sensor_fusion_4_1

April 1, 2021

[227]:

```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

TEST_NAME = "euler_angles_2"

dt = 1/960

GRAVITY = 9.80665
RAD_TO_DEG = 180 / math.pi
DEG_TO_RAD = math.pi / 180
```

[228]:

```
# read test params from CSVP
csvp = open(f"data/{TEST_NAME}.csvp")

# create params array
params = np.array([eval(line) for line in csvp])

print(params)
```

```
[   1 3848    0  300    0    1   30  960   96   16 2000   92   92   63
   63   14   14   86   86    0    0    0    0    0    0    0    0    0
    0    0    0]
```

[229]:

```
# load saved mag offsets (this approach isn't being used right now,
# since min/max averaged offsets are recalculated for each data set)

# read mag offsets from file
offsets_dat = open(f"data/mag_offsets.dat")

# create mag offsets array
mag_offsets = np.array([eval(line) for line in offsets_dat])
```

[230]:

```
# read data from CSV
data = pd.read_csv(f"data/{TEST_NAME}.csv", names=["AccelX", "AccelY",␣
↪"AccelZ", "GyroX", "GyroY", "GyroZ", "MagX", "MagY", "MagZ"],␣
↪index_col=False)
```

```python
sample_rate = params[7]

# add time axis to data set
time = np.arange(0, len(data)/sample_rate, 1/sample_rate)
data.insert(0, "Time", time)

# sign data
data = data.applymap(lambda x: x-65535 if x > 32767 else x)

# apply accel sensitivity
acc_cols = ["AccelX", "AccelY", "AccelZ"]
acc_sens = params[9]
data[acc_cols] = data[acc_cols].applymap(lambda x: x * acc_sens * GRAVITY /
 ↪32768)

# apply gyro sensitivity
gyro_cols = ["GyroX", "GyroY", "GyroZ"]
gyro_sens = params[10]
data[gyro_cols] = data[gyro_cols].applymap(lambda x: x * gyro_sens / 32768)

# apply mag sensitivity
mag_cols = ["MagX", "MagY", "MagZ"]
mag_sens = 4800
data[mag_cols] = data[mag_cols].applymap(lambda x: x * mag_sens / 8192)

# FIXME copy for debugging, remove later
original_mag_data = data[mag_cols]

# calculate offsets for each sensor (first 0.5s of data)
acc_offsets = data[acc_cols].head(480).mean()
gyro_offsets = data[gyro_cols].head(480).mean()
#mag_offsets = data[mag_cols].mean()

# min/max method
mag_offsets = (data[mag_cols].max() + data[mag_cols].min()) / 2

# apply offsets to each sensor (remove sensor bias)
# TODO: hold off on accel until actual IMU calibration is implemented
for i, axis in enumerate(gyro_cols):
    data[axis] = data[axis].map(lambda x: x - gyro_offsets[i])

for i, axis in enumerate(mag_cols):
    data[axis] = data[axis].map(lambda x: x - mag_offsets[i])

# create new mag dataframe by removing all NaNs
mag_data = data[mag_cols + ["Time"]].dropna()
```

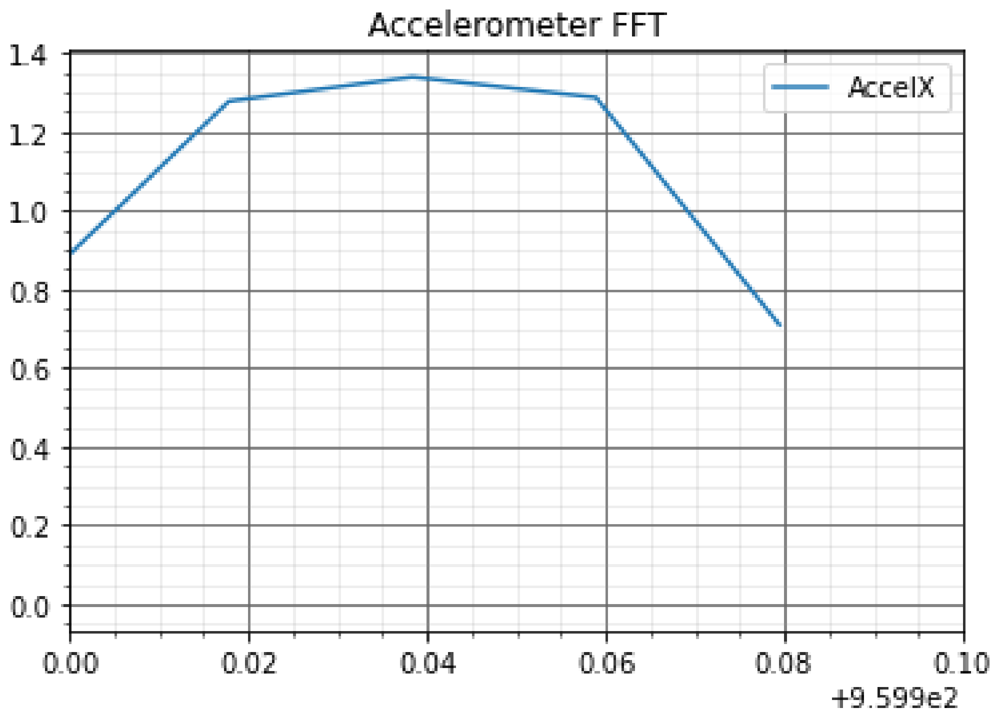
```python
# for some reason, the first mag data point is always erroneous, so remove it
mag_data = mag_data.iloc[1:]

print(mag_offsets)
```

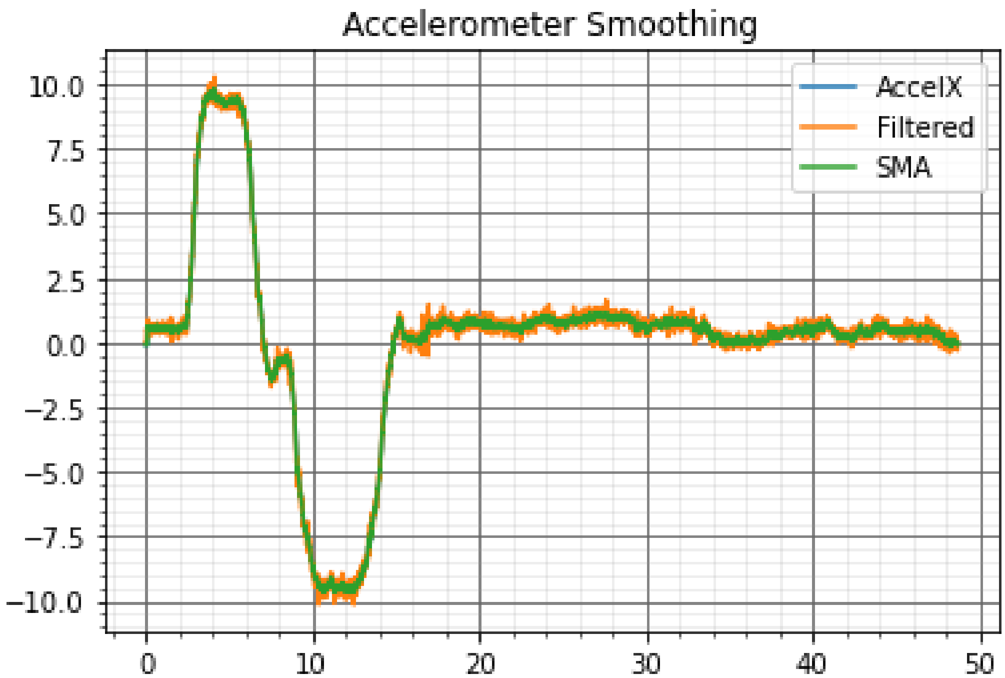
```
MagX   -222.070312
MagY     67.675781
MagZ     -7.910156
dtype: float64
```

[231]:
```python
def show_plot(title=""):
    '''Utility method to graph a plot with grid lines, a legend, and a title.'''
    plt.title(title)
    plt.grid(b=True, which='major', color='#666666', linestyle='-')
    plt.minorticks_on()
    plt.grid(b=True, which='minor', color='#999999', linestyle='-', alpha=0.2)
    plt.legend()
    plt.show()
```

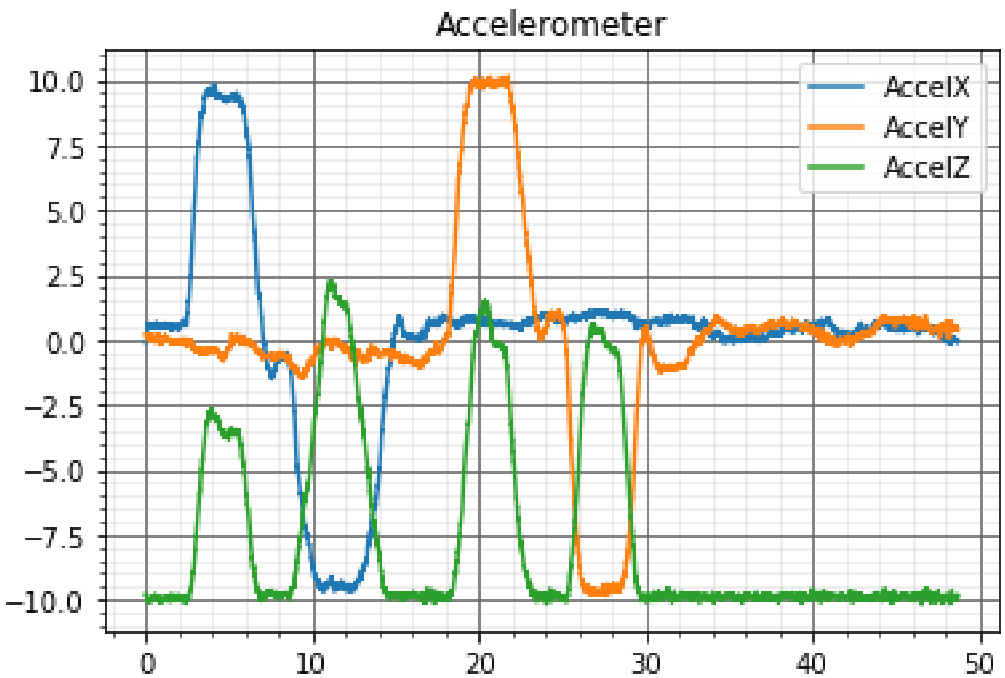
[232]:
```python
# FFT analysis of accelerometer
# (not currently being used, experimental)
import scipy.signal

# calculate fft for AccelX
fourierTransform = np.fft.fft(data["AccelX"])/len(data["AccelX"])

tpCount = len(data["AccelX"])
values = np.arange(tpCount)
timePeriod = tpCount/sample_rate
frequencies = values/timePeriod

plt.plot(frequencies, abs(fourierTransform), label="AccelX")

# display the plot
plt.xlim(959.9,960)
show_plot("Accelerometer FFT")

# normalized cutoff frequency = cutoff frequency / (2 * sample rate)
ORDER = 10
# 959.93838386
CUTOFF_FREQ = 100
NORM_CUTOFF_FREQ = CUTOFF_FREQ / (2 * 960)

# Butterworth filter
num_coeffs, denom_coeffs = scipy.signal.butter(ORDER, NORM_CUTOFF_FREQ)
filtered_data = scipy.signal.lfilter(num_coeffs, denom_coeffs, data["AccelX"])
```

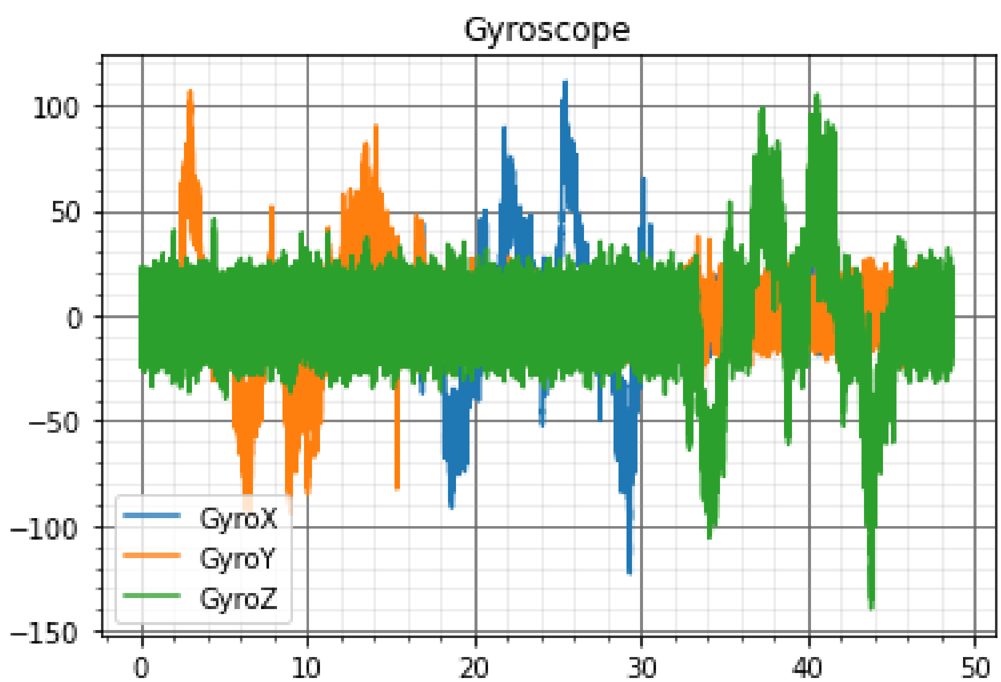
```
# simple moving average
rolling_data = data[acc_cols].rolling(window=100).mean().fillna(data[acc_cols].
↪iloc[49])

# apply simple moving average
data[acc_cols] = rolling_data

plt.plot(data["Time"], data["AccelX"], label="AccelX")
plt.plot(data["Time"], filtered_data, label="Filtered")
plt.plot(data["Time"], rolling_data["AccelX"], label="SMA")
show_plot("Accelerometer Smoothing")
```

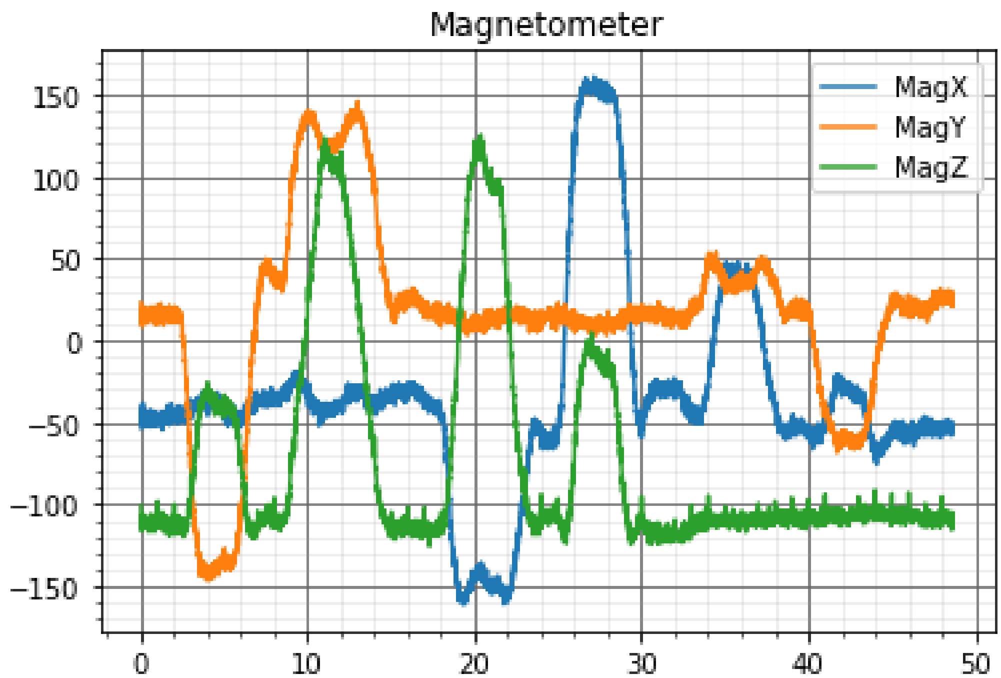




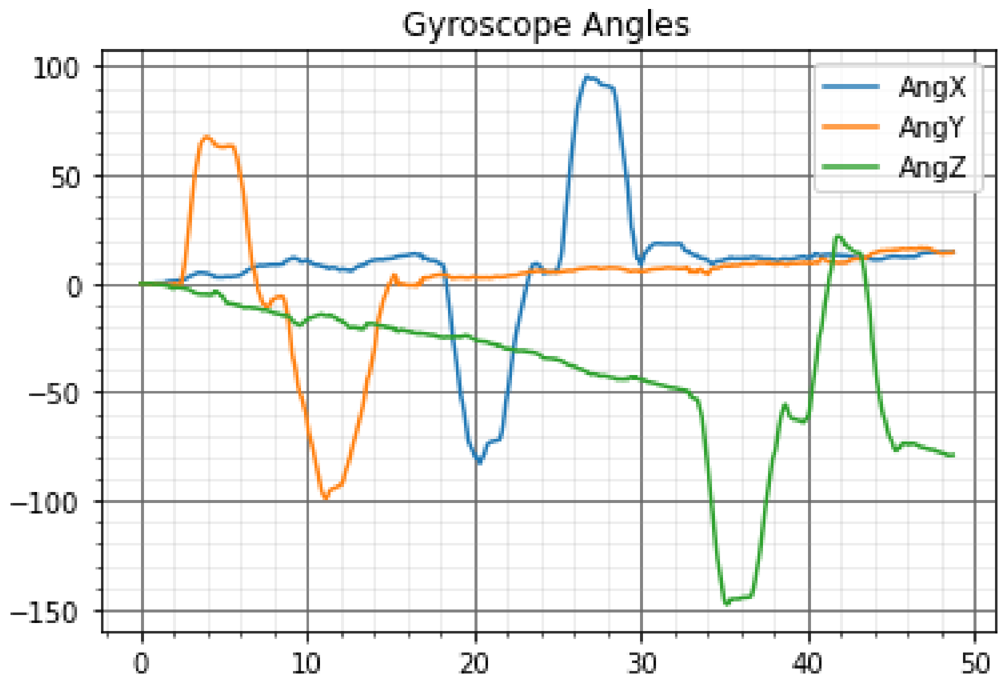
[233]:
```
# plot acceleration
plt.plot(data["Time"], data["AccelX"], label="AccelX")
plt.plot(data["Time"], data["AccelY"], label="AccelY")
plt.plot(data["Time"], data["AccelZ"], label="AccelZ")
show_plot("Accelerometer")

# plot gyroscope
plt.plot(data["Time"], data["GyroX"], label="GyroX")
plt.plot(data["Time"], data["GyroY"], label="GyroY")
plt.plot(data["Time"], data["GyroZ"], label="GyroZ")
show_plot("Gyroscope")

# plot magnetometer
plt.plot(mag_data["Time"], mag_data["MagX"], label="MagX")
plt.plot(mag_data["Time"], mag_data["MagY"], label="MagY")
plt.plot(mag_data["Time"], mag_data["MagZ"], label="MagZ")
show_plot("Magnetometer")
```


Accelerometer
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
10
20
30
40
50
AccelX
AccelY
AccelZ


Gyroscope
100
50
0
−50
−100
−150
0
10
20
30
40
50
GyroX
GyroY
GyroZ



[234]:
```
from scipy import integrate

# calculate angles from gyroscope
ang_x = integrate.cumtrapz(y=data["GyroX"], x=data["Time"], initial=0)
ang_y = integrate.cumtrapz(y=data["GyroY"], x=data["Time"], initial=0)
ang_z = integrate.cumtrapz(y=data["GyroZ"], x=data["Time"], initial=0)

# plot gyroscope angles
plt.plot(data["Time"], ang_x, label="AngX")
plt.plot(data["Time"], ang_y, label="AngY")
plt.plot(data["Time"], ang_z, label="AngZ")
show_plot("Gyroscope Angles")
```



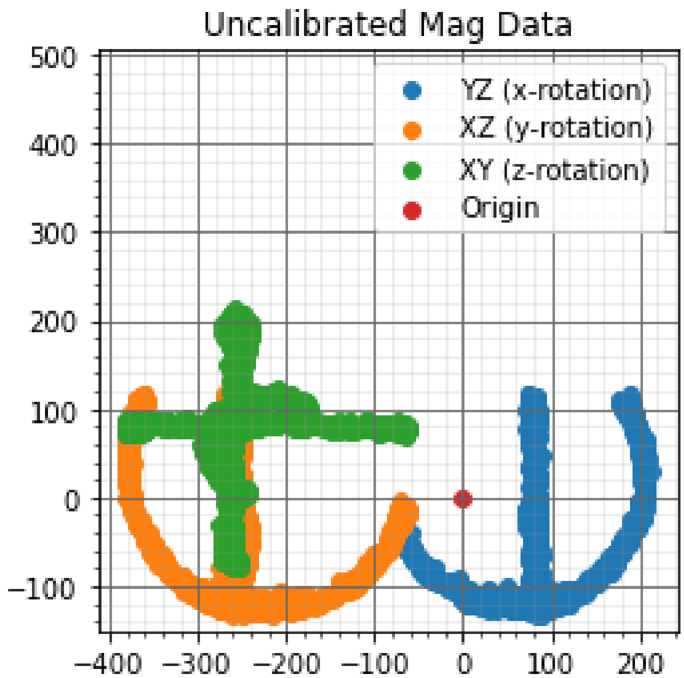
[235]:
```
# least squares ellipse fitting
# from https://www.hep.princeton.edu//mumu/target/Yan/ellipse_fit.pdf
#
# (*) denotes tensor product,
# @ denotes matrix multiplication
def ellipse_fit(x_data, y_data):
    # create x and y vectors
    x = np.array(x_data)
    y = np.array(y_data)

    # X = [x (*) x, x (*) y, y (*) y, x, y]
    #  (transpose to account for x and y not being column vectors)
    X = np.array([np.multiply(x, x), np.multiply(x, y), np.multiply(y, y), x,␣
 ↪y]).T

    # create n-dimensional column vector filled with 1s
    ones = np.c_[[1] * len(x)]

    # calculate coefficients [A,B,C,D,E] where:
    # Ax^2 + Bxy + Cy^2 + Dx + Ey = 1
    beta = (np.linalg.inv(X.T @ X) @ X.T) @ ones
    return beta

# fit data to 3D ellipsoid
```

```python
# from: https://teslabs.com/articles/magnetometer-calibration/
def ellipsoid_fit(s):
    # D (samples)
    D = np.array([s[0]**2., s[1]**2., s[2]**2.,
                  2.*s[1]*s[2], 2.*s[0]*s[2], 2.*s[0]*s[1],
                  2.*s[0], 2.*s[1], 2.*s[2], np.ones_like(s[0])])

    # S, S_11, S_12, S_21, S_22 (eq. 11)
    S = np.dot(D, D.T)
    S_11 = S[:6,:6]
    S_12 = S[:6,6:]
    S_21 = S[6:,:6]
    S_22 = S[6:,6:]

    # C (Eq. 8, k=4)
    C = np.array([[-1,  1,  1,  0,  0,  0],
                  [ 1, -1,  1,  0,  0,  0],
                  [ 1,  1, -1,  0,  0,  0],
                  [ 0,  0,  0, -4,  0,  0],
                  [ 0,  0,  0,  0, -4,  0],
                  [ 0,  0,  0,  0,  0, -4]])

    # v_1 (eq. 15, solution)
    E = np.dot(np.linalg.inv(C),
               S_11 - np.dot(S_12, np.dot(np.linalg.inv(S_22), S_21)))

    E_w, E_v = np.linalg.eig(E)

    v_1 = E_v[:, np.argmax(E_w)]
    if v_1[0] < 0: v_1 = -v_1

    # v_2 (eq. 13, solution)
    v_2 = np.dot(np.dot(-np.linalg.inv(S_22), S_21), v_1)

    # quadric-form parameters
    M = np.array([[v_1[0], v_1[3], v_1[4]],
                  [v_1[3], v_1[1], v_1[5]],
                  [v_1[4], v_1[5], v_1[2]]])
    n = np.array([[v_2[0]],
                  [v_2[1]],
                  [v_2[2]]])
    d = v_2[3]

    return M, n, d
```

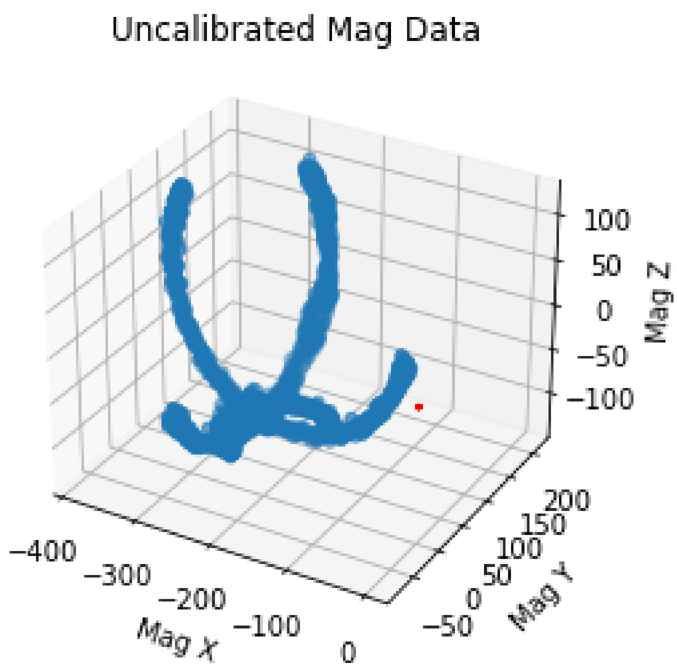
[236]:
```python
# magnetometer calibration techniques
from scipy.linalg import sqrtm
```

```python
def draw_mag_sphere(x_data, y_data, z_data):
    ax.scatter(x_data, y_data, z_data)
    limits = np.array([getattr(ax, f'get_{axis}lim')() for axis in 'xyz'])
    ax.set_box_aspect(np.ptp(limits, axis=1))
    ax.set_xlabel("Mag X")
    ax.set_ylabel("Mag Y")
    ax.set_zlabel("Mag Z")

def draw_sphere(r=1, c=(0,0,0)):
    '''
    Draws a wireframe sphere in a 3D plot.
    By default, this function draws a unit sphere
    (a sphere with a radius of 1 and centered at the origin).
    '''
    u = np.linspace(0, np.pi, 30)
    v = np.linspace(0, 2 * np.pi, 30)
    x = np.outer(np.sin(u), np.sin(v))
    y = np.outer(np.sin(u), np.cos(v))
    z = np.outer(np.cos(u), np.ones_like(v))
    ax.plot_wireframe(c[0]+r*x, c[1]+r*y, c[2]+r*z, color="r", alpha=0.25)

# plot x-axis rotation (MagY, MagZ)
plt.scatter(original_mag_data["MagY"], original_mag_data["MagZ"], label="YZ␣
↪(x-rotation)")

# plot y-axis rotation (MagX, MagZ)
plt.scatter(original_mag_data["MagX"], original_mag_data["MagZ"], label="XZ␣
↪(y-rotation)")

# plot z-axis rotation (MagX, MagY)
plt.scatter(original_mag_data["MagX"], original_mag_data["MagY"], label="XY␣
↪(z-rotation)")

# plot origin (0,0)
plt.scatter([0], [0], label="Origin")

# display the plot
plt.axis("square")
show_plot("Uncalibrated Mag Data")

fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# 3D plot of original mag data
draw_mag_sphere(original_mag_data["MagX"], original_mag_data["MagY"],␣
↪original_mag_data["MagZ"])
```

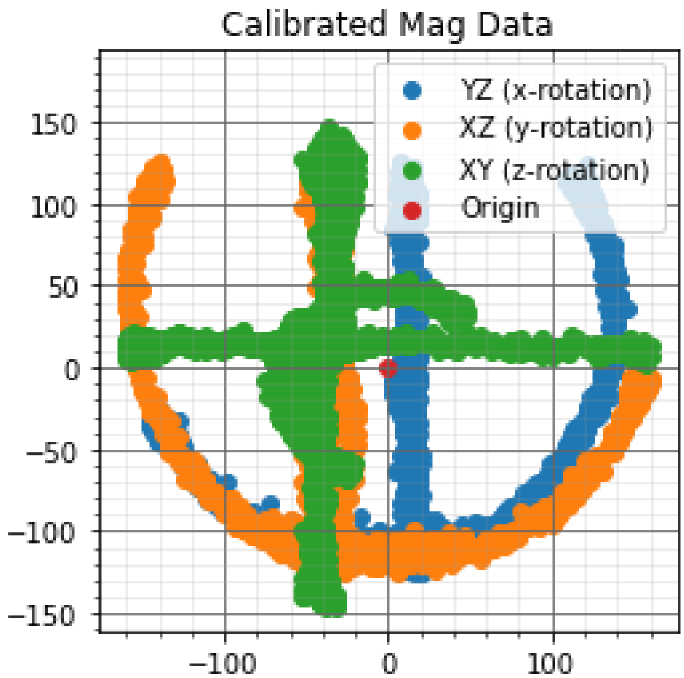
```python
draw_sphere()
plt.title("Uncalibrated Mag Data")
plt.show()

# plot x-axis rotation (MagY, MagZ)
plt.scatter(mag_data["MagY"], mag_data["MagZ"], label="YZ (x-rotation)")

# plot y-axis rotation (MagX, MagZ)
plt.scatter(mag_data["MagX"], mag_data["MagZ"], label="XZ (y-rotation)")

# plot z-axis rotation (MagX, MagY)
plt.scatter(mag_data["MagX"], mag_data["MagY"], label="XY (z-rotation)")

# plot origin (0,0)
plt.scatter([0], [0], label="Origin")

# display the plot
plt.axis("square")
show_plot("Calibrated Mag Data")

fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# 3D plot of mag data
draw_mag_sphere(mag_data["MagX"], mag_data["MagY"], mag_data["MagZ"])
draw_sphere()
plt.title("Calibrated Mag Data")
plt.show()
```

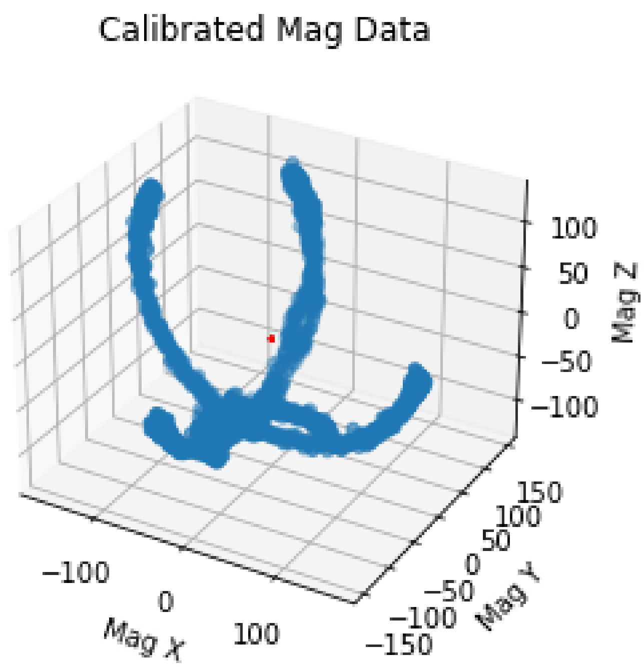


Uncalibrated Mag Data
YZ (x-rotation)
XZ (y-rotation)
XY (z-rotation)
Origin
500
400
300
200
100
0
−100
−400
−300
−200
−100
0
100
200



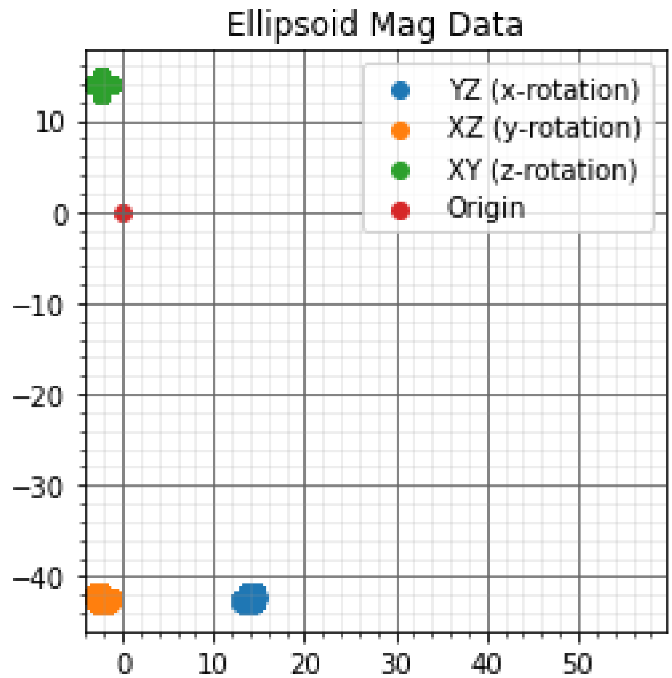
Uncalibrated Mag Data
100
50
0
−50
−100
Mag Z
200
150
100
50
0
−50
Mag Y
−400
−300
−200
−100
0
Mag X

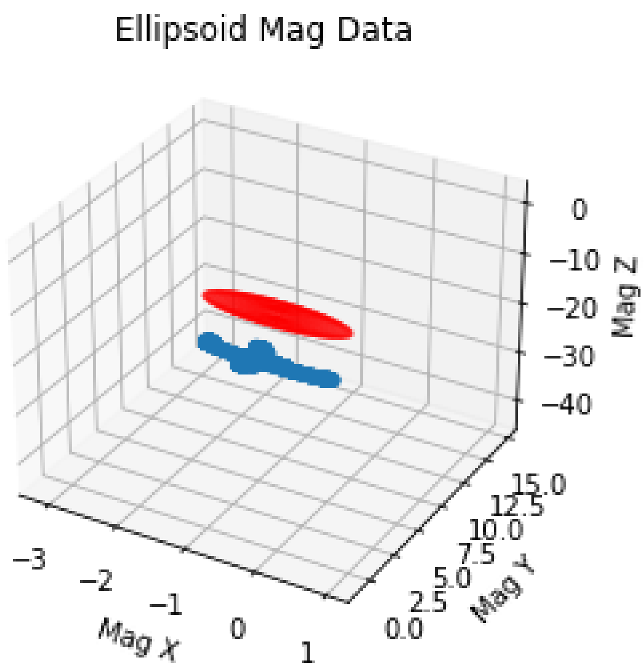




[237]:
```
# ellipsoid fit magnetometer calibration (NOT CURRENTLY IN USE)
# (from https://teslabs.com/articles/magnetometer-calibration/)
```

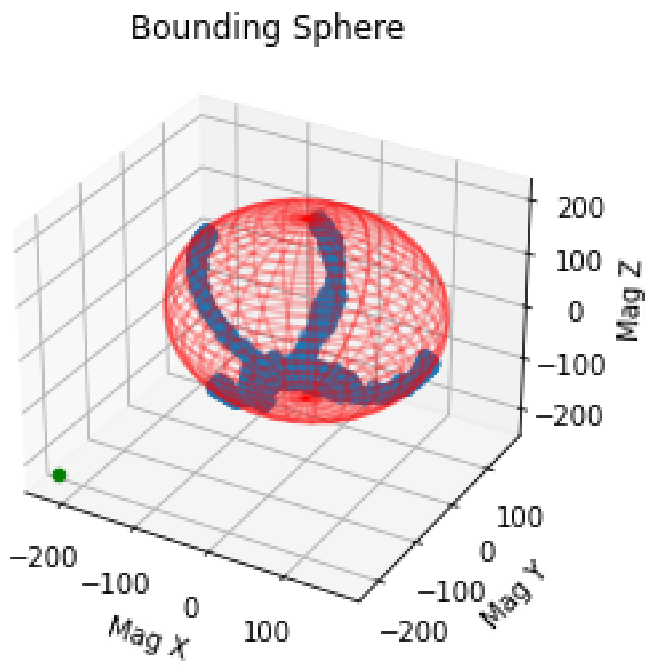
```python
# calculate ellipsoid for data
s = np.array(mag_data[mag_cols]).T
M, n, d = ellipsoid_fit(s)

# calculate calibration parameters for:
# h_m = A @ h + b where h = A^-1 @ (h_m - b)
M_1 = np.linalg.inv(M)
b = -np.dot(M_1, n)
A_1 = np.real(1 / np.sqrt(np.dot(n.T, np.dot(M_1, n)) - d) * sqrtm(M))

# calculate h = A^-1 @ (h_m - b)
def calibrate_mag(row):
    res = A_1 @ np.c_[row] - b
    return res.flatten().tolist()

# calibrate magnetometer
ell_data[mag_cols] = mag_data[mag_cols].apply(calibrate_mag, axis=1,␣
↪result_type='expand')

# plot x-axis rotation (MagY, MagZ)
plt.scatter(ell_data["MagY"], ell_data["MagZ"], label="YZ (x-rotation)")

# plot y-axis rotation (MagX, MagZ)
plt.scatter(ell_data["MagX"], ell_data["MagZ"], label="XZ (y-rotation)")

# plot z-axis rotation (MagX, MagY)
plt.scatter(ell_data["MagX"], ell_data["MagY"], label="XY (z-rotation)")

# plot origin (0,0)
plt.scatter([0], [0], label="Origin")

# display the plot
plt.axis("square")
show_plot("Ellipsoid Mag Data")

fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# 3D plot of mag data
draw_mag_sphere(ell_data["MagX"], ell_data["MagY"], ell_data["MagZ"])
draw_sphere()
plt.title("Ellipsoid Mag Data")
plt.show()
```





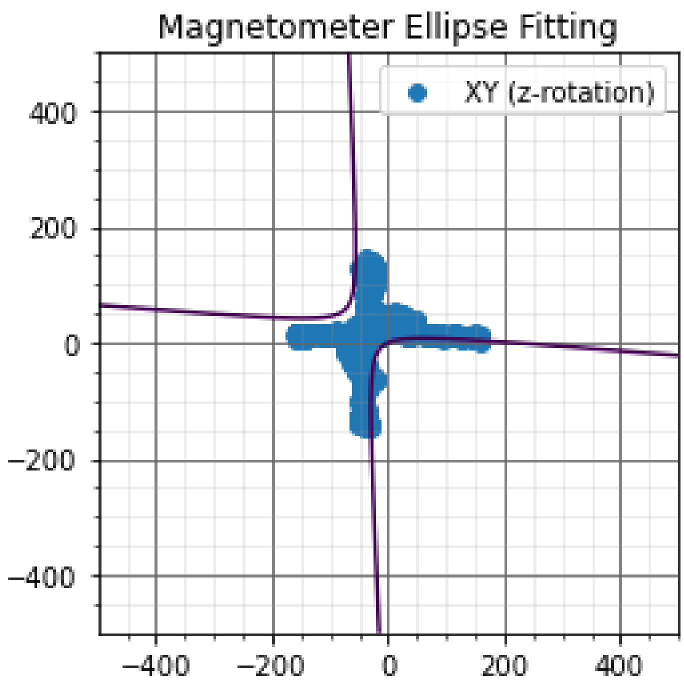
[238]:
```
# bounding sphere test (EXPERIMENTAL)
```

```python
import miniball

# calculate 100% bounding sphere
C, r2 = miniball.get_bounding_ball(mag_data[mag_cols].to_numpy())

fig = plt.figure()
ax = fig.add_subplot(projection="3d")

# draw 100% bounding sphere
draw_mag_sphere(mag_data["MagX"], mag_data["MagY"], mag_data["MagZ"])
draw_sphere(math.sqrt(r2), C)
ax.scatter(mag_offsets[0], mag_offsets[0], mag_offsets[0], color="g")
plt.title("Bounding Sphere")
plt.show()
```



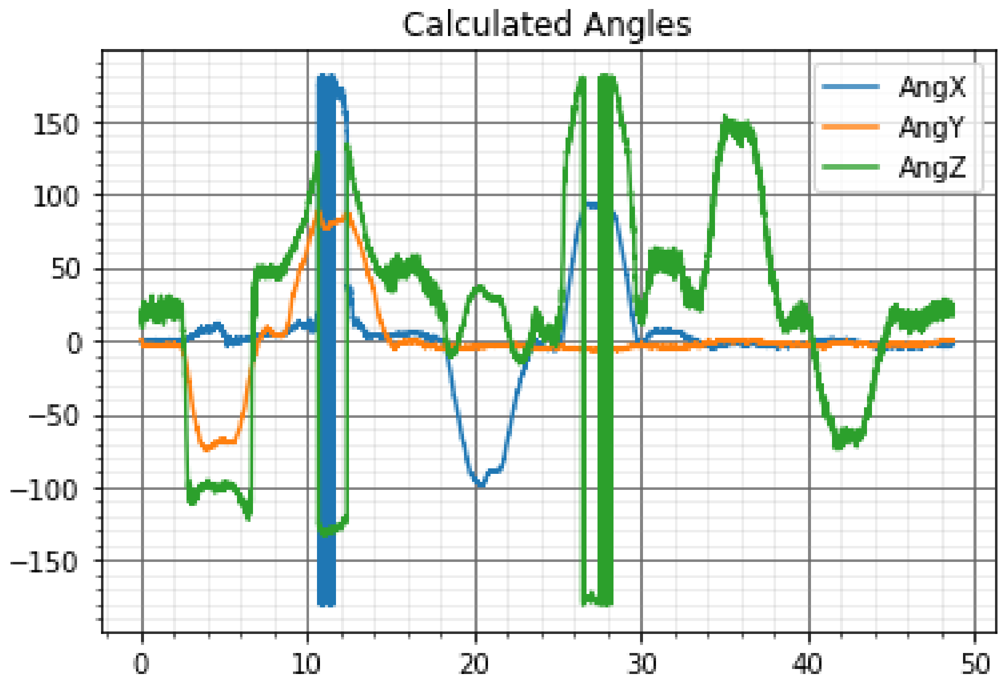
[239]:
```python
# ellipse fitting test (z-axis)
# (NOT IN USE)

# plot z-axis rotation (MagX, MagY)
plt.scatter(mag_data["MagX"], mag_data["MagY"], label="XY (z-rotation)")
plt.plot()

# calculate coefficients for ellipse fitting
A,B,C,D,E = ellipse_fit(mag_data["MagX"], mag_data["MagY"])
```

```
x = np.linspace(-500, 500, 1000)
y = np.linspace(-500, 500, 1000)
x, y = np.meshgrid(x, y)

plt.contour(x, y, A*x**2 + B*x*y + C*y**2 + D*x + E*y, [0])

# display the plot
plt.axis("square")
show_plot("Magnetometer Ellipse Fitting")
```



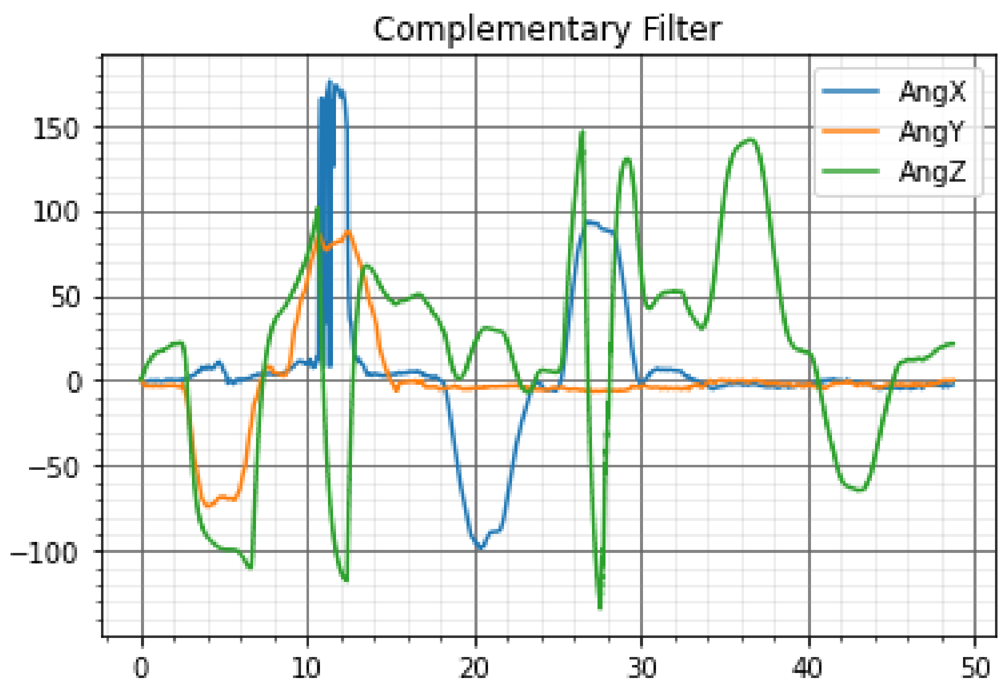
[240]:
```
# OLD: calculate angles from accelerometer
# acc_ang_x = np.arctan2(-data["AccelY"], -data["AccelZ"]) * RAD_TO_DEG
# acc_ang_y = np.arctan2(-data["AccelX"], np.sqrt(data["AccelY"]**2 +␣
 ↪data["AccelZ"]**2)) * RAD_TO_DEG

# better angle calculation from accelerometer
# (from https://docplayer.net/
 ↪23187793-Tilt-sensing-using-a-three-axis-accelerometer.html)

# value dictated by paper, either 0.01 or 0.1
MU = 0.01

# calculate better angles from accelerometer
```

```
# TODO: added 2 negatives to acc_ang_x, doesn't work without it: why?
acc_ang_x = np.arctan2(-data["AccelY"], -np.sign(data["AccelZ"]) * np.
↪sqrt(data["AccelZ"]**2 + MU * data["AccelX"]**2))
acc_ang_y = np.arctan2(-data["AccelX"], np.sqrt(data["AccelY"]**2 +␣
↪data["AccelZ"]**2))

# calculate magnetometer + accelerometer parameters
M_x = mag_data["MagX"] * np.cos(acc_ang_y) + mag_data["MagZ"] * np.
↪sin(acc_ang_y)
M_y = mag_data["MagX"] * np.sin(acc_ang_x) * np.sin(acc_ang_y) +␣
↪mag_data["MagY"] * np.cos(acc_ang_x) - mag_data["MagZ"] * np.sin(acc_ang_x)␣
↪* np.cos(acc_ang_y)

# remove NaNs from mag params
M_x = M_x[~np.isnan(M_x)]
M_y = M_y[~np.isnan(M_y)]

# calculate z-angle (yaw) from accelerometer & magnetometer
mag_ang_z = np.arctan2(M_y, -M_x)

# convert all angles to degrees
acc_ang_x *= RAD_TO_DEG;
acc_ang_y *= RAD_TO_DEG;
mag_ang_z *= RAD_TO_DEG;

# plot accelerometer angles
plt.plot(data["Time"], acc_ang_x, label="AngX")
plt.plot(data["Time"], acc_ang_y, label="AngY")

# plot mag+accel z-axis angle
plt.plot(mag_data["Time"], mag_ang_z, label="AngZ")

# display the plot
show_plot("Calculated Angles")
```



[241]:
```
# complementary filter
HP_weight = 0.98
LP_weight = 0.02

# create empty array w/ 3 axes
# set first elements to 0 (will be removed)
cf_ang = [[0],[0],[0]]

# group all axes of calculated angles together
gyro_ang = np.array([data["GyroX"].to_numpy(), data["GyroY"].to_numpy(),␣
↪data["GyroZ"].to_numpy()])
calc_ang = np.array([acc_ang_x, acc_ang_y, mag_ang_z], dtype=object)

# pair the calculated arrays for each axis together and loop
for i, (gyro_arr, calc_arr) in enumerate(zip(gyro_ang, calc_ang)):

    # pair the gyro & calc samples together and loop
    for j, (gyro, calc) in enumerate(zip(gyro_arr, calc_arr)):

        cf_ang_prev = cf_ang[i][-1]
        cf_ang_samp = HP_weight * (cf_ang_prev + gyro * dt) + LP_weight * calc
        cf_ang[i].append(cf_ang_samp)

        # if this is the magnetometer axis,
```

```
        # repeat sample 9x to ensure it lines up with accel/gyro
        if i == 2:
            for _ in range(9):
                cf_ang[i].append(cf_ang_samp)

# since accel/gyro can have additional samples past 10 x mag,
# repeat the last sample to line up mag axis w/ accel/gyro
num_extra = len(cf_ang[0]) - len(cf_ang[2])
for _ in range(num_extra):
    cf_ang[2].append(cf_ang[2][-1])

# remove initial 0 values
for axis in cf_ang: del axis[0]

plt.plot(data["Time"], cf_ang[0], label="AngX")
plt.plot(data["Time"], cf_ang[1], label="AngY")
plt.plot(data["Time"], cf_ang[2], label="AngZ")

# display the plot
show_plot("Complementary Filter")
```



[242]:
```
import csv

# save angle data as CSV
```

```
with open("out.csv", "w", newline="") as csvfile:

    # define CSV writer
    writer = csv.writer(csvfile, delimiter=",", quotechar='"', quoting=csv.
↪QUOTE_MINIMAL)

    # loop over each row of (x,y,z),
    # and write each line to CSV
    for row in zip(*cf_ang):
        writer.writerow(row)
```