Assignment 1

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission.

*Note that this assignment was designed to run in the Jupyter Notebook** environment.*

The assignment has three tasks:

- Task 1: Understanding the limitations of double-integrating acceleration time-series to obtain position time-series and exploring the efficacy of low-pass filter (LPF) in reducing noice.
- Task 2: Experimenting with sensor fusion algorithms to obtain acceleration in the global coordinate system while the sensor is in motion and assessing the associated challenges to gain insights into the complexity of the task.
- Task 3: Experimenting with sensor fusion algorithms to obtain acceleration in the global coordinate system while the sensor is moving randomly.

In [ ]:
```
# Run some setup code for this notebook
import os
import copy
import numpy as np
import pandas as pd
from types import SimpleNamespace
from scipy.signal import find_peaks, resample, butter, filtfilt
from scipy.integrate import cumtrapz
from scipy.spatial.transform import Rotation as R

import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

from cs690r.data_utils import load_sensor_from_csv, load_mocap_from_tsv, trim_data
from cs690r.plot_utils import plot_time_series, animate_trajectory, compare_traject

# The commands will allow the notebook to reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

`Attitude and Heading Reference Systems (AHRS)` is a Python library for sensor fusion, specifically for estimating orientation from inertial measurement unit (IMU) data, which typically includes acceleration and angular velocity measurements.

To download and install the package, you can use the following command in their terminal or command prompt:

```
pip install ahrs
```

Refer to this page for more detailed information about ahrs installation.

Once you install the package, you can import and use the ahrs.filters module for sensor fusion algorithms in you Python scripts or Jupyter notebooks.

In [ ]:
```
from ahrs.filters import *
```

In [ ]:
```
# Define gravity constant
GRAVITY_CONSTANT = 9.80665
```

Task 1

You can find the data for Task 1 in the "data" folder. In this task, a sensor equipped with accelerometer, gyroscope, and magnetometer was placed on a stationary horizontal surface. To facilitate this process, a function for loading sensor data from text files has been implemented for students. This function is located in the "cs690r.data_utils" module.

You can access the attributes of sensor data using the following code

```
sample_rate = sensor_data.sample_rate # sampling rate of the sensor, 100Hz
acc = sensor_data.acc # acceleration, unit: m/s^2
gyr = sensor_data.gyr # angular velocity, unit: rad/s
mag = sensor_data.mag # magnetometer data, unit: a.u. (arbitrary units;
normalized to earth field strength)

free_acc = sensor_data.free_acc # the sensor's manufacturer (Movella
XSens) has its proprietary algorithm to estimate the gravity-free
acceleration in the global coordinates
```
Please note that the data in the file is raw and unfiltered.

In [ ]:
```
# Load stationary data
task1_sensor_file = os.path.join('data', 'task1_sensor_data.csv')
task1_sensor_data = load_sensor_from_csv(task1_sensor_file)
```

In [ ]:
```
# Remove gravity from the z-axis, given that the sensor was placed stationary
task1_sensor_data.acc[:, 2] -= GRAVITY_CONSTANT
```

Task 1.1:

1. Implement the `filtering_and_integrate` function provided below. In this function, use a 6th order Butterworth filter with a cut-off frequency of 8 Hz to low-pass filter the raw acceleration and angular velocity, respectively. Integrate the filtered acceleration to derive the velocity time-series, followed by band-pass filtering (2nd order Butterworth) with cut-off frequency between 0.1 Hz and 8 Hz to address integration drift and high-frequency noise. Repeat the integration and band-pass filtering process to the derived velocity time-series to get position time-series.

2. Apply `filtering_and_integrate` to `task1_sensor_data`. Specifically, double-integrate the acceleration time-series to obtain position time-series both **with and without filtering**.

Hint:

1. For integration, consider `scipy.integrate.cumtrapz` or `scipy.integrate.cumulative_trapezoid` for integration depending on the version of `scipy` you are using. You are also free to use other integration implementations you are familiar with.

2. For filtering, checkout `Filtering Example.ipynb` in this folder to learn how to design different types of filters.

In [ ]:
```python
# Exercise 1.1.1: Implement the `filtering_and_integrate` function provided below
def filtering_and_integrate(data, use_filter=True):
    '''
    Input Parameters
    ----------
    data : sensor data object
        To access the acceleration, use:
        acc = data.acc

    use_filter : boolean
        If use_filter is set to True,
        low-pass and band-pass filters will be applied to the data

    Output Parameters
    -----------
    data : sensor data object with filtered data
        data.acc: filtered acceleration
        data.vel: integrated, filtered velocity
        data.pos: integrated, filtered position
        data.gyr: filtered angular velocity
    '''
    data = copy.deepcopy(data)

    # Get acceleration
    acc = data.acc.copy()

    # Get angular velocity
    gyr = data.gyr.copy()

    # Get the sampling rate of the sensor
    sample_rate = data.sample_rate

    ####################################################################
    # TODO: Define low-pass filter and band-pass filter
    # apply the filters on the sensor acceleration and angular velocity
    # if use_filter is True
    # integrate the acceleration to get velocity and position
    vel = np.zeros_like(acc)
    pos = np.zeros_like(vel)
    for i in range(3):
        if(use_filter):
            cutoff = 8
            b, a = butter(6,cutoff/(sample_rate/2),btype='lowpass')
            acc[:,i] = filtfilt(b,a,acc[:,i])
            gyr[:,i] = filtfilt(b,a,gyr[:,i])

        vel[:,i] = cumtrapz(acc[:,i],dx=1/sample_rate,initial=0)

        if(use_filter):
            low_cutoff = 0.1  # Lower cutoff frequency in Hz
            high_cutoff = 8  # Upper cutoff frequency in Hz

            # Design Butterworth band-pass filter
            b, a = butter(2, [low_cutoff / (sample_rate / 2), high_cutoff / (sample
            vel[:,i] = filtfilt(b,a,vel[:,i])

        pos[:,i] = cumtrapz(vel[:,i],dx=1/sample_rate,initial=0)

        if(use_filter):
            low_cutoff = 0.1  # Lower cutoff frequency in Hz
            high_cutoff = 8  # Upper cutoff frequency in Hz

            # Design Butterworth band-pass filter
            b, a = butter(2, [low_cutoff / (sample_rate / 2), high_cutoff / (sample
            pos[:,i] = filtfilt(b,a,pos[:,i])



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

    # Save acceleration, velocity, position, and angular velocity
    data.acc = acc
    data.vel = vel
    data.pos = pos
    data.gyr = gyr
    return data
```

In [ ]:

```
# Exercise 1.1.2: Apply `filtering_and_integrate` on task1_sensor_data
# with and without filters
filt_data = filtering_and_integrate(task1_sensor_data)
no_filt_data = filtering_and_integrate(task1_sensor_data,use_filter=False)
```

Task 1.2

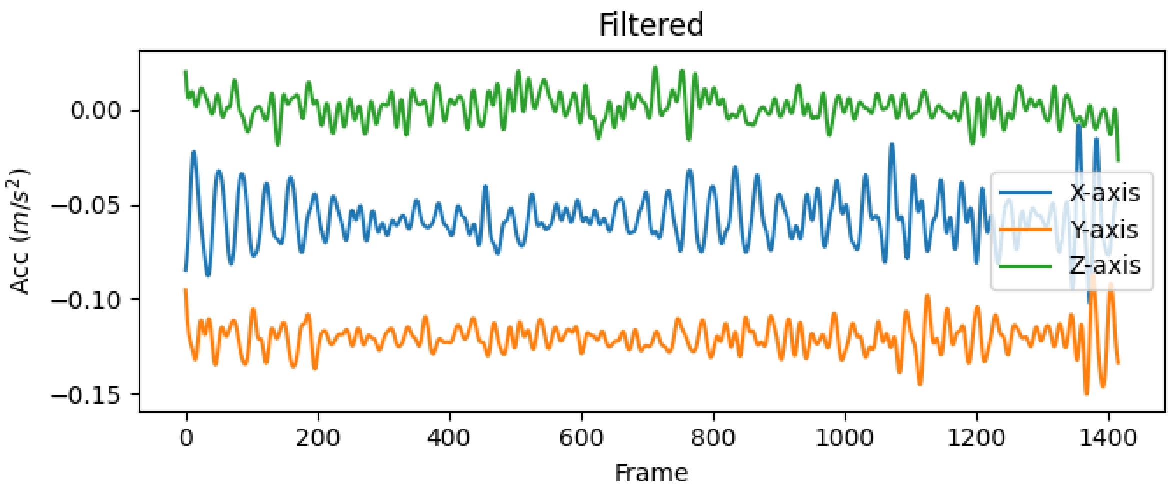
1. Apply `plot_time_series` (located in plot_utils.py) to generate 1D time series plots for acceleration, velocity, and position in each scenario (with and without filtering).

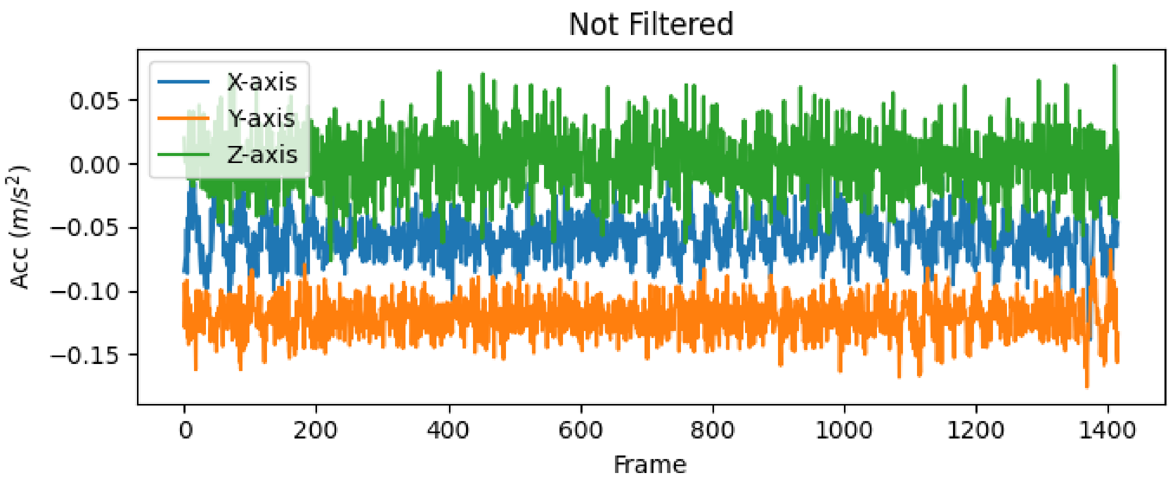
2. Animate the position time-series derived from with/without a low-pass filter using the provided `animate_trajectory` (located in plot_utils.py).

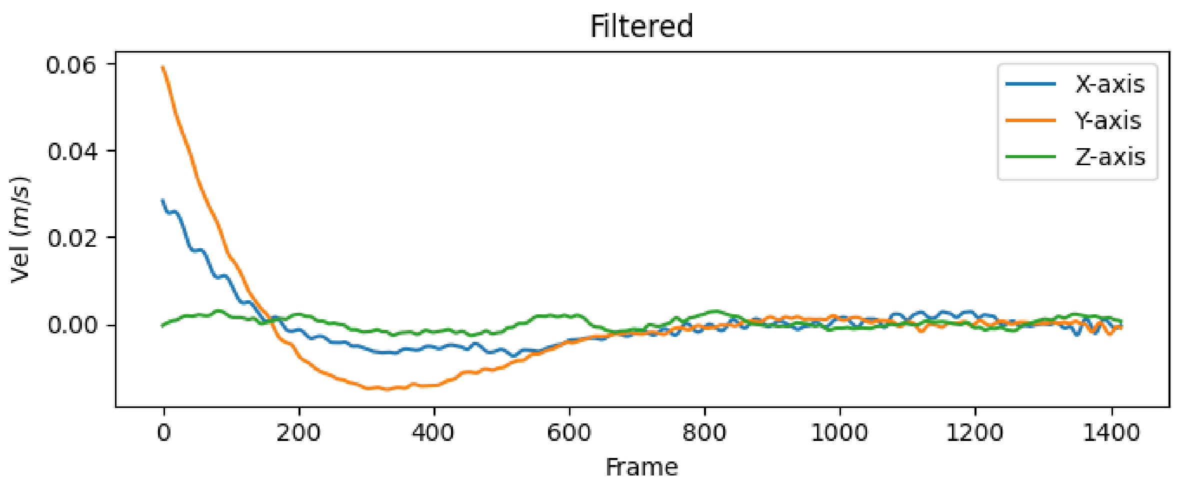
Even though the sensor was placed on a horizontal surface, the raw unfiltered acceleration does not seem to be perfectly zero due to imperfect calibration and inherent sensor noise.

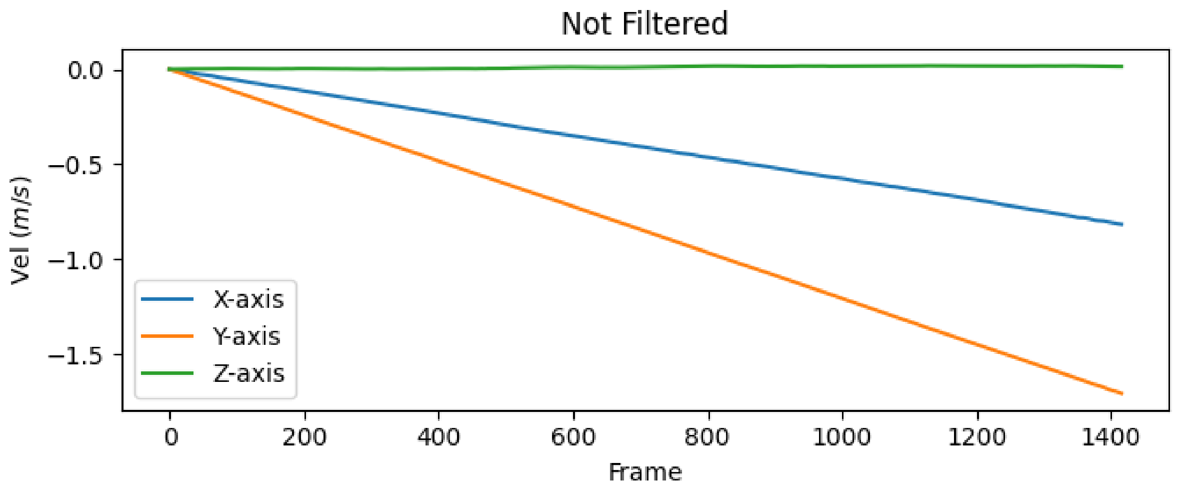
Integrating the raw unfiltered acceleration to obtain velocity and position may cause significant drift over time. However, utilizing a low-pass filter and a band-pass filter can effectively mitigate this drift.

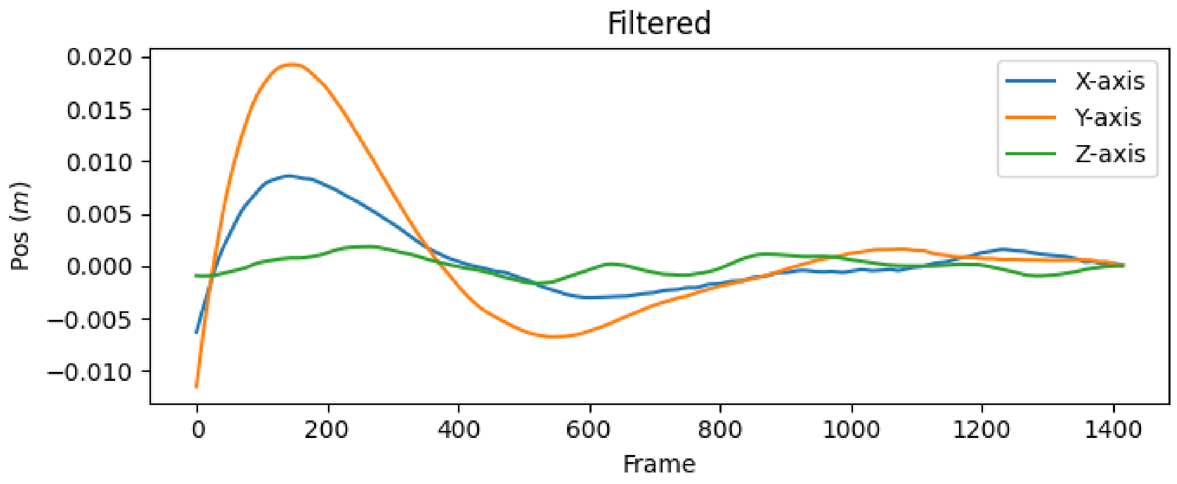
In [ ]:
```
# Exercise 1.2.1: Plot 1D time series for acceleration, velocity, and position
# in each scenario (i.e. with/without filtering)
plot_time_series(filt_data.acc,'acc','Filtered')
plot_time_series(no_filt_data.acc,'acc','Not Filtered')
plot_time_series(filt_data.vel,'vel','Filtered')
plot_time_series(no_filt_data.vel,'vel','Not Filtered')
plot_time_series(filt_data.pos,'pos','Filtered')
plot_time_series(no_filt_data.pos,'pos','Not Filtered')
```

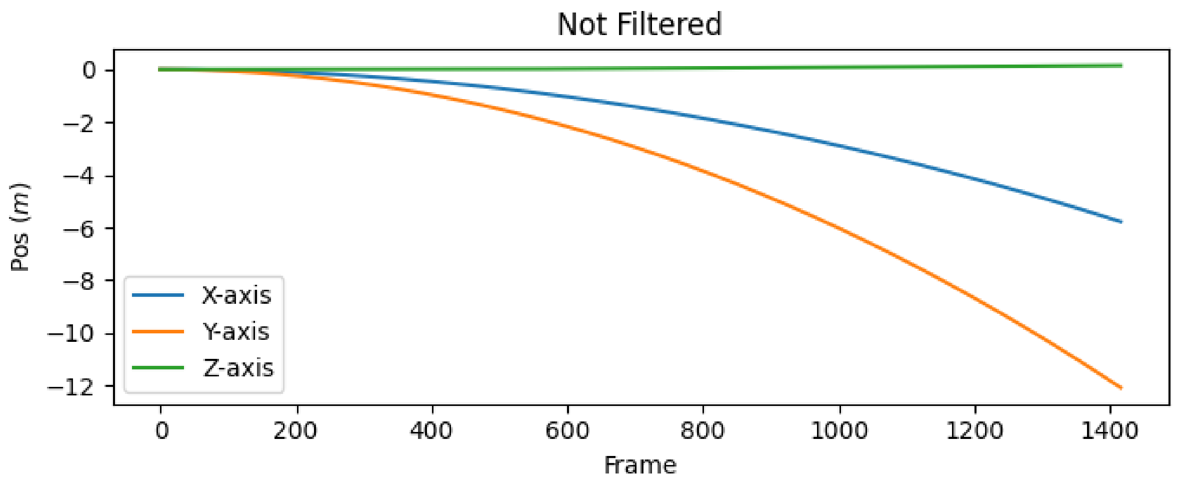




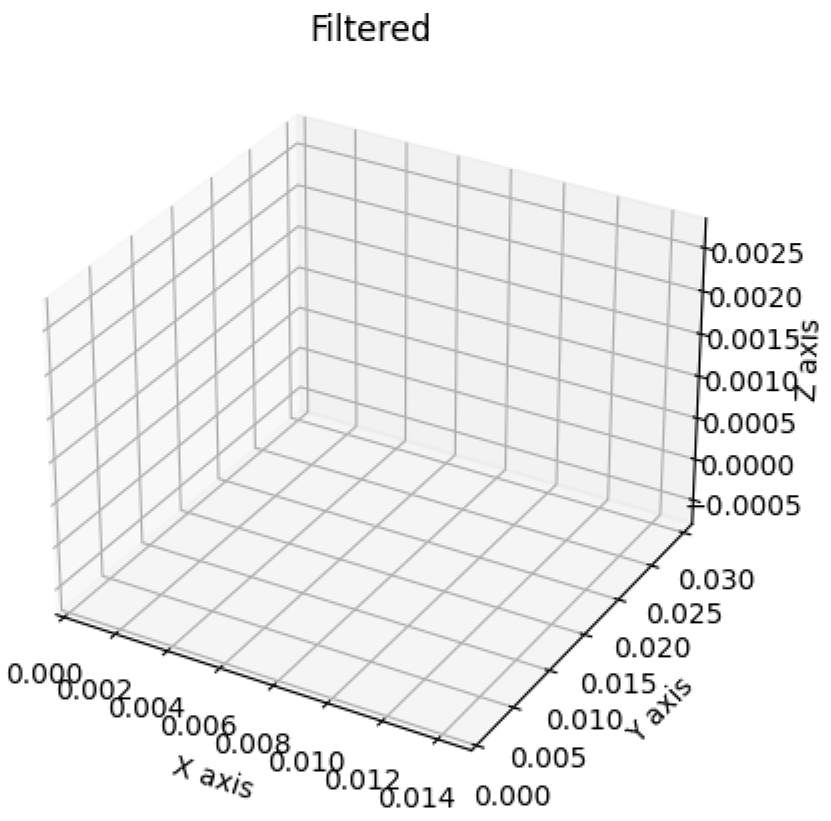

Filtered
Vel (m/s)
Frame
X-axis
Y-axis
Z-axis

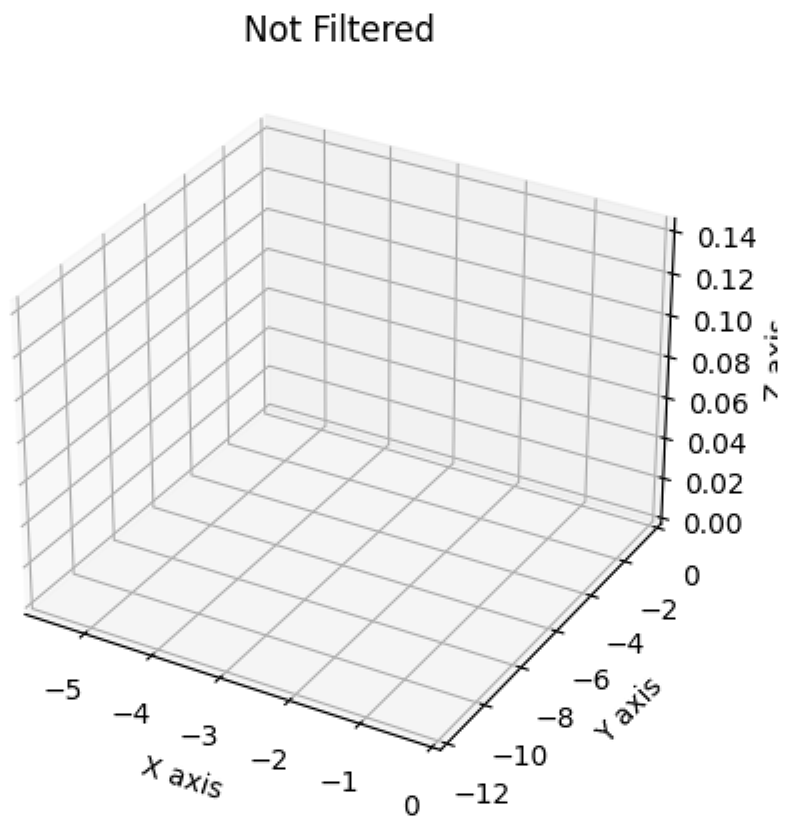

Not Filtered
Vel (m/s)
Frame
X-axis
Y-axis
Z-axis


Filtered
Pos (m)
Frame
X-axis
Y-axis
Z-axis

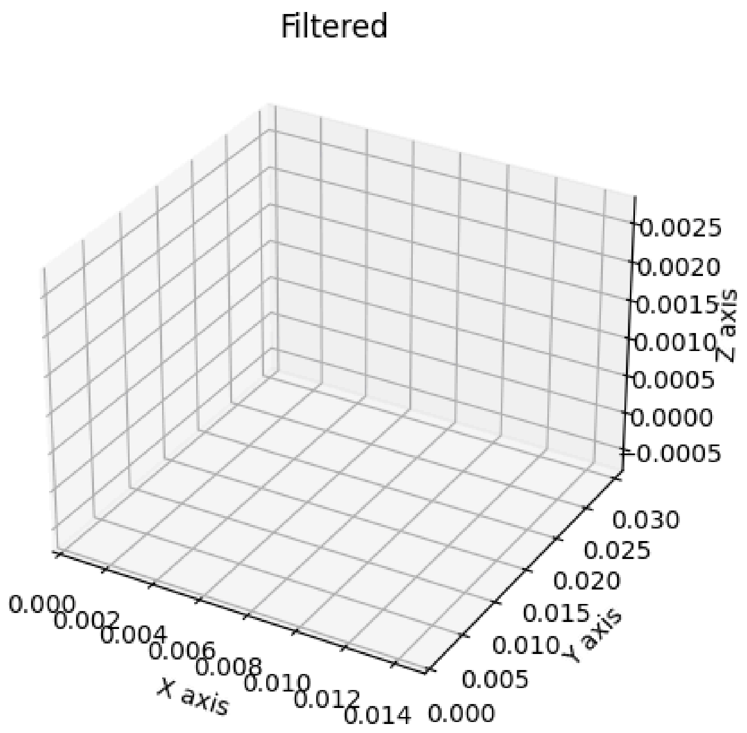


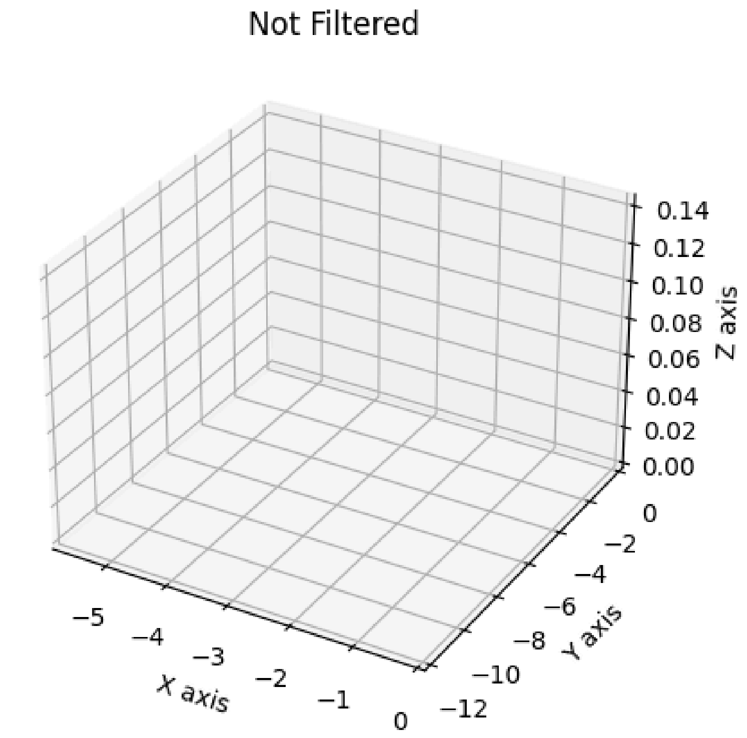
```
In [ ]: # Exercise 1.2.2: Animate the position time-series
        # obtained with filtering and without filtering
        filt_anim = animate_trajectory(filt_data.pos,'Filtered')
        no_filt_anim = animate_trajectory(no_filt_data.pos,'Not Filtered')
        filt_anim.save('filt.gif')
        no_filt_anim.save('no_filt.gif')
```





I was unable to get the animations to show inline, but I was able to save them and add them to a new markdown block


Filtered
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
−0.0005
Z axis
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Y axis
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
X axis


Not Filtered
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Z axis
0
−2
−4
−6
−8
−10
−12
Y axis
−5
−4
−3
−2
−1
0
X axis

Task 2

In this task, an 9-axis inertial measurement unit (referred to as *sensor*) was moved along a rectangular trajectory within 3D space (i.e., the trajectory discussed in class). The sensor captured the three-axis accelerometer, three-axis gyroscope, and three-axis magnetometer time-series. Moreover, a reflective marker was also placed on the sensor to capture the three-axis position time-series using a motion caption system (referred to as *mocap*).

Note that we have used *hand clapping* to synchronize the mocap and sensor data.

You can find the data for Task 2 in the "data" folder. The sensor data were saved in `task_2_sensor_data.csv`, whereas the mocap data were saved in `task3_mocap_data.tsv`. You may use `load_sensor_from_csv` and `load_mocap_from_tsv` functions provided in `data_utils.py` to load the data.

You can access the attributes of mocap data by using the following code

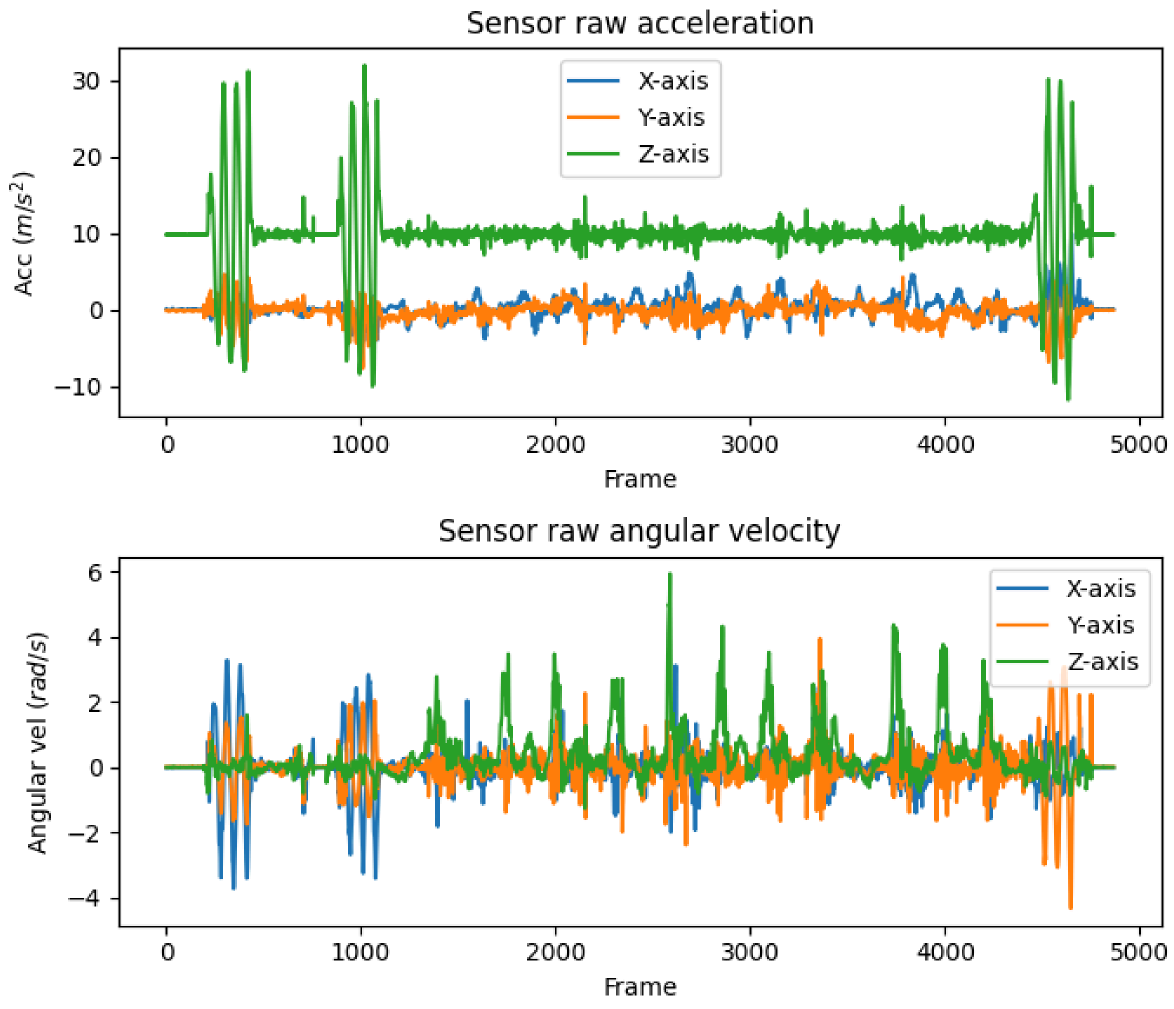
```
sample_rate = mocap_data.sample_rate # sampling rate of the mocap system, 150Hz
raw_pos = mocap_data.raw_pos # position, unit: m
```

Please note that the data in the files are raw and unfiltered.

In [ ]:
```
# Load sensor data
task2_sensor_file = os.path.join('data', 'task2_sensor_data.csv')
task2_sensor_data = load_sensor_from_csv(task2_sensor_file)

# Load mocap data
task2_mocap_file = os.path.join('data', 'task2_mocap_data.tsv')
task2_mocap_data = load_mocap_from_tsv(task2_mocap_file)
```

Run the following code to visualize the raw acceleration and angular velocity captured by the sensor. Note that the data collector clapped multiple times before and after the data collection, resulting in large peaks in the data.

In [ ]:
```
# Plot the sensor's raw acceleration and angular velocity
n_rows = 2
n_cols = 1
row_sz = 3
col_sz = 7

%matplotlib inline
fig = plt.figure(figsize=(n_cols*col_sz, n_rows*row_sz))
ax = fig.add_subplot(211)
ax.plot(task2_sensor_data.acc)
ax.set_xlabel('Frame')
ax.set_ylabel('Acc ($m/s^2$)')
ax.set_title('Sensor raw acceleration')
ax.legend(['X-axis', 'Y-axis', 'Z-axis'])

ax = fig.add_subplot(212)
ax.plot(task2_sensor_data.gyr)
ax.set_xlabel('Frame')
ax.set_ylabel('Angular vel ($rad/s$)')
ax.set_title('Sensor raw angular velocity')
ax.legend(['X-axis', 'Y-axis', 'Z-axis'])
fig.tight_layout()
plt.show()
```



Task 2.1

1. Implement the `filtering_and_gradient` function provided below. In this function, use a 6th order Butterworth filter with a cut-off frequency of 8 Hz to low-pass filter the raw position captured by the mocap system. Compute the gradient of the filtered position to derive the velocity time-series, followed by the same low-pass filter. Repeat the process of taking gradient and low-pass filtering to the velocity times-series to acquire acceleration time-series.

2. Apply `filtering_and_gradient` to `task2_mocap_data`.

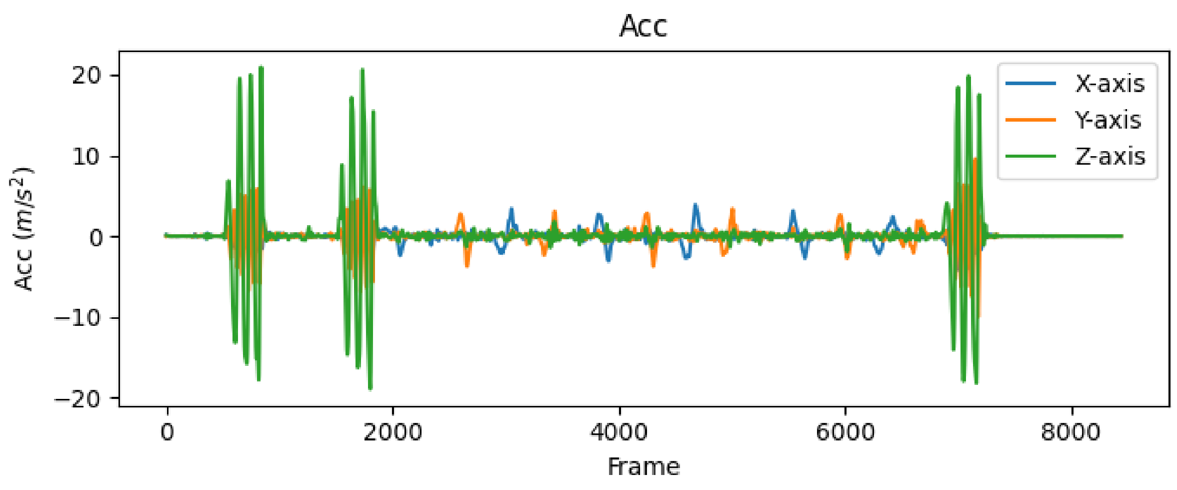
3. Apply `filtering_and_integrate` from Task 1 to `task2_sensor_data`.

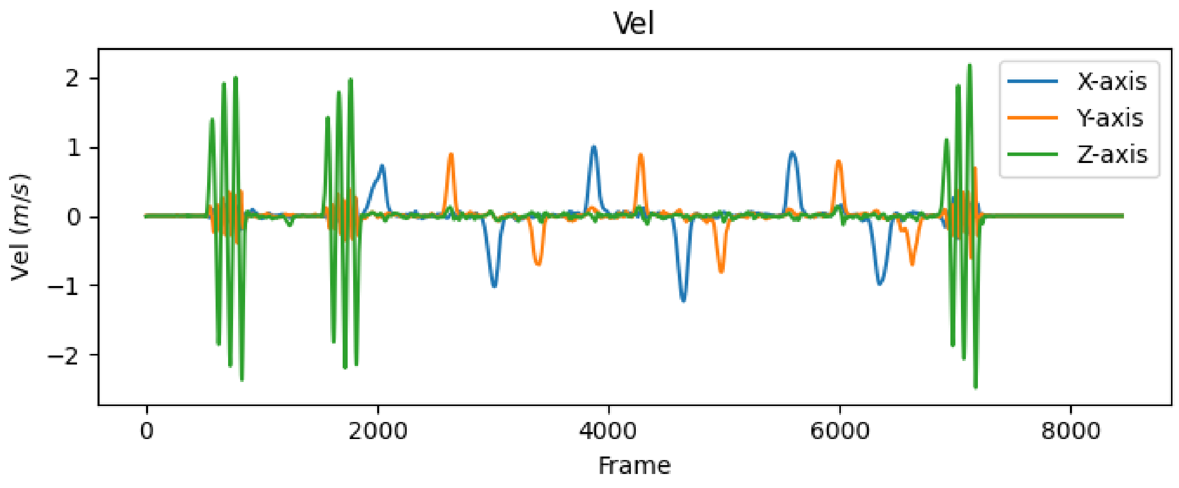
Hint

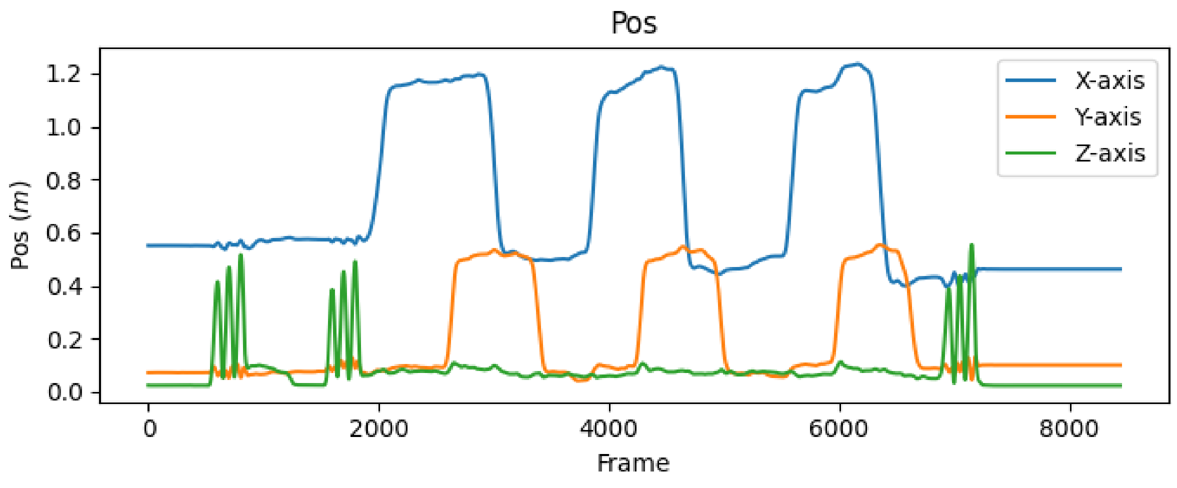
1. For gradient, you can use `numpy.gradient`. You are also free to use other integration implementations you are familiar with.

In [ ]:
```
# Exercise 2.1.1 Implement the `filtering_and_gradient` function provided below
def filtering_and_gradient(data):
    '''
    Input Parameters
    ----------
    data : mocap data object
        To access the position, use:
        pos = data.pos

    Output Parameters
    -----------
    data : mocap data object with filtered data
        data.pos: filtered position
        data.vel: filtered velocity
        data.acc: filtered position
    '''
    data = copy.deepcopy(data)

    # Get raw position
    pos = data.pos.copy()

    # Get the sampling rate of the sensor
    sample_rate = data.sample_rate

    ####################################################################
    # TODO: Define low-pass filter
    # apply the filters on the data
    # take the gradient of the position to get velocity and acceleration

    vel = np.zeros_like(pos)
    acc = np.zeros_like(vel)

    for i in range(3):
        cutoff = 8
        b, a = butter(6,cutoff/(sample_rate/2),btype='lowpass')
        pos[:,i] = filtfilt(b,a,pos[:,i])

        vel[:,i] = np.gradient(pos[:,i],1/sample_rate)

        b, a = butter(6,cutoff/(sample_rate/2),btype='lowpass')
        vel[:,i] = filtfilt(b,a,vel[:,i])

        acc[:,i] = np.gradient(vel[:,i],1/sample_rate)

        b, a = butter(6,cutoff/(sample_rate/2),btype='lowpass')
        acc[:,i] = filtfilt(b,a,acc[:,i])

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

    # Save filtered acceleration, velocity, and position
    data.pos = pos
    data.vel = vel
    data.acc = acc
    return data
```

In [ ]:
```
# Exercise 2.1.2: Apply `filtering_and_gradient` on task2_mocap_data
task2_mocap_data = filtering_and_gradient(task2_mocap_data)
plot_time_series(task2_mocap_data.acc,'acc','Acc')
plot_time_series(task2_mocap_data.vel,'vel','Vel')
plot_time_series(task2_mocap_data.pos,'pos','Pos')
```

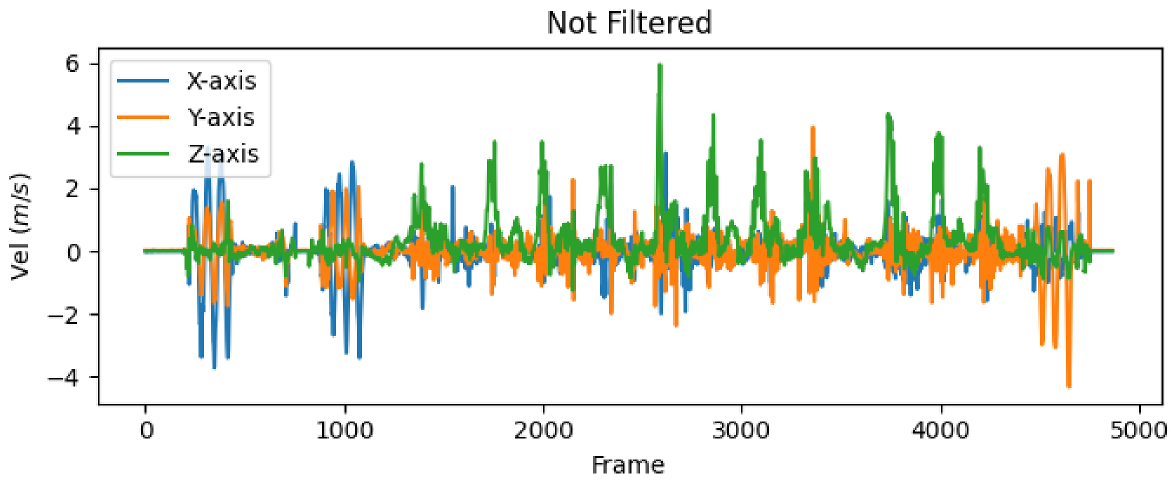


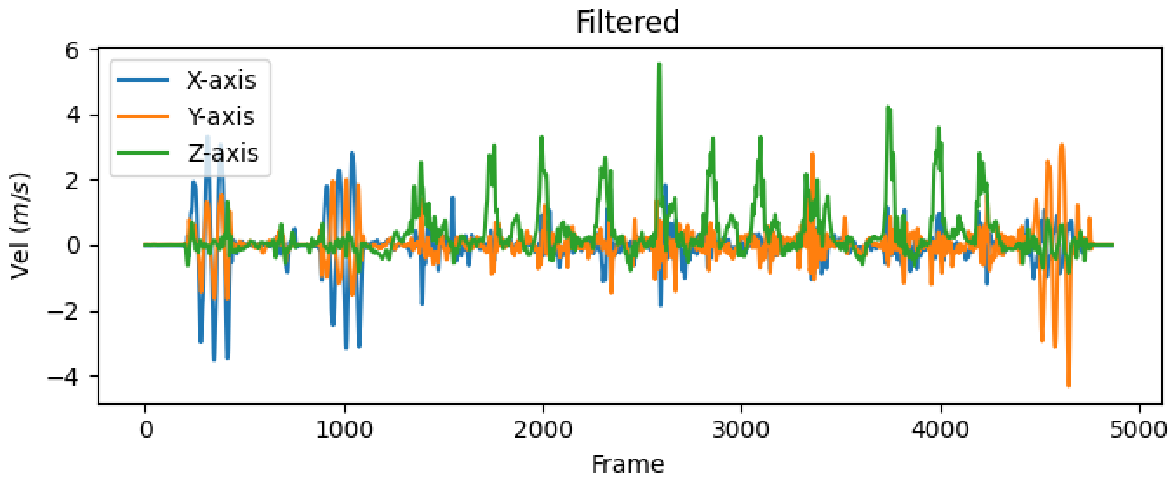




In [ ]:
```
# Exercise 2.1.3: Apply `filtering_and_integrate` on task2_sensor_data
plot_time_series(task2_sensor_data.gyr,'vel','Not Filtered')
```

```
task2_sensor_data = filtering_and_integrate(task2_sensor_data)
plot_time_series(task2_sensor_data.gyr,'vel','Filtered')
```





Task 2.2

In this task, you will delve into synchronizing data captured from disparate sensing systems. Variations in sampling rates or other factors often introduce discrepancies in the collected data. To facilitate synchronization, the data collector performed multiple claps while holding the sensor (which was attached with a mocap marker) before and after data collection. This resulted in large peaks in acceleration, as shown in our previous plots. Leveraging these claps, we can align different data by trimming segments between the claps and resampling to a uniform sampling rate.

1. Run the provided code to trim both sensor and mocap data.
2. Implement the `resampling_mocap` function. In the function, use `scipy.signal.resample` to resample the mocap acceleration, velocity, and position, in order to match the sampling rate of the sensor data. We opt to reduce the sampling rate of the mocap data to align with that of the sensor data, aiming to maintain data integrity (compared to upsampling the sensor data to match the mocap data, which entails data interpolation and may compromise integrity).

3. Apply `resampling_mocap` to `task2_mocap_data`.

In [ ]:
```
# Exercise 2.2.1: Run the provided code to trim both sensor and mocap data.
# The clapping indices were identified by manual inspection of the two data.
sensor_start, sensor_end = 1183, 4431 # task 2
task2_sensor_data = trim_data(task2_sensor_data, sensor_start, sensor_end, 'sensor'

mocap_start, mocap_end = 1981, 6812 # task 2
task2_mocap_data = trim_data(task2_mocap_data, mocap_start, mocap_end, 'mocap')
```

In [ ]:
```
# Exercise 2.2.2: Implement the `resampling_mocap` function provided below
def resampling_mocap(data, num):
    '''
    Input Parameters
    ----------
    data : mocap data object

    num : int, the number of samples in the resampled signal

    Output Parameters
    ----------
    data : mocap data object with resampled pos, vel and acc
    '''
    data = copy.deepcopy(data)

    ####################################################################
    # TODO: resample acceleration, velocity, and position

    data.acc = resample(data.acc,num)
    data.vel = resample(data.vel,num)
    data.pos = resample(data.pos,num)

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

    return data
```

In [ ]:
```
# Exercise 2.2.3: Apply `resampling_mocap` to `task2_mocap_data`.
print('Before Resample:')
print(f'Mocap:  {task2_mocap_data.acc.shape} {task2_mocap_data.vel.shape} {task2_mo
print(f'Sensor: {task2_sensor_data.acc.shape} {task2_sensor_data.vel.shape} {task2_
task2_mocap_data = resampling_mocap(task2_mocap_data, task2_sensor_data.acc.shape[0
print('After Resample')
print(f'Mocap:  {task2_mocap_data.acc.shape} {task2_mocap_data.vel.shape} {task2_mo
print(f'Sensor: {task2_sensor_data.acc.shape} {task2_sensor_data.vel.shape} {task2_
```

```
Before Resample:
Mocap:  (4831, 3) (4831, 3) (4831, 3)
Sensor: (3248, 3) (3248, 3) (3248, 3)
After Resample
Mocap:  (3248, 3) (3248, 3) (3248, 3)
Sensor: (3248, 3) (3248, 3) (3248, 3)
```

Task 2.3

This task will demonstrate the process of fusing acceleration and angular velocity data collected in the sensor's coordinate system to estimate the sensor's orientation in the global coordinate system. Once the orientation is determined, this information can be utilized to align or rotate the sensor to the global coordinate system.

1. Implement the `fuse_and_rotate` function. Firstly, select a method from `ahrs` to estimate the sensor's orientation. Secondly, utilize the estimated orientation to transform the acceleration and angular velocity from the sensor's coordinate system to the global coordinates. Thirdly, remove gravity from the vertical axis of the global coordinate acceleration. Finally, apply `filtering_and_integrate` from Task 1 to the transformed gravity-free, global-coordinate acceleration to derive velocity and position time-series in the global coordinates.
2. Apply `fuse_and_rotate` to `task2_sensor_data` to obtain acceleration, velocity, and position time-series in the global coordinates.
3. Utilize `compare_trajectory` to visualize the acceleration, velocity, and position from both the sensor and mocap systems in the global coordinates.

Hint

1. You can choose from the following methods provided by AHRS to estimate sensor orientation. The methods shown in the table below assume that the Z-axis of the global coordinate system aligns with gravity. Therefore, you can easily eliminate gravity from the transformed acceleration.

Fusion Method	Orientatio Representation	What to set inverse in `R.from_quat().apply()`
AQUA	global -> local	True
Madgwich	local -> global	False
Mahony	local -> global	False

Here's an example of sensor orientation estimation using `AQUA` from `ahrs`. Suppose you have acceleration data `acc` in m/s^2 and angular velocity data `gyr` in rad/s, sampled at a frequency of `sample_freq` Hz. To estimate the orientation, you can follow these steps:

```
ahrs_filter = AQUA(acc=acc, gyr=gyr, frequency=sample_freq) # define
sensor fusion method
orientation = ahrs_filter.Q # access the orientation
```

The `orientation` calculated by `AQUA` represents the global to local transformation, whereas for Madgwick and Mahony, it represents the local to global transformation. You need to be careful with this difference when using them for future coordinate transformation.

2. For coordinate transformation, you can use `scipy.spatial.transform.Rotation`. Assume the local acceleration captured in the sensor local frame is `local_acc` and the

`orientation` transforms the data from the sensor local frame to the global frame. To perform transformation,

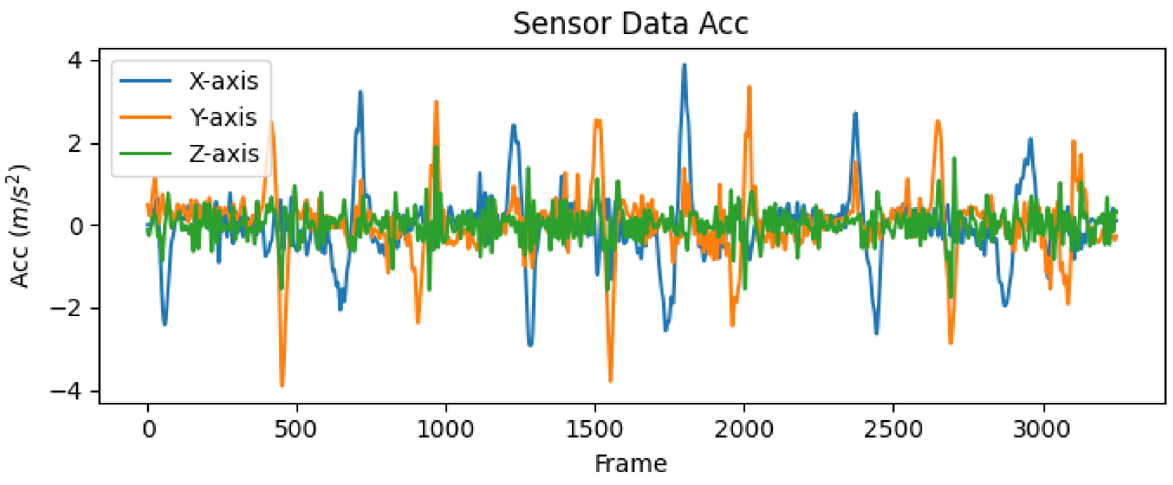
```
orientation = orientation[:, [1, 2, 3, 0]] # the orientation calculated
from ahrs has different form than what is required by R.from_quat

global_acc = R.from_quat(orientation).apply(local_acc, inverse=True) #
inverse needs to be set to `True` because the orientation calcuated from
AQUA represents the transformation from global coordinate system to the
sensor coordinate system. For Madgwich and Mahony methods, `inverse` needs
to be set to `False`.

global_acc[:, 2] -= GRAVITY_CONSTANT # remove gravity from Z-axis
```

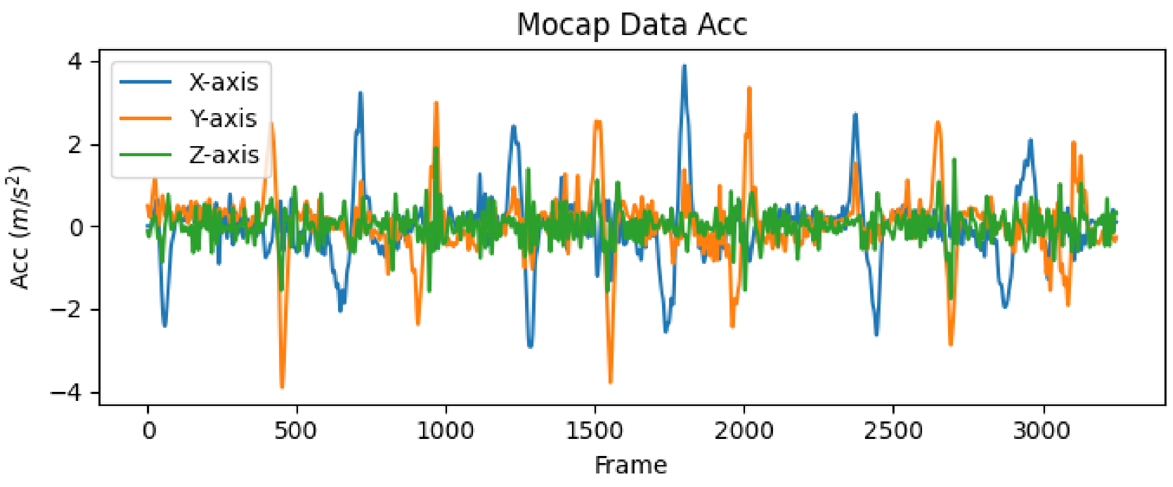
In the code above, if `inverse` is specificed `True` in `R.from_quat().apply()`, the inverse of the rotation(s) is applied to the input vectors. You can refer to the table above for guidance on how to set up this parameter.

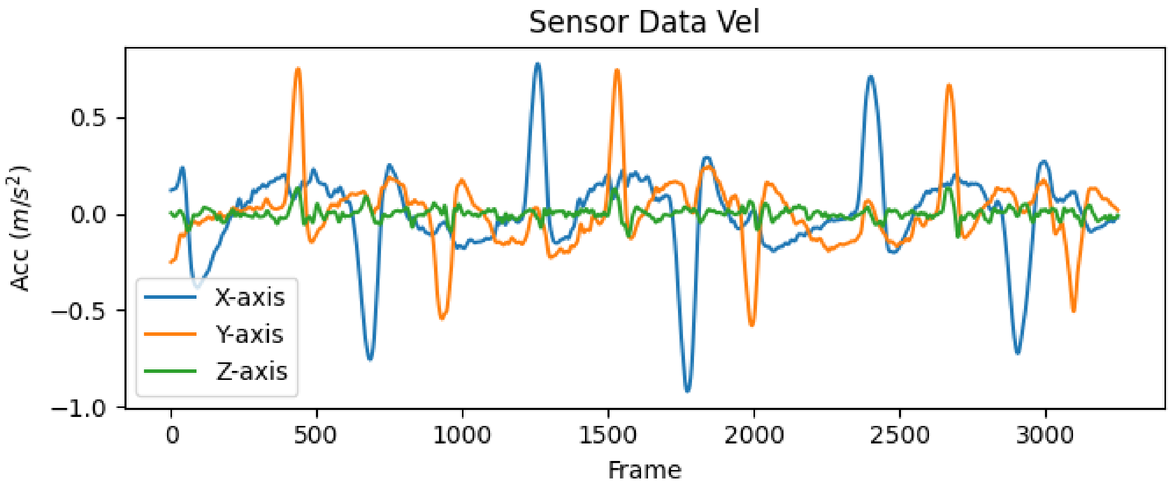
In [ ]:
```
# Exercise 2.3.1 Implement the `fuse_and_rotate` function
def fuse_and_rotate(local_data):
    '''
    Input Parameters
    ----------
    local_data : sensor data object in sensor's coordinates

    Output Parameters
    ----------
    global_data : transformed data object in global coordinates
        global_data.sample_rate: the synchornized sampling rate
        global_data.acc: global 3D accelerometer
        global_data.gyr: global 3D gyroscope
    '''
    local_data = copy.deepcopy(local_data)

    local_acc = local_data.acc
    local_gyr = local_data.gyr
    sample_rate = local_data.sample_rate

    ####################################################################
    # TODO: Calculate sensor orientation, then use orientation to
    # transform acceleration and angular velocity from sensor local
    # frame to global frame
    # Remove gravity from global acceleration

    ahrs = AQUA(acc=local_acc, gyr=local_gyr)
    orientation = ahrs.Q

    orientation = orientation[:, [1,2,3,0]]

    global_acc = R.from_quat(orientation).apply(local_acc, inverse=True)
    global_acc[:,2] -= GRAVITY_CONSTANT

    global_gyr = R.from_quat(orientation).apply(local_gyr, inverse=True)
    ####################################################################

    # Save the transformed data
    global_data = SimpleNamespace()
    global_data.sample_rate = local_data.sample_rate
    global_data.acc = global_acc
    global_data.gyr = global_gyr

    ####################################################################
    # TODO: Apply filtering_and_integrate to global data

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

    global_data = filtering_and_integrate(global_data)

    return global_data
```

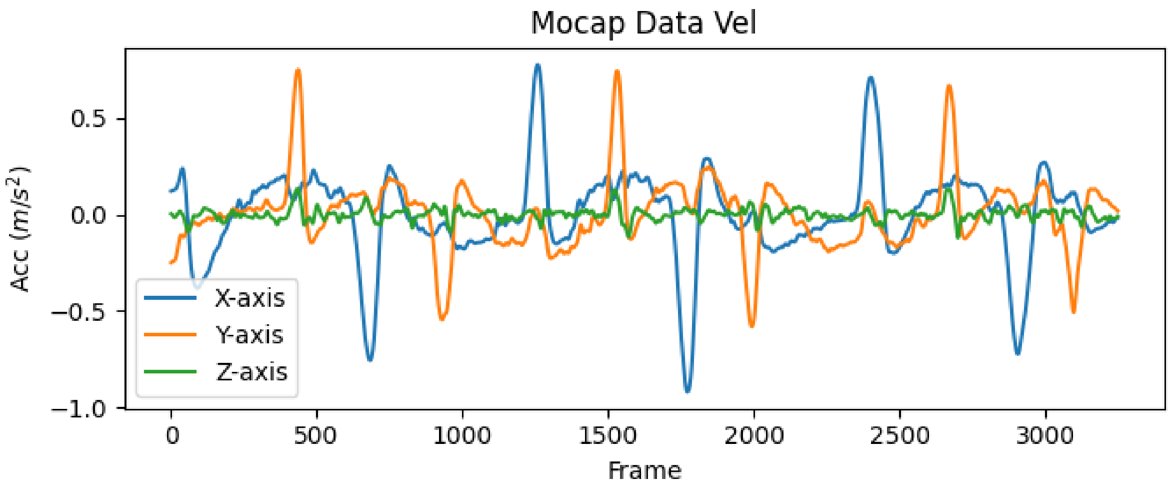
In [ ]:
```
# Exercise 2.3.2: Apply `fuse_and_rotate` on `task2_sensor_data`
task2_sensor_data_fuse = fuse_and_rotate(task2_sensor_data)
```

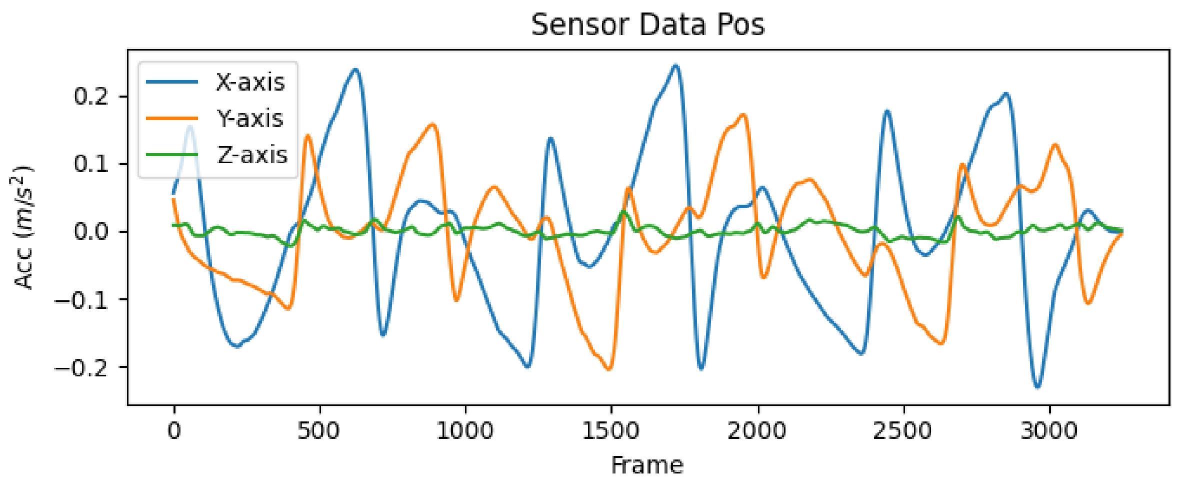
In [ ]:
```
# Exercise 2.3.3: Compare acceleration, velocity, and position
# from sensor fusion algorithm and mocap system
%matplotlib inline
plot_time_series(task2_sensor_data_fuse.acc, 'acc', 'Sensor Data Acc')
plot_time_series(task2_sensor_data_fuse.acc, 'acc', 'Mocap Data Acc')

plot_time_series(task2_sensor_data_fuse.vel, 'acc', 'Sensor Data Vel')
plot_time_series(task2_sensor_data_fuse.vel, 'acc', 'Mocap Data Vel')

plot_time_series(task2_sensor_data_fuse.pos, 'acc', 'Sensor Data Pos')
plot_time_series(task2_sensor_data_fuse.pos, 'acc', 'Mocap Data Pos')

plot_time_series(task2_sensor_data_fuse.gyr, 'vel', 'Sensor Gyr')
```

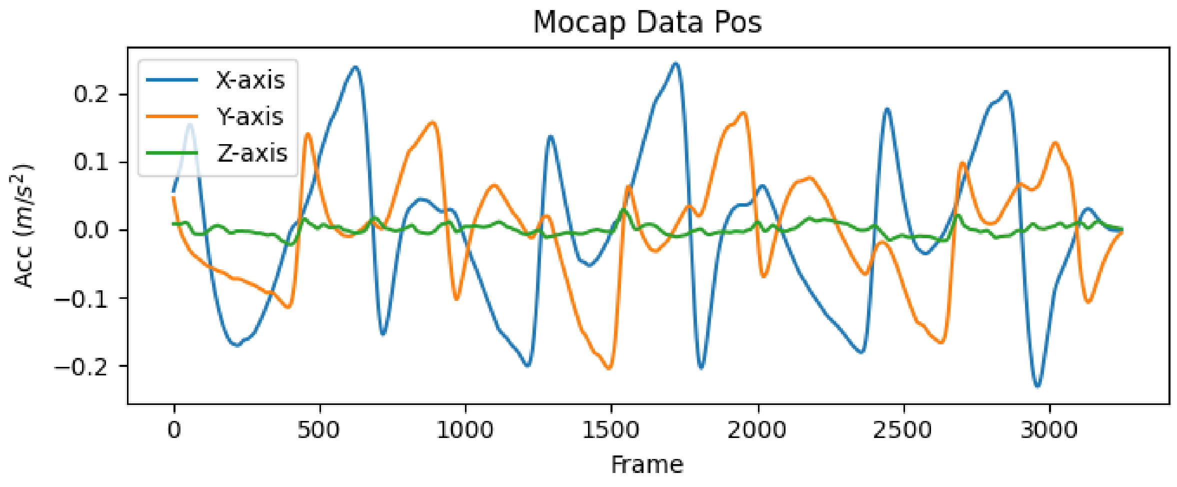


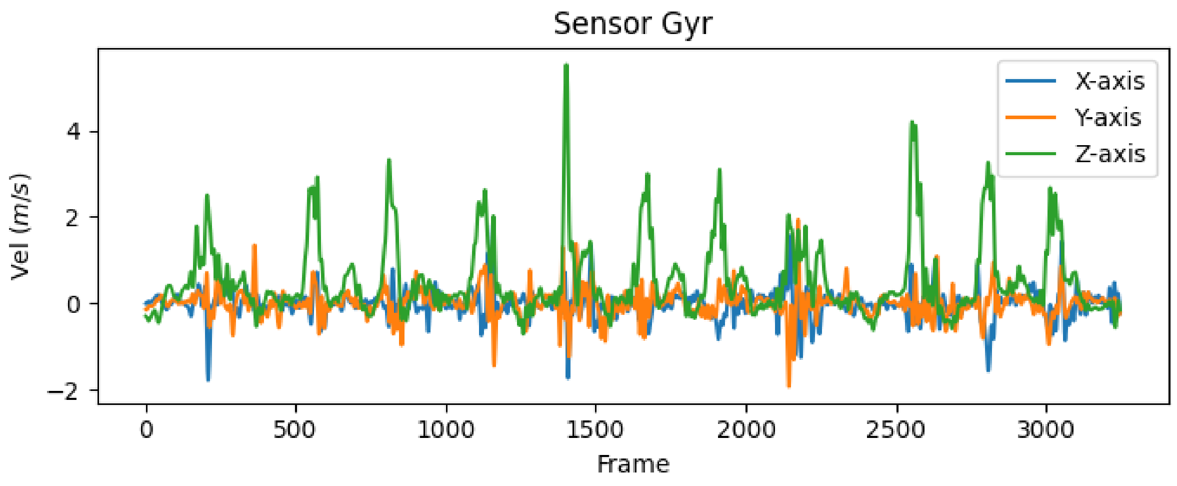

Mocap Data Acc
X-axis
Y-axis
Z-axis
Acc (m/s^2)
Frame
4
2
0
−2
−4
0
500
1000
1500
2000
2500
3000


Sensor Data Vel
X-axis
Y-axis
Z-axis
Acc (m/s^2)
Frame
0.5
0.0
−0.5
−1.0
0
500
1000
1500
2000
2500
3000


Mocap Data Vel
X-axis
Y-axis
Z-axis
Acc (m/s^2)
Frame
0.5
0.0
−0.5
−1.0
0
500
1000
1500
2000
2500
3000







Task 2.4

In this exercise, you will evaluate the performance of the `ahrs` algorithm both quantitatively and qualitatively.

1. Calculate the root mean square error (RMSE) and the normalized root mean square error (NRMSE) for acceleration, velocity, and position time-series obtained from your sensor fusion algorithm with the mocap data, respectively. NRMSE is obtained by

normalizing the RMSE to the range of mocap data, so that the error can be represented in percentage (%).

When you calculate RMSE and NRMSE for the position time-series, make sure that both position time-series start at 0 (for fair comparison).

This analysis will provide insights into the accuracy and reliability of your sensor fusion algorithm compared to the ground truth mocap data.

2. Create animations using `animate_trajectory` for the position time-series derived from the sensor fusion algorithm and captured by the mocap system, respectively. By visually comparing these animations, you can qualitatively assess how well the sensor fusion algorithm aligns with the mocap system in representing the sensor's movements.

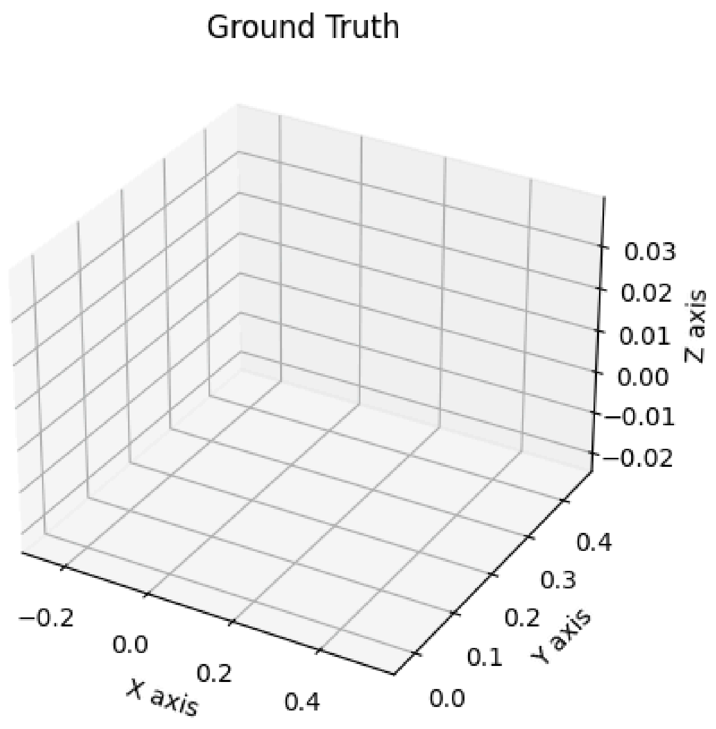
In [ ]:
```
# Exercise 2.4.1: Calculate RMSE and NRMSE for acceleration, velocity,
# and position time-series, respectively

task2_mocap_acc = np.copy(task2_mocap_data.acc)
task2_sensor_acc = np.copy(task2_sensor_data_fuse.acc)

task2_mocap_vel = np.copy(task2_mocap_data.vel)
task2_sensor_vel = np.copy(task2_sensor_data_fuse.vel)

task2_mocap_pos = np.copy(task2_mocap_data.pos)
task2_sensor_pos = np.copy(task2_sensor_data_fuse.pos)

for i in range(3):
    task2_mocap_acc[:,i] = task2_mocap_data.acc[:,i] - task2_mocap_data.acc[0,i]
    task2_sensor_acc[:,i] = task2_sensor_data_fuse.acc[:,i] - task2_sensor_data_fus
    task2_mocap_vel[:,i] = task2_mocap_data.vel[:,i] - task2_mocap_data.vel[0,i]
    task2_sensor_vel[:,i] = task2_sensor_data_fuse.vel[:,i] - task2_sensor_data_fus
    task2_mocap_pos[:,i] = task2_mocap_data.pos[:,i] - task2_mocap_data.pos[0,i]
    task2_sensor_pos[:,i] = task2_sensor_data_fuse.pos[:,i] - task2_sensor_data_fus

acc_rmse = np.sqrt(np.mean((task2_mocap_acc-task2_sensor_acc)**2))
vel_rmse = np.sqrt(np.mean((task2_mocap_vel-task2_sensor_vel)**2))
pos_rmse = np.sqrt(np.mean((task2_mocap_pos-task2_sensor_pos)**2))


print(f' RMSE of Acc: {acc_rmse}')
print(f'NRMSE of Acc: {acc_rmse/(np.max(task2_mocap_acc)-np.min(task2_mocap_acc))}'
print()
print(f' RMSE of Vel: {vel_rmse}')
print(f'NRMSE of Vel: {vel_rmse/(np.max(task2_mocap_vel)-np.min(task2_mocap_vel))}'
print()
print(f' RMSE of Pos: {pos_rmse}')
print(f'NRMSE of Pos: {pos_rmse/(np.max(task2_mocap_pos)-np.min(task2_mocap_pos))}'
 RMSE of Acc: 0.8573884919569196
NRMSE of Acc: 0.12051345476207738

 RMSE of Vel: 0.2896366283533415
NRMSE of Vel: 0.11560012118220273

 RMSE of Pos: 0.27063940787744417
NRMSE of Pos: 0.32353470845033966
```

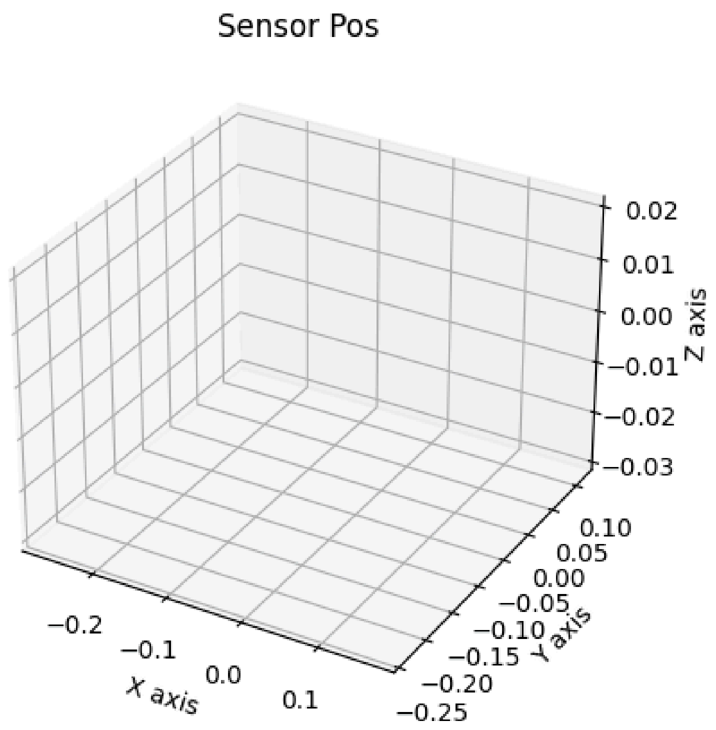
In [ ]:

```
# Exercise 2.4.2: Animate the position time-series derived from the motion capture
%matplotlib notebook
ani = animate_trajectory(task2_mocap_data.pos, 'Ground Truth')
ani.save('task2_mocap.gif')
```

In [ ]:

```
# Exercise 2.4.3: Animate the position time-series derived from the sensor fusion a
%matplotlib notebook
ani = animate_trajectory(task2_sensor_data_fuse.pos, 'Sensor Pos ')
ani.save('task2_sensor.gif')
```


Ground Truth
0.03
0.02
0.01
0.00
−0.01
−0.02
Z axis
−0.2
0.0
0.2
0.4
X axis
0.0
0.1
0.2
0.3
0.4
Y axis


Sensor Pos
0.02
0.01
0.00
−0.01
−0.02
−0.03
Z axis
−0.2
−0.1
0.0
0.1
X axis
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
Y axis

Task 2.5

In this exercise, you will leverage the gravity-free acceleration data provided by the manufacturer Xsens, known for its accuracy in fusing multi-modal data to estimate sensor orientation and obtain acceleration in global coordinates. This will enable you to evaluate the manufacturer's algorithm performance and compare it with mocap data. Note that the sensor fusion algorithm by Xsens uses all nine-axis inertial data (accelerometer + gyroscope + magnetometer) compared to the AQUA algorithm that uses only six-axis data (accelerometer + gyroscope).

1. Apply the `filtering_and_integrate` function to the manufacturer gravity-free acceleration data to derive velocity and position time-series.

2. Utilize `compare_trajectory` to visualize the acceleration, velocity, and position from both the manufacturer's gravity-free data and mocap systems in the global coordinates.

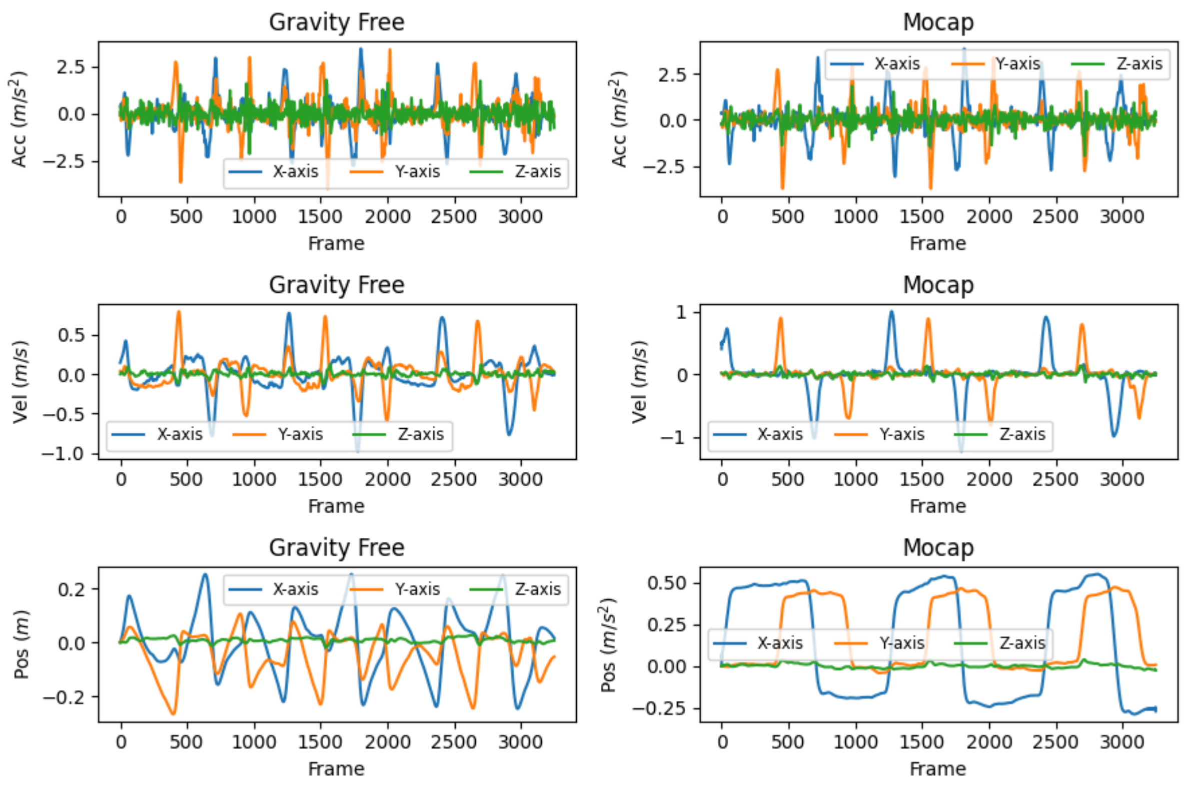
3. Calculate the RMSE and NRMSE between the acceleration, velocity, and position time-series derived from the manufacturer gravity-free acceleration and the mocap system, respectively.

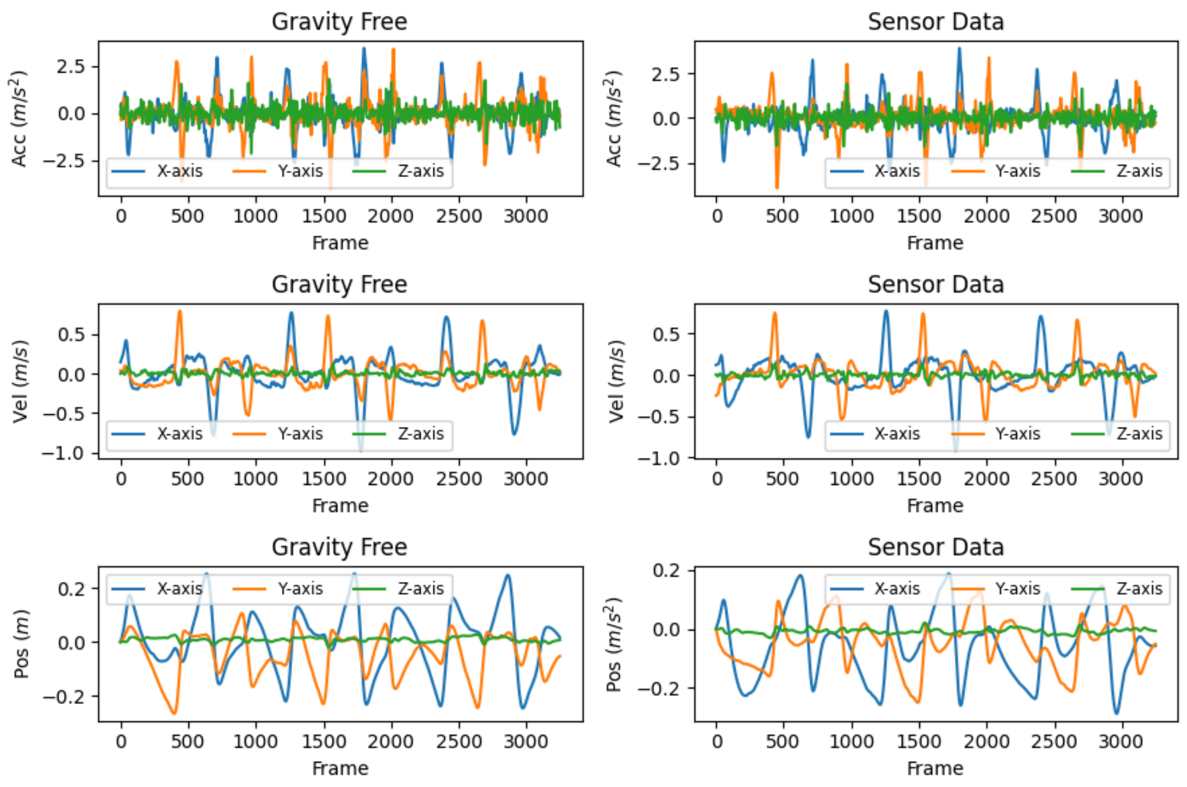
4. Generate animations for the position time-series derived from the gravity-free acceleration provided by the manufacturer.

In [ ]:
```
task2_gravity_free_data = SimpleNamespace()
task2_gravity_free_data.sample_rate = task2_sensor_data.sample_rate
task2_gravity_free_data.acc = task2_sensor_data.free_acc
task2_gravity_free_data.gyr = task2_sensor_data.gyr
```

In [ ]:
```
# Exercise 2.5.1: Apply the `filtering_and_integrate` function
# to the manufacturer gravity-free acceleration data
task2_gravity_free_data = filtering_and_integrate(task2_gravity_free_data)
```

In [ ]:
```
# Exercise 2.5.2: Compare acceleration, velocity, and position
# from manufecturer and mocap system
%matplotlib inline
compare_trajectory(task2_gravity_free_data,task2_mocap_data,'Gravity Free','Mocap')
compare_trajectory(task2_gravity_free_data,task2_sensor_data_fuse,'Gravity Free','S
```





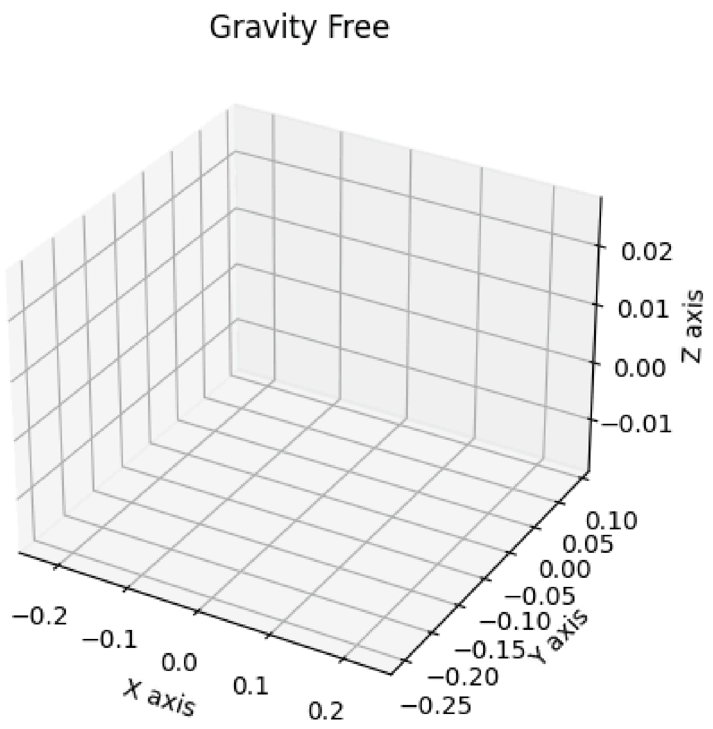
```
In [ ]: # Exercise 2.5.3: Calculate RMSE and NRMSE for acceleration, velocity,
        # and position time-series, respectively
        task2_mocap_acc = np.copy(task2_mocap_data.acc)
        task2_sensor_acc = np.copy(task2_gravity_free_data.acc)

        task2_mocap_vel = np.copy(task2_mocap_data.vel)
        task2_sensor_vel = np.copy(task2_gravity_free_data.vel)
```

```
task2_mocap_pos = np.copy(task2_mocap_data.pos)
task2_sensor_pos = np.copy(task2_gravity_free_data.pos)

for i in range(3):
    task2_mocap_acc[:,i] = task2_mocap_data.acc[:,i] - task2_mocap_data.acc[0,i]
    task2_sensor_acc[:,i] = task2_gravity_free_data.acc[:,i] - task2_gravity_free_d
    task2_mocap_vel[:,i] = task2_mocap_data.vel[:,i] - task2_mocap_data.vel[0,i]
    task2_sensor_vel[:,i] = task2_gravity_free_data.vel[:,i] - task2_gravity_free_d
    task2_mocap_pos[:,i] = task2_mocap_data.pos[:,i] - task2_mocap_data.pos[0,i]
    task2_sensor_pos[:,i] = task2_gravity_free_data.pos[:,i] - task2_gravity_free_d

acc_rmse = np.sqrt(np.mean((task2_mocap_acc-task2_sensor_acc)**2))
vel_rmse = np.sqrt(np.mean((task2_mocap_vel-task2_sensor_vel)**2))
pos_rmse = np.sqrt(np.mean((task2_mocap_pos-task2_sensor_pos)**2))


print(f' RMSE of Acc: {acc_rmse}')
print(f'NRMSE of Acc: {acc_rmse/(np.max(task2_mocap_acc)-np.min(task2_mocap_acc))}'
print()
print(f' RMSE of Vel: {vel_rmse}')
print(f'NRMSE of Vel: {vel_rmse/(np.max(task2_mocap_vel)-np.min(task2_mocap_vel))}'
print()
print(f' RMSE of Pos: {pos_rmse}')
print(f'NRMSE of Pos: {pos_rmse/(np.max(task2_mocap_pos)-np.min(task2_mocap_pos))}'
```

```
 RMSE of Acc: 0.7199661290253059
NRMSE of Acc: 0.10119753919542789

 RMSE of Vel: 0.21789808305523037
NRMSE of Vel: 0.08696774627491172

 RMSE of Pos: 0.25699942239422296
NRMSE of Pos: 0.30722884685689716
```

In [ ]:

```
# Exercise 2.5.4: Animate the position time-series derived from manufecturer data
%matplotlib notebook
ani = animate_trajectory(task2_gravity_free_data.pos,'Gravity Free')
ani.save('gravity_free.gif')
```



Task 3

You can find the data for Task 3 in the "data" folder. In this task, a nine-axis IMU and a mocap marker was placed on a human subject's wrist, who performed some arbitrary, patternless, and random upper-limb movements (e.g., swinging the arm in the air randomly). The sensor data were saved in `task_3_sensor_data.csv`, and the mocap data were saved in `task3_mocap_data.tsv`.

In [ ]:
```
# Load sensor data
task3_sensor_file = os.path.join('data', 'task3_sensor_data.csv')
task3_sensor_data = load_sensor_from_csv(task3_sensor_file)


# Load mocap data
task3_mocap_file = os.path.join('data', 'task3_mocap_data.tsv')
task3_mocap_data = load_mocap_from_tsv(task3_mocap_file)
```

The objective of Task 3 is to apply the same procedures (Task 2.1 - 2.5) to this dataset, aiming to assess the algorithm's capability to handle entirely unpredictable movements that simulate daily scenarios.

1. Apply `filtering_and_gradient` to `task3_mocap_data`.
2. Apply `filtering_and_integrate` to `task3_sensor_data` with filtering.

3. Synchronization and resampling. Trim the sensor data and mocap between the provided clapping indices. Then, apply `resampling_mocap` to resample to the mocap data to match the sampling rate of sensor data.
4. Apply `fuse_and_rotate` on the sensor data.
5. Utilize `compare_trajectory` to visualize the acceleration, velocity, and position from both the sensor and mocap systems in the global coordinates.
6. Calculate the RMSE and NRMSE between the acceleration, velocity, and position time-series derived from the `ahrs` and the mocap system, respectively.
7. Create animations using `animate_trajectory` for the position time-series derived from the sensor fusion algorithm and captured by the mocap system, respectively.
8. Repeat Task 2.5 with data provided by the manufecturer in `task3_sensor_data`.

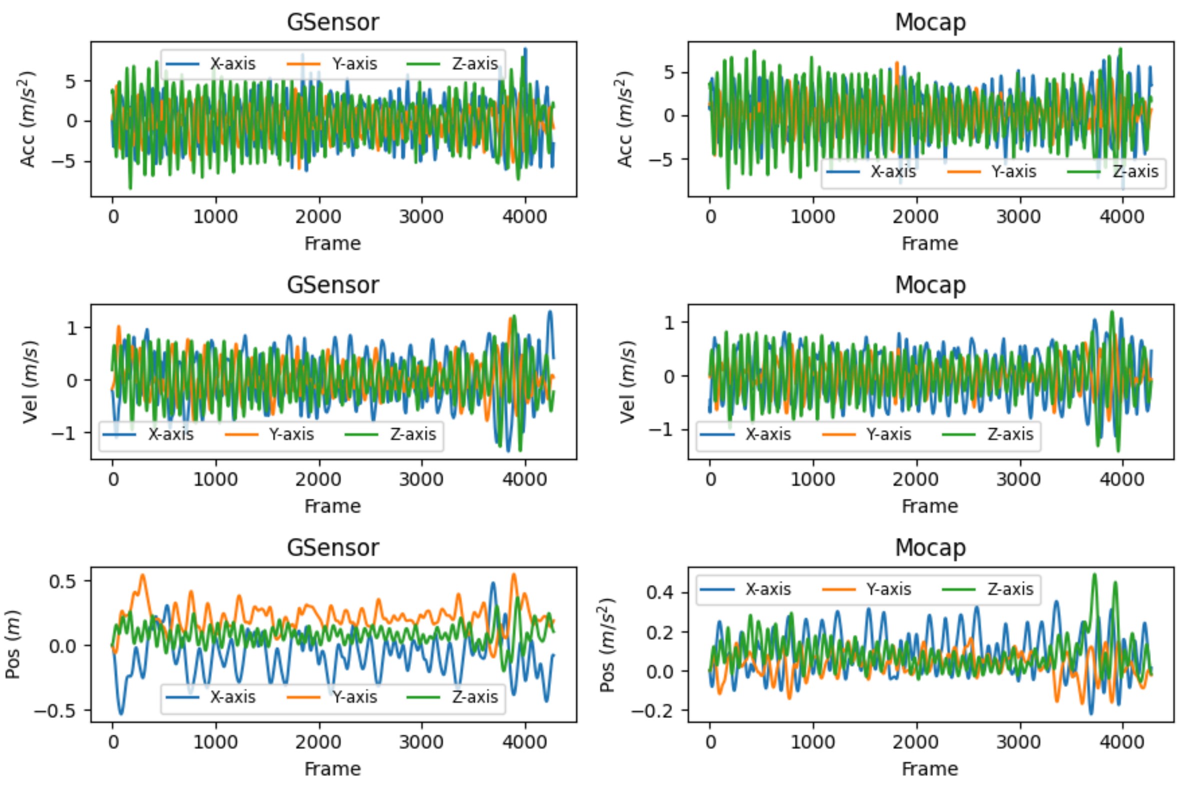
In [ ]:
```
# Exercise 3.1: Apply `filtering_and_gradient` to `task3_mocap_data`
task3_mocap_data = filtering_and_gradient(task3_mocap_data)
```

In [ ]:
```
# Exercise 3.2: Apply `filtering_and_integrate` to `task3_sensor_data` with filteri
task3_sensor_data = filtering_and_integrate(task3_sensor_data)
```

In [ ]:
```
# Exercise 3.3: Synchronization and resampling
sensor_start, sensor_end = 918, 5198 # task 3
task3_sensor_data = trim_data(task3_sensor_data, sensor_start, sensor_end, 'sensor'

mocap_start, mocap_end = 1300, 7719 # task 3
task3_mocap_data = trim_data(task3_mocap_data, mocap_start, mocap_end, 'mocap')

# TODO: Apply `resampling_mocap` to resample to the mocap data
task3_mocap_data = resampling_mocap(task3_mocap_data, task3_sensor_data.acc.shape[0
```

In [ ]:
```
# Exercise 3.4: Apply `fuse_and_rotate` on the sensor data.
task3_sensor_data = fuse_and_rotate(task3_sensor_data)
```

In [ ]:
```
# Exercise 3.5: Compare acceleration, velocity, and position from sensor fusion alg
# and mocap system
%matplotlib inline
compare_trajectory(task3_sensor_data,task3_mocap_data,'GSensor','Mocap')
```



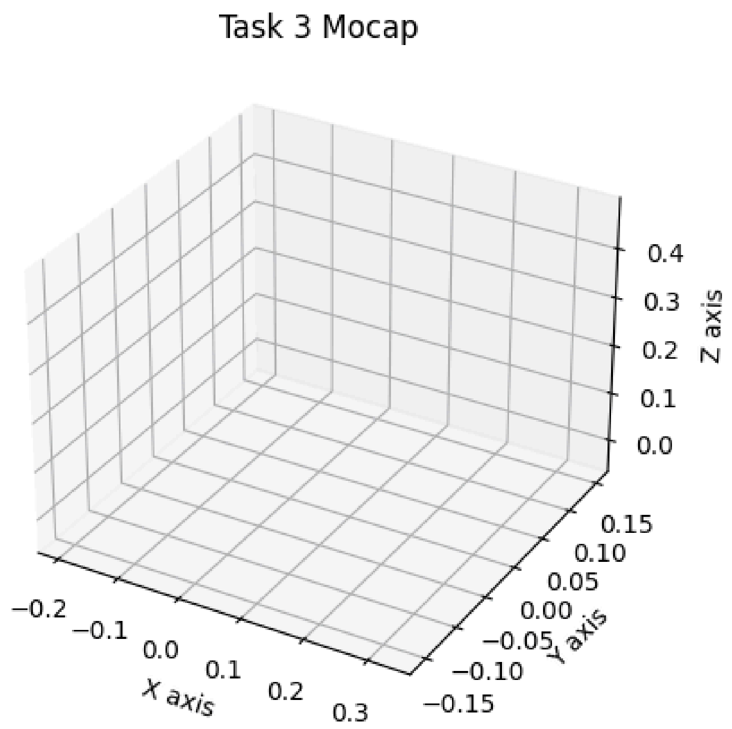
```
In [ ]: # Exercise 3.6: Calculate RMSE and NRMSE obtained from your sensor fusion algorithm
task3_mocap_acc = np.copy(task3_mocap_data.acc)
task3_sensor_acc = np.copy(task3_sensor_data.acc)

task3_mocap_vel = np.copy(task3_mocap_data.vel)
task3_sensor_vel = np.copy(task3_sensor_data.vel)

task3_mocap_pos = np.copy(task3_mocap_data.pos)
task3_sensor_pos = np.copy(task3_sensor_data.pos)

for i in range(3):
    task3_mocap_acc[:,i] = task3_mocap_data.acc[:,i] - task3_mocap_data.acc[0,i]
    task3_sensor_acc[:,i] = task3_sensor_data.acc[:,i] - task3_sensor_data.acc[0,i]
    task3_mocap_vel[:,i] = task3_mocap_data.vel[:,i] - task3_mocap_data.vel[0,i]
    task3_sensor_vel[:,i] = task3_sensor_data.vel[:,i] - task3_sensor_data.vel[0,i]
    task3_mocap_pos[:,i] = task3_mocap_data.pos[:,i] - task3_mocap_data.pos[0,i]
    task3_sensor_pos[:,i] = task3_sensor_data.pos[:,i] - task3_sensor_data.pos[0,i]

acc_rmse = np.sqrt(np.mean((task3_mocap_acc-task3_sensor_acc)**2))
vel_rmse = np.sqrt(np.mean((task3_mocap_vel-task3_sensor_vel)**2))
pos_rmse = np.sqrt(np.mean((task3_mocap_pos-task3_sensor_pos)**2))


print(f' RMSE of Acc: {acc_rmse}')
print(f'NRMSE of Acc: {acc_rmse/(np.max(task3_mocap_acc)-np.min(task3_mocap_acc))}'
print()
print(f' RMSE of Vel: {vel_rmse}')
print(f'NRMSE of Vel: {vel_rmse/(np.max(task3_mocap_vel)-np.min(task3_mocap_vel))}'
print()
print(f' RMSE of Pos: {pos_rmse}')
print(f'NRMSE of Pos: {pos_rmse/(np.max(task3_mocap_pos)-np.min(task3_mocap_pos))}'
 RMSE of Acc: 3.6253203416917423
NRMSE of Acc: 0.20747181367384773

 RMSE of Vel: 0.6693174192739818
NRMSE of Vel: 0.226146231433866

 RMSE of Pos: 0.22491026643013856
NRMSE of Pos: 0.3156269977763123
```

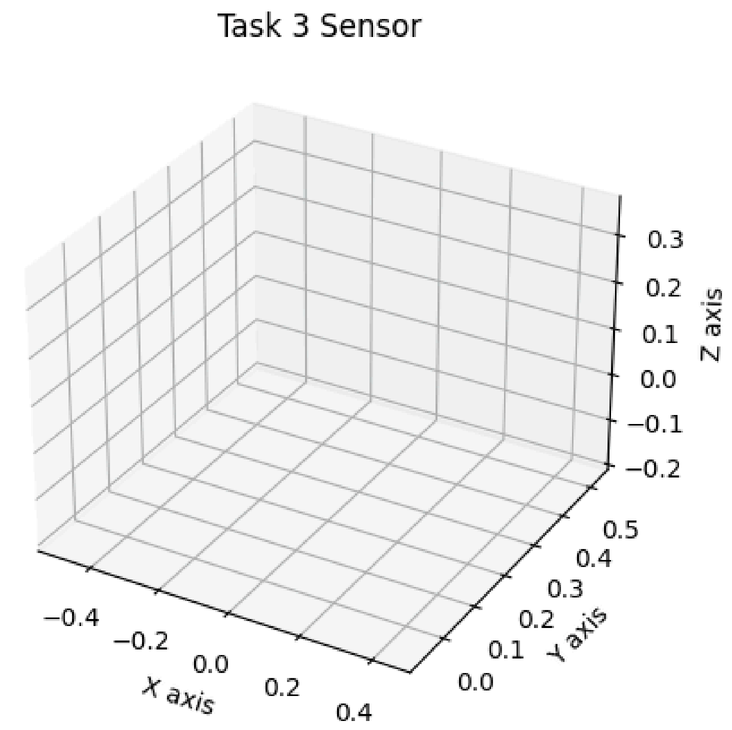
In [ ]:

```
# Exercise 3.7: Animate the position time-series captured by the mocap system
%matplotlib notebook
ani = animate_trajectory(task3_mocap_data.pos,'Task 3 Mocap')
ani.save('task3_mocap.gif')
```

In [ ]:

```
# Exercise 3.7: Animate the position time-series derived from the sensor fusion alg
%matplotlib notebook
ani = animate_trajectory(task3_sensor_data.pos,'Task 3 Sensor')
ani.save('task3_sensor.gif')
```



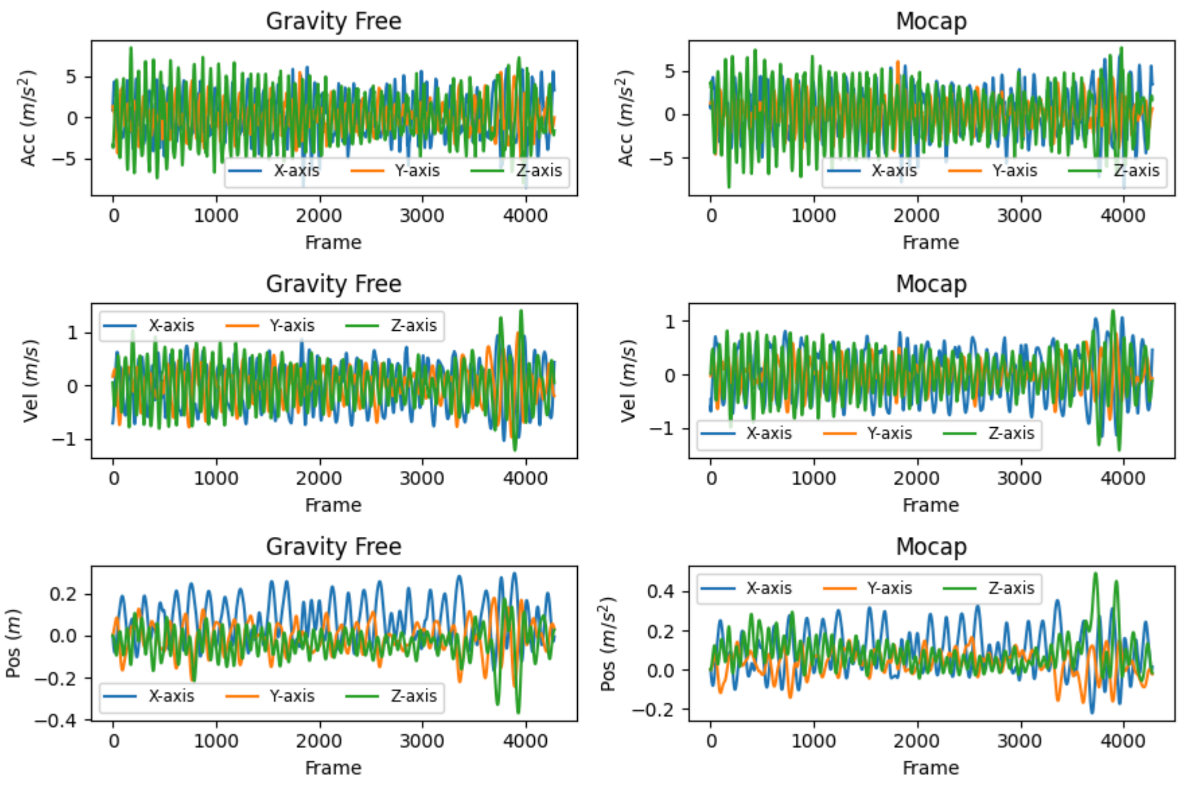


```
In [ ]: task3_sensor_data = load_sensor_from_csv(task3_sensor_file)

        task3_gravity_free_data = SimpleNamespace()
```

```
task3_gravity_free_data.sample_rate = task3_sensor_data.sample_rate
task3_gravity_free_data.acc = task3_sensor_data.free_acc
task3_gravity_free_data.gyr = task3_sensor_data.gyr
```

In [ ]:
```
# Exercise 3.8: Repeat Task 2.5 with data provided by the manufecturer in `task3_se
# TODO: apply filtering_and_integrate on the manufecturer data
task3_gravity_free_data = filtering_and_integrate(task3_gravity_free_data)
task3_gravity_free_data.free_acc = task3_sensor_data.free_acc

sensor_start, sensor_end = 918, 5198 # task 3
task3_gravity_free_data = trim_data(task3_gravity_free_data, sensor_start, sensor_e
```

In [ ]:
```
# TODO: Compare acceleration, velocity, and position
# from manufecturer and mocap system
%matplotlib inline
compare_trajectory(task3_gravity_free_data,task3_mocap_data,'Gravity Free', 'Mocap'
```

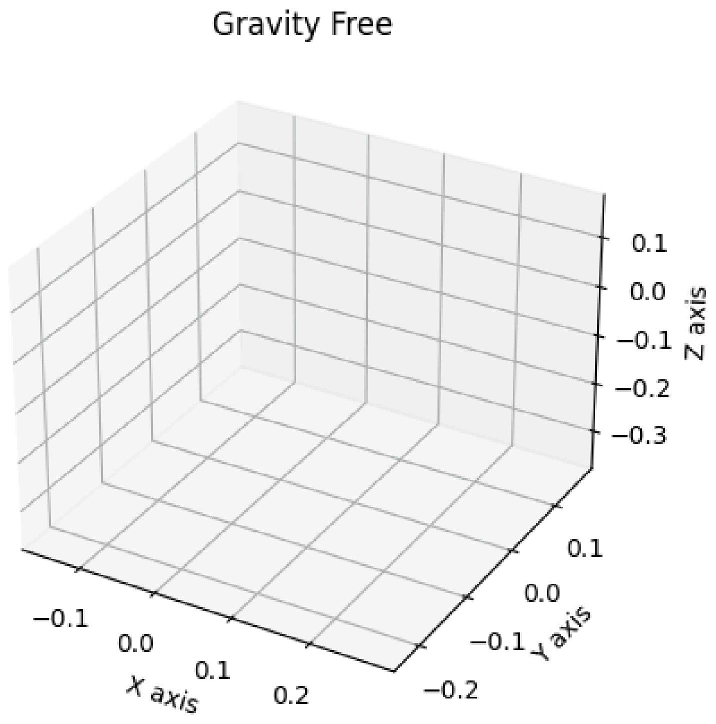


In [ ]:
```
# TODO: Calculate RMSE and NRMSE for acceleration, velocity,
# and position time-series, respectively

task3_mocap_acc = np.copy(task3_mocap_data.acc)
task3_sensor_acc = np.copy(task3_gravity_free_data.acc)

task3_mocap_vel = np.copy(task3_mocap_data.vel)
task3_sensor_vel = np.copy(task3_gravity_free_data.vel)

task3_mocap_pos = np.copy(task3_mocap_data.pos)
task3_sensor_pos = np.copy(task3_gravity_free_data.pos)

for i in range(3):
    task3_mocap_acc[:,i] = task3_mocap_data.acc[:,i] - task3_mocap_data.acc[0,i]
    task3_sensor_acc[:,i] = task3_sensor_vel[:,i] - task3_gravity_free_data.acc[0,i
    task3_mocap_vel[:,i] = task3_mocap_data.vel[:,i] - task3_mocap_data.vel[0,i]
    task3_sensor_vel[:,i] = task3_sensor_vel[:,i] - task3_gravity_free_data.vel[0,i
    task3_mocap_pos[:,i] = task3_mocap_data.pos[:,i] - task3_mocap_data.pos[0,i]
    task3_sensor_pos[:,i] = task3_sensor_vel[:,i] - task3_gravity_free_data.pos[0,i

acc_rmse = np.sqrt(np.mean((task3_mocap_acc-task3_sensor_acc)**2))
vel_rmse = np.sqrt(np.mean((task3_mocap_vel-task3_sensor_vel)**2))
pos_rmse = np.sqrt(np.mean((task3_mocap_pos-task3_sensor_pos)**2))


print(f' RMSE of Acc: {acc_rmse}')
print(f'NRMSE of Acc: {acc_rmse/(np.max(task3_mocap_acc)-np.min(task3_mocap_acc))}'
print()
print(f' RMSE of Vel: {vel_rmse}')
print(f'NRMSE of Vel: {vel_rmse/(np.max(task3_mocap_vel)-np.min(task3_mocap_vel))}'
print()
print(f' RMSE of Pos: {pos_rmse}')
print(f'NRMSE of Pos: {pos_rmse/(np.max(task3_mocap_pos)-np.min(task3_mocap_pos))}'
```

```
 RMSE of Acc: 4.558757945365048
NRMSE of Acc: 0.26089109096041696

 RMSE of Vel: 0.5260588294997043
NRMSE of Vel: 0.1777426051945774

 RMSE of Pos: 0.5583460746895655
NRMSE of Pos: 0.7835529167771287
```

In [ ]:

```
# TODO: Animate the position time-series derived from manufecturer data
%matplotlib notebook
ani = animate_trajectory(task3_gravity_free_data.pos, 'Gravity Free')
ani.save('task3_gravity_free.gif')
```


Gravity Free
0.1
0.0
−0.1
−0.2
−0.3
Z axis
0.1
0.0
−0.1
−0.2
Y axis
−0.1
0.0
0.1
0.2
X axis