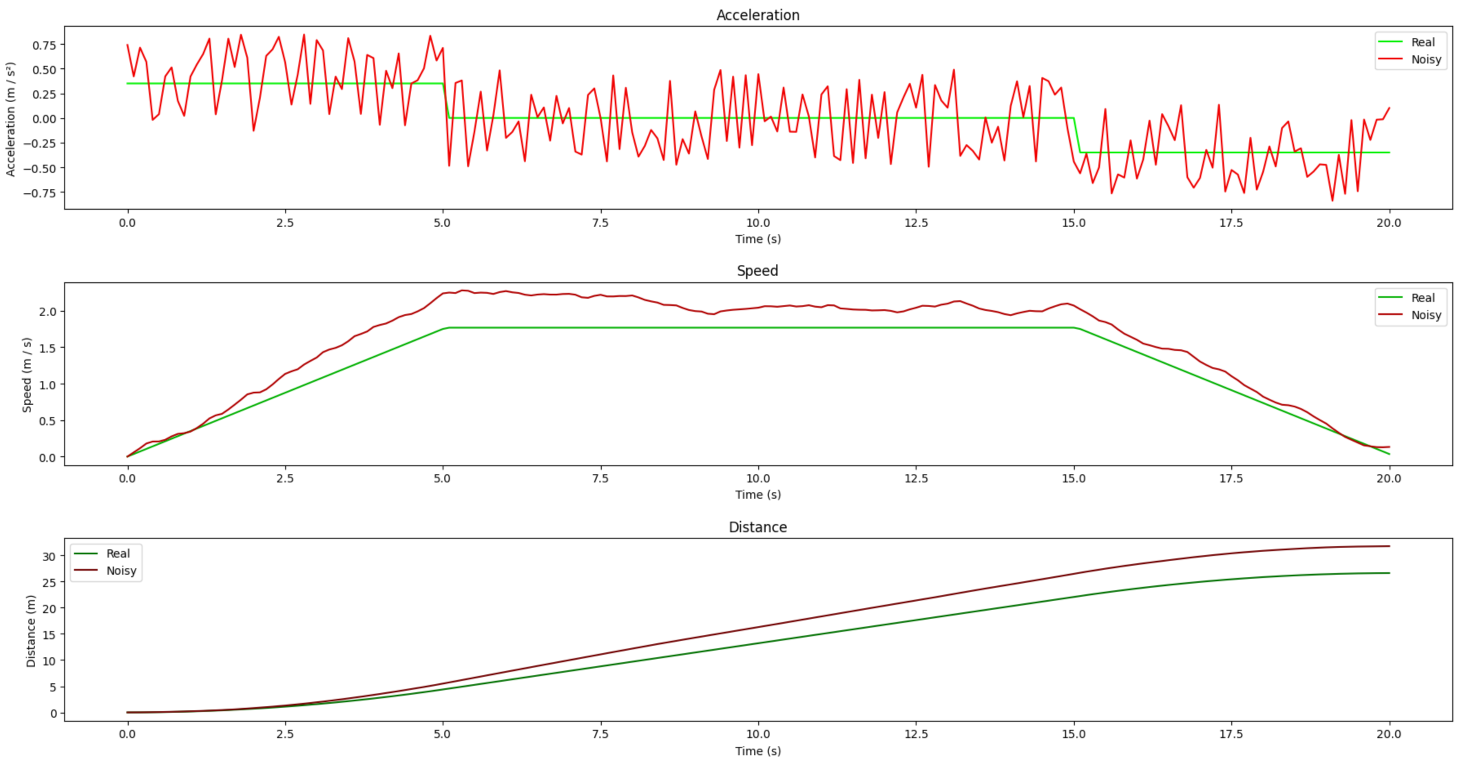
Lab 8 Report

James Vollmer, jrvollmer@wisc.edu
Nathan Abdullah, nabdullah2@wisc.edu

Milestone 1: Understanding Sensor Data Errors



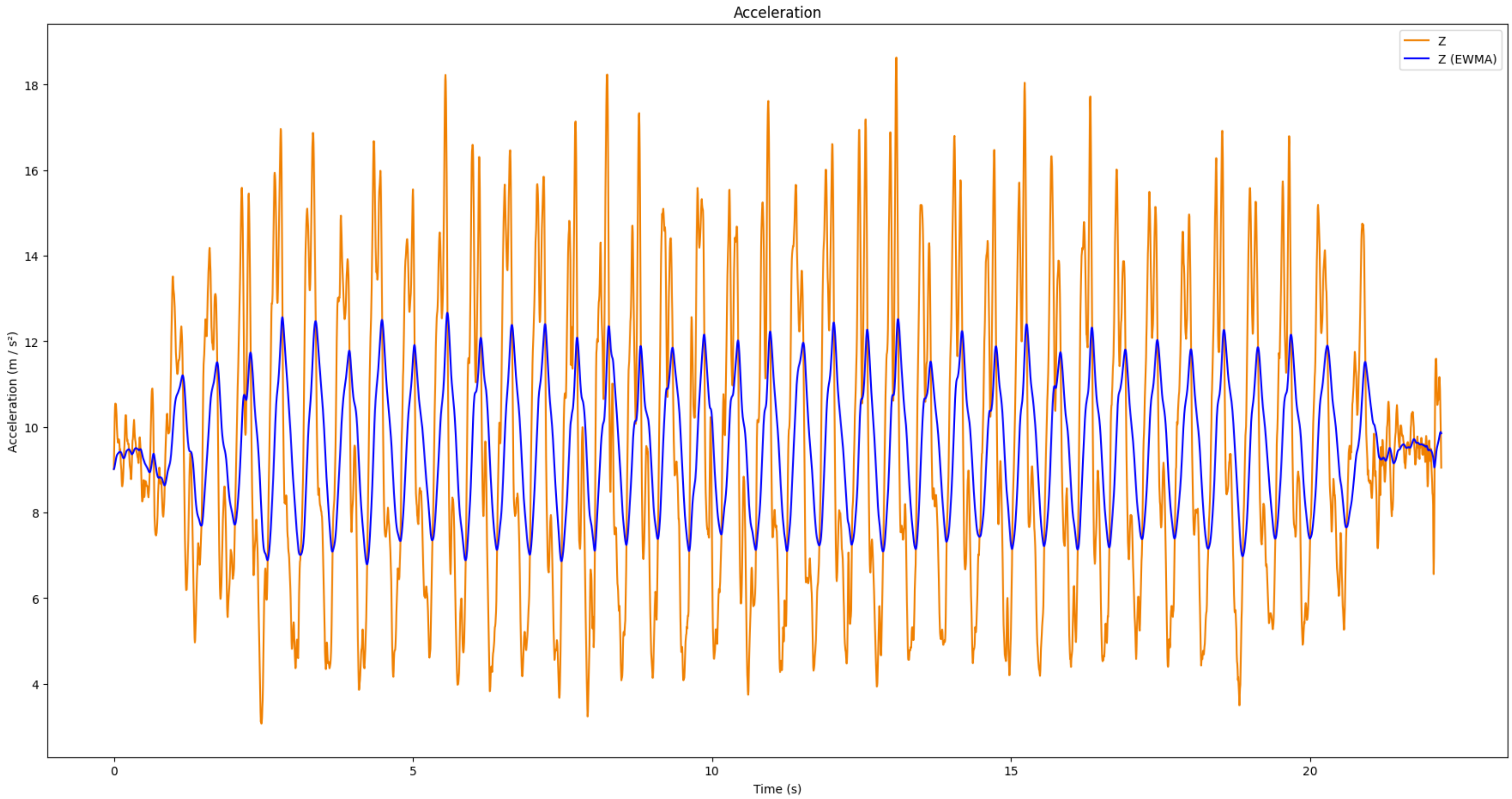
Real and noisy acceleration, speed, and distance

Final distances:

- Noisy: 31.734 m
- Real: 26.598 m
- Difference: 5.136 m

Milestone 2: Step Detection

2.1 Data Preparation



Raw (orange) and smoothed (blue) accel_z data. Note that the timestamps have been converted to seconds. The following is the smoothing algorithm used in code:

```python
def smooth_ewma(data, a):
    '''
    Smooth the provided data using an exponential weighted moving average

    Params:
        data: The raw data to be smoothed
        a: alpha value for the EWMA
    '''
    assert 0 <= a <= 1
    # Apply the smoothing algorithm element-wise to the array, where:
    #   - s is the previous element, which has already been smoothed
    #   - x is the current element to be smoothed
    return np.frompyfunc(lambda s,x: a * x + (1 - a) * s, 2, 1).accumulate(data)
```

The data was smoothed using the above exponential weighted moving average algorithm (the same as that shown in lecture) with an alpha value of 0.03. Additionally, the timestamps were converted to seconds for presentation purposes. This is shown below:

```
# Get data
# usecols is specified to handle the trailing commas in the dataset
df = pd.read_csv('datasets/WALKING.csv', usecols=['timestamp', 'accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z', 'mag_x', 'mag_y', 'mag_z'])
accel_z = df['accel_z'].values

# Smooth data
# Exponential weighted moving average with alpha=0.03
accel_z_filtered = smooth_ewma(accel_z, 0.03)

# Scale timestamps to be in seconds
df['timestamp'] -= df['timestamp'][0]
df['timestamp'] /= 10**9
```

2.2 Step Detection Algorithm

```
5   # Threshold for step detection
6   threshold = 10.5
```

```
36  # Count steps by counting intersections of the smoothed data with a threshold value
37  num_steps = 0
38  intersection_times = []
39  for i in range(len(df['timestamp'].values) - 1):
40      if accel_z_filtered[i] <= threshold and accel_z_filtered[i+1] > threshold:
41          num_steps += 1
42          intersection_times += [df['timestamp'].values[i]]
43
44  print(num_steps)
45  print(intersection_times)
```

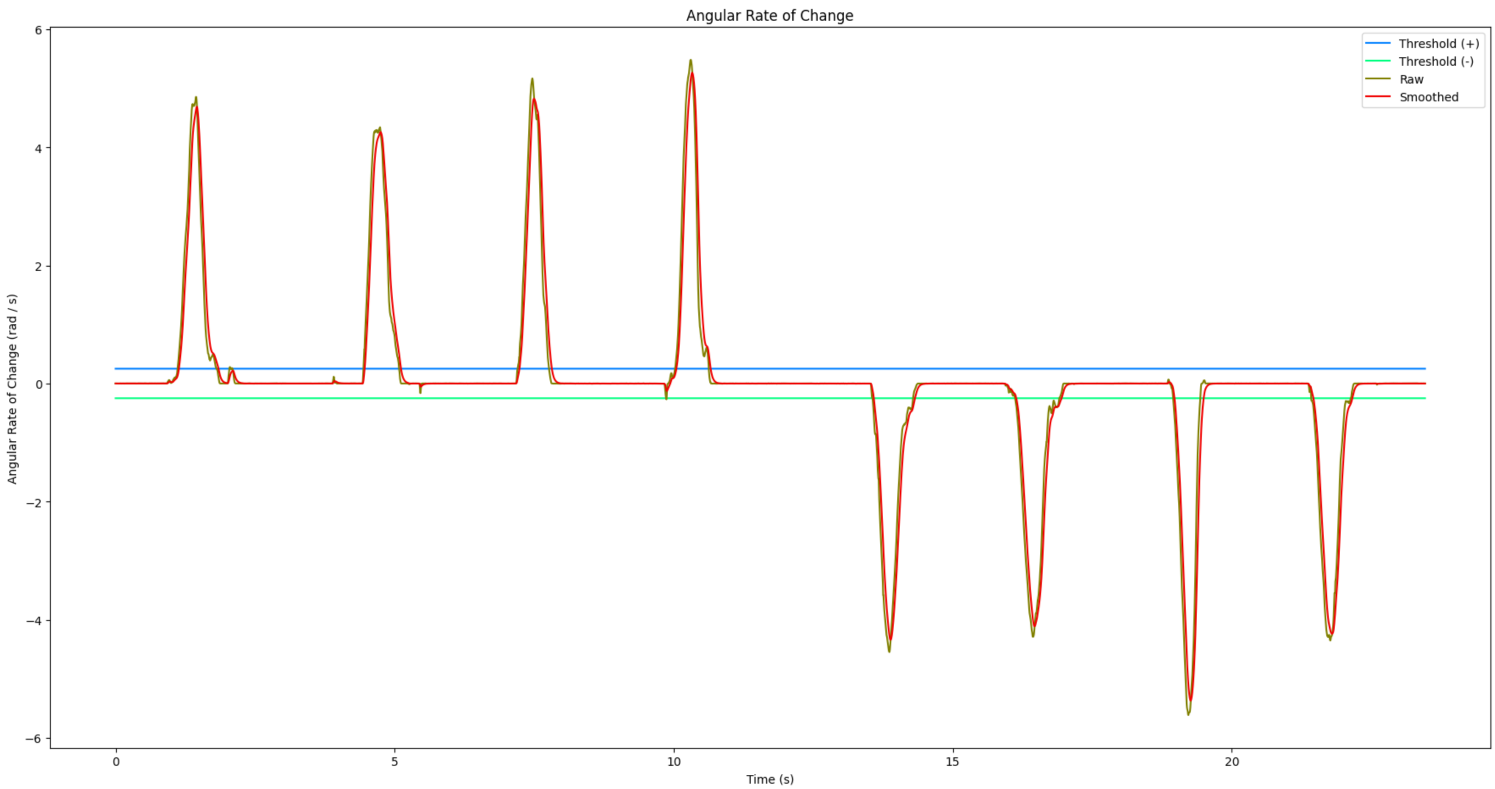
The above code snippets are the step detection algorithm used, as well as the threshold for that algorithm. To determine the number of steps and location of each step, we iterate through each index of the smoothed data, checking if a predetermined threshold that indicates a step is greater than or equal to the element at that index and is less than the element at the next index (line 6 is the threshold, and lines 37-42 contain the step detection). By doing this, we find the index of the rising edge where the threshold intersects the acceleration data, indicating the start of a peak that represents a step. A threshold of 10.5 m/s^2 was used because it was outside of the range of any leftover noise and was well below the peaks of each step.

The following screenshot shows the number of detected steps (37), as well as the approximate timestamps (in seconds) at which each step occurred:

```
james@james-p53:~/class_repos/cs407/lab8$ python3 ms2/milestone2.py
37
[1.017119005, 1.60626085, 2.155119493, 2.709013536, 3.27801378, 3.821837022, 4.365660264, 4.9346
60509, 5.478483751, 6.012236192, 6.561094835, 7.089811876, 7.678953721, 8.212706163, 8.751494005
, 9.214750841, 9.778715684, 10.302397325, 10.901635877, 11.395105116, 11.918786756, 12.4777162,
12.99636244, 13.530114881, 14.058831922, 14.67818617, 15.176690809, 15.690301649, 16.249231092,
16.777948133, 17.331842176, 17.905877821, 18.449701063, 19.038842909, 19.562524549, 20.161737196
, 20.856605576]
james@james-p53:~/class_repos/cs407/lab8$ |
```

Milestone 3: Direction Detection

3.1 Data Preparation



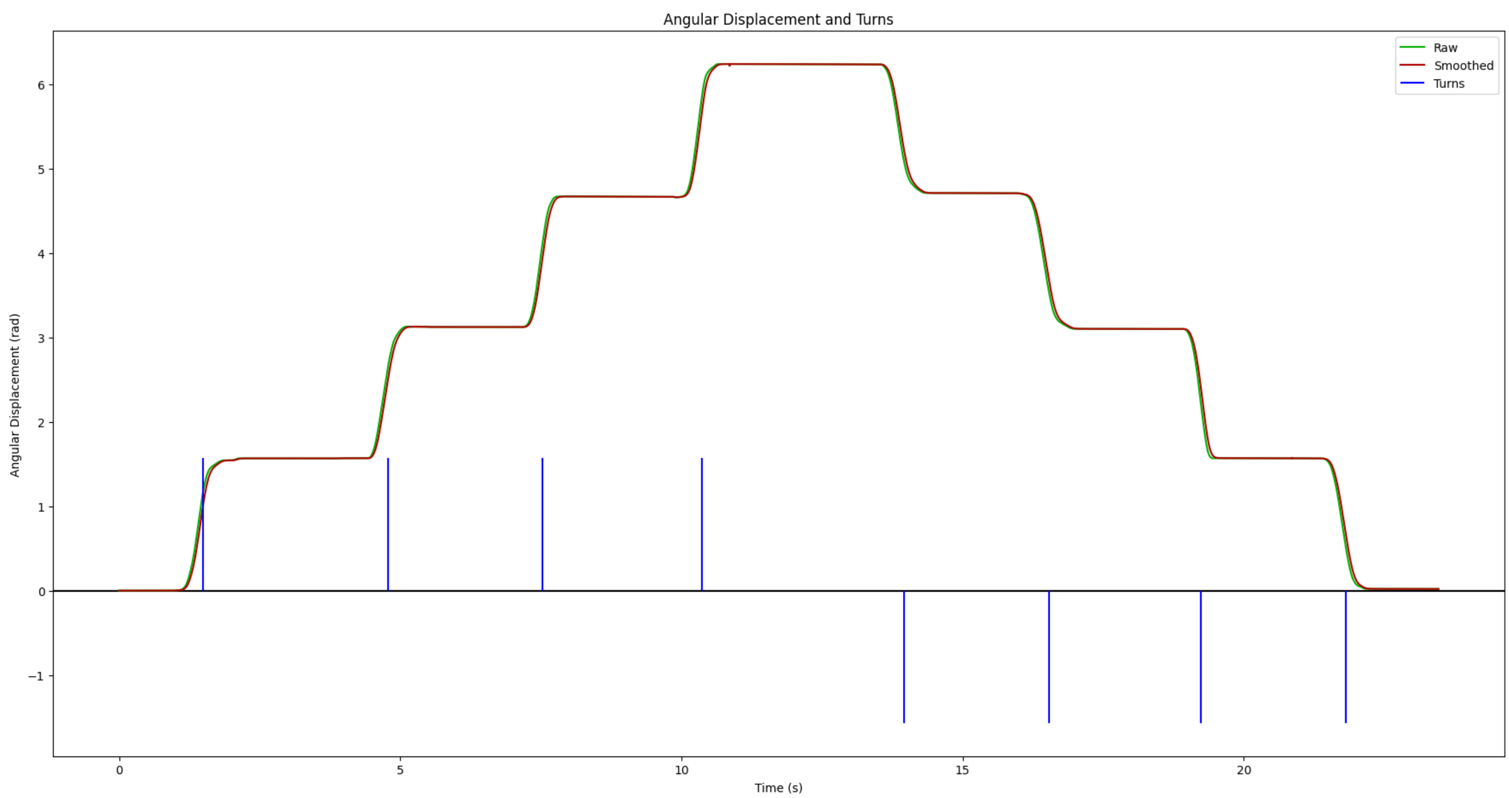
The plot above shows the raw and smoothed gyro_z data, as well as the thresholds used for the turn detection algorithm. The smoothing algorithm in section 2.1 was used here as well with an alpha value of 0.12. The following code snippet shows data acquisition and smoothing:

```
# Get data from csv
df = pd.read_csv('datasets/TURNING.csv', usecols=['timestamp', 'accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z', 'mag_x', 'mag_y', 'mag_z'])

# Change timestamps to be seconds since start
df['timestamp'] -= df['timestamp'][0]
df['timestamp'] /= 10**9
time = df['timestamp'].to_numpy()

# Smooth angular rate of change using an EWMA with specified alpha (0.12)
smooth_gyro_z = ms2.smooth_ewma(df['gyro_z'].values, 0.12)
# Integrate angular rate of change from smoothed gyro_z data to get angular displacement
theta_z = np.concatenate(([0], ms1.integrate(smooth_gyro_z, time)))
```

The data acquisition and smoothing is largely similar to that of milestone 2, but in addition to smoothing the gyro_z data, the smoothed data is then integrated (see code snippet below) to get angular displacement, theta_z, which is shown below

```python
def integrate(data, time):
    '''
    Integrate the provided dataset, data, over the time series, time

    data: dataset to integrate. Its Nth dimension must match the length of the time series
    time: time series to integrate over
    '''
    if data.shape != time.shape:
        time = np.repeat(time, data.shape[1]).reshape(data.shape)
        return (data[:-1,...] + ((data[1:,...] - data[:-1,...]) / 2)).cumsum(axis=0) * (time[1:,...] - time[:-1,...])
    else:
        return (data[:-1] + ((data[1:] - data[:-1]) / 2)).cumsum(axis=0) * (time[1:] - time[:-1])
```

Integration function



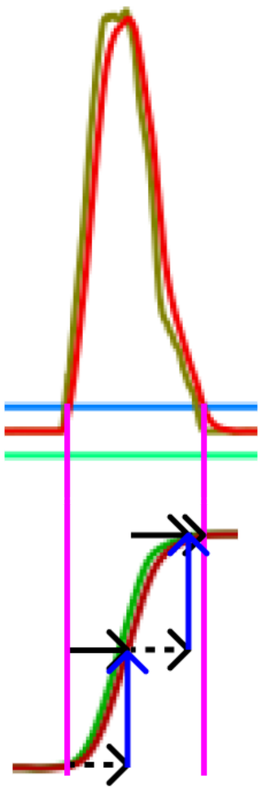
Angular displacement and detected turns (see below for algorithm)

3.2 Direction Detection Algorithm

```python
def find_turns(gyro_data, data, turn_increment, threshold=0.25, tolerance=np.pi/32):
    '''
    Returns two 2D arrays containing the midpoint index and angle of each CW and CCW turn

    data: array containing angular rate of change data
    data: array containing angular displacement data
    turn_increment: the expecte increments of which valid turns will be
        - e.g. turn_increment=pi/2 means valid turns are ..., -pi, -pi/2, 0, pi/2, pi, ...
    threshold: the minimum angular rate of change (rad/s) to be considered the start/end of a turn
    tolerance: the maximum amount by which a recorded turn can be closer to 0 than the final multiple of turn_increment
    '''
    # Limit the tolerance to at most half the turn increment
    tolerance = (turn_increment / 2) if (turn_increment <= tolerance * 2) else tolerance
    cw_bounds, ccw_bounds = get_turn_bounds(gyro_data, threshold)
    cw_turns = []
    ccw_turns = []
    idx = 0

    # Record CW turns
    for i in range(len(cw_bounds)):
        # Bounds and initial angle of the current turn
        idx = cw_bounds[i][0]
        turn_end = cw_bounds[i][1]
        init_angle = data[idx]
        increments = 0
        # Find the next index of data where the angle changes by turn_increment
        idx_tmp = np.argmax(data[idx:turn_end] <= init_angle - turn_increment + tolerance)
        while idx_tmp != 0:
            idx += idx_tmp
            increments += 1
            idx_tmp = np.argmax(data[idx:turn_end] <= init_angle - (increments+1) * turn_increment + tolerance)
        if increments > 0:
            # Record middle index and turn angle
            cw_turns.append([np.floor((cw_bounds[i][0] + turn_end) / 2).astype(int), -increments * turn_increment])
    # Record CCW turns
    for i in range(len(ccw_bounds)):
        # Bounds and initial angle of the current turn
        idx = ccw_bounds[i][0]
        turn_end = ccw_bounds[i][1]
        init_angle = data[idx]
        increments = 0
        # Find the next index of data where the angle changes by turn_increment
        idx_tmp = np.argmax(data[idx:turn_end] >= init_angle + turn_increment - tolerance)
        while idx_tmp != 0:
            idx += idx_tmp
            increments += 1
            idx_tmp = np.argmax(data[idx:turn_end] >= init_angle + (increments+1) * turn_increment - tolerance)
        if increments > 0:
            # Record middle index and turn angle
            ccw_turns.append([np.floor((ccw_bounds[i][0] + turn_end) / 2).astype(int), increments * turn_increment])
    return cw_turns, ccw_turns
```

```python
def get_turn_bounds(data, threshold=0.25):
    '''
    Returns two 2D arrays containing the start and end indices of each CW and CCW turn

    data: array containing angular rate of change data from which turn bounds will be determined
    threshold: the minimum angular rate of change (rad/s) to be considered the start/end of a turn
    '''
    cw_turn_bounds = []
    ccw_turn_bounds = []
    i = 0
    while i < len(data):
        # Step through the data until the start of a turn is reached
        if data[i] <= -threshold:
            # Record the start and (temporary) end bounds of the CW turn
            cw_turn_bounds.append([i,i])
            # Step through the data until the end of the CW turn is reached
            while (i < len(data)) and (data[i] <= -threshold):
                i += 1
            # Record the actual end index of the CW turn
            cw_turn_bounds[-1][1] = i
        if data[i] >= threshold:
            # Record the start and (temporary) end bounds of the CCW turn
            ccw_turn_bounds.append([i,i])
            # Step through the data until the end of the CCW turn is reached
            while (i < len(data)) and (data[i] >= threshold):
                i += 1
            # Record the actual end index of the CCW turn
            ccw_turn_bounds[-1][1] = i
        i += 1
    return cw_turn_bounds, ccw_turn_bounds
```

```python
    theta_z = np.concatenate(([0], msi.integrate(smooth_gyro_z, time)))
    # Detect turns of specified increments from angular rate of change and angular displacement
    cw_turns, ccw_turns = find_turns(smooth_gyro_z, theta_z, np.pi/2)
    print("CW:", cw_turns)
    print("CCW:", ccw_turns)
```



Black = search for index (dashed arrow represents current angle during search)
Blue = index found; increment angle for next search/final angle
Magenta = turn bounds detected from gyro data

The code snippets above show the algorithm used to detect turns. Additionally, the image on the left shows a visual representation of the algorithm. The find_turns() method takes in gyro data (gyro_data), angular displacement data (data), a turn increment (turn_increment), a threshold (threshold), and tolerance for error of the final angle (tolerance). First the function calls get_turn_bounds(), passing in the gyro data and threshold for angular rate of change that indicates the start and end of a turn. get_turn_bounds() determines and returns the bounds of clockwise and counterclockwise turns by finding intersection points of the threshold lines (light blue and light green lines in the top image: threshold reflected about the x-axis) with the gyro data. find_turns() then steps through these turn bounds and determines how many increments of turn_increment are found within the turn bounds in the angular displacement data. It does this by checking for an angle within the bounds that exceeds the next multiple of turn_increment, allowing for a specified tolerance. If such an angle is found, the function jumps to the index of that angle, increments the current detected angle by turn_increment, and continues to look between the new index and the end of the turn bounds for the next multiple of turn_increment. This is repeated for a given bound until either no more multiples of turn_increment are found or the end of the turn bounds are reached. Once either case is satisfied, if the detected angle is greater than 0, it is appended to the list of detected angles, along with the middle index of the turn. This is done separately for clockwise and counterclockwise turns, where clockwise turns are recorded as negative angles and counterclockwise turns are recorded as positive angles.

The main function of the milestone passes in smoothed gyro data, smoothed angular displacement, and a turn increment of pi/2 to find_turns() to detect turns that are multiples of pi/2; it uses the other default values in the function.

The results of the turn detection are shown below (numerically) and at the end of section 3.1 (graphically). The snippet below shows the detected angles (4 clockwise and 4 counterclockwise) and their corresponding indices within the dataset. This can be cross-referenced with the plot at the end of section 3.1 to see the times of each turn and how they relate to the angular displacement (and gyro) data.

```
james@james-p53:~/class_repos/cs407/lab8$ python3 -m ms3.milestone3
CW: [[2771, -1.5707963267948966], [3285, -1.5707963267948966], [3820, -1.5707963267948966], [4332, -1.5707963267948966]]
CCW: [[296, 1.5707963267948966], [951, 1.5707963267948966], [1495, 1.5707963267948966], [2057, 1.5707963267948966]]
james@james-p53:~/class_repos/cs407/lab8$ |
```

Milestone 4: Trajectory Plotting

Data Acquisition and Smoothing, Turn Detection, and Step Detection

The steps for acquiring and smoothing data, as well as detecting steps and turns, are identical to those explained in milestones 2 and 3, with the exception of the function parameters and a slight addition to the step detection algorithm, both of which are explained below.

```python
    # Get data from csv
    df = pd.read_csv('datasets/WALKING_AND_TURNING.csv', usecols=['timestamp', 'accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z', 'mag_x', 'mag_y', 'mag_z'])

    # Change timestamps to be seconds since start
    df['timestamp'] -= df['timestamp'][0]
    df['timestamp'] /= 10**9
    time = df['timestamp'].to_numpy()


    # -------------------------------------------------------------
    # Turn Detection
    # -------------------------------------------------------------

    # Smooth angular rate of change using an EWMA with specified alpha
    smooth_gyro_z = ms2.smooth_ewma(df['gyro_z'].values, 0.07)
    # Integrate angular rate of change from smoothed gyro_z to get angular displacement
    theta_z = np.concatenate(([0], ms1.integrate(smooth_gyro_z, time)))
    # Detect turns of specified increments from angular rate of change and angular displacement
    cw_turns, ccw_turns = ms3.find_turns(smooth_gyro_z, theta_z, np.pi/4, 0.125)
    cw_turns = np.array(cw_turns)
    ccw_turns = np.array(ccw_turns)
    print("CW:", cw_turns)
    print("CCW:", ccw_turns)


    # -------------------------------------------------------------
    # Step Detection
    # -------------------------------------------------------------

    threshold = 9.75

    # Smooth data
    # Exponential weighted moving average with alpha=0.02
    smooth_accel_z = ms2.smooth_ewma(df['accel_z'].values, 0.02)

    # Count steps by counting intersections of the smoothed data with a threshold value
    num_steps = 0
    intersection_indices = []
    intersection_times = []
    for i in range(len(time) - 1):
        if smooth_accel_z[i] <= threshold and smooth_accel_z[i+1] > threshold:
            num_steps += 1
            intersection_indices += [i]
            intersection_times += [time[i]]

    print(num_steps)
    print(intersection_times)
```

As can be seen in the code snippet above, the only differences for turn detection between this milestone and milestone 3 are that this one uses an alpha value of 0.07 for smoothing gyro_z data, and for turn detection, pi/4 is the turn_increment and 0.125 is the angular rate of change threshold for determining turn bounds. For step detection, 0.02 is used as the alpha value for smoothing the accel_z data, and a threshold of 9.75 m/s^2 is used for determining intersections/steps. Also, in addition to recording the times of intersection, indices within the dataset at the points of intersection are recorded for use when plotting the trajectory (see below).

Trajectory Plotting and Results

The following code snippet shows the trajectory construction:

```python

    # ------------------------------------------------------------
    # Assemble Position Data
    # ------------------------------------------------------------

    # Current heading angle (start going north)
    heading = np.pi/2
    # Location info for plotting
    x_loc = [0]
    y_loc = [0]

    # Variables for tracking position in turn arrays
    cw_idx = 0
    ccw_idx = 0
    for i in range(len(time) - 1):
        step = 0
        # Check if we're stepping forward
        if i in intersection_indices:
            step = 1
        # Adjust heading according to turns
        if len(cw_turns) > 0 and i in cw_turns[:, 0]:
            heading += cw_turns[cw_idx, 1]
            cw_idx += 1
        elif len(ccw_turns) > 0 and i in ccw_turns[:, 0]:
            heading += ccw_turns[ccw_idx, 1]
            ccw_idx += 1
        # Move according to heading and step
        x_loc += [x_loc[-1] + step * np.cos(heading)]
        y_loc += [y_loc[-1] + step * np.sin(heading)]

```

This works by stepping through every index in the dataset and checking whether or not it's contained in the turn lists or the list of intersection/step indices. If it's within a turn list, change the heading by the corresponding turn angle. If it's within the intersection/step list, move the trajectory forward by 1m in the direction of the current heading. Otherwise, stay at the current location.

The following plot shows the trajectory of the walking path. Each dot is 1m apart, and the distance between each dot represents a single step. The y-axis increases northward and the x-axis increases eastward.