# unit_test

November 25, 2023

## 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.

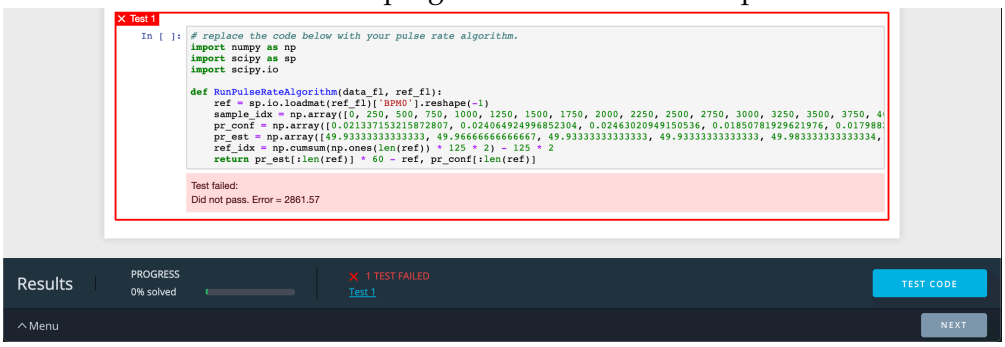
2. In the bottom right, click the Test Run button.

#### 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.

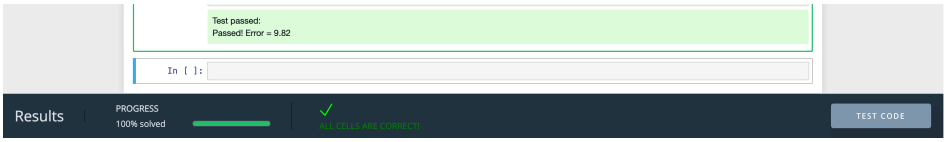
#### 1.1.2 Pass

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.




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.io

from matplotlib import pyplot as plt
from scipy import signal as sg


FS = 125  # All signals were sampled at 125 Hz

# The following parameters match the structure of the test data
WINDOW_LENGTH_SEC = 8  # Window length used in to compute the spectogram frequencies
OVERLAP = 6  # Two successive time windows overlap by 6 seconds


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 LoadTroikaRefFile(ref_fl):
    """
    Loads and extracts the ground-truth of heart rate from a troika data file.

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

    Returns:
        numpy arrays for the BPM value in every 8-second time window. Note that two succ
overlap by 6 seconds. Thus the first value in 'BPM0' gives the calcualted heart rate gro
the first 8 seconds, while the second value in 'BPM0' gives the calculated heart rate gr
from the 3rd second to the 10th second.
    """
    return sp.io.loadmat(ref_fl)['BPM0']


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 correspondin
            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 bpm_to_hz(bpm):
    """
    Convert a magnitude given in Beats per Minute (BPM) to Hertz (Hz)
    """
    return bpm / 60


def hz_to_bpm(hz):
    """
    Convert a magnitude given in Hertz (Hz) to Beats per Minute (BPM)
    """
    return hz * 60


def BandpassFilter(signal):
    """
    A Bandpass filter to remove frequencies outside the range 40-240 Beats per Minute
```

```
    Returns:
        A ndarray containing the filtered signal
    """
    low_freq, high_freq = bpm_to_hz(40), bpm_to_hz(240)
    b, a = sg.butter(5, (low_freq, high_freq), btype='bandpass', fs=FS)
    return sg.filtfilt(b, a, signal)


def Spectogram(signal):
    """
    Compute the spectogram and frequencies of a given signal.

    Data are split into segments of 8 seconds with an overlap of 6 seconds.

    Returns:
        spec: 2D ndarray
            Columns containing the periodograms of successive segments.
        freqs: 1D ndarray
            The frequencies corresponding to the rows in spectrum.
    """
    _ = plt.figure()
    plt.title(f'Spectrogram')
    spec, freqs, _, _ = plt.specgram(
        signal,
        Fs=FS,
        NFFT=WINDOW_LENGTH_SEC * FS,
        noverlap=OVERLAP * FS,
    )
    plt.xlabel('Time (sec)')
    plt.ylabel('Frequency (Hz)')
    plt.close()  # Comment to show plot
    return spec, freqs


def SimilarFreqs(freq_a, window_hz, freq_b):
    """
    Indicates whether the difference between two frequencies fall within a window.
    """
    return (
        (freq_a >= freq_b - window_hz / 2) &
        (freq_a <= freq_b + window_hz / 2)
    )


def EstimationConfidence(spec, hr_freqs, all_freqs):
    """
    Computes the confidence of the estimations
```

```
    The confidence is given by the energy around the estimated frequency
    over the total energy along the spectrum.

    Returns:
        confidences: 1D ndarray
            Array with the energy ratios
    """
    # Get freqs around the heart rate
    w_f = bpm_to_hz(10)

    confidences = []
    for i, time_window in enumerate(hr_freqs):
        fund_freqs = (all_freqs >= hr_freqs[i] - w_f) & (all_freqs <= hr_freqs[i] + w_f)

        hr_signal = np.sum(spec[:, i][fund_freqs])
        all_signals = np.sum(spec[:, i])

        confidences.append(hr_signal / all_signals)
    return np.array(confidences)


def SimilarToACC(ppg_freq, window, freq_accx, freq_accy, freq_accz):
    """
    Compares a PPG frequency with the 3 axis of the Accelerometer

    Returns:
        Boolean indicating whether the frequencies fall into a similar range
    """
    return (
        SimilarFreqs(ppg_freq, window, freq_accx) |
        SimilarFreqs(ppg_freq, window, freq_accy) |
        SimilarFreqs(ppg_freq, window, freq_accz)
    )


def FindHeartbeatFrequency(
    top_ppg_freqs,
    dom_freq_accx,
    dom_freq_accy,
    dom_freq_accz,
    avg_latest_freqs,
):
    """
    Find the heart beat frequency given by the PPG sensor

    The algorithm iterates over the top PPG frequencies to find the one related to
    heart beats. Frequencies similar to the dominant frequency of the accelerometer
    are skipped. In case all frequencies are similar to the accelerometer, the
```

```
    frequency that resembles the past estimations is considered.

    Returns:
        heartbeat_freq: float
            Value of the estimated heartbeat frequency in Hz
    """
    heartbeat_freq = None
    for top_ppg_freq in top_ppg_freqs:
        bpm_window = 15  # Window of frequencies around the
                         # PPG to compare against the ACC
        window = bpm_to_hz(bpm_window)

        if SimilarToACC(top_ppg_freq, window, dom_freq_accx, dom_freq_accy, dom_freq_acc
            continue

        heartbeat_freq = top_ppg_freq
        break

    # If the cadence of the arm swing is the same as the heartbeat, take
    # the most similar frequency to the latest samples
    if not heartbeat_freq:
        if avg_latest_freqs > 0:
            diffs = np.abs(top_ppg_freqs - avg_latest_freqs)
            heartbeat_freq = top_ppg_freqs[np.argmin(diffs)]
        else:
            heartbeat_freq = top_ppg_freqs[0]
    return heartbeat_freq


def GetTopKFreqs(k, specs, freqs, time_window):
    """
    Iterates over the frequency spectrum and return the top K frequencies
    that maximise the spectrum value.

    Return:
        top_k_freqs: 1D ndarray
            Array of top frequencies sorted in decreasing order
    """
    top_k_freq_idx_asc = np.argpartition(specs[:, time_window], -k)[-k:]
    return freqs[top_k_freq_idx_asc[::-1]]


def FindFrequenciesOverTime(ppg, accx, accy, accz):
    """
    Computes the estimated heart beat frequencies through time.

    Returns:
        motion_compensated_freqs: 1D ndarray
```

```
            Estimated heart beat frequencies in Hz
        spec_ppg: 2D ndarray
            Columns containing the PPG spectrum of successive segments.
        freqs_ppg: 1D ndarray
            The frequencies corresponding to the rows in PPG spectrum.
    """
    signals = (ppg, accx, accy, accz)
    ppg, accx, accy, accz = (BandpassFilter(sig) for sig in signals)
    spec_ppg, freqs_ppg = Spectogram(ppg)
    spec_accx, freqs_accx = Spectogram(accx)
    spec_accy, freqs_accy = Spectogram(accy)
    spec_accz, freqs_accz = Spectogram(accz)

    # Find dominant frequencies for every time window
    motion_compensated_freqs = []
    for time_window in range(spec_ppg.shape[1]):
        top_5_dom_freq_ppg = GetTopKFreqs(5, spec_ppg, freqs_ppg, time_window)

        dom_freq_accx = freqs_accx[np.argmax(spec_accx[:, time_window])]
        dom_freq_accy = freqs_accy[np.argmax(spec_accy[:, time_window])]
        dom_freq_accz = freqs_accz[np.argmax(spec_accz[:, time_window])]

        latest_freqs = np.mean(motion_compensated_freqs[-2:]) if motion_compensated_freq

        heartbeat_freq = FindHeartbeatFrequency(
            top_5_dom_freq_ppg,
            dom_freq_accx,
            dom_freq_accy,
            dom_freq_accz,
            latest_freqs
        )
        motion_compensated_freqs.append(heartbeat_freq)
    return np.array(motion_compensated_freqs), spec_ppg, freqs_ppg


def RunPulseRateAlgorithm(data_fl, ref_fl):
    """
    Estimate pulse rate from the PPG sensor and compare it with ground truth data from d

    Returns:
        errors: 1D ndarray
            Contains the absolute error of the estimations
        confidence: 1D ndarray
            Contains the estimations confidence
    """
    ppg, accx, accy, accz = LoadTroikaDataFile(data_fl)
    y_true = LoadTroikaRefFile(ref_fl).flatten()
```

```
            estimated_freqs, spec_ppg, freqs_ppg = FindFrequenciesOverTime(ppg, accx, accy, accz
            confidence = EstimationConfidence(spec_ppg, estimated_freqs, freqs_ppg)

            y_pred = hz_to_bpm(estimated_freqs)
            errors = np.abs(y_true - y_pred)
            return errors, confidence
```

In [ ]: