mems.py

```python
# Micro Electro Mechanical Systems
# #############################################################################

# This python script...

# Authors:
# Christopher Mahn
# Parastoo Oghlansheikhha
# Lukas-René Schulz
# Silas Teske
# Joshua Wolf
# Hajar Zare

# #############################################################################

# Import of Libraries
# -----------------------------------------------------------------------------

# import string as st
# import random as r
# import re
from turtle import position
import matplotlib.pyplot as plt
# from scipy import interpolate
import numpy as np
import math as m
# import sys
import os
# from scipy.fft import fft, fftfreq
# from scipy import signal


# -----------------------------------------------------------------------------
# Settings

verbose = False  # Shows more debugging information

# The following values are the stationary parts of measurements, where the IMU
# wasn't moved.
stationary_track = {"before": [{"start": 15, "end": 710},
                               {"start": 7, "end": 597},
                               {"start": 11, "end": 485},
                               {"start": 7, "end": 522},
                               {"start": 7, "end": 250},
                               {"start": 16, "end": 300},
                               {"start": 16, "end": 509},
                               {"start": 34, "end": 548},
                               {"start": 14, "end": 556},
                               {"start": 23, "end": 432},
                               {"start": 0, "end": 0},
                               {"start": 17, "end": 576},
                               {"start": 23, "end": 392}],
                    "after": [{"start": 1125, "end": 1654},
                              {"start": 973, "end": 1320},
                              {"start": 923, "end": 1691},
                              {"start": 772, "end": 1467},
                              {"start": 821, "end": 1321},
                              {"start": 939, "end": 1391},
                              {"start": 1438, "end": 2041},
                              {"start": 1914, "end": 2338},
                              {"start": 1535, "end": 2229},
                              {"start": 1931, "end": 2294},
                              {"start": 0, "end": 0},
                              {"start": 1658, "end": 1999},
                              {"start": 1503, "end": 2348}]}
stationary_turntable = {"before": [{"start": 452, "end": 2574},
                                   {"start": 38, "end": 1427},
                                   {"start": 24, "end": 768},
                                   {"start": 17, "end": 589},
                                   {"start": 12, "end": 617},
                                   {"start": 21, "end": 1967},
                                   {"start": 7, "end": 140},
                                   {"start": 4, "end": 213},
                                   {"start": 10, "end": 818},
                                   {"start": 9, "end": 389},
                                   {"start": 6, "end": 354},
                                   {"start": 6, "end": 202}],
                        "after": [{"start": 5262, "end": 5583},
                                  {"start": 3621, "end": 3802},
                                  {"start": 2365, "end": 2840},
```

```python
                                  {"start": 2250, "end": 2406},
                                  {"start": 1642, "end": 1879},
                                  {"start": 2901, "end": 3004},
                                  {"start": 1405, "end": 1645},
                                  {"start": 1249, "end": 1304},
                                  {"start": 1762, "end": 1834},
                                  {"start": 1760, "end": 1820},
                                  {"start": 1245, "end": 1346},
                                  {"start": 1158, "end": 1406}]}


# Functions
# -----------------------------------------------------------------------------

def printf(string, filename="output", end="\n"):
    """
    This function writes a string to a file instead of the command line. The
    output will always be appended to the file.

    Args:
        string (str): This string will be written
        filename (str, optional): This is the name of the file, the string will
                                  be written to. Defaults to "output.txt".
        end (str, optional): This is a appended return-character, that can also
                             be changed. Defaults to "\n".
    """
    if(verbose):
        print(f'[Info] Writing "{string}" to "{filename}.txt"')
    with open(os.path.join("data", f'{filename}.txt'), "a") as file:
        file.write(f'{string}{end}')


def clearf(filename="output"):
    """
    This function removes the contents of a file.

    Args:
        filename (str, optional): This specifies the file to be cleared.
                                  Defaults to "output".
    """
    if(verbose):
        print(f'[Info] Clearing "{filename}.txt"')
    with open(os.path.join("data", f'{filename}.txt'), "w") as file:
        file.write("")


def import_data(input_filename):
    """
    This function is used to import a singe measurement string of ins-data.

    Args:
        input_filename (str): This specifies the name of the file, that will be
        imported.
    """
    # Opening and reading file from disk
    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    if(verbose):
        print(f'[Info] Opening file "{input_filename}"', end="\r")
    with open(os.path.join("data", input_filename)) as file:
        if(verbose):
            print(f'[Info] Reading file "{input_filename}"', end="\r")
        data = file.readlines()
    if(verbose):
        print(f'[Info] Read file "{input_filename}" successfully')

    # Formating the data from disk into a two-dimentional list
    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    for i, e in enumerate(data):
        try:
            data[i] = e.split(';')
            for j, e in enumerate(data[i]):
                if(verbose):
                    print(f'[Info][{i+1}/{len(data)}][{j+1}/{len(data[i])}] Importing entries', end="\r")
                if(j==0):
                    data[i][j] = int(e.strip())
                else:
                    data[i][j] = float(e.strip())
        except(ValueError):
            if(verbose):
                print(f'[Warn] Found weiredly formatted data at line {i+1}{20*" "}')
    if(verbose):
        print("")
```

```
    # Return the loaded data
    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    return(data)


def calc_offset(data, start=0, end=None):
    """
    This function determines the offset/bias of a specific set of values from a
    list. Using the optional start and end indexes a specific set of values can
    be selected from the data.

    Args:
        data ([float]): A list with values serves as the provided data
        start (int, optional): By providing a starting value the data partially
                               selected
        end (int, optional): By providing an ending value the data partially
                             selected
    """
    if(verbose):
        print(f'[Info] Determing a bias')
    if(end==None):
        end = len(data)
    data = np.array(data)
    bias = np.mean(data[start:end])
    return(bias)


def remove_bias(data, bias):
    """
    This function substracts an offset of a list of sensor-values.

    Args:
        data ([float]): This is the sensor-data that will be removed of bias
        bias (float): This is the bias, that will be removed from the data
    """
    for i, e in enumerate(data):
        if(verbose):
            print(f'[Info][{i+1}/{len(data)}] Removing bias from data', end="\r")
        data[i] = e-bias
    if(verbose):
        print("")
    return(data)


def remove_bias_advanced(data, bias1, bias2, index1, index2):
    """
    This function removes bias from data. For anything before index1, bias1
    will be removed and after index2, bias2 will be removed. In between index1
    and index2 the removal of the bias will be interpolated in a linear way.

    Args:
        data ([float]): This data the bias-removal will be applied to
        bias1 (float): bias at the beginning of the measurement
        bias2 (_type_): bias at the end of the measurement
        index1 (int): index marking the beginning of the movement
        index2 (int): index marking the end of the movement
    """
    for i, e in enumerate(data):
        if(verbose):
            print(f'[Info][{i+1}/{len(data)}] Removing bias from data (advanced)', end="\r")
        if(i <= index1):
            data[i] = e-bias1
        elif(i >= index2):
            data[i] = e-bias2
        else:
            bias2_part = (i-index1)/(index2-index1)
            bias1_part = 1-bias2_part
            data[i] = e-(bias1*bias1_part)-(bias2*bias2_part)
    if(verbose):
        print("")
    return(data)


def calc_velocity(accelerometer_data, timestamp_data):
    """
    This function calculates the velocity from data of an accelerometer and
    their designated timestamps.

    Args:
        accelerometer_data ([{"x": float, "y": float, "z": float}]): This is the accelerometer-data
                                                                      and must be formatted as
  dictionaries
```

```
                                                                      inside of a list
        timestamp_data ([float]): This is timestamp-data from the measurements
                                  of the accelerometer
    """
    velocity = {"x": [], "y": [], "z": []}
    velocity_i = {"x": 0.0, "y": 0.0, "z": 0.0}
    for i, e in enumerate(timestamp_data):
        for j, xyz in enumerate(["x", "y", "z"]):
            if(verbose):
                print(f'[Info][{i+1}/{len(timestamp_data)}][{j+1}/3] Calculating velocities', end="\r")
            if(i==0):
                velocity_i[xyz] += accelerometer_data[xyz][i]*e
            else:
                velocity_i[xyz] += accelerometer_data[xyz][i]*(e-timestamp_data[i-1])
            velocity[xyz].append(velocity_i[xyz])
    if(verbose):
        print("")
    return(velocity)


def calc_position(accelerometer_data, timestamp_data, velocity_data):
    """
    This function calculates positions based on velocity-data and timestamps.

    Args:
        accelerometer_data ([{"x": float, "y": float, "z": float}]): This is the accelerometer-data
                                                                      and must be formatted as dictionaries
                                                                      inside of a list
        timestamp_data ([float]): This is timestamp-data for the velocity-data
    """
    position = {"x": [], "y": [], "z": []}
    position_i = {"x": 0.0, "y": 0.0, "z": 0.0}
    for i, e in enumerate(timestamp_data):
        for j, xyz in enumerate(["x", "y", "z"]):
            if(verbose):
                print(f'[Info][{i+1}/{len(timestamp_data)}][{j+1}/3] Calculating positions', end="\r")
            if(i==0):
                position_i[xyz] += 0.5*accelerometer_data[xyz][i]*(e**2)+velocity_data[xyz][i]*e
            else:
                position_i[xyz] += 0.5*accelerometer_data[xyz][i]*((e-timestamp_data[i-1])**2)+velocity_data[xyz][i]*(e-timestamp_data[i-1])
            position[xyz].append(position_i[xyz])
    # if(verbose):
    #     print("")
    return(position)


def calc_rotation(gyroscope_data, timestamp_data):
    """
    This function calculates the rotation based on velocity-data and timestamps.

    Args:
        gyroscope_data ([{"x": float, "y": float, "z": float}]):      his is the gyroscope-data
                                                                      and must be formatted as dictionaries
                                                                      inside of a list
        timestamp_data ([float]): This is timestamp-data for the velocity-data
    """
    rotation = {"x": [], "y": [], "z": []}
    rotation_i = {"x": 0.0, "y": 0.0, "z": 0.0}
    for i, e in enumerate(timestamp_data):
        for j, xyz in enumerate(["x", "y", "z"]):
            if(verbose):
                print(f'[Info][{i+1}/{len(timestamp_data)}][{j+1}/3] Calculating turnrates', end="\r")
            if(i==0):
                rotation_i[xyz] += gyroscope_data[xyz][i]*e
            else:
                rotation_i[xyz] += gyroscope_data[xyz][i]*(e-timestamp_data[i-1])
            rotation[xyz].append(rotation_i[xyz])
    if(verbose):
        print("")
    return(rotation)


def calc_distance(positions, stationary_start, stationary_end):
    """
    This function calculates the distance of a IMU-measurement
    """
    if(verbose):
        print("[Info] Calculating distance")
    delta = []
    for xyz in ["x", "y", "z"]:
```

```
        delta.append((positions[xyz][stationary_end["start"]]-positions[xyz][stationary_start["end"]])**2)
    distance = np.sqrt(delta[0]+delta[1]+delta[2])
    return(distance)


def calc_trajectory(position_data, rotation_data):
    """
    This function calculates a 2D trajectory of an IMU using integrated values as
    distance and rotation.

    Args:
        position_data ({"x": [float], "y": [float], "z": [float]}): position data as lists in a dictionary
        rotation_data ({"x": [float], "y": [float], "z": [float]}): rotation data as lists in a dictionary
    """
    if(verbose):
        print("[Info] Calculating trajectory")
    last_value = {"x": 0.0, "y": 0.0, "z": 0.0}
    position_change = {"x": [], "y": [], "z": []}
    for xyz in ["x", "y", "z"]:
        for i, e in enumerate(position_data[xyz]):
            position_change[xyz].append(e-last_value[xyz])
            last_value[xyz] = e
    trajectory = {"x": [0.0], "y": [0.0]}
    rho = m.pi/180
    for i, e in enumerate(position_change["x"]):
        if(verbose):
            print(f'[Info][{i+1}/{len(position_change["x"])}] Calculating trajectory', end="\r")
        trajectory["x"].append(trajectory["x"][i] + (m.sin(rotation_data["z"][i]*rho)*position_change["x"][i]) + (m.cos(rotation_data["z"][i]*rho)*position_change["y"][i]))
        trajectory["y"].append(trajectory["y"][i] + (m.sin(rotation_data["z"][i]*rho)*position_change["y"][i]) + (m.cos(rotation_data["z"][i]*rho)*position_change["x"][i]))
    if(verbose):
        print("")
    return(trajectory)


def write_bias_acceleration(bias_data, filename, measurement):
    """
    This funtion writes the bias of the measurements to a file on disk.

    Args:
        bias_data ([float]): The bias of the data must be formatted in
                          [x, y, z] and the unit of measurement must be m/s²
        filename (str): This is the filename under wich the data will be
                        appended to.
        measurement (str): This specifies the description of the measurement
                           and will be printed only for user-friendlyness
                           purposes
    """
    if(verbose):
        print(f'[Info] Writing biases of measurement {measurement} to "{filename}.txt"')
    printf(f'The following biases were determined for measurement "{measurement}":', filename)
    printf(f'X: {bias_data[0]:.6f} m/s^2', filename)
    printf(f'Y: {bias_data[1]:.6f} m/s^2', filename)
    printf(f'Z: {bias_data[2]:.6f} m/s^2', filename)
    printf(f'{15*"- "}-', filename)
    printf("", filename)


def write_bias_rotation(bias_data, filename, measurement):
    """
    This funtion writes the bias of the measurements to a file on disk.

    Args:
        bias_data ([float]): The bias of the data must be formatted in
                          [x, y, z] and the unit of measurement must be m/s²
        filename (str): This is the filename under wich the data will be
                        appended to.
        measurement (str): This specifies the description of the measurement
                           and will be printed only for user-friendlyness
                           purposes
    """
    if(verbose):
        print(f'[Info] Writing biases of measurement {measurement} to "{filename}.txt"')
    printf(f'The following biases were determined for measurement "{measurement}":', filename)
    printf(f'X: {bias_data[0]:.6f} degree/s', filename)
    printf(f'Y: {bias_data[1]:.6f} degree/s', filename)
    printf(f'Z: {bias_data[2]:.6f} degree/s', filename)
    printf(f'{15*"- "}-', filename)
    printf("", filename)
```

```python


def plot_results(datasets, title_label, x_label, y_label, data_label, timestamps=None):
    """
    This function plots graphs.

    Args:
        datasets ([[float]]): A list with datasets a lists with floating-point

                              numbers
        title_label (str): This is the tile of the plot
        x_label (str): This is the label of the x-axis
        y_label (str): This is the label of the y-axis
        data_label ([str]): This is a list with labels of the datasets
        timestamps ([float], optional): By using a list of floating-point
                                        numbers the data get's plotted on a
                                        time-axis. If nothing is provided the
                                        values will be plotted equidistant.
    """
    for i, dataset in enumerate(datasets):
        if(timestamps==None):
            timestamps = range(len(dataset))
        plt.plot(timestamps, dataset)
    plt.legend(data_label)
    plt.grid()
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(title_label)
    plt.show()


def process_data(measurement, number_of_measurements, stationary_indices, plot=False):
    """
    This function processes datasets labeled to the following:
    "<mesurement>_01.csv"

    Args:
        measurement (str): name of the dataset
        number_of_measurements (int): the total amount of measurements to interate over
        stationary_indices ({"before": [{"start": int, "end": int}], "after": [{"start": int, "end": int}]}): A complex dictionary/list of indices for stationary part during the measurement.
        plot (bool, optional): If enabled all the plots will be created (a lot of window-popups). Defaults to False.
    """
    # Clearing output-files
    clearf(f"{measurement}_biases")
    clearf(f"{measurement}_distances")

    # Running data-processing for each measurement
    for measurement_id in range(number_of_measurements):
        data = import_data(f"{measurement}_{measurement_id+1:02d}.csv")

        # Putting the data into sensor-streams as lists
        accelerometer = {"x": [], "y": [], "z": []}
        gyroscope = {"x": [], "y": [], "z": []}
        timestamps = []
        for sensor_info in data:
            timestamps.append(sensor_info[0]/1000)
            for i, e in enumerate(["x", "y", "z"]):
                accelerometer[e].append(sensor_info[i+1])
                gyroscope[e].append(sensor_info[i+4])

        # Determine the biases before and after the movement for the accelerometer
        accelerometer_bias = {"before": {"x": 0.0, "y": 0.0, "z": 0.0},
                              "after": {"x": 0.0, "y": 0.0, "z": 0.0}}
        for i in ["before", "after"]:
            for xyz in ["x", "y", "z"]:
                accelerometer_bias[i][xyz] = calc_offset(accelerometer[xyz],
                                                         stationary_indices[i][measurement_id]["start"],
                                                         stationary_indices[i][measurement_id]["end"])
            write_bias_acceleration([accelerometer_bias[i]["x"],
                                     accelerometer_bias[i]["y"],
                                     accelerometer_bias[i]["z"]],
                                     f'{measurement}_biases',
                                     f'{measurement_id+1:02d} ({i} movement)')

        # Determine the biases before and after the movement for the gyroscope
        gyroscope_bias = {"before": {"x": 0.0, "y": 0.0, "z": 0.0},
                          "after": {"x": 0.0, "y": 0.0, "z": 0.0}}
        for i in ["before", "after"]:
            for xyz in ["x", "y", "z"]:
                gyroscope_bias[i][xyz] = calc_offset(gyroscope[xyz],
```

```python
                                                stationary_indices[i][measurement_id]["start"],
                                                stationary_indices[i][measurement_id]["end"])
        write_bias_rotation([gyroscope_bias[i]["x"],
                             gyroscope_bias[i]["y"],
                             gyroscope_bias[i]["z"]],
                             f'{measurement}_biases',
                             f'{measurement_id+1:02d} ({i} movement)')

    # Removing biases
    accelerometer_without_bias = {"x": [], "y": [], "z": []}
    gyroscope_without_bias = {"x": [], "y": [], "z": []}
    for xyz in ["x", "y", "z"]:
        accelerometer_without_bias[xyz] = remove_bias_advanced(accelerometer[xyz],
                                                               accelerometer_bias["before"][xyz],
                                                               accelerometer_bias["after"][xyz],
                                                               stationary_indices["before"][measurement_id]["end"],
                                                               stationary_indices["after"][measurement_id]["start"])
        gyroscope_without_bias[xyz] = remove_bias_advanced(gyroscope[xyz],
                                                           gyroscope_bias["before"][xyz],
                                                           gyroscope_bias["after"][xyz],
                                                           stationary_indices["before"][measurement_id]["end"],
                                                           stationary_indices["after"][measurement_id]["start"])

    # Calculation of velocity

    velocity = calc_velocity(accelerometer_without_bias, timestamps)

    # Determine offset of velocity before and after movement

    velocity_offset = {"before": {"x": 0.0, "y": 0.0, "z": 0.0},
                       "after": {"x": 0.0, "y": 0.0, "z": 0.0}}
    for i in ["before", "after"]:
        for xyz in ["x", "y", "z"]:
            velocity_offset[i][xyz] = calc_offset(velocity[xyz],
                                                  stationary_indices[i][measurement_id]["start"],
                                                  stationary_indices[i][measurement_id]["end"])

    # Remove offset of velocity before and after movement

    velocity_without_bias = {"x": [], "y": [], "z": []}
    for xyz in ["x", "y", "z"]:
        velocity_without_bias[xyz] = remove_bias_advanced(velocity[xyz],
                                                          velocity_offset["before"][xyz],
                                                          velocity_offset["after"][xyz],
                                                          stationary_indices["before"][measurement_id]["end"],
                                                          stationary_indices["after"][measurement_id]["start"])

    # Calculation of position and rotation

    position = calc_position(accelerometer_without_bias, timestamps, velocity)
    rotation = calc_rotation(gyroscope_without_bias, timestamps)

    # Calculation of the measured distance
    distance = calc_distance(position, stationary_indices["before"][measurement_id], stationary_indices["after"][measurement_id])
    if(measurement_id != 10):
        list_of_distances.append(distance)
    printf(f'The distance from measurement {measurement_id+1:02d} is {distance:6.3f} m.', f'{measurement}_distances')

    # Calculation of a trajectory
    trajectory = calc_trajectory(position, rotation)

    # Plot of data
    if(plot == True):
        plot_results([gyroscope["x"], gyroscope["y"], gyroscope["z"]],
                     f'Raw gyroscope-data from {measurement} with measurement {measurement_id+1:02d}',
                     "time [s]",
                     "rotation change [°/s]",
                     ["x", "y", "z"],
                     timestamps)
        plot_results([accelerometer["x"], accelerometer["y"], accelerometer["z"]],
                     f'Raw accelerometer-data from {measurement} with measurement {measurement_id+1:02d}',
                     "time [s]",
                     "acceleration [m/s²]",
```

```
                         ["x", "y", "z"],
                         timestamps)
            plot_results([gyroscope_without_bias["x"], gyroscope_without_bias["y"], gyroscope_without_bias["z"]],
                         f'Gyroscope-data from {measurement} with measurement {measurement_id+1:02d} with bias removed',
                         "time [s]",
                         "rotation change [°/s]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([accelerometer_without_bias["x"], accelerometer_without_bias["y"], accelerometer_without_bias["z"]],
                         f'Accelerometer-data from {measurement} with measurement {measurement_id+1:02d} with bias removed',
                         "time [s]",
                         "acceleration [m/s²]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([velocity["x"], velocity["y"], velocity["z"]],
                         f'Velocity from {measurement} with measurement {measurement_id+1:02d} (without offset-removal)',
                         "time [s]",
                         "velocity [m/s]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([velocity_without_bias["x"], velocity_without_bias["y"], velocity_without_bias["z"]],
                         f'Velocity from {measurement} with measurement {measurement_id+1:02d} (with the offset removed)',
                         "time [s]",
                         "velocity [m/s]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([position["x"], position["y"], position["z"]],
                         f'Position from {measurement} with measurement {measurement_id+1:02d}',
                         "time [s]",
                         "position [m]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([rotation["x"], rotation["y"], rotation["z"]],
                         f'Rotation from {measurement} with measurement {measurement_id+1:02d}',
                         "time [s]",
                         "rotation [°]",
                         ["x", "y", "z"],
                         timestamps)
            plot_results([trajectory["x"]],
                        f'Trajectory from {measurement} with measurement {measurement_id+1:02d}',
                        "Y",
                        "X",
                        ["trajectory"],
                        trajectory["y"])

# Classes
# ----------------------------------------------------------------------------


# Beginning of the Programm
# ----------------------------------------------------------------------------

if __name__ == '__main__':
    diagrams = False

    list_of_distances = []
    with open(os.path.join("data", "distances.txt"), "w") as file:
        file.write("")

    process_data("data_track", 13, stationary_track, diagrams)

    for i in range(12):
        printf(f'The distance for measurement {i+1:2.0f} is: {list_of_distances[i]:6.3f} m', "distances")
    distance_min = np.min(np.array(list_of_distances))
    printf(f'Min: {distance_min:6.3f} m', "distances")
    distance_max = np.max(np.array(list_of_distances))
    printf(f'Max: {distance_max:6.3f} m', "distances")
    distance_avg = np.average(np.array(list_of_distances))
    printf(f'Avg: {distance_avg:6.3f} m', "distances")
    distance_std = np.std(np.array(list_of_distances))
    printf(f'Std: {distance_std:6.3f} m', "distances")

    diagrams = False
    process_data("data_turntable", 12, stationary_turntable, diagrams)
```