# unit_test

April 22, 2021

## 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.

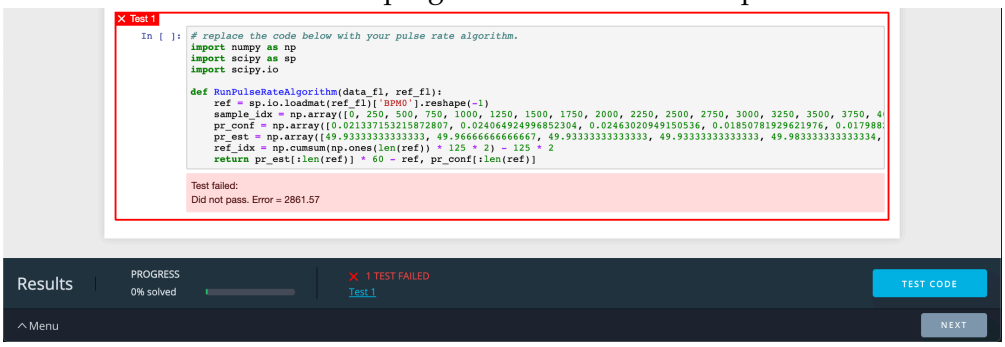
2. In the bottom right, click the Test Run button.

#### 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.

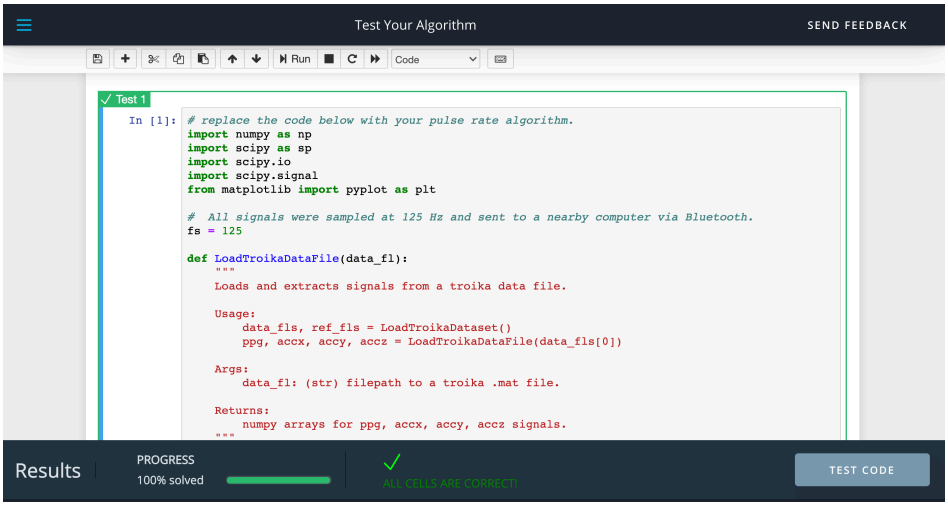
#### 1.1.2 Pass

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 [1]: # replace the code below with your pulse rate algorithm.
        import numpy as np
        import scipy as sp
        import scipy.io
        import scipy.signal
        from matplotlib import pyplot as plt

        #  All signals were sampled at 125 Hz and sent to a nearby computer via Bluetooth.
        fs = 125

        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 RunPulseRateAlgorithm(data_fl, ref_fl):
            """
            Runs the pulse rate algorithm on the Troika dataset and returns an aggregate error m
```

```
    Args:
        data_fl: a path of the .mat files that contain signal data
        ref_fl: a path of the .mat files that contain reference data

    Returns:
        A 2-tuple of numpy arrays of per-estimate mean absolute error and confidence as
    """

    ecgdata = sp.io.loadmat(ref_fl)['BPM0']

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

    # Compute pulse rate estimates and estimation confidence.

    # Get spectogram (FFT) data for ppg, accx, accy, accz
    _, data_spec = get_spectogram_data(ppg, accx, accy, accz, fs)

    filt_ppg_specs, filt_ppg_freqs = data_spec['ppg']['specs'], data_spec['ppg']['freqs'
    accx_specs, accx_freqs = data_spec['accx']['specs'], data_spec['accx']['freqs']
    accy_specs, accy_freqs = data_spec['accy']['specs'], data_spec['accy']['freqs']
    accz_specs, accz_freqs = data_spec['accz']['specs'], data_spec['accz']['freqs']

    # Get the FFT peaks information for ppg, accx, accy, accz
    data_peaks_ppg = get_freq_peaks(data_spec['ppg'])
    data_peaks_accx = get_freq_peaks(data_spec['accx'])
    data_peaks_accy = get_freq_peaks(data_spec['accy'])
    data_peaks_accz = get_freq_peaks(data_spec['accz'])

    # --- Compute an estimated frequency of a ppg signal that
    # will be used to compute a pulse rante for each 8-second time window
    # (overlapped by 6 seconds with a successive window)

    ppg_max_freqs = []

    # For each timeslot,
    for i in range(filt_ppg_specs.shape[1]):

        # Get a frequency of the highest FFT peak of each accelerometer signals
        accx_max_freq = data_peaks_accx[i]['peak_freqs'][0]
        accy_max_freq = data_peaks_accy[i]['peak_freqs'][0]
        accz_max_freq = data_peaks_accz[i]['peak_freqs'][0]

        # Initialize the selected ppg frequency with a frequency for the highest peak
        cur_max_ppg_freq = data_peaks_ppg[i]['peak_freqs'][0]

        # For the top-3 frequencies of the highest FFT peaks of the ppg signals
        lim_f = np.min((len(data_peaks_ppg[i]['peak_freqs']), 3))
```

```
    for f in range(0, lim_f):

        # If a ppg frequency of the current peak matches a frequency of
        # the highest peak of any accelerometer signal
        ppg_freq = data_peaks_ppg[i]['peak_freqs'][f]

        if ( (ppg_freq == accx_max_freq) or
             (ppg_freq == accy_max_freq) or
             (ppg_freq == accz_max_freq) ):

            # continue to check a frequency of the next peak
            continue
        else:

            # choose this ppg frequency for the current time slot
            cur_max_ppg_freq = ppg_freq
            break

    ppg_max_freqs.append(cur_max_ppg_freq)

# --- Compute per-estimate mean absolute error and confidence of
# the estimated pulse rate frequency for each 8-second time window
# (overlapped by 6 seconds with a successive window)

# Mean absolute error: is an absolute differece of
#    an estimated bpm and a reference bpm
# Confidence: is a ratio of the energy around the
#    estimated pulse rate frequence of the ppg signal over the overal energy.
#    where the chosen frequency window used to compute the confidence value is 10 bp

# Window used to compute a confidence value
bpm_sum_window = 10
bps_sum_window = bpm_sum_window/60

# Pulse rate will be restricted between 40BPM (beats per minute) and 240BPM
freq_low = 40 / 60
freq_high = 240 / 60

errors, confidence = [], []

# For each estimated pulse rate frequency of each time window
for i in range(len(ppg_max_freqs)):

    # cap the estimated pulse rate frequency to between 40-240 bpm
    cur_max_freq = ppg_max_freqs[i]

    if cur_max_freq <= freq_low:
        cur_max_freq = freq_low
```

```
        elif cur_max_freq >= freq_high:
            cur_max_freq = freq_high

        ppg_max_freqs[i] = cur_max_freq

    # For each time window of a ppg signal,
    for i in range(filt_ppg_specs.shape[1]):

        # Compute a pulse rate (bpm) of the current time window
        cur_max_freq = ppg_max_freqs[i]
        cur_bpm = cur_max_freq * 60


        # If the current pulse rate (bpm) increases more than 15 bpm comparing to
        # the previous time window (2-second difference), we will use an average of
        # the pulse rate of the last 10 time windows instead to smooth out
        # the estimated pulse rate
        max_diff_bpm = 15
        num_prev = 10
        if (i > num_prev):
            prev_freq = ppg_max_freqs[i-1]
            if abs(prev_freq*60 - cur_bpm) >= max_diff_bpm:
                cur_max_freq = np.mean(ppg_max_freqs[i-num_prev : i])

        # --- Compute an absolute error between the estimated and reference
        # pulse rate of the current time window and append it to the output errors array

        cur_bpm = cur_max_freq * 60
        cur_error = np.abs(cur_bpm - ecgdata[i][0])

        errors.append(cur_error)

        # --- Compute a confidence value

        # First, compute a frequency window that will be used to compute an energy value
        # around the estimated pulse rate
        low_window = cur_max_freq - bps_sum_window
        high_window = cur_max_freq + bps_sum_window
        window = (filt_ppg_freqs >= low_window) & (filt_ppg_freqs <= high_window)

        # Then, compute a confidence value by getting a ratio of the energy of the peak
        # over the entire FFT signal
        cur_specs = filt_ppg_specs[:,i]
        cur_confidence = np.sum(cur_specs[window])/np.sum(cur_specs)

        confidence.append(cur_confidence)

    # Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
```

```
    return errors, confidence

def BandpassFilter(signal, fs=125):
    """
    Bandpass filter the signal between 40-240BPM (40/60=0.667 and 240/60=4 Hz).

    Returns:
        A bandpass filtered signal
    """

    freq_low = 40 / 60
    freq_high = 240 / 60

    b, a = sp.signal.butter(3, (freq_low, freq_high), btype='bandpass', fs=fs)
    return sp.signal.filtfilt(b, a, signal)

def get_spectogram_data(ppg, accx, accy, accz, fs=125):
    """
    Compute bandpass filtered signals and spectogram data (FFT)
    for ppg, accx, accy, accz, and aggregated accelerometer signals

    Args:
        ppg:    PPG signal
        accx:   Accelerometer signal in the x-direction
        accy:   Accelerometer signal in the y-direction
        accz:   Accelerometer signal in the z-direction

    Optional Arg:
        fs: Frequency used to compute the spectogram data (FFT). Default is 125 Hz

    Returns:
        A 2-tuple of a dictionary of bandpass filtered signals and spectogram (FFT)
        data for ppg, accx, accy, accz, and aggregated accelerometer signals
    """

    # Center Y values to remove a gravity value in the y-direction
    accy = accy-np.mean(accy)

    # Getting an aggregated acc signal
    acc_agg = np.sqrt(accx**2 + accy**2 + accz**2)

    # --- Compute a bandpass filtered signal of all input signals
    # (ppg, accx, accy, accz) and an aggregated accelerometer signal
    signals = {
        'ppg': BandpassFilter(ppg, fs),
        'accx': BandpassFilter(accx, fs),
        'accy': BandpassFilter(accy, fs),
```

```
        'accz': BandpassFilter(accz, fs),
        'acc_agg': BandpassFilter(acc_agg, fs),
    }

    # From README of the data:
    # the dataset is collected using a 8-second window with 6 seconds overlapped
    # between each subsecutive window
    nfft_window = fs*8
    noverlap = fs*6

    # --- Compute a spectogram data for each signal
    spectogram = {}

    for key in signals:
        filt_specs, filt_freqs, filt_ts, _ = plt.specgram(signals[key], NFFT = nfft_wind
        plt.close()
        spectogram[key] = {
            'specs': filt_specs,
            'freqs': filt_freqs,
            'ts': filt_ts
        }

    return signals, spectogram

def get_freq_peaks(spec_data, max_freq = 5):
    """
    Compute peaks of the input spectogram data

    Args:
        spec_data: Spectogram data computed from the get_spectogram_data function

    Optional Arg:
        max_freq: Maximum frequency that will be used to find peaks. Default is 5 Hz

    Returns:
        A dictionary of the sorted peak information
    """

    filt_specs, filt_freqs = spec_data['specs'], spec_data['freqs']

    out_peaks = []

    # For a FFT signal of each time window
    for cur_win in range(filt_specs.shape[1]):

        # We only interested up to max_freq
        cur_spec = filt_specs[(filt_freqs <= max_freq), cur_win]
```

```
            # Get the indices of the FFT peaks.
            # Criteria of a peak is that it must have a height greater than 10% of the highe
            peak_indice = sp.signal.find_peaks(cur_spec, height=np.max(np.abs(cur_spec)*0.1)

            # Sort the peak values descendingly
            ii = np.argsort(cur_spec[peak_indice])[::-1]

            out_peaks.append({
                'peak_indice': peak_indice,
                'peak_freqs': filt_freqs[peak_indice[ii]]
            })

        return out_peaks
```

In [ ]: