MEMORANDUM

To: Charlie Refvem, Department of Mechanical Engineering, Cal Poly SLO
crefvem@calpoly.edu

From: Antonio Ventimiglia
ventimig@calpoly.edu

Caiden Bonney
clbonney@calpoly.edu

Date: 11/30/2025

RE: Lab 0x05: State Estimation

The following will cover the team's Lab 0x05 results after integrating both an IMU module that tracks Romi's Euler angles and rotational speeds, and an observer class that uses a discretized observer model to best estimate Romi's pose (position and orientation).

IMU

A fully successful integration of the IMU was accomplished. When the IMU's calibration function is called, the IMU object searches for existing calibration data. If it exists, the values are loaded into the respective registers. If calibration data does not exist, the IMU object checks the calibration status byte to determine whether proper calibration has been performed. While the status byte reads inadequate calibration, blocking code halts operations until full calibration efforts have been executed. Once calibration is confirmed, the proper offsets are "mem_read" and written to a new text file on Romi. Table 1 shows the calibration data, in decimal, gathered while calibrating Romi on the center table.

Table 1. IMU Calibration Data

Acceleration Offsets			Magnetometer Offsets			Gyroscope Offsets			Radii	
X	Y	Z	X	Y	Z	X	Y	Z	ACC	MAG
65530	0	19	65478	65136	768	65535	2	0	1000	784

The overflow/underflow that occurs at $\psi = 0$ and $\psi = 2\pi$ has been properly accounted for in the exact same manner it was addressed with the encoders. Since the observer runs at such a short period, it is confidently assumed that Romi will not be able to turn 1/3 of a revolution in the time between observer task executions, providing the team with an upper limit value to check for overflow

State Estimator

Our group decided to implement a discretized model rather than the RK4 solver. To begin implementing this discretized model, we set up a MATLAB script to calculate the discretized A_D and B_D matrices.

Calculating A_D and B_D

Since the observer is a fourth-order system, it requires four eigenvalues. Two of these are chosen to form a second-order pair that achieve a desired overshoot and settling-time; these poles shape the dominant estimation dynamics. The remaining two poles are placed far to the left in the complex plane, making them fast-decaying modes whose influence quickly diminish. As a result, the observer behaves primarily like the designed second-order system while still ensuring full-order stability and rapid convergence of all estimation error modes.

First, we needed to decide on a maximum overshoot percentage, M_p, and settling-time, T_s. Our group chose an overshoot percentage of 5%, $M_p = 0.05$, and a settling time of one-tenth the average time constant of

our motors, $T_s = 0.1\tau_{m,ave}$. We determined the time constants and steady state motor gains for each motor from data collected in Lab 0x02. We used the average of these properties, $\tau_{m,ave}$ and $K_{m,ave}$, for all subsequent calculations.

Based on the maximum overshoot percentage equation,

$$M_p = e^{\left(-\frac{\zeta\pi}{\sqrt{1-\zeta^2}}\right)}$$

and chosen value for M_p, the dampening constant, ζ, can be determined. Based on the settling-time equation corresponding to the time required for the response to be within ±5% of steady-state,

$$T_s = \frac{3}{\zeta\omega_n}$$

and the known values of T_s and ζ, the corresponding natural frequency, ω_n, for the system can be determined. With the known values of ζ and ω_n, the first two poles for the observer gain matrix, L, can be calculated. The second-order equation created from these parameters are as follows,

$$P(s) = s^2 + 2\zeta\omega_n s + \omega_n^2$$

The discretized model requires four poles, so two additional arbitrary poles were added to the system. Emphasizing the importance of the pole placement being far on the negative real axis, the poles were placed at $s = -10\omega_n$ and $s = -10\omega_n + 1$, resulting in the full system outlined in Equation 1

$$P(s) = (s^2 + 2\zeta\omega_n s + \omega_n^2)\big(s - (-10\omega_n)\big)\big(s - (-10\omega_n + 1)\big) \quad (1)$$

The four roots for Equation 1 will be the pole placements for the observer gain matrix.

Matrices A, B, C, and D were determined in the lecture notes, based on the wheel radius, $r = 35\ [mm]$, and track width, $w = 141[mm]$. Future testing yielded results slightly deviated from expectations, so calipers were used to obtain a more accurate wheel radius of $r = 34.75\ [mm]$, which was then substituted into the system equations.

Using MATLAB's "place" function, the observer gain matrix, L, was calculated from matrices A and C, with the pole locations equal to the roots of Equation 1.

The continuous system matrices A_O, and B_O were then determined from the equations provided in the notes for the discretized model,

$$A_O = A - LC \quad \text{and} \quad B_O = [B - LD \quad L]$$

Then, using MATLAB's continuous to discrete function, "c2d", the continuous system matrices A_O, and B_O, were converted to discrete system matrices A_D, and B_D, for a given time step, $t_{step} = 0.02[s]$. This t_{step} must match the period that the state estimator is running at.

<u>Romi's Observer Class</u>

Using the discretized observer model equations,

$$\hat{\mathrm{x}} = \begin{bmatrix} \Omega_{\mathrm{L}} \\ \Omega_{\mathrm{R}} \\ s \\ \psi \end{bmatrix}, \quad u = \begin{bmatrix} V_L \\ V_R \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} \mathrm{s}_{\mathrm{L}} \\ \mathrm{s}_{\mathrm{R}} \\ \psi \\ \dot{\psi} \end{bmatrix}$$

$$\hat{\mathrm{x}} = \begin{bmatrix} \Omega_{\mathrm{L}} \\ \Omega_{\mathrm{R}} \\ s \\ \psi \end{bmatrix}, \quad u^* = \begin{bmatrix} u \\ y \end{bmatrix} = \begin{bmatrix} V_L \\ V_R \\ \mathrm{s}_{\mathrm{L}} \\ \mathrm{s}_{\mathrm{R}} \\ \psi \\ \dot{\psi} \end{bmatrix}$$

$$\hat{\mathrm{x}}_{k+1} = A_D \hat{\mathrm{x}}_k + B_D u_k^*$$

$$\hat{y}_k = C \hat{\mathrm{x}}_k$$

and the aforementioned A_D, B_D, and C matrices found in MATLAB, $\hat{\mathrm{x}}_{k+1}$ and $\hat{y}_k$ can be determined.

$$A_D = \begin{bmatrix} 0.0012 & 0.0012 & 0.0140 & 0 \\ 0.0012 & 0.0012 & 0.0140 & 0 \\ -0.0001 & -0.0001 & 0.0039 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B_D = \begin{bmatrix} 0.0506 & 0.0436 & -0.0070 & -0.0070 & 0 & -2.0241 \\ 0.0436 & 0.0506 & -0.0070 & -0.0070 & 0 & 2.0241 \\ 0.0029 & 0.0029 & 0.4980 & 0.4980 & 0 & 0 \\ 0 & 0 & -0.0071 & 0.0071 & 0.0001 & 0.0002 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 1 & -70.5 \\ 0 & 0 & 1 & 70.5 \\ 0 & 0 & 0 & 1 \\ -0.2465 & 0.2465 & 0 & 0 \end{bmatrix}$$

The input matrix, u^*, is updated at the start of each run of the observer class. u^* obtains its values from three different sources: voltages from motor objects, displacements from encoder objects, and orientation data from the IMU object. To calculate the voltage supplied to each motor, the current pulse-width-percentage applied to each motor is multiplied by the current voltage of the batteries, which inherently accounts for battery droop over operation time. The remaining four inputs are read directly from their corresponding objects.

In addition, the global X and Y positions of Romi were calculated by determining the global velocities, $\dot{\mathrm{X}}$ and $\dot{\mathrm{Y}}$. The velocities are obtained from the instantaneous heading angles and Romi-centroid velocities. Due to B_D's 5th column containing nearly all 0s, the yaw determined through the observer is always near-zero, regardless of Romi's orientation. Therefore, for the global velocity calculations, the measured yaw is used instead of the "observed" yaw. Future restructuring of the matrices is to be done to resolve the inadequate prediction from the observer.

Romi's centroid velocity is based on a weighted average of the centroid velocity calculated from the observer-determined Ω_{L} and Ω_{R}, $\hat{v}$, and the centroid velocity determined by the average of the derivatives of the encoders' positions, v. The weighted average, $v_{avg} = (3\hat{v} + 2v)/5$, was chosen to incorporate both inputs; however, due to the previously seen noise in the velocity determined through the encoders, v, future testing will rely solely on $\hat{v}$.

Results:

The first test conducted to validate the Observer's data was moving Romi forward a fixed distance based on the Observer's estimated distance traveled. The first and second tests shown in Figure 1 recorded the left and right encoder positions against a target distance. After confirmation that the observer results were within reason, implementation of the center position, heading, and X and Y positions were completed shown by the additional data shown in the final observations noted in Figure 1.

```
Going straight at 100 mm/s:

    Target distance was 100 mm
    - Left position was 103 mm
    - Right position was 104 mm
    - Observed position was ~110 mm
    --------------------------------
    Target distance was 500 mm
    - Left position was 503 mm
    - Right position was 504 mm
    - Observed position was ~513 mm
    --------------------------------
    Target distance was 500 mm
    - Left position was 503 mm
    - Right position was 504 mm
    - Center position was 501 mm
    - Observed position was ~513 mm
    - heading was ~0.00 rad
    - X position was ~0.00 mm
    - Y position was ~0.00 mm
-----------------------------------
```

Figure 1. Results of the first three tests of moving Romi forward a fixed distance

The X and Y positions in Figure 1 referenced the incorrect matrix, adjusting this error and doubling the speed and target distance led to the results shown in Figure 2.

```
-------------------------------------------------------
Going straight at 200 mm/s:

    Target distance was 1000 mm
    - Left position was 1001.963 mm
    - Right position was 1001.608 mm
    - Center position was 1001.786 mm
    - Observed position was ~1025 mm
    - heading was ~0 rad
    - X position was ~999.4849 mm (SCALING = 1/14.578334)
    - Y position was ~0.00643 mm (SCALING = 1/14.578334)
-------------------------------------------------------
```

Figure 2. Results of the fourth test of moving Romi forward a fixed distance

After adjusting to the correct matrix, the X and Y positions were off by a factor of 14.578334, as noted by Figure 2. All future testing scales the X and Y positions calculated by the observer by this constant. Future testing will be required to determine why this is the case; however, note that all future distances are accurate within reason after this adjustment.

To validate this final Observer implementation, Romi was placed in line-following mode (segment 2 of Path Director) and instructed to traverse a circular path with a 600 mm diameter. The segment concluded once the Observer estimated that Romi had completed a full revolution. Figure 3. IMU Heading versus Data Index presents the IMU's heading data during the traversal. The visible linear trend reflects Romi's constant rotational velocity. The IMU readings reveal noise arising from electromagnetic interference (EMI) impacting the hardware. Figure 4. Observer's Estimated Center Arc Length versus Data Index depicts the Observer's estimated arc length for Romi's centroid along the circular path. Compared to Figure 3, the Observer's effective smoothing of the data is clearly demonstrated. Figure 5 illustrates the trajectory of Romi as reconstructed by the Observer. A notable outcome is that the circular path's dimensions were remarkably well preserved in the estimated X and Y positions. Taken together, these results suggest that the Observer provided a satisfactory estimation of Romi's pose in global space.

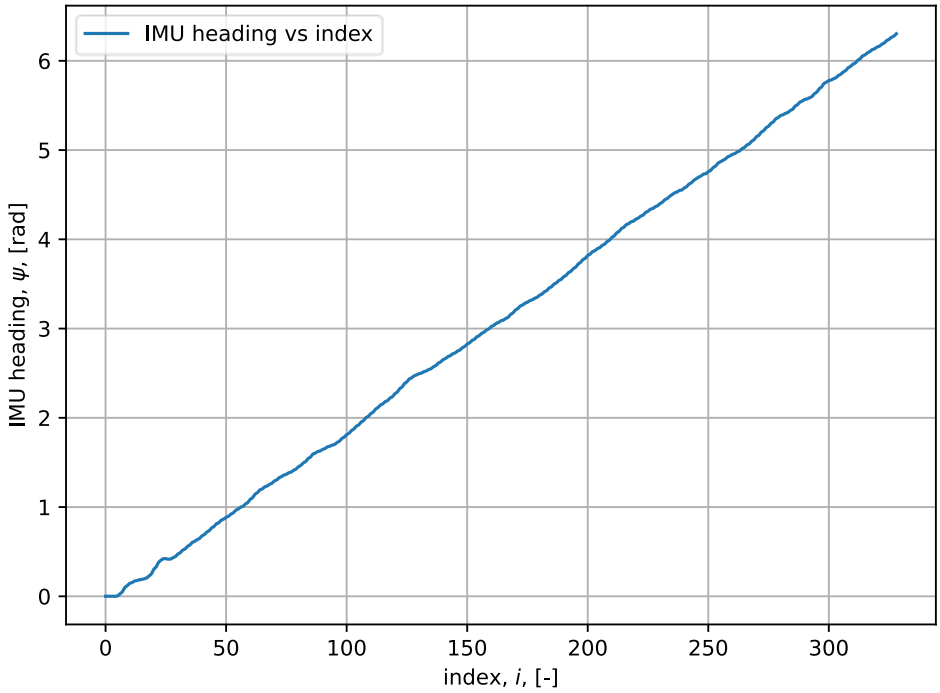

Figure 3. IMU Heading versus Data Index

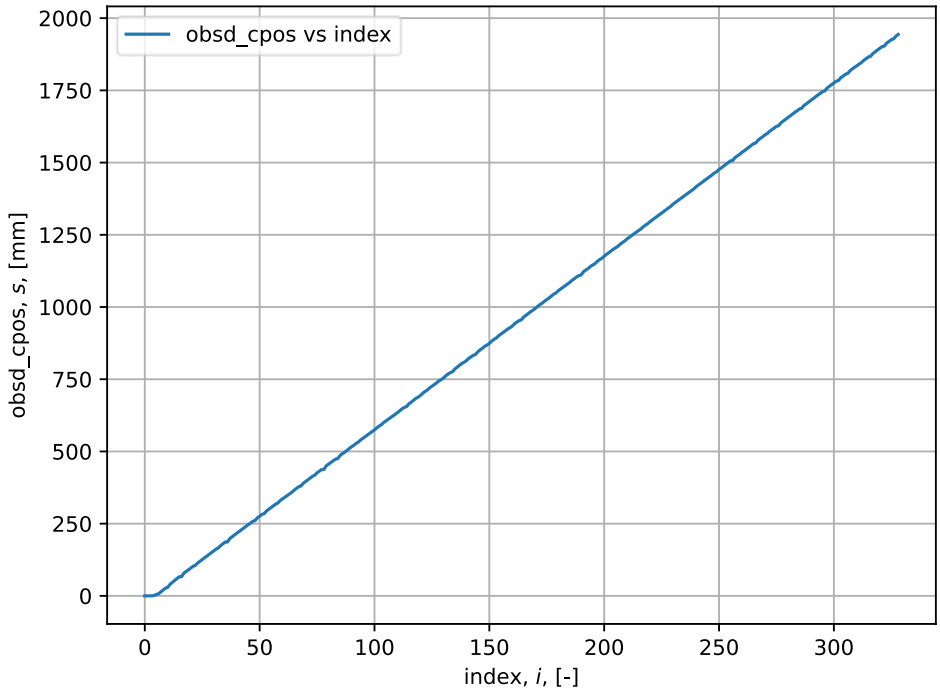

Figure 4. Observer's Estimated Center Arc Length versus Data Index

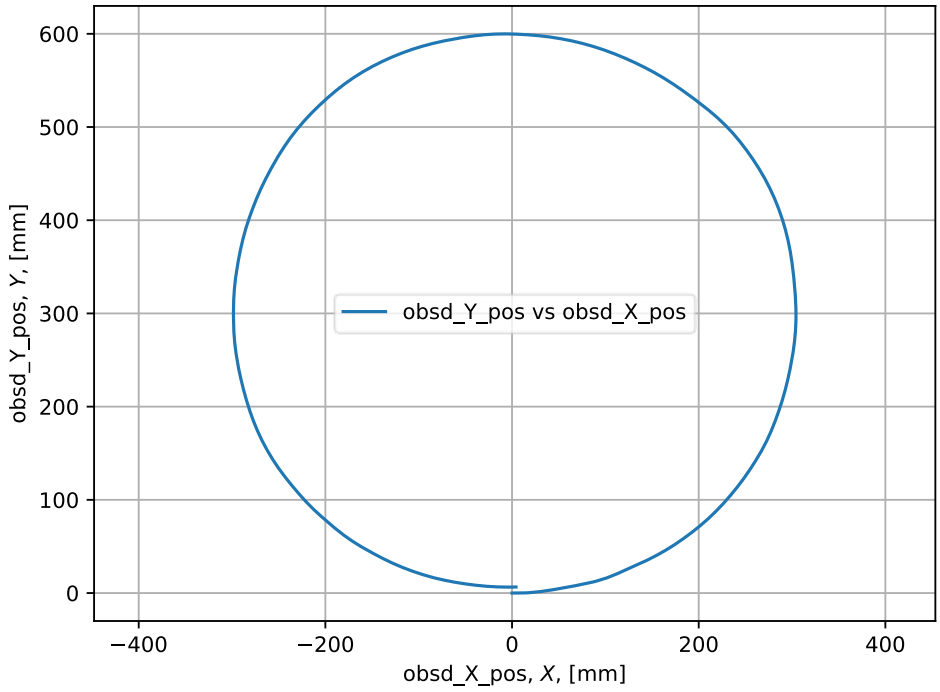

Figure 5. Observer's Estimated Y-Position versus Estimated X-Position

Overall, the observer's state estimation proved effective in predicting Romi's pose, with the exception of the heading data. Figure 5 illustrates this satisfactory estimation, since Romi followed the 600mm diameter circular track for one revolution, and the positions reported are incredible accurate to the track's nominal

size. It is important to note Romi ended roughly 30 degrees past a full revolution showing that the estimation is not perfect, but satisfactory for such a large path.

Task Diagram

Figure 6 shows how each task communicates with the others using shares and queues, including each task's period and priority.

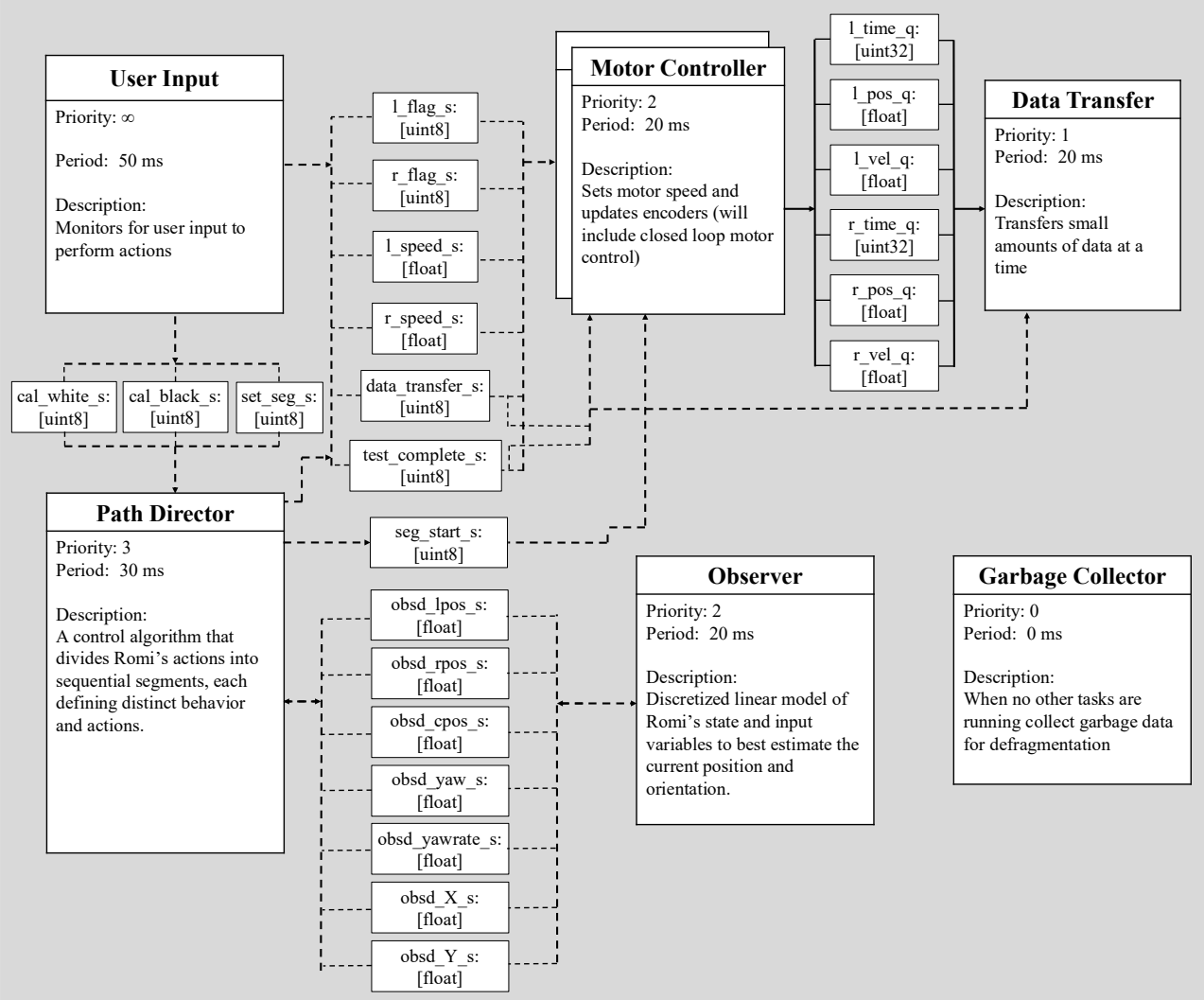

Figure 6. Task Diagram for Romi

Figures 7 through 12 illustrate the functionality of each task. Since a finite-state machine (FSM) architecture was not chosen for our code, the following figures illustrate the tasks using their "logic diagram."

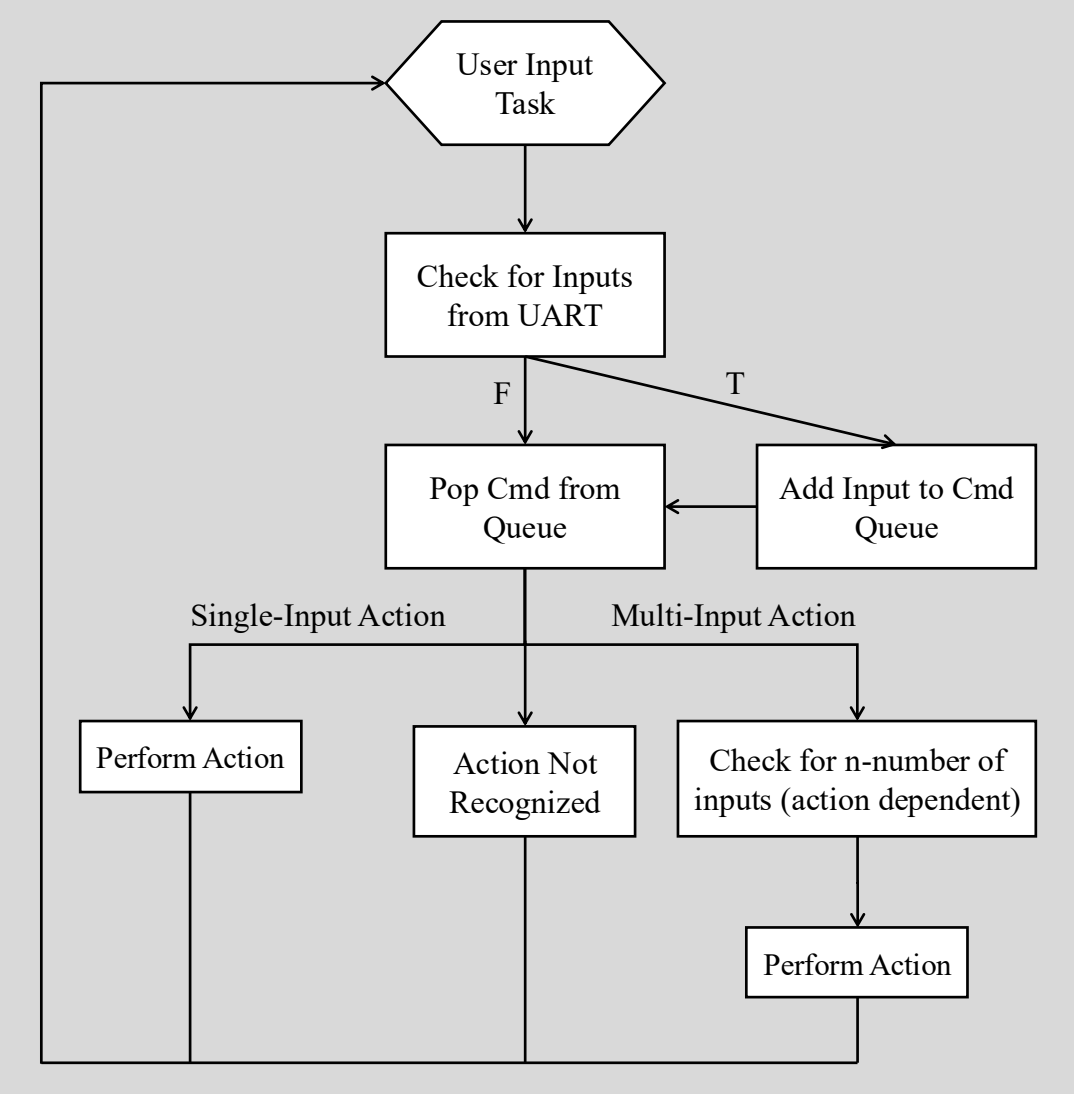

Figure 7. User Input Task Logic Diagram

The User Input Task receives user input through the Bluetooth UART port and performs the given command if one exists; otherwise, the command is ignored. User Input actions include, but are not limited to, linear control of Romi (WASD), updating any gain value in the Motor Controller's closed loop control, changing Path Director's current segment, and soft rebooting.

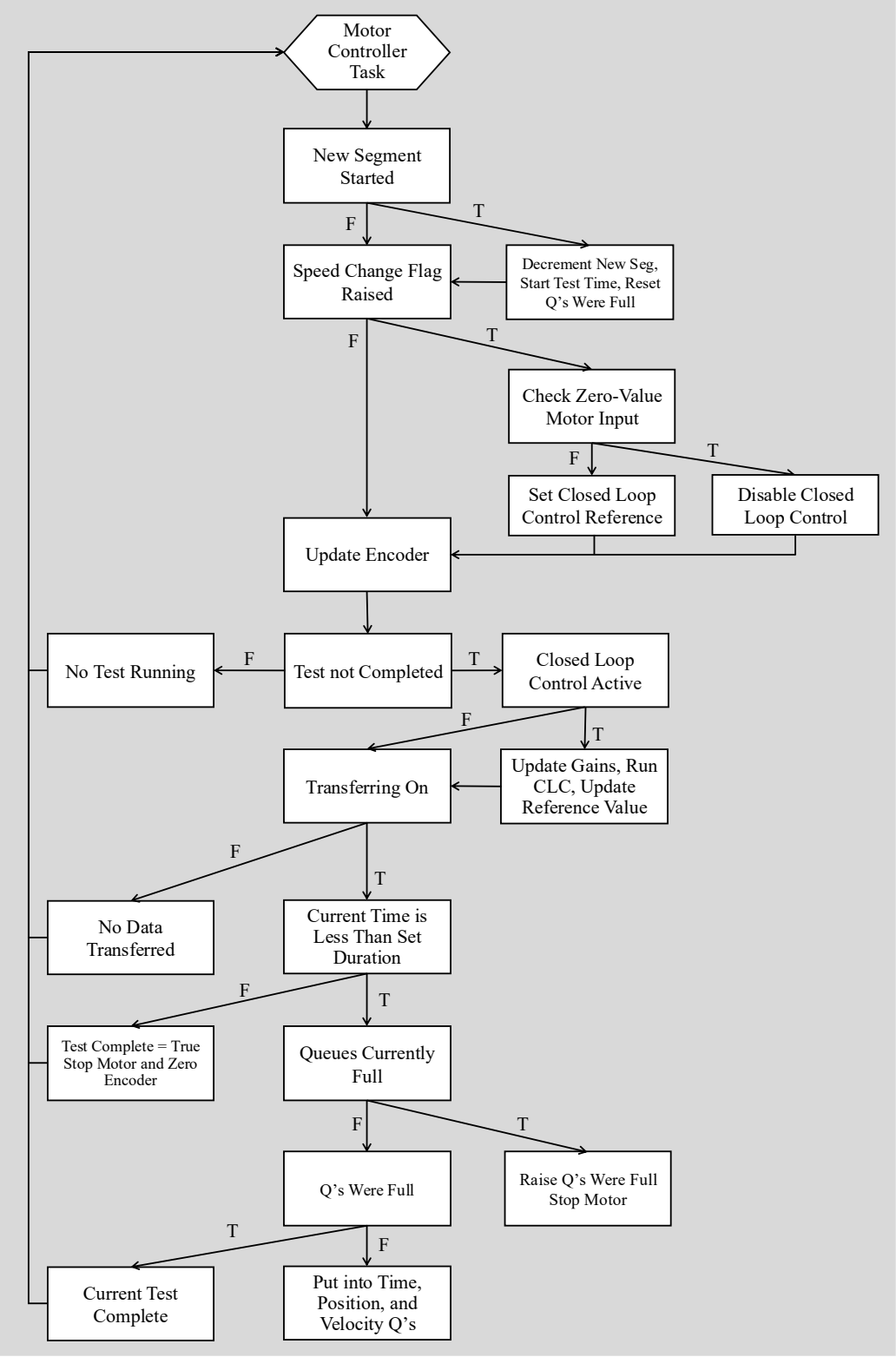

Figure 8. Motor Controller Task Logic Diagram

The Motor Controller Tasks control the speeds of the motor it is assigned to and can be changed via the appropriate speed share. The speed of each wheel is internally modified using closed-loop P+I control. It only places data into the queues if the data transfer flag is raised and a test is running.

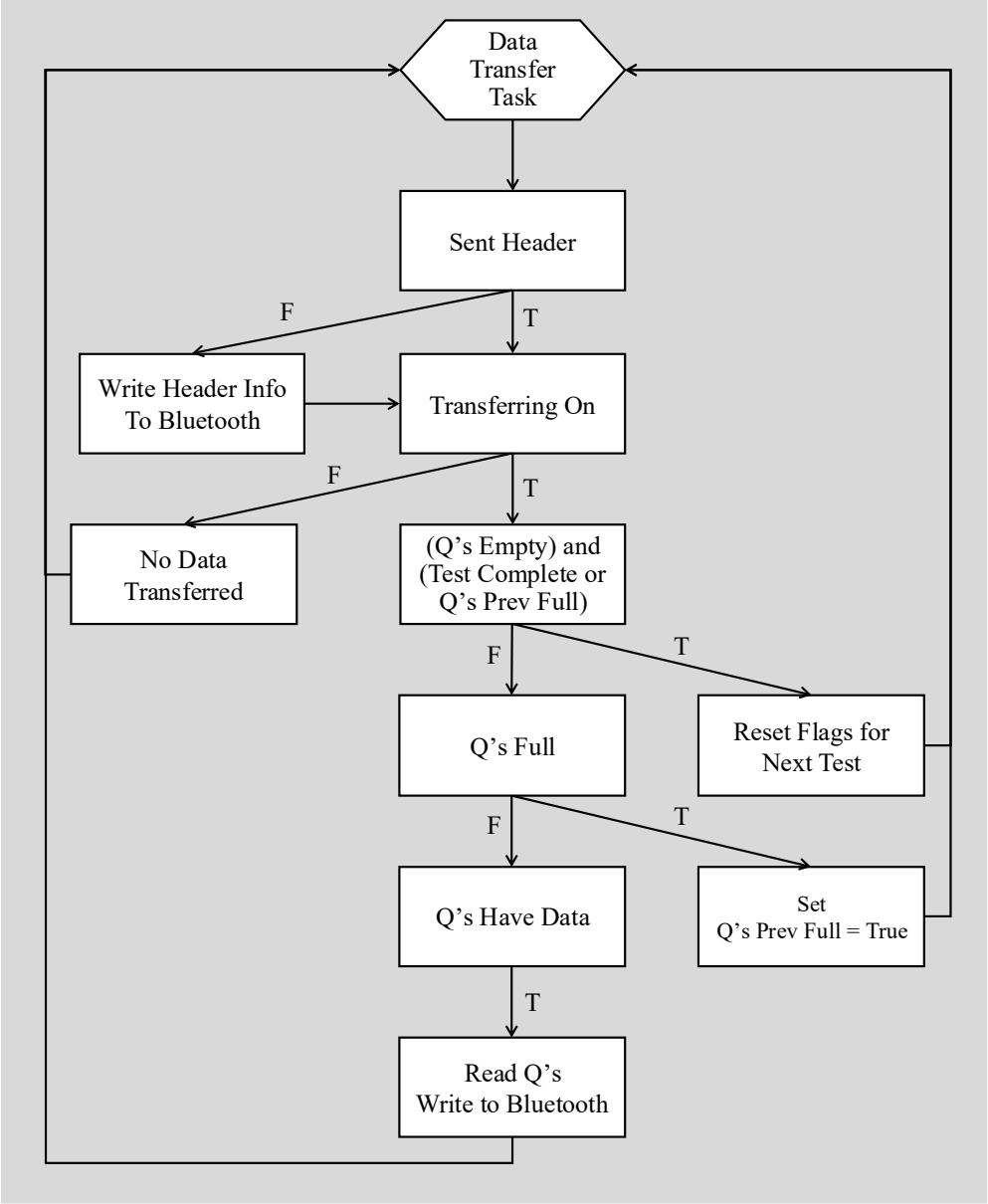

Figure 9. Data Transfer Task Logic Diagram

The Data Transfer Task transfers data from the Romi across Bluetooth through the use of the Romi's UART port to the host of the Bluetooth connection. If this is the first transfer of the test then the headers are transmitted ensuring each tests data is clearly separated. A data transfer flag determines whether the data should be transmitted across Bluetooth or not. If the data transfer flag is raised and the queues contain data then one data point from all queues are transmitted simultaneously.

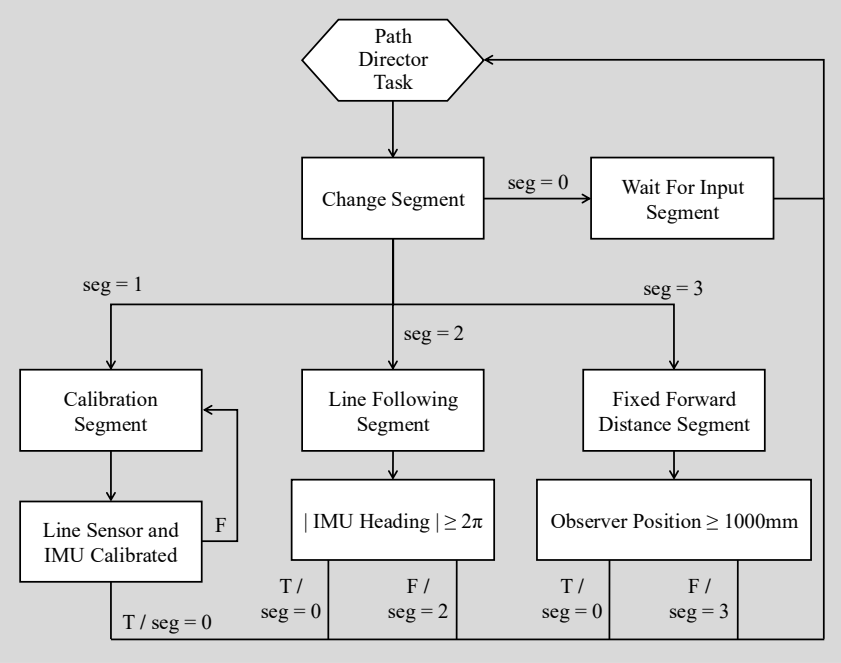

Figure 10. Path Director Task Logic Diagram

The Path Director Task directs Romi's behavior according to the segment it is in. Segment 0 is the waiting state where it idles until a user input forces it into one of the other segments. Segment 1 is the calibration state that will enact blocking code until the line sensor and IMU is calibrated. Segment 2 currently performs closed loop control on the line centroid data, performing line following. The line will be followed until the Observer estimates Romi has performed an entire revolution. Segment 3 drives Romi forward at a fixed velocity until the Observer estimates it has reached a certain distance (1000 mm). Overall, Path Director will be utilized to "segmentize" the obstacle course to perform unique behaviors for each distinct portion of the course.

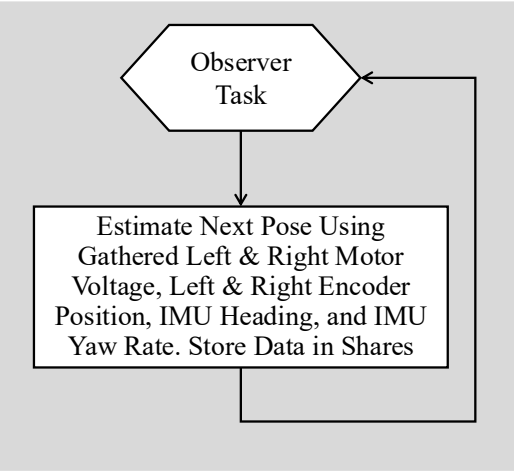

Figure 11. Observer Task Logic Diagram

The Observer Task utilizes a discretized linear model of Romi's state and input variables to best estimate the current position and orientation (pose). Utilizing shares, the estimated pose from the Observer is used in Path Director to change segments.

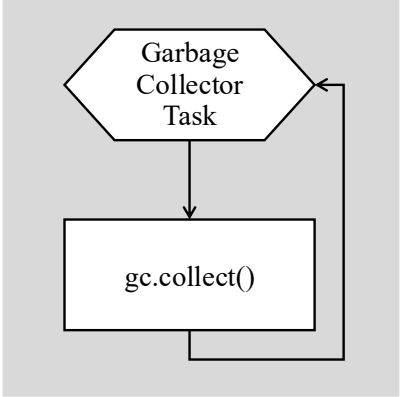

Figure 12. Garbage Collector Task Logic Diagram

The Garbage Collector Task runs in the free time between the higher priority tasks previously noted. The only purpose of the task is to defragment as much of the memory as possible to ensure as much of the data is stored in memory efficiently.

All new relevant files that facilitate the above functionalities are listed in Appendix A.

References:

- Refvem, C. (2025). *Lab 0x05 Closed Loop Control* [Unpublished lab instructions]. Department of Mechanical Engineering, Cal Poly. Retrieved from https://canvas.calpoly.edu/courses/161863/assignments/1405866
- Refvem, C. (2025). *Lecture 17 – State Feedback* [Unpublished lecture notes]. Department of Mechanical Engineering, Cal Poly.
- Refvem, C. (2025). *Lecture 18 – Observer Design* [Unpublished lecture notes]. Department of Mechanical Engineering, Cal Poly.
- Refvem, C. (2025). *Lecture 19 – Discrete Systems* [Unpublished lecture notes]. Department of Mechanical Engineering, Cal Poly.

Appendix A: Python File Transcript

Romi Files

1. Battery.py
   - A class object for Romi's battery which allows other files to query the current battery percentage and voltage.
2. Closed_Loop_Control.py
   - An abstract closed-loop control architecture that requires a reference value and Sensor object. Depending on implementation, P, PI, or PID control can be enacted with windup, saturation, and battery droop corrections included, with each respective gain value capable of being set and changed at any time.
3. IMU.py
   - A handler class for the BNO055 IMU that supports register reads, partial-byte to full-word writes, and manages relevant register addresses internally. It also performs calibration in a blocking manner if a saved calibration data file does not already exist.
4. IR_Sensor.py
   - Each infrared sensor wired into Romi is an object of IR_Sensor, each holding the low-level MCU pin data. Each IR_Sensor object holds its own calibration data and only returns processed readings.
5. Line_Sensor.py
   - The line sensor class handles each infrared sensor, allowing for system-level actions such as calibration and centroid data calculation.
6. main.py
   - Sets up motor timers, motors, encoder, infrared sensors, line sensor, and IMU objects with their correct pins on the Nucleo and initializes the shared variables that allow the cooperative tasks to communicate. Instantiates User Input, Observer, Path Director, Motor Controller, Data Transfer, and Garbage Collection task objects and registers them with the cotask scheduler. Runs the priority scheduler in a loop, and on any exception stops the motors, clears data queues, prints task diagnostics, and re-raises the error.
7. Observer.py
   - A discretized linear model of Romi's state and input variables to best estimate the current position and orientation.
8. Path_Director.py
   - A partitioning of Romi's path into "segments" such that Romi can switch between unique behaviors with ease. Ultimately, it will be used to segmentize the final obstacle course so that each unique portion can be approached differently.
9. Romi_Props.py
   - Holds important Romi properties so repeat calculations and variables can be avoided across classes and files.
10. Sensor.py
    - An abstract base class for all sensors used on the Romi platform. It provides a unified interface that seamlessly integrates any sensor type into the closed-loop control architecture.

11. User_Input.py
    - Initializes a UART connection, buffers incoming single-character commands into a queue. The run co-routine pulls the user input from the queue, echoes commands, and performs the respective hardcoded routine. Inputs can either correspond to single-input actions, which will execute upon their respective keystrokes, while the multi-input actions will not execute until the adequate amount of inputs are received. User Input can be used to physically control Romi, update gain values for Motor Controller's closed loop control, or change Path Director's current segment.