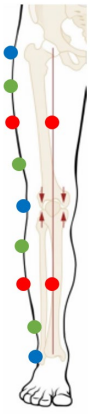
P05: Data Analyses IMU (Inertial Measurement Unit)

An Inertial Measurement Unit (IMU) is a device that typically consists of gyroscopes to measure angular rate (deg/s) and accelerometers to measure force/acceleration (including gravitational acceleration g).

IMUs can be found in many devices, including airbags, phones, watches, pacemakers, navigation devices, and robotics, and are used for various applications including activity monitoring, motion analysis, stabilization, prosthetic control, and more. In real-time controlling and navigation applications, the sensor signals are usually directly integrated into the control loop such as the stabilization of an airplane or detection of portrait/landscape mode of the phone.

For motion analyses, offline data can be used to study movement patterns. This is the topic for P5.



In the practicum of the physics course, you have measured several gait cycles with IMUs attached to the legs. The goal is to detect different gait events and phases and display the results in a manner that can be interpreted for example by a rehabilitation clinician.

Before you start coding, discuss with your group members how you structure the program. Which functions or classes do you need? What are the interfaces of these functions and methods, meaning which are the required inputs and outputs for each function? Divide the tasks and review each other's code.

Data import and cleaning

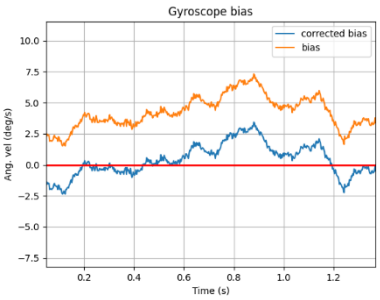
The data measured in the Delsys software is exported to a CSV file. Analyze the file, import it into Python using Pandas and clean it up.

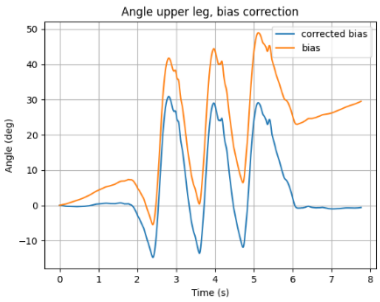
- Requirements:
  - Make a DataFrame for each sensor.
  - Use the time series as the index for each DataFrame, and remove any redundant time series columns from the DataFrame.
  - See hints below how to clean up the DF.

Data Analyses

Plot the accelerometer and gyroscope signals for the sensors on the upper leg and lower leg. What do you see? Can you detect some kind of pattern? Which signal seems most suitable to detect gait phases, such as the heel strike and toe off?

- Write a function that calculates the knee angle based on the angular velocity signals from the upper and lower leg.
    - Hint: the gyroscope output is in deg/s. To calculate an angle for each sensor, you need to integrate the signal over time.
    - Hint: the data set has 4 sensors, for each body segment you only need one signal. Use the most suitable position (lateral or anterior) and channel (X, Y, or Z).
    - We expect that the sensor measures 0 deg/s when it is not moving. However, due to a wrong sensor calibration, this might not be the case and you measure a signal with an offset (biased). Use the first few seconds of the recording in which the person is standing still to determine the bias and subtract that value from the total signal.



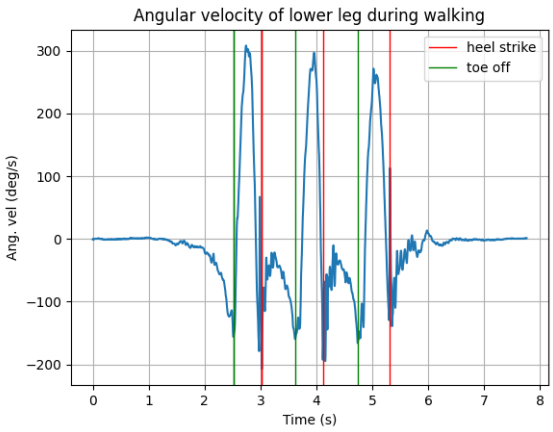


    - Note, the angle graph above does not display the full knee angle yet, you need to take the lower leg angle into account too.
    - Remark: more advanced IMUs typically have a sensor fusion algorithm implemented on the embedded microprocessor taking care of the sensor bias and drift.

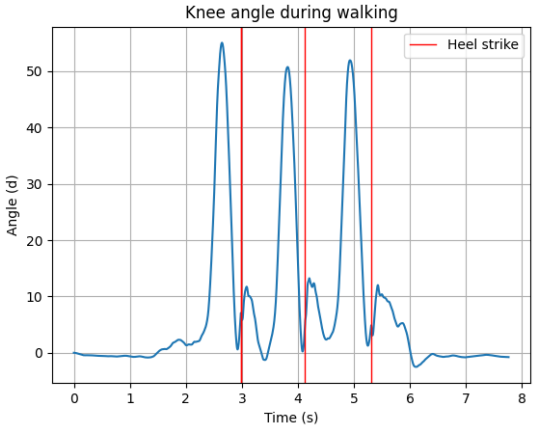
Gait Analyses

Write a function that detects the beginning of the gait cycle. This is where the knee is fully extended, and the heel touches the ground after the swing phase. A good signal to detect this is the angular velocity signal from the lower leg, but you may use another suitable signal. The graph below shows both the heel strike and the toe off events, only the heel strike event is required.

- Hint: to implement a detection algorithm you need to understand the signal. The first step seems to be taken with the leg without the sensor. The angular velocity starts with negative values, meaning the lower leg is rotating backward when you start to walk. At toe off, the leg swings forward and you see a big positive peak. Before the foot hits the ground (heel strike), the leg slows down again.
- One way to implement an event detection is to check for the signal to cross a specified threshold. To prevent multiple detected events within one gait cycle, you can check for a minimum time between two events, for example two heel strikes cannot happen within 0.5 s. It's not necessary to implement a perfect detection, but the general principle should work.



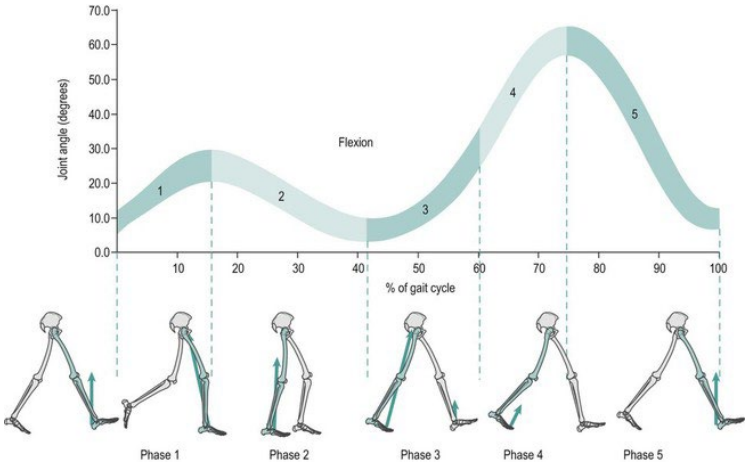
- Plot the detected gait events in the knee angle graph. This should give you a graph like this:



- During the heel strike event, you see a little bump in the data, can you explain this?

Step Analyses

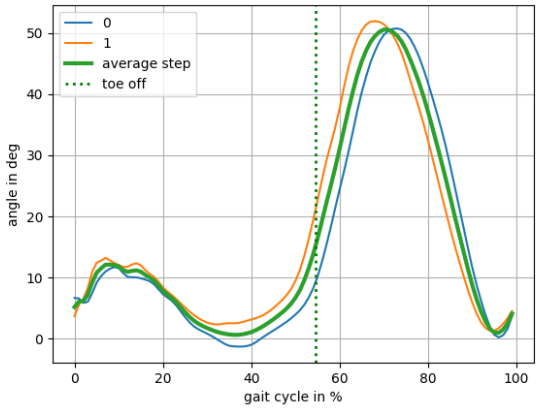
In a typical graph for clinical gait analyses, the joint angles are plotted based on a time-normalized gait cycle, from one heel strike to the next heel strike of the same leg.



- Separate/slice each step based on the detected heel strike. In case your heel strike algorithm isn't working robustly enough, you can also manually determine the time indices by zooming in the graph: e.g.,
  heelstrike = [3.011, 4.149, 5.295, 6.167] #time index in seconds
- Resample the sliced steps to 100 data points, each point representing the percentage of the gait cycle. Use linear interpolation for this, which is supported by the method: np.interp() , see hints below.
- Add the resampled steps to a DF and plot the curves. What do you observe?
- Plot the average curve on top.

```
            0          1
0    6.677111   3.689939
1    6.577193   5.556091
2    5.866025   6.514229
3    6.040155   8.297302
4    7.449101  10.926313
..        ...        ...
95   0.645131   1.318958
96   0.230111   1.742115
97   0.584639   2.446897
98   1.632954   3.372902
99   3.689939   4.518592

[100 rows x 2 columns]
```



Hint for interpolation implementation
https://en.wikipedia.org/wiki/Linear_interpolation

```
"""
data interpolation example
"""
import numpy as np

# original data
data = np.array([1, 2, 3, 2.5, 5])

# resample data to 9 data points
new_length = 9

# target indices for new number of data points
x = np.linspace(0, len(data) - 1, new_length)
# original indices
xp = np.arange(len(data))
# interpolate
data_new = np.interp(x, xp, data)

print("orignal data:", data)
print("interpolated:", data_new)
```

Hint for cleaning up sensor DF

In a DF, it's good practice to have unique column names. Therefore, duplicate names in the CSV are renamed automatically with an index number, for example "xx (s).3" as can be seen in the example below.

```
In [59]: print(data.head())
   ACC X Time Series (s)  ACC X (G)  ...  GYRO Z Time Series (s).3  GYRO Z (deg/s).3
1                 0.0000  -0.159180  ...                    0.0000          2.987805
2                 0.0027  -0.158691  ...                    0.0027          2.865854
3                 0.0054  -0.155273  ...                    0.0054          3.170732
4                 0.0081  -0.151856  ...                    0.0081          3.353659
5                 0.0108  -0.146973  ...                    0.0108          3.292683
```

Instead of having all sensor data in a single DF, split it up into 4 DFs, each containing the data for each sensor.

Find the index numbers, for each sensor and use slicing to generate separate DFs. For each signal, the same time series is also in the DF. Use df.drop(…) to delete redundant time series.

```
In [72]: upperleg_data
Out[72]:
                 ACC X (G).1  ACC Y (G).1  ...  GYRO Y (deg/s).1  GYRO Z (deg/s).1
Time Series (s)                            ...
0.0000              0.005371    -0.975586  ...          5.548780          3.292683
0.0027              0.006836    -0.978516  ...          6.524390          2.987805
0.0054              0.008789    -0.978027  ...          7.560976          3.170732
0.0081              0.007324    -0.981934  ...          8.414635          3.170732
0.0108              0.006836    -0.983398  ...          8.719512          3.353659
```

The two DF for the upper and lower leg look like this now:

```
In [79]: upperleg_data.iloc[1]
Out[79]:
ACC X (G).1         0.006836
ACC Y (G).1        -0.978516
ACC Z (G).1        -0.313477
GYRO X (deg/s).1    9.390244
GYRO Y (deg/s).1    6.524390
GYRO Z (deg/s).1    2.987805
Name: 0.0027, dtype: float64
```

```
In [81]: lowerleg_data.iloc[1]
Out[81]:
ACC X (G).3          0.126465
ACC Y (G).3         -0.989258
ACC Z (G).3         -0.236816
GYRO X (deg/s).3   -14.390244
GYRO Y (deg/s).3    -3.719512
GYRO Z (deg/s).3     2.865854
Name: 0.0027, dtype: float64
```

To get rid of the ".1", ".2" or ".3) in the column names, think how you would split the string. It is not per se required to rename the columns. If you want a specific signal, there are multiple ways.

If you know the index of the signal you are looking for, you can access it directly via df.iloc[x]. So, if you want the Z Channel Gyroscope signal, it's on index 5:

```
In [73]: upperleg_data.iloc[:,5]
Out[73]:
Time Series (s)
0.0000    3.292683
0.0027    2.987805
0.0054    3.170732
0.0081    3.170732
0.0108    3.353659
            ...
7.7490    4.817073
7.7517    5.060976
7.7544    5.121951
7.7571    4.878049
7.7598    5.060976
Name: GYRO Z (deg/s).1, Length: 2875, dtype: float64
```

Or, in case you have the column name:

```
In [75]: upperleg_data["GYRO Z (deg/s).1"]
Out[75]:
Time Series (s)
0.0000    3.292683
0.0027    2.987805
0.0054    3.170732
0.0081    3.170732
0.0108    3.353659
            ...
7.7490    4.817073
7.7517    5.060976
7.7544    5.121951
7.7571    4.878049
7.7598    5.060976
Name: GYRO Z (deg/s).1, Length: 2875, dtype: float64
```

If you delete the ".1" from the column names, you can use the same column name for each sensor.

```
In [84]: upperleg_data["GYRO Z (deg/s)"]
Out[84]:
Time Series (s)
0.0000    3.292683
0.0027    2.987805
0.0054    3.170732
0.0081    3.170732
0.0108    3.353659

In [88]: lowerleg_data["GYRO Z (deg/s)"]
Out[88]:
Time Series (s)
0.0000    2.987805
0.0027    2.865854
0.0054    3.170732
0.0081    3.353659
0.0108    3.292683
```