ASTE101_Project2_Team3

#ASTE

- Kwanyoung Jeremy Park (kwanyoun@usc.edu)
- Elizabeth Yoseph (yoseph@usc.edu)
- Tucker Collins (tecollin@usc.edu)
- Brandon Hyo Sung Ahn (ahnbrand@usc.edu)
- London Monjet (monjet@usc.edu)
- Tianlun Ni (tni@usc.edu)

Code

IMU_lib.py

```
#!/usr/bin/env python3

import IMU
import sys
import math

# ================================================================
# CALIBRATION VALUES
# ================================================================

magXmin = -1683
magYmin = -1087
magZmin = -1923
magXmax = 764
magYmax = 1661
magZmax = 777

# ================================================================
# INITIALIZE IMU
# ================================================================
IMU.detectIMU()
if IMU.BerryIMUversion == 99:
print("No BerryIMU found... exiting")
sys.exit()
IMU.initIMU()
```

```
print("IMUclean.py: BerryIMU initialized (clean library).")


# ============================================================
# ACCELEROMETER READ FUNCTIONS
# ============================================================
def readACCx():
return IMU.readACCx()

def readACCy():
return IMU.readACCy()

def readACCz():
return IMU.readACCz()

def readACC():
return readACCx(), readACCy(), readACCz()

# ============================================================
# GYROSCOPE READ FUNCTIONS
# ============================================================

def readGYRx():
return IMU.readGYRx()

def readGYRy():
return IMU.readGYRy()

def readGYRz():
return IMU.readGYRz()

def readGYR():
return readGYRx(), readGYRy(), readGYRz()

# ============================================================
# MAGNETOMETER READ FUNCTIONS
# ============================================================

def readMAGx():
raw = IMU.readMAGx()
return raw - (magXmin + magXmax) / 2

def readMAGy():
```

```
raw = IMU.readMAGy()
return raw - (magYmin + magYmax) / 2

def readMAGz():
raw = IMU.readMAGz()
return raw - (magZmin + magZmax) / 2

def readMAG():
return readMAGx(), readMAGy(), readMAGz()

# ============================================================
# END
# ============================================================
```

Project2.py

```
#!/usr/bin/env python3
import time
import math
import numpy as np
import sys
import IMU_lib as sensors

# ============================================================
#                  CONFIGURATION
# ============================================================
GYRO_GAIN = 0.070
ACCEL_DEADZONE = 0.15
VELOCITY_DECAY = 0.98

DECLINATION = math.radians(12.3)
RAD_TO_DEG = 180 / math.pi
DEG_TO_RAD = math.pi / 180

# ============================================================
#                  HELPER FUNCTIONS
# ============================================================
def Rx(phi):
    return np.array([
        [1, 0, 0],
        [0, math.cos(phi), -math.sin(phi)],
```

```
        [0, math.sin(phi),  math.cos(phi)]
    ])

def Ry(theta):
    return np.array([
        [math.cos(theta), 0, math.sin(theta)],
        [0, 1, 0],
        [-math.sin(theta), 0, math.cos(theta)]
    ])

def Rz(psi):
    return np.array([
        [math.cos(psi), -math.sin(psi), 0],
        [math.sin(psi),  math.cos(psi), 0],
        [0, 0, 1]
    ])

def initialize_system():
    print("\n=== Initializing: HOLD SENSOR FLAT & STILL ===")
    print("   (Calibrating... do not move)")

    N = 100
    gyro_sum_x, gyro_sum_y, gyro_sum_z = 0, 0, 0
    acc_sum_norm = 0

    for i in range(N):
        gx, gy, gz = sensors.readGYR()
        ax, ay, az = sensors.readACC()

        gyro_sum_x += gx
        gyro_sum_y += gy
        gyro_sum_z += gz

        acc_mag = math.sqrt(ax**2 + ay**2 + az**2)
        acc_sum_norm += acc_mag

        time.sleep(0.01)

    # Averages
    g_off_x = gyro_sum_x / N
    g_off_y = gyro_sum_y / N
    g_off_z = gyro_sum_z / N
```

```
    avg_acc_mag = acc_sum_norm / N

    # Scale Factor: We know gravity is 9.81.
    # If raw mag is 4000, scale = 9.81 / 4000 = 0.00245
    acc_scale = 9.81 / avg_acc_mag

    print(f"   [Calibration] Gyro Offsets: X:{g_off_x:.1f} Y:
{g_off_y:.1f} Z:{g_off_z:.1f}")
    print(f"   [Calibration] Accel Scale:  {acc_scale:.6f} (Raw 1g =
{avg_acc_mag:.0f})")

    # --- Lock Orientation ---
    print("   (Locking orientation...)")
    buffer_roll = []
    buffer_pitch = []
    buffer_yaw_x = []
    buffer_yaw_y = []

    stability_count = 0
    REQUIRED_STABLE = 20

    while True:
        ACCx, ACCy, ACCz = sensors.readACC()
        MAGx, MAGy, MAGz = sensors.readMAG()

        # Calculate Angles
        r = math.atan2(ACCy, ACCz)
        p = math.atan2(-ACCx, math.sqrt(ACCy**2 + ACCz**2))

        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm

        mx_c = mx * math.cos(p) + mz * math.sin(p)
        my_c = (mx * math.sin(r) * math.sin(p) +
                my * math.cos(r) - mz * math.sin(r) * math.cos(p))

        y = math.atan2(-my_c, mx_c) + DECLINATION

        buffer_roll.append(r)
        buffer_pitch.append(p)
        buffer_yaw_x.append(math.cos(y))
        buffer_yaw_y.append(math.sin(y))
```

```
        if len(buffer_roll) > 20:
            buffer_roll.pop(0)
            buffer_pitch.pop(0)
            buffer_yaw_x.pop(0)
            buffer_yaw_y.pop(0)

            # Check stability
            r_var = max(buffer_roll) - min(buffer_roll)
            p_var = max(buffer_pitch) - min(buffer_pitch)

            if r_var < (1.0 * DEG_TO_RAD) and p_var < (1.0 *
DEG_TO_RAD):
                stability_count += 1
            else:
                stability_count = 0
                print(f" ... Stabilizing ...", end='\r')

            if stability_count >= REQUIRED_STABLE:
                final_r = sum(buffer_roll) / len(buffer_roll)
                final_p = sum(buffer_pitch) / len(buffer_pitch)

                avg_cx = sum(buffer_yaw_x) / len(buffer_yaw_x)
                avg_cy = sum(buffer_yaw_y) / len(buffer_yaw_y)
                final_y = math.atan2(avg_cy, avg_cx)

                print(f"\n>>> SYSTEM READY. Start: R=
{math.degrees(final_r):.1f}°, P={math.degrees(final_p):.1f}°, Y=
{math.degrees(final_y):.1f}°\n")

                return final_r, final_p, final_y, g_off_x, g_off_y,
g_off_z, acc_scale

        time.sleep(0.02)

def read_all_sensors():
    return sensors.readACC(), sensors.readGYR(), sensors.readMAG()


# ============================================================
#                  MODE 1: NAVIGATION
# ============================================================
def run_navigation_mode():
```

```
    print("\n--- STARTING NAVIGATION MODE ---\n")

    # 1. Initialize
    roll, pitch, yaw, goff_x, goff_y, goff_z, acc_scale =
initialize_system()

    ALPHA = 0.75
    vel = np.zeros(3)
    pos = np.zeros(3)
    gravity = np.array([0, 0, 9.81])

    last_time = time.time()

    while True:
        (ACCx, ACCy, ACCz), (GYRx, GYRy, GYRz), (MAGx, MAGy, MAGz) =
read_all_sensors()

        now = time.time()
        dt = now - last_time
        last_time = now

        # --- GYRO (Calibrated) ---
        gx = math.radians((GYRx - goff_x) * GYRO_GAIN)
        gy = math.radians((GYRy - goff_y) * GYRO_GAIN)
        gz = math.radians((GYRz - goff_z) * GYRO_GAIN)

        # --- ACCEL (Scaled to m/s^2) ---
        ax_m = ACCx * acc_scale
        ay_m = ACCy * acc_scale
        az_m = ACCz * acc_scale

        # --- ORIENTATION TRACKING ---
        roll_acc  = math.atan2(ay_m, az_m)
        pitch_acc = math.atan2(-ax_m, math.sqrt(ay_m**2 + az_m**2))

        # Gyro Integration
        roll_gyro  = roll  + gx * dt
        pitch_gyro = pitch + gy * dt
        yaw_gyro   = yaw   + gz * dt

        # Magnetometer
        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
```

```
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm

        mx_c = mx * math.cos(pitch_acc) + mz * math.sin(pitch_acc)
        my_c = (mx * math.sin(roll_acc) * math.sin(pitch_acc) +
                my * math.cos(roll_acc) - mz * math.sin(roll_acc) *
math.cos(pitch_acc))

        yaw_mag = math.atan2(-my_c, mx_c) + DECLINATION
        if yaw_mag > math.pi:  yaw_mag -= 2*math.pi
        elif yaw_mag < -math.pi: yaw_mag += 2*math.pi

        # Filter
        roll  = ALPHA*roll_gyro  + (1-ALPHA)*roll_acc
        pitch = ALPHA*pitch_gyro + (1-ALPHA)*pitch_acc
        yaw   = ALPHA*yaw_gyro   + (1-ALPHA)*yaw_mag

        # Rotation Matrix
        R = Rz(yaw) @ Ry(pitch) @ Rx(roll)

        # --- POSITION MATH ---
        # 1. Rotate body accel to world frame
        acc_body = np.array([ax_m, ay_m, az_m])
        acc_world = R @ acc_body

        # 2. Remove Gravity
        acc_act = acc_world - gravity

        # 3. NOISE DEADBAND (The "Better" Logic)
        # If acceleration is tiny, treat it as zero to stop jitter
        if np.linalg.norm(acc_act) < ACCEL_DEADZONE:
            acc_act = np.zeros(3)
            # Optional: Slow down velocity if no accel (Simulate
Friction)
            vel *= VELOCITY_DECAY

        # 4. Integrate
        vel += acc_act * dt
        pos += vel * dt

        print("\n============================")
        print(f"Mode: NAV | dt: {dt:.3f}s")
        print(f"Vel: {vel[0]:.2f}, {vel[1]:.2f}, {vel[2]:.2f}")
        print(f"Pos: {pos[0]:.2f}, {pos[1]:.2f}, {pos[2]:.2f}")
```

```
        time.sleep(0.02)


# ============================================================
#                   MODE 2: NED DCM
# ============================================================
def run_dcm_mode():
    print("\n--- STARTING NED DCM MODE ---\n")
    roll, pitch, yaw, goff_x, goff_y, goff_z, acc_scale =
initialize_system()
    ALPHA = 0.98
    last_time = time.time()

    while True:
        (ACCx, ACCy, ACCz), (GYRx, GYRy, GYRz), (MAGx, MAGy, MAGz) =
read_all_sensors()
        now = time.time()
        dt = now - last_time
        last_time = now

        gx = math.radians((GYRx - goff_x) * GYRO_GAIN)
        gy = math.radians((GYRy - goff_y) * GYRO_GAIN)
        gz = math.radians((GYRz - goff_z) * GYRO_GAIN)

        ax_m = ACCx * acc_scale
        ay_m = ACCy * acc_scale
        az_m = ACCz * acc_scale

        roll_acc  = math.atan2(ay_m, az_m)
        pitch_acc = math.atan2(-ax_m, math.sqrt(ay_m**2 + az_m**2))

        roll_gyro  = roll  + gx * dt
        pitch_gyro = pitch + gy * dt
        yaw_gyro   = yaw   + gz * dt

        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm
        mx_c = mx * math.cos(pitch_acc) + mz * math.sin(pitch_acc)
        my_c = (mx * math.sin(roll_acc) * math.sin(pitch_acc) +
                my * math.cos(roll_acc) - mz * math.sin(roll_acc) *
math.cos(pitch_acc))
```

```
        yaw_mag = math.atan2(-my_c, mx_c) + DECLINATION
        if yaw_mag > math.pi:  yaw_mag -= 2*math.pi
        elif yaw_mag < -math.pi: yaw_mag += 2*math.pi

        roll  = ALPHA*roll_gyro  + (1-ALPHA)*roll_acc
        pitch = ALPHA*pitch_gyro + (1-ALPHA)*pitch_acc
        yaw   = ALPHA*yaw_gyro   + (1-ALPHA)*yaw_mag
        R = Rz(yaw) @ Ry(pitch) @ Rx(roll)

        print("\n============================")
        print(f"Mode: DCM | dt: {dt:.3f}s")
        print(f"Roll:  {math.degrees(roll):7.2f}°")
        print(f"Pitch: {math.degrees(pitch):7.2f}°")
        print(f"Yaw:   {math.degrees(yaw):7.2f}°")
        print("\nDCM Rotation Matrix:")
        for row in R:
            print("[ {:8.4f}  {:8.4f}  {:8.4f} ]".format(*row))

        time.sleep(0.02)

def main():
    while True:
        print("\n========================================")
        print("   IMU SYSTEM SELECTION (v3.1 Enhanced)")
        print("========================================")
        print("1. Navigation")
        print("2. NED DCM")
        print("q. Quit")

        choice = input("\nEnter choice: ").strip().lower()
        if choice == '1':
            try: run_navigation_mode()
            except KeyboardInterrupt: pass
        elif choice == '2':
            try: run_dcm_mode()
            except KeyboardInterrupt: pass
        elif choice == 'q':
            sys.exit()

if __name__ == "__main__":
    main()
```

Written Description

`IMU_lib.py`

`IMU_lib.py` was created as a file to import sensors from `IMU` without printing. The only purpose of this file is to separate the code from original sensors.py which ran while loop at the end. It imports sensors from `IMU` and define functions that was used in `Project2.py`

`Project2.py`

`Project2.py` is the code with both NED coordinate run and navigation functions integrated in one.

Below breakdowns code in detail:

Block 1: Imports & Configuration

Lines: 1 – 18

- Imports `numpy` for math and `IMU_lib` to integrate hardware.
- `GYRO_GAIN` **(Line 11):** A conversion factor directly from `BerryIMU.py` to get actual degrees/second.
- `ACCEL_DEADZONE` **(Line 12):** First filter to articulate small accelerations to zero.
- `VELOCITY_DECAY` **(Lines 13):** Second filter to slowly reduce velocity to stop the position from drifting to infinity when the sensor is still–act like resistance.

Lines 2-18(`Project2.py`)

```
import time
import math
import numpy as np
import sys
import IMU_lib as sensors

# ============================================================
#                  CONFIGURATION
# ============================================================
GYRO_GAIN = 0.070
ACCEL_DEADZONE = 0.15
VELOCITY_DECAY = 0.98
```

```
DECLINATION = math.radians(12.3)
RAD_TO_DEG = 180 / math.pi
DEG_TO_RAD = math.pi / 180
...
```

Block 2: Rotation Matrices (Helpers)

Lines: 20 – 42

- Defines the mathematical functions for rotating a 3D point in space around the X, Y, or Z axes.
  - Uses Each function returns a 3x3 matrix filled with sine and cosine values of an angle.
  - When you multiply a vector–in our case, acceleration–by this matrix, it "rotates" that vector into a new frame of reference without changing its length (as it's noramlized)

LINES 20-42 (`PROJECT2.PY`)

```
...
# ============================================================
#                   HELPER FUNCTIONS
# ============================================================
def Rx(phi):
    return np.array([
        [1, 0, 0],
        [0, math.cos(phi), -math.sin(phi)],
        [0, math.sin(phi),  math.cos(phi)]
    ])

def Ry(theta):
    return np.array([
        [math.cos(theta), 0, math.sin(theta)],
        [0, 1, 0],
        [-math.sin(theta), 0, math.cos(theta)]
    ])

def Rz(psi):
```

```
    return np.array([
        [math.cos(psi), -math.sin(psi), 0],
        [math.sin(psi),  math.cos(psi), 0],
        [0, 0, 1]
    ])
...
```

Block 3: System Initialization & Calibration

Lines: 44 – 147 This is the startup sequence for both Navigation and NED_DCM

Part A: Sensor Calibration

Lines: 55 – 85

- Finds the "Zero" point for the Gyroscope.
  - It runs a loop `100` times (Line 57) while the pi-module is still.
  - It calculates the average reading during this run. If the Gyro reads `X` when it should be `0`, the code saves `X` as an offset to subtract later.
  - It calculates `acc_scale` (Line 84) by comparing the raw accelerometer magnitude to Earth's gravity ($9.81 m/s^2$).

LINES 44-85 (`PROJECT2.PY`)

```
def initialize_system():
    print("\n=== Initializing: HOLD SENSOR FLAT & STILL ===")
    print("   (Calibrating... do not move)")

    N = 100
    gyro_sum_x, gyro_sum_y, gyro_sum_z = 0, 0, 0
    acc_sum_norm = 0

    for i in range(N):
        gx, gy, gz = sensors.readGYR()
        ax, ay, az = sensors.readACC()

        gyro_sum_x += gx
        gyro_sum_y += gy
        gyro_sum_z += gz
```

```
        acc_mag = math.sqrt(ax**2 + ay**2 + az**2)
        acc_sum_norm += acc_mag

        time.sleep(0.01)

    # Averages
    g_off_x = gyro_sum_x / N
    g_off_y = gyro_sum_y / N
    g_off_z = gyro_sum_z / N

    avg_acc_mag = acc_sum_norm / N

    # Scale Factor: We know gravity is 9.81.
    # If raw mag is 4000, scale = 9.81 / 4000 = 0.00245
    acc_scale = 9.81 / avg_acc_mag

    print(f"   [Calibration] Gyro Offsets: X:{g_off_x:.1f} Y:
{g_off_y:.1f} Z:{g_off_z:.1f}")
    print(f"   [Calibration] Accel Scale:  {acc_scale:.6f} (Raw 1g =
{avg_acc_mag:.0f})")
```

Part B: Orientation Lock (Tilt-Compensated Compass)

Lines: 87 – 147

- Waits for the sensor to stabilize and finds which way is "North" and "Down" before the main `while` loop runs.
  - It uses a `while True` loop to constantly read angles.
  - Compass Math (Lines 103-112): This is the Tilt Compensation.
  - Stability Check (Lines 125-135): It keeps a history of the last 20 readings. It only allows the code to proceed if the variance (shaking) is less than 1.0 degree.
- This part of the code was largely powered by AI. Needed support to make sure the code only runs when the sensor reading is stabilized.

LINE 87 -147 (`PROJECT2.PY`)

```
...
    # --- Lock Orientation ---
    print("   (Locking orientation...)")
```

```
    buffer_roll = []
    buffer_pitch = []
    buffer_yaw_x = []
    buffer_yaw_y = []

    stability_count = 0
    REQUIRED_STABLE = 20

    while True:
        ACCx, ACCy, ACCz = sensors.readACC()
        MAGx, MAGy, MAGz = sensors.readMAG()

        # Calculate Angles
        r = math.atan2(ACCy, ACCz)
        p = math.atan2(-ACCx, math.sqrt(ACCy**2 + ACCz**2))

        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm

        mx_c = mx * math.cos(p) + mz * math.sin(p)
        my_c = (mx * math.sin(r) * math.sin(p) +
                my * math.cos(r) - mz * math.sin(r) * math.cos(p))

        y = math.atan2(-my_c, mx_c) + DECLINATION

        buffer_roll.append(r)
        buffer_pitch.append(p)
        buffer_yaw_x.append(math.cos(y))
        buffer_yaw_y.append(math.sin(y))

        if len(buffer_roll) > 20:
            buffer_roll.pop(0)
            buffer_pitch.pop(0)
            buffer_yaw_x.pop(0)
            buffer_yaw_y.pop(0)

            # Check stability
            r_var = max(buffer_roll) - min(buffer_roll)
            p_var = max(buffer_pitch) - min(buffer_pitch)

            if r_var < (1.0 * DEG_TO_RAD) and p_var < (1.0 *
DEG_TO_RAD):
```

```
                stability_count += 1
            else:
                stability_count = 0
                print(f" ... Stabilizing ...", end='\r')

            if stability_count >= REQUIRED_STABLE:
                final_r = sum(buffer_roll) / len(buffer_roll)
                final_p = sum(buffer_pitch) / len(buffer_pitch)

                avg_cx = sum(buffer_yaw_x) / len(buffer_yaw_x)
                avg_cy = sum(buffer_yaw_y) / len(buffer_yaw_y)
                final_y = math.atan2(avg_cy, avg_cx)

                print(f"\n>>> SYSTEM READY. Start: R=
{math.degrees(final_r):.1f}°, P={math.degrees(final_p):.1f}°, Y=
{math.degrees(final_y):.1f}°\n")

                return final_r, final_p, final_y, g_off_x, g_off_y,
g_off_z, acc_scale

        time.sleep(0.02)

def read_all_sensors():
    return sensors.readACC(), sensors.readGYR(), sensors.readMAG()
...
```

Block 4: Navigation Mode

Lines: 151 – 224 This is the code for navigation function.

Part A: Sensor Fusion

Lines: 174 – 207

- It determines orientation (Roll, Pitch, Yaw) by integrating three sensors.
  - Gyro Integration (Lines 185-187): Calculates angle by adding speed over time ($Angle = Angle + Speed \times dt$)–directly from `sensors.py` .
  - Accelerometer & Magnetometer Calculation (Lines 181, 190-201): Calculates angle using gravity and magnetic north.

- Complementary Filter (Lines 204-206): Merges gyro and accel/mag using the constant `ALPHA = 0.75`. It takes 75% of the Gyro estimate and corrects it with 25% of the Accel/Mag estimate.

LINE 151-207 (`PROJECT2.PY`)

```
...
# ============================================================
#                   MODE 1: NAVIGATION
# ============================================================
def run_navigation_mode():
    print("\n--- STARTING NAVIGATION MODE ---\n")

    # 1. Initialize
    roll, pitch, yaw, goff_x, goff_y, goff_z, acc_scale =
initialize_system()

    ALPHA = 0.75
    vel = np.zeros(3)
    pos = np.zeros(3)
    gravity = np.array([0, 0, 9.81])

    last_time = time.time()

    while True:
        (ACCx, ACCy, ACCz), (GYRx, GYRy, GYRz), (MAGx, MAGy, MAGz) =
read_all_sensors()

        now = time.time()
        dt = now - last_time
        last_time = now

        # --- GYRO (Calibrated) ---
        gx = math.radians((GYRx - goff_x) * GYRO_GAIN)
        gy = math.radians((GYRy - goff_y) * GYRO_GAIN)
        gz = math.radians((GYRz - goff_z) * GYRO_GAIN)

        # --- ACCEL (Scaled to m/s^2) ---
        ax_m = ACCx * acc_scale
        ay_m = ACCy * acc_scale
        az_m = ACCz * acc_scale
```

```
        # --- ORIENTATION TRACKING ---
        roll_acc  = math.atan2(ay_m, az_m)
        pitch_acc = math.atan2(-ax_m, math.sqrt(ay_m**2 + az_m**2))

        # Gyro Integration
        roll_gyro  = roll  + gx * dt
        pitch_gyro = pitch + gy * dt
        yaw_gyro   = yaw   + gz * dt

        # Magnetometer
        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm

        mx_c = mx * math.cos(pitch_acc) + mz * math.sin(pitch_acc)
        my_c = (mx * math.sin(roll_acc) * math.sin(pitch_acc) +
                my * math.cos(roll_acc) - mz * math.sin(roll_acc) *
math.cos(pitch_acc))

        yaw_mag = math.atan2(-my_c, mx_c) + DECLINATION
        if yaw_mag > math.pi:  yaw_mag -= 2*math.pi
        elif yaw_mag < -math.pi: yaw_mag += 2*math.pi

        # Filter
        roll  = ALPHA*roll_gyro  + (1-ALPHA)*roll_acc
        pitch = ALPHA*pitch_gyro + (1-ALPHA)*pitch_acc
        yaw   = ALPHA*yaw_gyro   + (1-ALPHA)*yaw_mag
...
```

Part B: Matrices Formation

Lines: 209 – 212

- Converts the acceleration from the Body Frame (relative to the pi-module) to the World Frame (North/East/Down).
  - It creates a "Rotation Matrix" `R` using the current Roll, Pitch, and Yaw.
  - It performs a dot product(`@`) between `R` and the acceleration vector.

LINES 209 - 212 (`PROJECT2.PY`)

```
        # Rotation Matrix
        R = Rz(yaw) @ Ry(pitch) @ Rx(roll)

        # --- POSITION MATH ---
        # 1. Rotate body accel to world frame
        acc_body = np.array([ax_m, ay_m, az_m])
        acc_world = R @ acc_body

...
```

Part C: Dead Reckoning

Lines: 215 – 221

- Calculates Velocity and Position from Acceleration and `R`.
  - Gravity Removal (Line 215): Subtracts `9.81` from the Z-axis.
  - Deadzone Check (Line 218): If the movement is smaller than `ACCEL_DEADZONE`, forces acceleration to zero.
  - Integration (Line 220-221):
    - Velocity adds acceleration ($v = v + a \cdot dt$).
    - Position adds velocity ($p = p + v \cdot dt$).

LINE 215 - 221 (`PROJECT2.PY`)

```
...
        # 2. Remove Gravity
        acc_act = acc_world - gravity

        # 3. NOISE DEADBAND (The "Better" Logic)
        # If acceleration is tiny, treat it as zero to stop jitter
        if np.linalg.norm(acc_act) < ACCEL_DEADZONE:
            acc_act = np.zeros(3)
            vel *= VELOCITY_DECAY

        # 4. Integrate
        vel += acc_act * dt
        pos += vel * dt

        print("\n==========================")
        print(f"Mode: NAV | dt: {dt:.3f}s")
```

```
        print(f"Vel: {vel[0]:.2f}, {vel[1]:.2f}, {vel[2]:.2f}")
        print(f"Pos: {pos[0]:.2f}, {pos[1]:.2f}, {pos[2]:.2f}")

        time.sleep(0.02)
...
```

Block 5: NED DCM Mode

Lines: 226 – 271

- A mode that only outputs orientation (Perform Task2 of Project2.)
  - It runs the exact same Sensor Integration logic as Block 4A.
  - It prints the raw Direction Cosine Matrix (Rotation Matrix) instead of XYZ coordinates.

LINE 226 - 271 (`PROJECT2.PY`)

```
...
# ============================================================
#                   MODE 2: NED DCM
# ============================================================
def run_dcm_mode():
    print("\n--- STARTING NED DCM MODE ---\n")
    roll, pitch, yaw, goff_x, goff_y, goff_z, acc_scale =
initialize_system()
    ALPHA = 0.98
    last_time = time.time()

    while True:
        (ACCx, ACCy, ACCz), (GYRx, GYRy, GYRz), (MAGx, MAGy, MAGz) =
read_all_sensors()
        now = time.time()
        dt = now - last_time
        last_time = now

        gx = math.radians((GYRx - goff_x) * GYRO_GAIN)
        gy = math.radians((GYRy - goff_y) * GYRO_GAIN)
        gz = math.radians((GYRz - goff_z) * GYRO_GAIN)
```

```
        ax_m = ACCx * acc_scale
        ay_m = ACCy * acc_scale
        az_m = ACCz * acc_scale

        roll_acc  = math.atan2(ay_m, az_m)
        pitch_acc = math.atan2(-ax_m, math.sqrt(ay_m**2 + az_m**2))

        roll_gyro  = roll  + gx * dt
        pitch_gyro = pitch + gy * dt
        yaw_gyro   = yaw   + gz * dt

        mag_norm = math.sqrt(MAGx**2 + MAGy**2 + MAGz**2)
        if mag_norm == 0: mag_norm = 1
        mx, my, mz = MAGx/mag_norm, MAGy/mag_norm, MAGz/mag_norm
        mx_c = mx * math.cos(pitch_acc) + mz * math.sin(pitch_acc)
        my_c = (mx * math.sin(roll_acc) * math.sin(pitch_acc) +
                my * math.cos(roll_acc) - mz * math.sin(roll_acc) *
math.cos(pitch_acc))
        yaw_mag = math.atan2(-my_c, mx_c) + DECLINATION
        if yaw_mag > math.pi:  yaw_mag -= 2*math.pi
        elif yaw_mag < -math.pi: yaw_mag += 2*math.pi

        roll  = ALPHA*roll_gyro  + (1-ALPHA)*roll_acc
        pitch = ALPHA*pitch_gyro + (1-ALPHA)*pitch_acc
        yaw   = ALPHA*yaw_gyro   + (1-ALPHA)*yaw_mag
        R = Rz(yaw) @ Ry(pitch) @ Rx(roll)

        print("\n============================")
        print(f"Mode: DCM | dt: {dt:.3f}s")
        print(f"Roll:  {math.degrees(roll):7.2f}°")
        print(f"Pitch: {math.degrees(pitch):7.2f}°")
        print(f"Yaw:   {math.degrees(yaw):7.2f}°")
        print("\nDCM Rotation Matrix:")
        for row in R:
            print("[ {:8.4f}  {:8.4f}  {:8.4f} ]".format(*row))

        time.sleep(0.02)
...
```

Block 6: Main Entry

Lines: 273 – 286

- Prints out the user interface.
- A simple input loop that lets you select Mode 1, Mode 2, or Quit (`q`).

Instead of inserting lines of code, printed output when code is ran is inserted:

INITIAL TERMINAL INTERFACE

```
**aste101@pi3**:**~/Projects/aste101_2 $** python Project2.py

Found BerryIMUv3 (LSM6DSL and LIS3MDL)
IMUclean.py: BerryIMU initialized (clean library).

========================================
   IMU SYSTEM SELECTION (v3.1 Enhanced)
========================================

1. Navigation
2. NED DCM
q. Quit

Enter choice:
```

Reflection

Challenges Faced:

1. The initial rotational function setting took a while due to limited conceptual understanding of roll, pitch, and yaw angles in 3D coordinates.
2. Accelerometer integration using `atan2` didn't come out straightforward.
3. Magnetometer integration took the longest, again, due to limited understanding of 3D coordinate system, particularly how to account for the sensor's tilt.
4. The filtering part using three different sensors to deduce an accurate `R` wasn't intuitive. We initially sought to approach this by resetting the system to accelerometer and magnetometer readings whenever the gyroscope reading deviated too far. We then discovered a method called complementary filtering (the `ALPHA` method). We do not believe this is the most accurate way to achieve this... but it gets the job done–to some extent...! A Kalman Filter would perform a better job. A thing to note is that a different `ALPHA` values were used for navigation function and NED_DCM function, 0.75 and 0.98 each after testing various numbers.
5. Though unsure whether necessary, the initial coordinates were "tared(calibrated)" at the beginning of each run to establish a starting point as we thought it would

eliminate the error with magnetometer.

6. Just like other groups, had an issue of the position constantly increasing(or decreasing) due to hardware issue and mathematical limitation of the gyro-sensor integration method. Achieved to filter it by setting a Deadzone and Decay. While Decay does not seem practically useful, Deadzone could be a useful solution to this problem. Yet, the final result remains imperfect.