JulyImuOdom

July 21, 2021

Importing the required libraries

[21]:
```
import bagpy
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import signal
from scipy.spatial.transform import Rotation as R
# from tf.transformations import unit_vector, vector_norm,␣
↪quaternion_conjugate, quaternion_multiply, euler_matrix, quaternion_matrix

# Standard plotly imports
import plotly.graph_objects as go
from plotly.offline import iplot, init_notebook_mode
import plotly.express as px
import plotly.io as pio
pio.templates.default = 'plotly_dark'
```

Importing the data

[22]:
```
os = 'ubuntu'
# os = 'windows'

bag_file_name = '2021-06-30-14-48-17.bag'
```

[23]:
```
if os == 'ubuntu':
    filepath = '/home/user/rosbags/' + bag_file_name
if os == 'windows':
    filepath = r"C:\Users\FSaal\Google Drive\Uni\Masterarbeit\{}".
↪format(bag_file_name)

# Load rosbag
bag = bagpy.bagreader(filepath)
# bag.topic_table

# Extract recorded topics/messages
imu = bag.message_by_topic('/imu/data_raw')
tab_imu = pd.read_csv(imu)
```

```
ang_vel = np.asarray(tab_imu.iloc[:,18:21])
lin_acc = np.asarray(tab_imu.iloc[:,30:33])

odom = bag.message_by_topic('/eGolf/sensors/odometry')
tab_odom = pd.read_csv(odom)
odom = np.asarray(tab_odom.iloc[:,-5:])

# Frequency defintion
f_imu = 100 # Hz
f_odom = 100 # Hz

t = np.linspace(0, lin_acc.shape[0]/f_imu, lin_acc.shape[0])
```

```
[INFO]  Data folder /home/user/rosbags/2021-06-30-14-48-17 already exists. Not
creating.
```

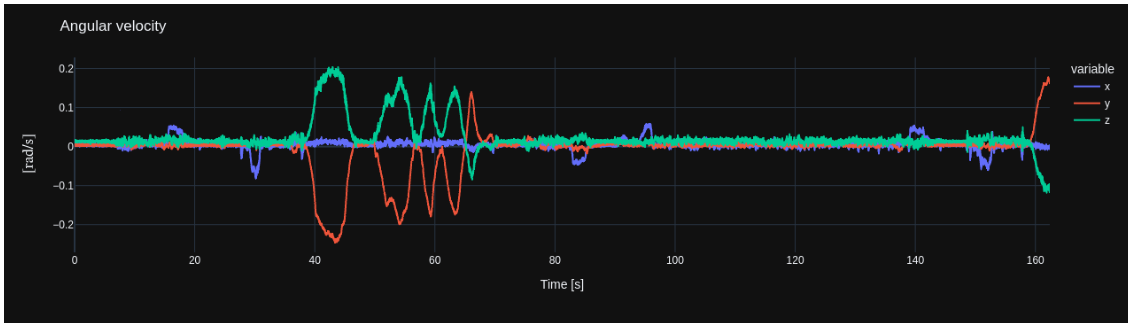
Helper functions

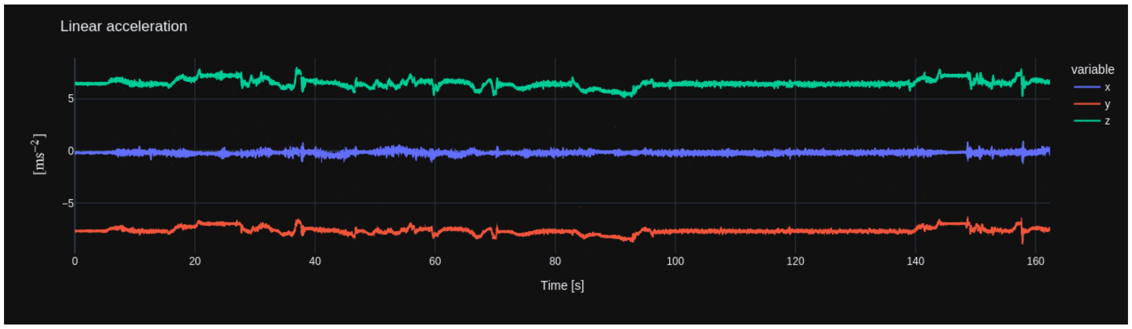
[24]:
```
def plotAngle(title_ext, car_angle, *additional_data):
    if not additional_data:
        fig = px.line(pd.DataFrame(data=car_angle, index=t))
        fig.update_layout(yaxis_range=[-10,10], xaxis_title='Time [s]',␣
 ↪yaxis_title='$[deg]$', title='Car pitch angle using {}'.format(title_ext),␣
 ↪showlegend=False)
    else:
        signal = additional_data[0]
        custom_legend = additional_data[1]
        fig = px.line(pd.DataFrame(data=np.vstack([car_angle, signal]).T,␣
 ↪index=t, columns=custom_legend))
        fig.update_layout(yaxis_range=[-10,10], xaxis_title='Time [s]',␣
 ↪yaxis_title='$[deg]$', title='Car pitch angle using {}'.format(title_ext))
        if len(additional_data) == 3:
            hide_data = additional_data[2]
            fig.for_each_trace(lambda trace: trace.update(visible='legendonly')
                  if trace.name in hide_data else ())
    fig.show()

# Functions to replace the functions from tf.transformations (**only execute if␣
 ↪on Windows**)
def vector_norm(v):
    return np.linalg.norm(v)

def unit_vector(v):
    mag = vector_norm(v)
    return v / mag

def quaternion_matrix(q):
    return R.from_quat(q).as_matrix()

def quaternion_multiply(q0, q1):
    x0, y0, z0, w0 = q0
    x1, y1, z1, w1 = q1
    return np.array([-x1 * x0 - y1 * y0 - z1 * z0 + w1 * w0,
                     x1 * w0 + y1 * z0 - z1 * y0 + w1 * x0,
                     -x1 * z0 + y1 * w0 + z1 * x0 + w1 * y0,
                     x1 * y0 - y1 * x0 + z1 * w0 + w1 * z0], dtype=np.float64)

def quaternion_conjugate(q):
    return np.hstack([-q[:-1], q[-1]])

def euler_matrix(x, y, z, nvm):
    if x != 0:
        return R.from_euler('x', x).as_matrix()
    if y != 0:
        return R.from_euler('y', y).as_matrix()
    else:
        return R.from_euler('z', z).as_matrix()
```

0.0.1 Plot measurements

Beide rampen mehrmals hoch und runter (auch rückwärts)

[25]:
```
fig = px.line(pd.DataFrame(data=ang_vel, columns=['x','y','z'], index=t))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[rad/s]}$',␣
 ↪title='Angular velocity')
fig.show()

fig = px.line(pd.DataFrame(data=lin_acc, columns=['x','y','z'], index=t))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',␣
 ↪title='Linear acceleration')
fig.show()
```





1 Align device (IMU) frame with car frame

1.0.1 Orientation of IMU and car:

- IMU frame is shown in picture
- Car frame has x-axis forward, y-axis to the left and z-axis up
- The ZED Cam and IMU are mounted on the car as seen in the picture (but usually the camera is slightly tilted up), with the lenses facing in forward direction

1.0.2 Steps to get rotation:

Step 1) Get correct z-axis

- Find rotation matrix, such that measured **g vector aligns with z-axis** (of car) $\rightarrow$ results in correct inclination and roll
- This is done by using the linear acceleration measurement during stand still (record 1s and take average)
- Calculate vector norm and unit vector, because the car stands still it can be assumed that the only measured accleration is the gravity acceleration and thus the vector `g_imu = unit_vector(lin_acc_imu)` is the gravity vector
- Rotate the gravity vector on to the desired vector `g_car = [0,0,1]` by calculating the needed rotation axis

  $$ax = \frac{g_{\text{imu}} \cdot g_{\text{car}}}{\|g_{\text{imu}} \times g_{\text{car}}\|}$$

  and the needed rotation angle

  $$\alpha = \arctan\left(\frac{\|g_{\text{imu}} \times g_{\text{car}}\|}{g_{\text{imu}} \cdot g_{\text{car}}}\right)$$

  which then leads to the following quaternion

  $$q = [ax \cdot \sin(\frac{\alpha}{2}), \quad \cos(\frac{\alpha}{2})]$$

- atm the quaternion will be converted to a rotation matrix `rotmat1`
- Apply rotation to IMU measurements

Step 2) Get correct x-axis and y-axis

- Find rotation matrix to **correct the heading**
- In the previous step the z-axis was already aligned, to get the correct x-axis and y-axis a rotation around the z-axis is neccessary
- This is done by accelerating the car in forward direction and recording the linear acceleration measurement during the process
- Because the z-axis is already aligned, the forward acceleration should only be measured by the x- and y-axis
- Now rotate around the z-axis with different rotation angles and search for the angle, which maximizes the linear acceleration values of the x-axis (or minimizes the y-axis values)
- Rotation axis ([0, 0, 1]) and rotation angle is now known and the second rotation matrix `rotmat2` can be calculated

Step 3) Get final rotation matrix

- Find rotation to transform from the IMU frame to the car frame
- Rotation matrix concatenation is always in reverse order and hence the final rotation matrix can be calculated using `mat_imu_car = rotmat2 * rotmat1`

1.1 Step 1) Align IMU g vector with z-axis

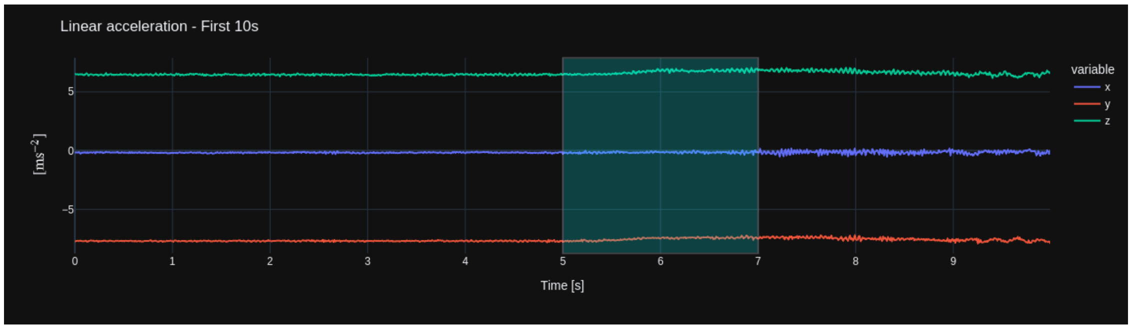
[26]:
```
# Ideal normed g measurement in car frame
g_car = [0, 0, 1]

def trafo1(lin_acc):
    """Rotation to align IMU measured g-vector with car z-axis
    :param lin_acc: Linear acceleration while car stands still
    :return:        Quaternion
    """
    g_mag = vector_norm(np.mean(lin_acc, axis=0))
    print('Average linear acceleration magnitude: {}  (should ideally be 9.81)'.
 ↪format(round(g_mag, 2)))
    g_imu = unit_vector(np.mean(lin_acc, axis=0))
    quat = quat_from_vectors(g_imu, g_car)
    return quat

def quat_from_vectors(vec1, vec2):
    """Quaternion that aligns vec1 to vec2
    :param vec1: A 3d "source" vector
    :param vec2: A 3d "destination" vector
    :return quat: A quaternion [x, y, z, w] which when applied to vec1, aligns␣
 ↪it with vec2
    """
    a, b = unit_vector(vec1), unit_vector(vec2)
    c = np.cross(a, b)
    d = np.dot(a, b)
    # Rotation axis
    ax = c / vector_norm(c)
    # Rotation angle
    a = np.arctan2(vector_norm(c), d)

    quat = np.append(ax*np.sin(a/2), np.cos(a/2))
    return quat

# Use first second of linear acceleration data (car stands still)
quat1 = trafo1(lin_acc[:100])
# Convert quaternion to rotation matrix
rot_mat1 = quaternion_matrix(quat1)[:3,:3]
# Apply rotation
acc_rot1 = np.inner(rot_mat1, lin_acc).T
```

```
Average linear acceleration magnitude: 10.05  (should ideally be 9.81)
```

1.2 Step 2) Get correct x-axis and y-axis

By looking at the data (linear acceleration after first rotation) the start of acceleration was determined to be at around 5-7s

[27]:
```
fig = px.line(pd.DataFrame(data=lin_acc[:1000], columns=['x','y','z'], index=t[:
↪1000]))
fig.add_vrect(x0=5, x1=7, fillcolor='aqua', opacity=0.2)
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',␣
↪title='Linear acceleration - First 10s')

fig.show()
```

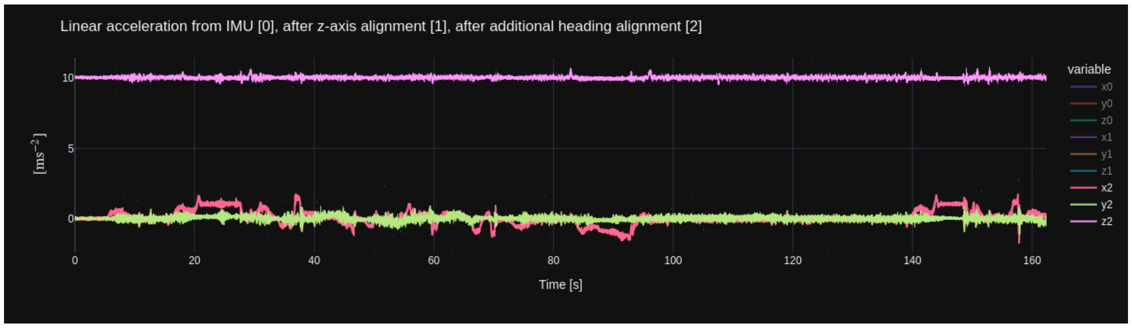


[28]:
```
def find_z_angle(lin_acc_rot):
    """Yaw angle to align IMU x-axis with x-axis of car
    :param lin_acc:     Linear acceleration while car accelerates after first␣
↪rotation (z-axes aligned)
```

```
    :return ang_opt:    Rotation angle around z-axis in rad
    """
    x_mean_old = 0
    # Precision of returned rotation angle in deg
    ang_precision = 0.1
    # Max expected yaw angle deviation of IMU from facing straight forward␣
 ↪(expected to be -90)
    ang_error = np.deg2rad(180)

    # Find rotation angle around z-axis which maximizes the measurements of␣
 ↪x-axis
    for i in np.arange(-ang_error, ang_error, np.deg2rad(ang_precision)):
        # Apply rotation of i rad around z-axis
        rot_mat = euler_matrix(0, 0, i, 'sxyz')[:3,:3]
        z_rot = np.inner(rot_mat, lin_acc_rot).T
        x_mean = np.mean(z_rot[:,0])
        if x_mean > x_mean_old:
            x_mean_old = x_mean
            ang_opt = i

    print('Yaw angle was corrected by {} degree'.format(np.degrees(ang_opt)))
    return ang_opt

def trafo2(lin_acc, rot_mat1):
    """Second rotation to align IMU with car frame
    :param lin_acc:             Linear acceleration while car accelerates
    :param rot_mat1:            Rotation matrix from the first transform step
    :return rot_mat_imu_car:    Rotation matrix to transform IMU frame to car␣
 ↪frame
    """
    # Apply first rotation (trafo1)
    lin_acc_rot1 = np.inner(rot_mat1, lin_acc).T
    # Get second rotation
    z_angle = find_z_angle(lin_acc_rot1)
    # Second rotation matrix for yaw correction
    rot_mat2 = euler_matrix(0, 0, z_angle, 'sxyz')[:3,:3]

    # Concatenation of rotation matrices is in reverse order
    rot_mat_imu_car = np.matmul(rot_mat2, rot_mat1)
    return rot_mat_imu_car

# Use 5s-7s of linear acceleration data (car accelerates)
tf_imu_car = trafo2(lin_acc[500:700], rot_mat1)
# Apply rotation
acc_rot2 = np.inner(tf_imu_car, lin_acc).T
```

```
Yaw angle was corrected by -86.30000000000582 degree
```

[29]:
```
fig = px.line(pd.DataFrame(data=np.hstack([lin_acc, acc_rot1, acc_rot2]),␣
↪columns=['x0','y0','z0','x1','y1','z1','x2','y2','z2'], index=t))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',
                  title='Linear acceleration from IMU [0], after z-axis␣
↪alignment [1], after additional heading alignment [2]')
fig.for_each_trace(lambda trace: trace.update(visible='legendonly')
                  if trace.name in ['x0','y0','z0','x1','y1','z1'] else ())
fig.show()
```



2 Pitch angle car calculation

The first idea to calculate the pitch angle would be using the so called acceleration method. Using the forward (x-axis after tf to car frame) linear acceleration measurements of the IMU and the odometry measurements the angle can be calculated using

$$\alpha = \arcsin\left(\frac{a_x - a_{at}}{g}\right) = \arcsin\left(\frac{a_x - \frac{dv}{dt}}{g}\right) = \arctan 2(a_x, a_z)$$

with:

- a_x: IMU acceleration measurements of forward-axis in car frame (x-axis)
- $\frac{dv}{dt}$: Longitudinal acceleration (vehicle acceleration from odometry)
- g: Earths gravitational acceleration (magnitude)

Using the Einspurmodell the car velocity can be calculated in the following way

$$\alpha(t) = \frac{v_{\text{rear,left}} - v_{\text{rear,right}}}{d} \tag{1}$$

$$yaw(t) = \frac{\alpha}{f_{\text{odom}}} \tag{2}$$

$$v_{\text{car}}(t) = \frac{v_{\text{rear,left}} + v_{\text{rear,right}}}{2} \cdot \cos(yaw) \tag{3}$$

with: - $v_{\mathrm{rear,[left,right]}}$: Wheel speed of the left/right rear wheel [km/h] - d: Wheelbase of the car (2.631 m for the eGolf) - α: ? - yaw: Yaw angle of the car - v_{car}: Longitudinal car velocity [m/s]

The car acceleration can then be calculated using numerical differentiation, e.g. when using forward difference

$$a_{\mathrm{car}}(t) = \frac{v_{\mathrm{car}}(x+h) - v_{\mathrm{car}}(x)}{h}$$

EDIT: Either change all time continious functions to time discrete or do not use numerical differentation

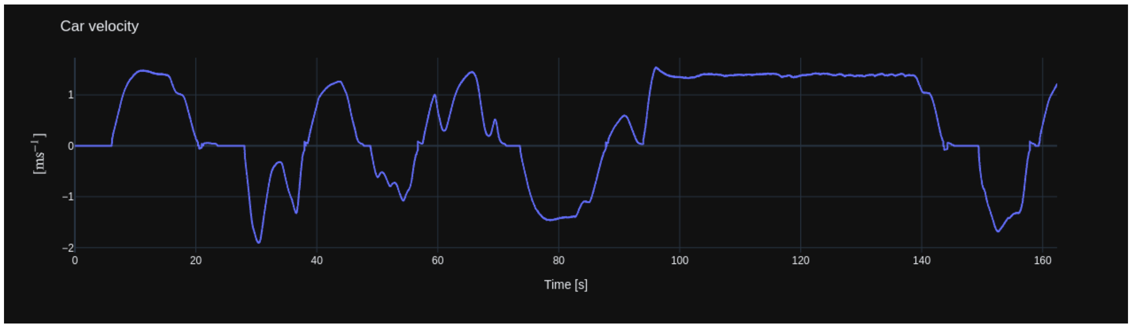
2.0.1 Checking how noisy the data is

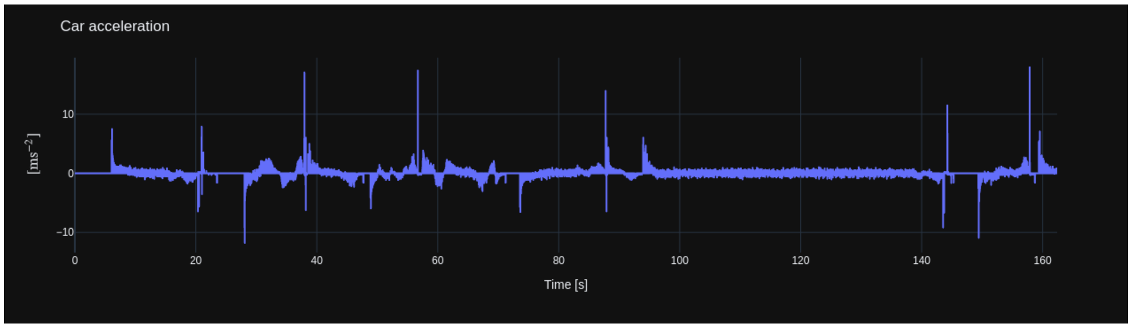
Because the car velocity is time discrete a derivation of the signal can lead to large jumps. This can be seen in the car acceleration plot, without filtering the car velocity the derived signal is not really usable.

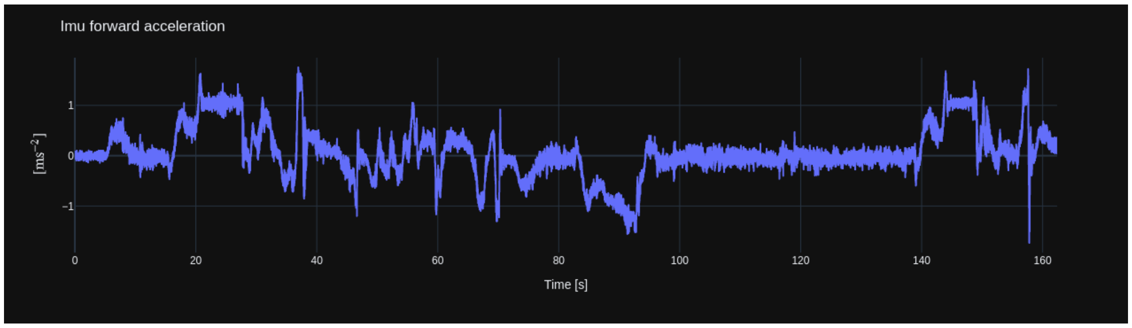
The IMU measurement is also quite a noisy.

[30]:
```python
def car_vel_acc_from_odom(odom):
    wheelbase = 2.631 # m
    alpha = (odom[:,2] - odom[:,3]) / wheelbase
    yaw = alpha * 1.0 / f_odom
    # 3.6 to convert from km/h to m/s
    vel_x_car = ((odom[:,2] + odom[:,3]) / 2) * np.cos(yaw) / 3.6
    acc_x_car = acc_from_vel(vel_x_car)
    return vel_x_car, acc_x_car

def acc_from_vel(vel):
    acc = []
    for i in range(len(vel)-1):
        a = (vel[i+1] - vel[i]) / (1.0/f_odom)
        acc.append(a)
    acc.append(a)
    # Add points if odometry topic recorded less msgs than imu
    msgs_diff = len(lin_acc) - len(odom)
    while msgs_diff > 0:
        acc.append(a)
        msgs_diff -= 1
    return acc

vel_x_car, acc_x_car = car_vel_acc_from_odom(odom)
acc_x_imu = acc_rot2[:,0]

fig = px.line(pd.DataFrame(data=vel_x_car, index=t[:-1]))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-1}]}$',␣
 ↪title='Car velocity', showlegend=False)
fig.show()
fig = px.line(pd.DataFrame(data=acc_x_car, index=t))
```

```
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',␣
↪title='Car acceleration', showlegend=False)
fig.show()
fig = px.line(pd.DataFrame(data=acc_x_imu, index=t))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',␣
↪title='Imu forward acceleration', showlegend=False)
fig.show()
```





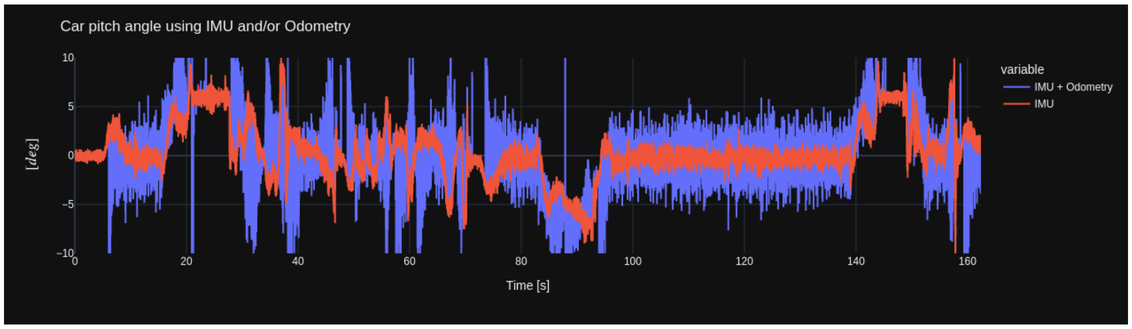


2.0.2 Calculate car pitch angle using the unfiltered data

Demonstration of the results without the use of any filtering. Using only the IMU measurements (no $-a_{at}$ part) the course is somewhat recognizable (e.g. still stand on ramp at 20-30s), but every acceleration on a flat surface also leads to a change of the angle (e.g. at 7s).
The result when also using the odometry data is not useable at all though, because the car acceleration from the derived car velocity is very volatile.

[31]:
```
def pitch_car(acc_x):
    deg = np.degrees(np.arcsin(acc_x/g_mag))
    return deg

def pitch_car_odom(acc_x_imu, acc_x_car = None):
    if acc_x_car == None:
        _, acc_x_car = car_vel_acc_from_odom(odom)
    deg = np.degrees(np.arcsin((acc_x_imu - acc_x_car) / g_mag))
    return deg

g_mag = vector_norm(np.mean(lin_acc[:100], axis=0))
car_angle = pitch_car(acc_x_imu)
car_angle_odom = pitch_car_odom(acc_x_imu)

plotAngle('IMU and/or Odometry', car_angle_odom, car_angle, ['IMU + Odometry',␣
 ↪'IMU'])
```

```
/home/user/.local/lib/python3.6/site-packages/ipykernel_launcher.py:8:
RuntimeWarning:

invalid value encountered in arcsin
```



2.0.3 Filtering the data

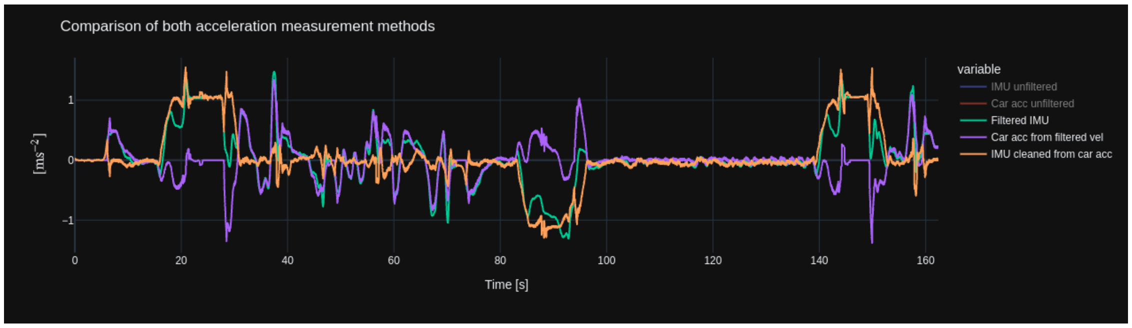
Note: When driving on a flat surface the measurements from IMU and odometry should be about the same, but when driving on a ramp they can and should be different

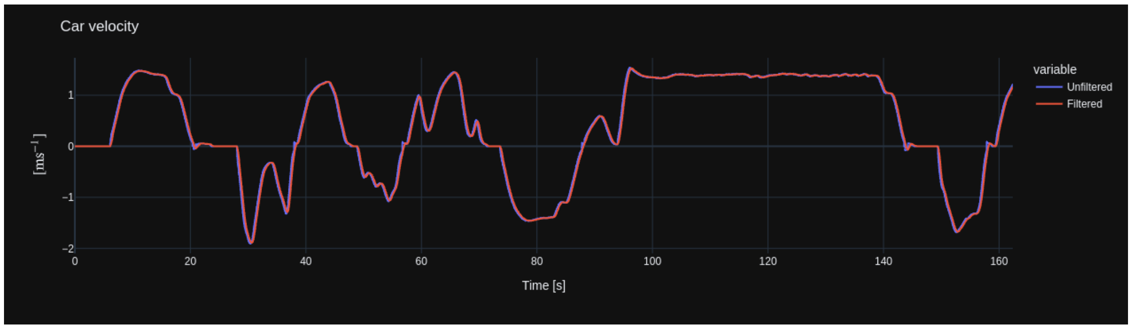
In the first plot it can be seen, that measurements from IMU and odometry almost match each other as long the car is not driving onto a ramp. Because the angle is depended on the difference between those two measurements, this leads to a better car angle, which can be seen in plot 3 (orange vs purple line). The angle now significantly changes when driving onto a ramp, but sometimes the angle still deviates up to 3 degrees on a flat surface (e.g. at 70s).

For the filtering a moving average filter with a window length of 0.5s was used, because it easy to implement and real time ready. But it leads to a slight delay (half of the window length) and also does not perform as well as e.g. a butterworth filter. Further investigation is neccessary to get a better real time ready filtering method.

[35]:
```python
def moving_average(signal, window_size):
    """Moving average filter, acts as lowpass filter
    :param val:          Measured value (scalar)
    :param window_size: Window size, how many past values should be considered
    :return filtered:   Filtered signal
    """
    sum = 0
    values = []
    filtered = []
    for i,v in enumerate(signal):
        values.append(v)
        sum += v
        if len(values) > window_size:
            sum -= values.pop(0)
        filtered.append(float(sum) / len(values))
    return np.array(filtered)

acc_x_imu_filt = moving_average(acc_x_imu, 50)
vel_x_car_filt = moving_average(vel_x_car, 50)
acc_x_car_filt = acc_from_vel(vel_x_car_filt)

# Scipy Butterworth filter (also online possible? (filtfilt is definitely only␣
↪offline))
fc = 1 # Cutoff frequency in Hz
w = fc / (f_imu/2) # Normalize fc (Nyquist)
# num, denom of IIR 4-th order butterworth filter with cutoff of 1 Hz
b, a = signal.butter(4, w, 'low')
acc_x_imu_filtButter = signal.filtfilt(b,a, acc_x_imu)
vel_x_car_filtButter = signal.filtfilt(b,a, vel_x_car)
acc_x_car_filtButter = acc_from_vel(vel_x_car_filtButter)

fig = px.line(pd.DataFrame(data=np.vstack([acc_x_imu, acc_x_car,␣
↪acc_x_imu_filt, acc_x_car_filt, acc_x_imu_filt-acc_x_car_filt]).T, index=t,
                           columns=['IMU unfiltered','Car acc unfiltered',␣
↪'Filtered IMU', 'Car acc from filtered vel', 'IMU cleaned from car acc']))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-2}]}$',␣
↪title='Comparison of both acceleration measurement methods')
```

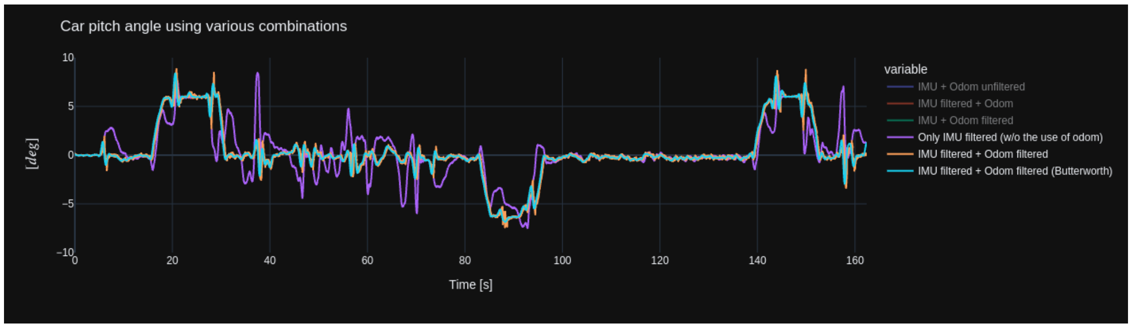
```
fig.for_each_trace(lambda trace: trace.update(visible='legendonly')
                   if trace.name in ['IMU unfiltered','Car acc unfiltered'] else␣
↪())
fig.show()
fig = px.line(pd.DataFrame(data=np.vstack([vel_x_car, vel_x_car_filt]).T,␣
↪index=t[:-1], columns=['Unfiltered', 'Filtered']))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='$\mathrm{[ms^{-1}]}$',␣
↪title='Car velocity')
fig.show()

plotAngle('various combinations', pitch_car_odom(acc_x_imu),
          [pitch_car_odom(acc_x_imu_filt), pitch_car_odom(acc_x_imu,␣
↪acc_x_car_filt), pitch_car(acc_x_imu_filt), pitch_car_odom(acc_x_imu_filt,␣
↪acc_x_car_filt),
          pitch_car_odom(acc_x_imu_filtButter, acc_x_car_filtButter)],
          ['IMU + Odom unfiltered', 'IMU filtered + Odom', 'IMU + Odom␣
↪filtered', 'Only IMU filtered (w/o the use of odom)', 'IMU filtered + Odom␣
↪filtered', 'IMU filtered + Odom filtered (Butterworth)'],
         ['IMU + Odom unfiltered', 'IMU filtered + Odom', 'IMU + Odom␣
↪filtered'])
```





```
/home/user/.local/lib/python3.6/site-packages/ipykernel_launcher.py:8:
RuntimeWarning:
```

```
invalid value encountered in arcsin
```



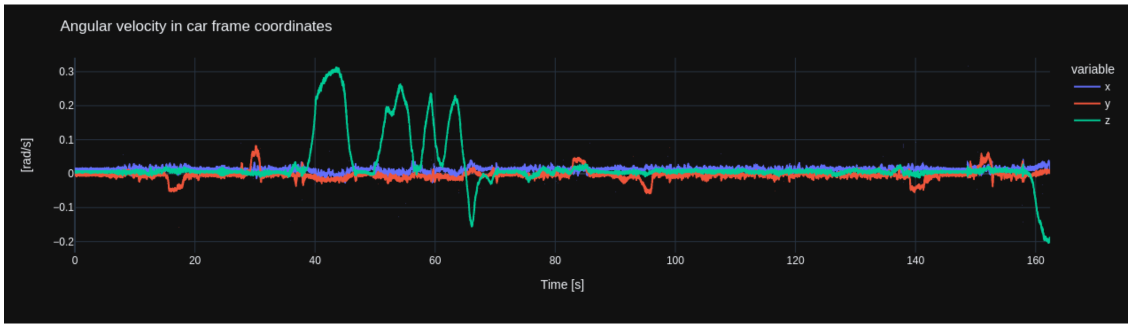
2.0.4 Using angular velocity measurements to further improve results

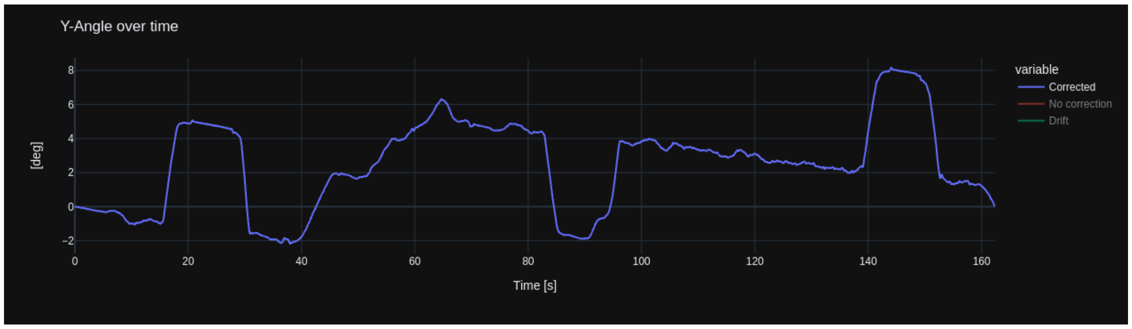
After the transformation from imu to car frame a rotation around the y-axis results in the inclination of the car. Therefore the measured angular velocity of the y-axis is important.
A negative value means the car drives a ramp up (or accelerates fast forward) and a positive value means that the car drives a ramp down (or breaks hard). To make the comparison with the previous car angle calculation easier, the sign of the angle from gyro measurement has been inverted. The z-axis measurement indicate a change of the yaw angle. It should change by 90 deg when driving around a corner. The x-axis measurement is the roll angle and should be near zero all the time.

The gryoscope measures the angular velocity. By integrating the velocity, the distance (in this case the angle in rad) over time can be calculated. But the gyroscope measurements has white noise, which leads to a drift when integrating, which has to be taken into account.
For offline data this can be done, by standing still at the start and the end of the recording and then calculating a straight line from start to finish and substract that line from the angle.

[33]:
```
vel_rot = np.inner(tf_imu_car, ang_vel).T
vel_y = vel_rot[:, 1]

fig = px.line(pd.DataFrame(data=vel_rot, index=t, columns=['x','y','z']))
fig.update_layout(xaxis_title='Time [s]', yaxis_title='[rad/s]', title='Angular␣
↪velocity in car frame coordinates')
fig.show()

ang_y = -np.degrees(np.cumsum(vel_y) / f_imu)
drift = np.linspace(ang_y[0], ang_y[-1], len(ang_y))
ang_y_no_drift = ang_y - drift

fig = px.line(pd.DataFrame(data=np.vstack([ang_y_no_drift, ang_y, drift]).T,␣
↪index=t, columns=['Corrected','No correction','Drift']))
```

```
fig.update_layout(xaxis_title='Time [s]', yaxis_title='[deg]', title='Y-Angle␣
↪over time')
fig.for_each_trace(lambda trace: trace.update(visible='legendonly')
                   if trace.name in ['No correction','Drift'] else ())
fig.show()
```





It can be seen, that the angle drifts about 16 deg per minute without any correction. But even after substracting the estimated drift, the angle still seems to drift slightly.

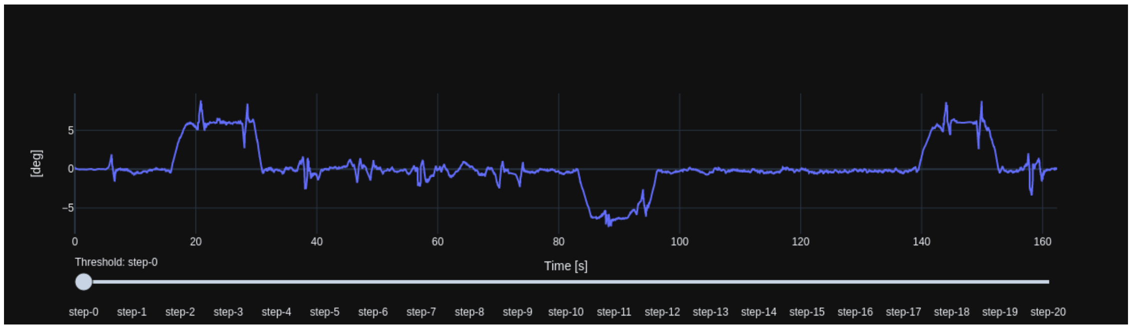
[34]:
```
# Integrate angular velocity and convert from rad to deg
def gyr_ang(ang_vel):
    return -np.degrees(np.cumsum(ang_vel)/f_imu)

def gyr_no_drift(ang_vel, ang_vel_still=vel_rot[:500, 1]):
    """Calculate drift free angle over time from angular velocity
    :param ang_vel:        Angular velocity from IMU
    :return ang_no_drift: Drift free angle over time
    """
    # Negative sign, so that angular change has the same sign as from odom and␣
↪imu
    ang_still = -np.degrees(np.cumsum(ang_vel_still / f_imu))
    drift = np.linspace(ang_still[0], ang_still[-1]/(len(ang_vel_still)/
↪len(ang_vel)), len(ang_vel))
```

```
    ang = -np.degrees(np.cumsum(ang_vel) / f_imu)
    ang_no_drift = ang - drift
    return ang_no_drift
```

Using gyroscope to limit outlier from lin_acc and odom Idea: Use a buffer of lets say 10 samples and calculate drift free angular change of this period. Now set an arbitrary threshold, below which every pitch angle change is ignored. Disadvantage of this approach is that a very slow change of the pitch angle might not be detected, that is why the threshold has to be selected carefully.
The drift is removed by having a recording at the beginning where the car stands still for several seconds and calculating the drift from that recording.

[91]:
```
def gyr_change_detection(ang_vel, win_len, threshold):
    """Returns True if sth happend
    :param ang_vel:   Angular velocity
    :param win_len:   Window length over which the angle difference from start␣
 ↪to finish is being calculated
    :param threshold: Angular changes per second below this threshold are␣
 ↪ignored.
    :return bool_lst_flat: Boolean list of length ang_vel, if angular change >␣
 ↪threshold then True, else False
    """
    buffer = []
    bool_lst = []
    # Convert threshold to deg/win_len
    threshold_norm = threshold * win_len/100
    for i,v in enumerate(ang_vel):
        buffer.append(v)
        if len(buffer) == win_len:
            ang_diff = gyr_no_drift(buffer)[-1]
            if abs(ang_diff) > threshold_norm:
                bool_lst.extend([True] * win_len)
            else:
                bool_lst.extend([False] * win_len)
            buffer = []
    # Fill missing values if len(ang_vel) is not cleanly divisble by win_len
    if len(ang_vel) % win_len != 0:
        bool_lst.extend([False] * (len(ang_vel) % win_len))
    return bool_lst

def fuse_gyr(car_angle, bool_lst):
    """Only use acc data if gyr data also detected a change, if not repeat last␣
 ↪value
    :param car_angle: Car angle calculated using IMU acc + odom
    :param bool_lst:  Boolean list, True when gyr change > threshold
    """
    fused_sig = []
    last_change = 0
    for i,v in enumerate(car_angle):
        if bool_lst[i]:
            fused_sig.append(v)
            last_change = v
        else:
            fused_sig.append(last_change)
    return np.array(fused_sig)
```

```
[94]: car_ang = pitch_car_odom(acc_x_imu_filt, acc_x_car_filt)

fig = go.Figure()
fig.add_trace(go.Scatter(x=t, y=car_ang))
win_len = 25
# Add traces, one for each slider step
for step in np.arange(0, 2, 0.1):
    fig.add_trace(
        go.Scatter(
            visible=False,
            x=t,
            y=fuse_gyr(car_ang, gyr_change_detection(vel_y, win_len, step))))
fig.data[0].visible = True
# Create and add slider
steps = []
for i in range(len(fig.data)):
    step = dict(
        method="update",
        args=[{"visible": [True] + [False] * len(fig.data)},
              {"title": "Fused signal with window length: {}s and threshold of:␣
 ↪{} deg/s".format(win_len/100.0, i/10)}],  # layout attribute
    )
    step["args"][0]["visible"][i] = True  # Toggle i'th trace to "visible"
    steps.append(step)
sliders = [dict(
    active=0,
    currentvalue={"prefix": "Threshold: "},
    steps=steps)]
fig.update_layout(sliders=sliders, xaxis_title='Time [s]', yaxis_title='[deg]')
fig.show()
```



A window length of 0.25s and a threshold of 0.9 deg/s seem to deliver resonably results. The angle, to be corrected by the gyr data, was calculated using the IMU acc + odom data filtered with a moving average filter with a window length of 50, resulting in a delay of 0.25 s. That's why I chose a window length of 0.25s, in an attempt to synchronice both signals.

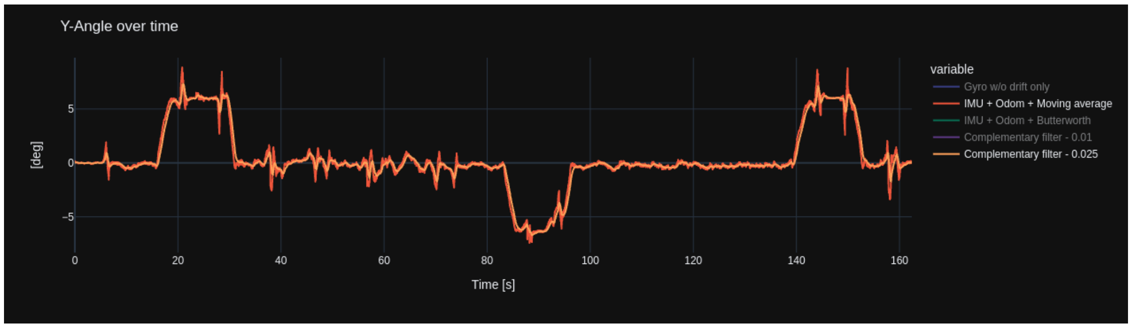
Complementary filter (similar results to Kalman filter but less complex) Complementary filter uses the good short term accuracy of the gyroscope and combines it with the accelerometer data, to reduce the drifting. The formula is as follows

$$\theta = (1-\alpha)(\theta + gyrData \cdot dt) + \alpha(accData)$$

with: - θ: Estimated angle from fusing measurements - α: Time constant response time in the range of [0-1]. With a value of 0 only the gyroscope is being used and with $\alpha = 1$ only the accelerometer. - $gyrData$: Angular velocity - $accData$: Angle from the low-pass filtered accelerometer data using $accData = \arctan 2(a_x, a_z)$ - (in our case $\arctan 2(a_x - a_{\text{car}}, a_z)$ to be precise)

[103]:
```
def complementary_filter(acc_data, gyr_data, alpha):
    angle = 0
    angle_fused = []
    for i,v in enumerate(acc_data):
        angle = (1-alpha)*(angle + gyr_data[i]*(1/f_imu)) + alpha*acc_data[i]
        angle_fused.append(angle)
    return angle_fused

fig = px.line(pd.DataFrame(data=np.vstack([ang_y_no_drift,
                                           car_ang,
                                           pitch_car_odom(acc_x_imu_filtButter,␣
 ↪acc_x_car_filtButter),
                                           complementary_filter(car_ang, vel_y,␣
 ↪0.01),
                                           complementary_filter(car_ang, vel_y,␣
 ↪0.025)]).T,
                            index=t, columns=['Gyro w/o drift only', 'IMU + Odom␣
 ↪+ Moving average', 'IMU + Odom + Butterworth',
                                              'Complementary filter - 0.01',␣
 ↪'Complementary filter - 0.025']))
```

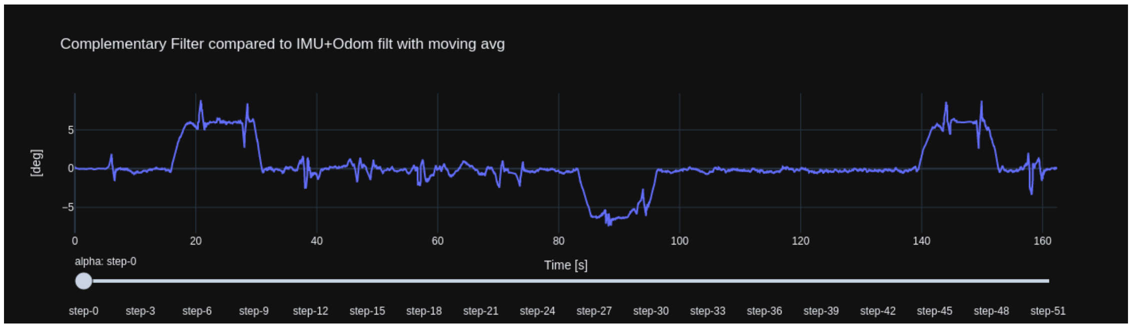
```
fig.for_each_trace(lambda trace: trace.update(visible='legendonly')
                  if trace.name in ['Gyro w/o drift only', 'IMU + Odom +␣
↪Butterworth', 'Complementary filter - 0.01'] else ())
fig.update_layout(xaxis_title='Time [s]', yaxis_title='[deg]', title='Y-Angle␣
↪over time')
```



Testing different parameter values

```
[120]: fig = go.Figure()
fig.add_trace(go.Scatter(x=t, y=car_ang))
# Add traces, one for each slider step
for step in np.linspace(0.1, 0, 51):
    fig.add_trace(
        go.Scatter(
            visible=False,
            x=t,
            y=complementary_filter(car_ang, vel_y, step)))
fig.data[0].visible = True
# Create and add slider
steps = []
for i in range(len(fig.data)):
    step = dict(
        method="update",
        args=[{"visible": [True] + [False] * len(fig.data)},
              {"title": "Angle using complementary filter with alpha: {:.3f}".
↪format(0.102-i*0.002)}],  # layout attribute
    )
    step["args"][0]["visible"][i] = True  # Toggle i'th trace to "visible"
    steps.append(step)
sliders = [dict(
    active=0,
    currentvalue={"prefix": "alpha: "},
    steps=steps)]
fig.update_layout(sliders=sliders, xaxis_title='Time [s]', yaxis_title='[deg]',␣
↪title='Complementary Filter compared to IMU+Odom filt with moving avg')
```

```
fig.show()
```



With e.g. $\alpha = 0.022$ the signal gets significantly smoother. The amplitude of undesired outliers gets more than halfed. But this comes at the cost of an increased time delay. The filtered signal is about 0.5s seconds behind.
Because the data from IMU accelerometer + odometry is already filtered and thus has a time delay of 0.25s (win size of 0.5s) the total final signal is delayed by 0.25s + 0.5s = 0.75s.

There has to be a trade-off between the accuarcy of the signal and the introduced time delay.

3 Improving live filtering and numerical differentiation

3.1 Numerical Differentation

Problems with numerical differentiation: - Signal is discrete and often noisy - Differentation acts as a high pass filter and amplifies noise - Rounding errors can occur

The most basic approach is to use finite difference apporaximations. There are three different possibilites (with h=f_s^{-1}): - forward difference:

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

- central difference:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

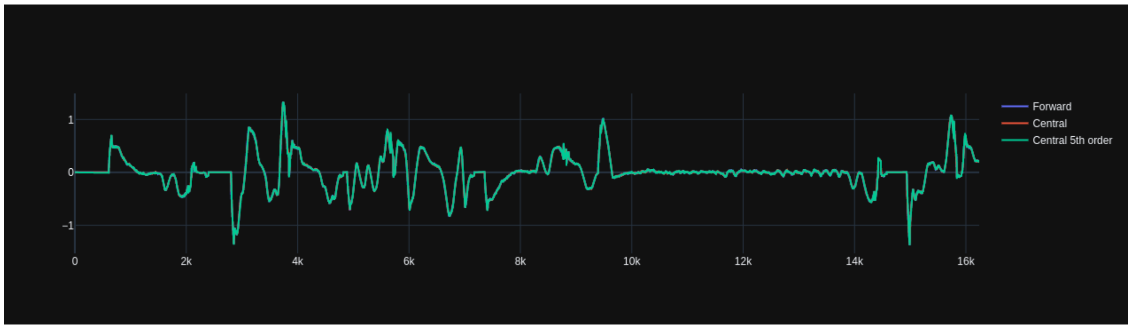
- backward difference:

$$f'(x) = \frac{f(x) - f(x-h)}{h}$$

Higher-order methods are also possible, where more than one future/past point is being used to calculate the current derivative. Finite difference coefficents to use already exist. Advantages are that the resulting signal gets smoother and is less affected by outliers, but at the cost of a slight time delay.

```
[121]: def acc_from_vel_fwd(vel):
           acc = []
```

```
    for i in range(len(vel)-1):
        a = (vel[i+1] - vel[i]) / (1.0/f_odom)
        acc.append(a)
    acc.append(a)
    # Add points if odometry topic recorded less msgs than imu
    msgs_diff = len(lin_acc) - len(odom)
    while msgs_diff > 0:
        acc.append(a)
        msgs_diff -= 1
    return acc

def acc_from_vel_central(vel):
    acc = [0]
    for i in range(1, len(vel)-1):
        a = (vel[i+1] - vel[i-1]) / (2*1.0/f_odom)
        acc.append(a)
    acc.append(a)
    # Add points if odometry topic recorded less msgs than imu
    msgs_diff = len(lin_acc) - len(odom)
    while msgs_diff > 0:
        acc.append(a)
        msgs_diff -= 1
    return acc

# Example of high-order method (here with an order of 5)
def acc_from_vel_5(vel):
    acc = [0, 0]
    for i in range(2, len(vel)-2):
        a = (-vel[i+2] + 8*vel[i+1] - 8*vel[i-1] + vel[i-2]) / (12*1.0/f_odom)
        acc.append(a)
    acc.extend([a, a])
    msgs_diff = len(lin_acc) - len(odom)
    while msgs_diff > 0:
        acc.append(a)
        msgs_diff -= 1
    return acc

fig = go.Figure()
fig.add_trace(go.Scatter(y=acc_from_vel_fwd(vel_x_car_filt), name='Forward'))
fig.add_trace(go.Scatter(y=acc_from_vel_central(vel_x_car_filt),␣
 ↪name='Central'))
fig.add_trace(go.Scatter(y=acc_from_vel_5(vel_x_car_filt), name='Central 5th␣
 ↪order'))
```



Comparison of the different numerical differentiation methods, by deriving the car velocity
It can be seen that the difference between the methods is very marginal, thus the use of the most simplest method (forward (or backward)) is sufficent.

3.2 Filtering

Moving average filter

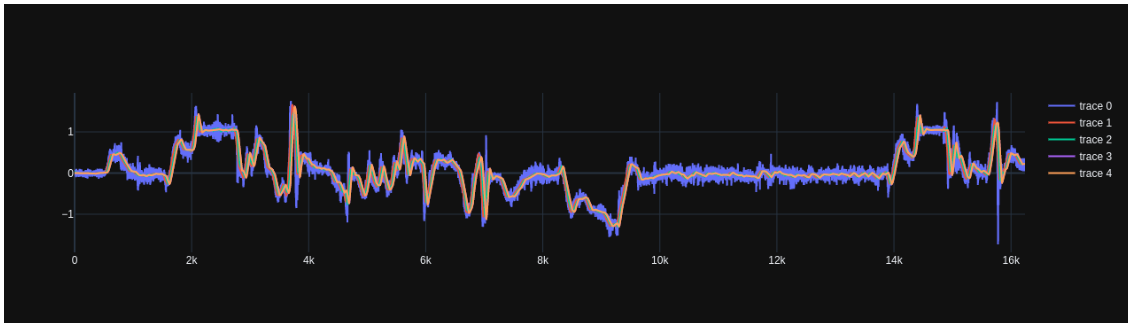
```
[ ]: def moving_average(signal, window_size):
         sum = 0
         values = []
         filtered = []
         for i,v in enumerate(signal):
             values.append(v)
             sum += v
             if len(values) > window_size:
                 sum -= values.pop(0)
             filtered.append(float(sum) / len(values))
         return np.array(filtered)
```

4-th order Butterworth filter with a cut-off frequency of 1 Hz

```
[122]: fc = 1 # Cutoff frequency in Hz
       w = fc / (f_imu/2) # norm fc
       b, a = signal.butter(4, w, 'low')

       b2, a2 = signal.iirfilter(4, w, btype='lowpass')
```

```
[123]: def lp_filter(sig, window_size):
           fc = 1 # Cutoff frequency in Hz
           w = fc / (f_imu/2) # Nyquist
           b, a = signal.butter(4, w, 'low')
           values = []
           filtered = []
           for i,v in enumerate(sig):
```

```
        values.append(v)
        filtered.append(signal.lfilter(b,a,values)[-1])
        if len(values) == window_size:
            values.pop(0)
    return np.array(filtered)

fig = go.Figure()
fig.add_trace(go.Scatter(y=acc_x_imu))
fig.add_trace(go.Scatter(y=signal.filtfilt(b,a,acc_x_imu)))
fig.add_trace(go.Scatter(y=moving_average(acc_x_imu, 50)))
# fig.add_trace(go.Scatter(y=lp_filter(acc_x_imu, 50)))
fig.add_trace(go.Scatter(y=signal.lfilter(b,a,acc_x_imu)))
fig.add_trace(go.Scatter(y=signal.lfilter(b2,a2,acc_x_imu)))
```


2k
4k
6k
8k
10k
12k
14k
16k
trace 0
trace 1
trace 2
trace 3
trace 4