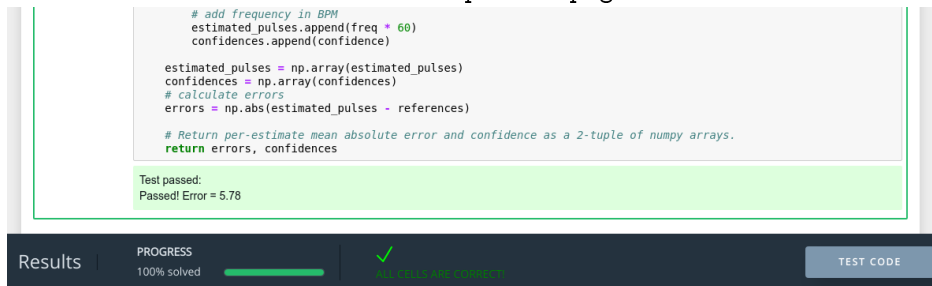
If your code passes the test, complete the following! You **must** include a screenshot of your code and the Test being **Passed**. Here is what the starter filler code looks like when the test is run and should be similar. A passed test will include in the notebook a green outline plus a box with **Test passed:** and in the Results bar at the bottom the progress bar will be at 100% plus a checkmark with



All cells passed.

1. Take a screenshot of your code passing the test, make sure it is in the format .png. If not a .png image, you will have to edit the Markdown render the image after Step 3. Here is an example of what the passed.png would look like
2. Upload the screenshot to the same folder or directory as this jupyter notebook.

3. Rename the screenshot to `passed.png` and it should show up below.




```
# add frequency in BPM
estimated_pulses.append(freq * 60)
confidences.append(confidence)

estimated_pulses = np.array(estimated_pulses)
confidences = np.array(confidences)
# calculate errors
errors = np.abs(estimated_pulses - references)

# Return per-estimate mean absolute error and confidence as a 2-tuple of numpy arrays.
return errors, confidences

Test passed:
Passed! Error = 5.78
```

Results | PROGRESS 100% solved |  All tests passed | TEST CODE

4. Download this jupyter notebook as a .pdf file.  
5. Continue to Part 2 of the Project.

In [2]: `import glob`

```
import numpy as np
import scipy as sp
import scipy.signal
import scipy.io
import time
import matplotlib.pyplot as plt
```

```
# import mpld3
# mpld3.enable_notebook()
```

```
def LoadTroikaDataset():
```

```
    """
```

```
    Retrieve the .mat filenames for the troika dataset.
```

```
    Review the README in ./datasets/troika/ to understand the organization of the .mat f
```

```
    Returns:
```

```
        data_fls: Names of the .mat files that contain signal data
```

```
        ref_fls: Names of the .mat files that contain reference data
```

```
        <data_fls> and <ref_fls> are ordered correspondingly, so that ref_fls[5] is the
            reference data for data_fls[5], etc...
```

```
    """
```

```
    data_dir = "./datasets/troika/training_data"
```

```
    data_fls = sorted(glob.glob(data_dir + "/DATA_*.mat"))
```

```
    ref_fls = sorted(glob.glob(data_dir + "/REF_*.mat"))
```

```
    return data_fls, ref_fls
```

```
def LoadTroikaDataFile(data_fl):
```

```
    """
```

```
    Loads and extracts signals from a troika data file.
```

```
    Usage:
```

```
        data_fls, ref_fls = LoadTroikaDataset()
```

```

        ppg, accx, accy, accz = LoadTroikaDataFile(data_fls[0])

    Args:
        data_fl: (str) filepath to a troika .mat file.

    Returns:
        numpy arrays for ppg, accx, accy, accz signals.
    """
    data = sp.io.loadmat(data_fl)['sig']
    return data[2:]

def AggregateErrorMetric(pr_errors, confidence_est):
    """
    Computes an aggregate error metric based on confidence estimates.

    Computes the MAE at 90% availability.

    Args:
        pr_errors: a numpy array of errors between pulse rate estimates and corresponding
            reference heart rates.
        confidence_est: a numpy array of confidence estimates for each pulse rate
            error.

    Returns:
        the MAE at 90% availability
    """
    # Higher confidence means a better estimate. The best 90% of the estimates
    # are above the 10th percentile confidence.
    percentile90_confidence = np.percentile(confidence_est, 10)

    # Find the errors of the best pulse rate estimates
    best_estimates = pr_errors[confidence_est >= percentile90_confidence]

    # Return the mean absolute error
    return np.mean(np.abs(best_estimates))

def Evaluate():
    """
    Top-level function evaluation function.

    Runs the pulse rate algorithm on the Troika dataset and returns an aggregate error m

    Returns:
        Pulse rate error on the Troika dataset. See AggregateErrorMetric.
    """
    # Retrieve dataset files
    data_fls, ref_fls = LoadTroikaDataset()

```

```

errs, confs = [], []
for data_fl, ref_fl in zip(data_fls, ref_fls):
    # Run the pulse rate algorithm on each trial in the dataset
    errors, confidence = RunPulseRateAlgorithm(data_fl, ref_fl)
    errs.append(errors)
    confs.append(confidence)
    # Compute aggregate error metric
errs = np.hstack(errs)
confs = np.hstack(confs)
return AggregateErrorMetric(errs, confs)

def test_function(idx, figures="", specgram=False):
    """
    Top-level function to test code.
    Runs the pulse rate algorithm on one subject of the Troika dataset
    and prints some metrics and display figures.
    Parameters: - idx: positive int, index of subject to evaluate
                - figures: array of strings, option to show signals' plot
                        can be ["raw"] or ["filtered"] or both
                - specgram: boolean, option to show signals' spectrogram
    Returns: None
    """
    # Retrieve a subject's dataset file
    data_fls, ref_fls = LoadTroikaDataset()
    data, ref = data_fls[idx], ref_fls[idx]
    # Run the pulse rate algorithm on one trial of the dataset
    errors, confidence = RunPulseRateAlgorithm(data, ref, figures=figures, specgram=specgram)

    print("Mean error: %.3f"%(AggregateErrorMetric(errors, confidence)))

def figure_params(kind=None):
    """
    Auxiliary function setting some parameters
    to plot figures.
    Parameters: - kind: type of figure to show, only 'specgram' available.
    Returns: None
    """
    if (kind == "specgram"):
        plt.ylabel("Frequency (Hz)", size=16)
        plt.ylim(40/60, 240/60)
    else:
        plt.ylabel("Signal (a.u.)", size=16)
        plt.legend(fontsize=15)

    plt.xlabel("Time (minutes)", size=16)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.show()

```

```

def plot_data(ppg, accx, accy, accz, fs, specgram=False, description='Raw'):
    """
    Function to plot and show the signals ppg, accx, accy, accz.
    Inputs: - ppg: Numpy array of PPG signal
            - accx: Numpy array of PPG signal
            - accy: Numpy array of PPG signal
            - accz: Numpy array of PPG signal
            - fs: sampling rate in Hz, assumed equal for all signals
            - specgram: boolean, option to show signals' spectrogram
            - description: string describing signal
    Returns: None
    """
    figsize = (10,6)
    # time stamps
    ts = np.arange(len(ppg)) / fs / 60
    print("-----")
    print(description + ' signal:')
    print("-----")
    # plot PPG signal
    plt.figure(figsize=figsize)
    plt.plot(ts, ppg, label="PPG")
    plt.title(description + " photoplethysmogram signal", size=20)
    figure_params()
    # plot accelerometers signal
    plt.figure(figsize=figsize)
    plt.plot(ts, accx, label="x")
    plt.plot(ts, accy, label="y")
    plt.plot(ts, accz, label="z")
    plt.title(description + " accelerometers signal", size=20)
    figure_params()
    plt.figure(figsize=figsize)
    # plot total acceleration amplitude
    acc = np.sqrt(np.square(accx) + np.square(accy) + np.square(accz))
    plt.plot(ts, acc, label="Total magnitude")
    plt.title(description + " accelerometer total amplitude signal", size=20)
    figure_params()
    # plot specgrams if selected
    if specgram:
        kind = 'specgram'
        # PPG signal
        plt.figure(figsize=figsize)
        plt.specgram(ppg, Fs=fs, NFFT=8 * fs, noverlap=6 * fs, xextent=(0, len(ppg) / fs))
        plt.title(description + " photoplethysmogram signal", size=20)
        figure_params(kind)
        # accx signal
        plt.figure(figsize=figsize)
        plt.specgram(accx, Fs=fs, NFFT=8 * fs, noverlap=6 * fs, xextent=(0, len(ppg) / fs))

```

```

plt.title(description + " accelerometer signal: x-direction", size=20)
figure_params(kind)
# accy signal
plt.figure(figsize=figsize)
plt.specgram(accy, Fs=fs, NFFT=8 * fs, noverlap=6 * fs, xextent=(0, len(ppg) / fs)
plt.title(description + " accelerometer signal: x-direction", size=20)
figure_params(kind)
# accz signal
plt.figure(figsize=figsize)
plt.specgram(accz, Fs=fs, NFFT=8 * fs, noverlap=6 * fs, xextent=(0, len(ppg) / fs)
plt.title(description + " accelerometer signal: x-direction", size=20)
figure_params(kind)
# total acc signal
plt.figure(figsize=figsize)
plt.specgram(acc, Fs=fs, NFFT=8 * fs, noverlap=6 * fs, xextent=(0, len(ppg) / fs)
plt.title(description + " accelerometer total magnitude signal", size=20)
figure_params(kind)

def BandpassFilter(signal, pass_band, fs):
    '''
    Applies a bandpass filter to the signal.
    Parameters: - signal: numpy array, input signal
                  - pass_band: tuple (frequency_min, frequency_max), frequency pass band,
                          components outside the tuple range will be removed.
                  - fs: sampling rate in Hz
    Returns: - Filtered signal as a numpy.array
    '''

    b, a = sp.signal.butter(3, pass_band, btype='bandpass', fs=fs)

    return sp.signal.filtfilt(b, a, signal)

def extract_frequency(freqs, ppg_fft_amp, acc_fft_amp, accx_fft_amp, accy_fft_amp, accz_
    accxy_fft_amp, accxz_fft_amp, accyz_fft_amp):
    '''
    Returns the frequency and position in freqs array corresponding to the
    larger amplitude in ppg_fft_amp that is also not present in acc_fft_amp.
    Parameters: - freqs: numpy array of positive values, frequencies of FFT
                  - ppg_fft_amp: numpy array, corresponding PPG's FFT amplitudes
                  - acc_fft_amp: numpy array, corresponding acceleration's magnitude FFT a
                  - accx_fft_amp: numpy array, corresponding x acceleration's FFT amplitud
                  - accy_fft_amp: numpy array, corresponding y acceleration's FFT amplitud
                  - accz_fft_amp: numpy array, corresponding z acceleration's FFT amplitud
                  - accxy_fft_amp: numpy array,
                          corresponding acceleration's projection in x-y plane FF
                  - accxz_fft_amp: numpy array,
                          corresponding acceleration's projection in x-z plane FF
                  - accyz_fft_amp: numpy array,
                          corresponding acceleration's projection in y-z plane FF

```

```

Returns: - freq: the corresponding frequency
         - idx: index corresponding to freq
'''
# indexes for input arrays
idxs = list(range(len(freqs)))
# sort amplitudes in descending order
# by keeping their position in original array
sort_ppg_fft_amp = sorted(list(zip(ppg_fft_amp, idxs)), reverse=True)
sort_acc_fft_amp = sorted(list(zip(acc_fft_amp, idxs)), reverse=True)
sort_accx_fft_amp = sorted(list(zip(accx_fft_amp, idxs)), reverse=True)
sort_accy_fft_amp = sorted(list(zip(accy_fft_amp, idxs)), reverse=True)
sort_accz_fft_amp = sorted(list(zip(accz_fft_amp, idxs)), reverse=True)
sort_accxy_fft_amp = sorted(list(zip(accxy_fft_amp, idxs)), reverse=True)
sort_accxz_fft_amp = sorted(list(zip(accxz_fft_amp, idxs)), reverse=True)
sort_accyz_fft_amp = sorted(list(zip(accyz_fft_amp, idxs)), reverse=True)

# maximum amplitude in PPG's FFT frequencies
max_mag = sort_ppg_fft_amp[0][0]
thresh = 0.22
# check that the frequency of max amplitude
# for ppg is not in the components other signals
# and if signals with smaller amplitude need to be
# considered, then don't consider amplitudes having
# below given percentage threshold of max amplitude
for ppg_tuple, acc_tuple, accx_tuple, accy_tuple, accz_tuple, \
    accxy_tuple, accxz_tuple, accyz_tuple \
    in zip(sort_ppg_fft_amp, sort_acc_fft_amp, \
           sort_accx_fft_amp, sort_accy_fft_amp, sort_accz_fft_amp, \
           sort_accxy_fft_amp, sort_accxz_fft_amp, sort_accyz_fft_amp
           ):
    if (ppg_tuple[1] != acc_tuple[1]) & (ppg_tuple[1] != accx_tuple[1]) \
        & (ppg_tuple[1] != accy_tuple[1]) & (ppg_tuple[1] != accz_tuple[1]) \
        & (ppg_tuple[1] != accxy_tuple[1]) & (ppg_tuple[1] != accxz_tuple[1]) \
        & (ppg_tuple[1] != accyz_tuple[1]) & (ppg_tuple[0] > thresh * max_mag):

        idx = ppg_tuple[1]
        freq = freqs[idx]

        return freq, idx

# if no solution in loop return ppg's frequency of max amplitude
idx = sort_ppg_fft_amp[0][1]
freq = freqs[idx]

return freq, idx

def RunPulseRateAlgorithm(data_fl, ref_fl, figures="", specgram=False):
'''

```

*Estimates pulse rate for PPG data in data\_fl using also accelerometer data. Assumes puls rate between 40 BPM and 240 BPM and gives an output every 2 seconds with a confidence estimation.*

*Parameters:*

- *data\_fl*: array containing data to analyze, each element corresponds to a subject
- *ref\_fl*: array containing the corresponding target frequencies in BPM
- *figures*: array of strings, option to show signals' plot can be ["raw"] or ["filtered"] or both
- *specgram*: boolean, option to show signals' spectrogram

```
'''
# Load data using LoadTroikaDataFile
ppg, accx, accy, accz = LoadTroikaDataFile(data_fl)

# subtract mean value, imperical decision
# shown to give a better frequeuncy estimate
ppg = ppg - np.mean(ppg)
accx = accx - np.mean(accx)
accy = accy - np.mean(accy)
accz = accz - np.mean(accz)
# calculate total acceleration
acc = np.sqrt(np.square(accx) + np.square(accy) + np.square(accz))
# acceleration in x-y plane
accxy = np.sqrt(np.square(accx) + np.square(accy))
# acceleration in x-z plane
accxz = np.sqrt(np.square(accx) + np.square(accz))
# acceleration in y-z plane
accyz = np.sqrt(np.square(accy) + np.square(accz))

# Load data from reference file
references = sp.io.loadmat(ref_fl)['BPM0'].flatten()

# Sampling rate in Hz according to information in Readme.pdf
fs = 125
# minimum frequency from BPM to Hz
freq_min = 40 / 60
# upper limit frequency from BPM to Hz
freq_max = 240 / 60
# band pass
bp_lims = (freq_min, freq_max)
# window of 2 secs for frequency estimation
# in number of samples
window_shift = 2 * fs
# window length to match window length of references
window_length = np.int(np.around((len(ppg) - len(references) * window_shift)))

# option to plot raw signals
if ("raw" in figures):
    plot_data(ppg, accx, accy, accz, fs, specgram=specgram)
```



```

# filtering signals bewteen freq_min and freq_max
bp_ppg = BandpassFilter(ppg, bp_lims, fs)
bp_acc = BandpassFilter(acc, bp_lims, fs)
bp_accx = BandpassFilter(accx, bp_lims, fs)
bp_accy = BandpassFilter(acy, bp_lims, fs)
bp_accz = BandpassFilter(accz, bp_lims, fs)
bp_accxy = BandpassFilter(accxy, bp_lims, fs)
bp_accxz = BandpassFilter(accxz, bp_lims, fs)
bp_accyz = BandpassFilter(acyz, bp_lims, fs)

# option to plot filtered signals
if ("filtered" in figures):
    plot_data(bp_ppg, bp_accx, bp_accy, bp_accz, fs, specgram=specgram, description=

# Compute pulse rate estimates and estimation confidence.
estimated_pulses = []
confidences = []
for i in range(0, len(bp_ppg) - window_length, window_shift):
    # windowed signals
    windowed_ppg = bp_ppg[i:i + window_length]
    windowed_acc = bp_acc[i:i + window_length]
    windowed_accx = bp_accx[i:i + window_length]
    windowed_accy = bp_accy[i:i + window_length]
    windowed_accz = bp_accz[i:i + window_length]
    windowed_accxy = bp_accxy[i:i + window_length]
    windowed_accxz = bp_accxz[i:i + window_length]
    windowed_accyz = bp_accyz[i:i + window_length]

    # calculate FFTs, and keep only frequencies
    # between freq_min and freq_max
    freqs = np.fft.rfftfreq(len(windowed_ppg), 1 / fs)
    filter_freqs = (freqs > freq_min) & (freqs < freq_max)

    # get FFTs amplitudes
    ppg_fft_amp = np.abs(np.fft.rfft(windowed_ppg))[filter_freqs]
    acc_fft_amp = np.abs(np.fft.rfft(windowed_acc))[filter_freqs]
    accx_fft_amp = np.abs(np.fft.rfft(windowed_accx))[filter_freqs]
    accy_fft_amp = np.abs(np.fft.rfft(windowed_accy))[filter_freqs]
    accz_fft_amp = np.abs(np.fft.rfft(windowed_accz))[filter_freqs]
    accxy_fft_amp = np.abs(np.fft.rfft(windowed_accxy))[filter_freqs]
    accxz_fft_amp = np.abs(np.fft.rfft(windowed_accxz))[filter_freqs]
    accyz_fft_amp = np.abs(np.fft.rfft(windowed_accyz))[filter_freqs]
    freqs = freqs[filter_freqs]

    # get frequency of max amplitude and its corresponding index
    freq, idx = extract_frequency(freqs, ppg_fft_amp, acc_fft_amp, \
                                   accx_fft_amp, accy_fft_amp, accz_fft_amp, \

```

```

        accxy_fft_amp, accxz_fft_amp, accyz_fft_amp
    )

    # calculate window's confidence near the estimated frequency
    # value determined empirically
    neighbors = 4
    confidence = np.sum(ppg_fft_amp[idx - neighbors:idx + neighbors]) / np.sum(ppg_f

    # add frequency in BPM
    estimated_pulses.append(freq * 60)
    confidences.append(confidence)

estimated_pulses = np.array(estimated_pulses)
confidences = np.array(confidences)
# calculate errors
errors = np.abs(estimated_pulses - references)

# Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
return errors, confidences

```