49329 – Control of Mechatronic Systems

Project Group 0 – Vision-Based Robotic Sorting System (VBRSS)

Authors: Abhijay Sekhar Choudhury & Jason Stewart

User Guide

Contents

1. System Overview

This project implements a vision-based robotic sorting system built around a modelled 3-axis Cartesian gantry robot. Two control strategies are provided:

1. Trajectory-Smoothened (Smooth+FF/TS) PID Model – in `robotic_sorting_system.slx`

- Control: PID feedback + feedforward with 5th-order polynomial trajectories
- Sorting paths follow 156 pre-computed smooth trajectories (proactive strategy)
- Performs with greater accuracy at the cost of longer cycle time, as presented in the accompanying report.

2. Direct/discrete PID (DPID) Model – in `dpid_robotic_sorting_system.slx`

- Control: PID feedback only with discrete waypoints
- Sorting “paths” follow real-time position calculation (reactive strategy)
- Performs with shorter cycle time at the cost of poorer accuracy and success rate, as presented in the accompanying report.

System Architecture (in Simulink) consists of the following subsystems:

- **Vision System**: Generates random dish types, provides the signal `dish_type` (to Database and Sorting Logic) from `dish_idx` (which also goes to Database). Placeholder noise I/O nodes have been introduced but have not been developed through in the current build (13th October 2025).

- **Database**: Maintains persistent states (`Area1_Types, Area1_Counts, Completed_Types`) via feedback loops and memory blocks. Inputs are `dish_idx` (unit delayed by 1 from the State Machine output), `dish_type` from the Vision System, `destination_slot` & `destination_area` from Sorting Logic, and `action_trigger` from Session Manager (which tells the Database each time the system has placed a dish). Outputs are sent to Sorting Logic.

- **Sorting Logic**: Determines dish destination based on dish type and current database state. Outputs are sent to simultaneously update the database state to the session's current state, and to set the State Machine up to calculate the target position for the session's current dish.

- **State Machine**: Three critical loops are managed by this subsystem: the one between Database, Sorting Logic and itself, the loop between Robot Control, Position FB and itself, and the loop between the Session Manager and itself (Session Manager as mentioned above also loops with the Database subsystem to communicate when a dish is placed, which itself is determined by the State Machine's Robot Control-Position FB loop – this is a relatively complicated system with interrelated dependencies that enable its function). The 9-state controller coordinates the entire workflow, where target position (and velocity and acceleration for the TS model) tell Robot Control where it needs to "go", tracks dishes going to buffer and disposal, and returns home when the Session Manager reads that a session has completed sorting 64 input dishes.

- **Robot Control**: 3-axis PID controllers (TS: +feedforward; DPID: feedback only). Inputs are streams that tell the robot where to go, and a loop is formed between Robot Control, Position FB and the State Machine to determine whether the target position has actually been reached or not (within tolerance). Once `position_reached==1` (and the time elapsed is enough to allow for a computed trajectory's duration to complete in the TS model), the State Machine progresses to the next state.

- **Position FB**: Calculates tracking error and checks whether this is within the tolerance mentioned above (in the simulations' distributed state this value is set to 0.5 (mm), which means that `position_reached==1` if the tracking error is within this tolerance).

- **Session Manager**: Tracks progress, manages database updates and triggers completion state once `dish_idx > 65` according to the rules set out in state transitions to the COMPLETE state in the State Machine, and if `position_reached==1`.

- **Accuracy Analyser**: Computes running `mean_error` and `success_rate` metrics for performance, debugging and comparative analysis workflows.

This list above is generalised to account for the functionality of both DPID and TS models. For complete system block diagrams for both models, please see Appendix B: Simulink Model Diagrams.

2. Installation & Prerequisites

The models were both developed on MATLAB R2025a and Simulink. It is recommended this version of MATLAB is used when running scripts and simulations.

Toolboxes:

- Simulink
- Control System Toolbox
- Optimization Toolbox (for Bayesian optimisation of motor parameters)
- Statistics and Machine Learning Toolbox (for Bayesian optimisation of motor parameters)

To ensure simplicity and functionality, it is recommended that all files in the distribution are saved in the same folder. Additional recommendations are to disable OneDrive (or iCloud if on an Apple device) and/or to have this working directory saved in a folder or drive that is not connected to a cloud service. In testing, some issues occurred with running simulations which were likely caused by these cloud services. If you want to have it saved on OneDrive to port it to another computer, once files are synchronised on the new computer, save them to an offline directory especially if runtime issues occur when running Simulink.

Files distributed (also referred to as the "package" in this document – from shared compressed file):

```
init_robotic_sorting.m                  # TS model initialisation
dpid_init_robotic_sorting.m             # DPID model initialisation
robotic_sorting_system.slx              # TS Simulink model
dpid_robotic_sorting_system.slx         # DPID Simulink model

configure_model.m                       # Fix configuration issues if models don't run

process_after_simulation.m          # Process results (generates database_state.mat 1st run)
check_state.m                       # View database state
visualise_state.m                   # Plot state graphs

compare_system_versions.m           # Compare TS vs DPID, manually enter motor parameters
measure_performance.m               # Extract metrics (function)
generate_performance_report.m       # Generates report .txt, code WIP – needs manual entry of
                                    time taken to run session
optimise_bayesian.m                 # Run Bayesian optimisation of motor parameters
resume_bayesian_optimisation.m      # Resume optimisation run if interrupted
analyse_bayesian_results.m          # Analyse optimisation results (code still WIP)
```

Note: Some scripts require manual reconfiguration to work on a different model, e.g., `optimise_bayesian.m`. Lines of code that require changing are specified at the top of each script this

condition pertains to. Also, a keen eye may notice a `clear sim` line in each init script. This is a remnant from a deprecated build that handled `sim.dt`, `sim.session_dishes` and `sim.total_sessions` when the models were intended to run multiple sorting runs in a simulation until the allotted time-steps finished. Instead, now a fixed maximum of 64 dishes are sorted, and the simulation stops automatically once this condition is reached, and can occur before the allotted time-steps are used up. These variables are still used to display to the user the configuration of the initialisation, before they are cleared so they don't interfere with other functions in the MATLAB environment such as launching simulations via CLI.

To verify MATLAB version, in the CLI input `ver`. To verify that key functions exist use `which sim` and `which bayesopt`. To check files are in your current working directory use `dir *.m` and `dir *.slx`

3. Quick Start

Before running any scripts or simulations, ensure proper directory structure with all files in the same folder as listed in section 2. Especially if the initialisation scripts are in a different directory to its respective Simulink file, variables won't be found. If the Simulink model is run before running the init script, red indicators occur and the simulation fails to run. If there are multiple copies of the model on the path (including in parent, child or branched directories), shadow copy errors may occur and cause unpredictable behaviour. If the model is run from the incorrect directory, a "Model not found" error will occur. For routine running of the Simulink models, the required workflow is as such:

1. cd to project directory
2. Verify single copies of files only
3. Run init script for the model you wish to run
4. Run simulation (either in CLI or by opening the model in Simulink and clicking the green "Play" button – both methods work)
5. Process results (and visualise and/or check database state)

The following example CLI code allows you to follow this workflow (for MATLAB CLI).

```
% Extract ALL files to a SINGLE directory

% Navigate MATLAB to that directory

cd 'C:\path\to\RoboticSorting'

% Verify you're in the correct location

pwd % Should show your project directory

% Check files are present

dir *.slx % Should show both .slx models

dir *.m % Should show all .m scripts

% Check for duplicate copies

which('robotic_sorting_system.slx', '-all')

% Should return ONLY ONE path

% If multiple paths appear, see Troubleshooting -> Path Shadowing
```

Different workflows are possible for the various files sent as part of this package. The following example CLI inputs pertain to running a single simulation, comparing both models, and conducting multiple sessions of a single model at a time respectively (note: `database_state.mat` has not been validated to work for concurrent multiple-session runs of both models at the same time, and processing after simulation of one model may overwrite the simulation data that would be processed into the `database_state.mat` file for the other model if this is performed. It is advised to only run multiple sessions in a row of one model at a time for database persistence demonstration purposes).

Scenario 1: Single simulation (here for Trajectory Smoothing):

```
% Step 1: Initialise

init_robotic_sorting

% Step 2: Run simulation (specified to allow full run parameters: 64 dishes in ~260 s)

sim('robotic_sorting_system','StopTime',‘300’)

% Step 3: Process results (Required immediately after simulation)

process_after_simulation

% Follow input prompt to manually process completed types if any slots are full

% Step 4: View state (optional)

check_state

% Step 5:

visualise_state
```

NB: Always run the simulation that matches the initialisation file you ran. Running the wrong simulation after initialisation will cause errors and/or incorrect behaviour (as there are key differences between the two as to trajectories being calculated for one model and not the other, most notably). When switching between models, re-run the correct init script first. It is also advised to run:

```
clear all; clear sim; close all; delete('database_state.mat');clc
```

in CLI before running a different model. Distribution contains `database_state.mat` for quick demo purposes of `check_state.m` and `visualise_state.m`.

Scenario 2: Comparing both models

```
% Runs both models with manually configurable test parameters (in distribution copies these are
set to equal the ones determined best according to Bayesian parameter optimisation)

compare_system_versions % You can ignore warnings for size [0x0] and [], sims will still run

% Generate .txt report

generate_performance_report

% Output saved to: Performance_Report.txt

% Manual adjustment of cycle time must occur from reading time taken from the scope for the sim
to complete, and hence the time taken to sort each dish should be manually fixed too
```

Scenario 3: Running multiple sessions for a given model:

```
% Session 1
init_robotic_sorting
sim('robotic_sorting_system','StopTime',‘300’)
process_after_simulation % Creates database_state.mat


% Session 2 (Database persists from Session 1)
sim('robotic_sorting_system','StopTime',‘300’)
process_after_simulation % Updates cumulative totals, session 2 sim appends to existing database


% View complete history
check_state
visualise_state
```

Note on Test Mode: Test mode (`test_mode=1; max_dishes_test=X`) is deprecated for distribution, and was used during simulation development to debug subsystem functionality with reduced dish counts. For system stability and compatibility, the distributed models use constant blocks configured for normal operation (64 dishes per session). To run shorter tests, simply stop the simulation early by calling `StopTime` using a smaller number in CLI, or manually entering a smaller number in Simulation -> Stop Time in Simulink.

Note on init scripts: There currently exists some code to load in an existing `database_state.mat` files to assign Area 1 (sorting area) counts and types and completed types data. This does not currently work in full. It is redundant for the scenario three workflow anyway, but remains in case future development necessitates this functionality with how the database may interact with the simulation environment.

Custom parameters may be set in MATLAB CLI between clearing the previous workspace and before running the model's init script by setting `test_*` values for damping, Kp, Ki and Kd for x-, y- and z-axes.

4. Simulink Configuration

The Simulink models were developed and validated on the following settings for their Solver and Data Import/Export Settings:

Solver Settings:

- Stop time: set to 300 (overestimation to allow for all dishes to be sorted), reconfigurable.
- Type: Fixed-step, required for consistent timestep
- Solver: ode4 (Runge-Kutta)
- Fixed-step size: 0.001, 1 ms sample time (sim.dt in init scripts).

If using variable-step solvers, simulation may fail or produce incorrect results. Must use fixed-step ode4.

If settings are found to be different, or if there are issues running the simulation, run `configure_model.m` in the same directory as the .slx files in the distribution to apply all required settings.

Data Import/Export Settings:

- Time: Checked check-box for `tout`, exports time vector
- Output: Checked check-box for `yout`, exports signals
- Signal logging: Checked check-box for `logsout`, enables scope data export
- Data stores: Checked check-box for `dsmout`, exports database state

These settings enable `process_after_simulation.m` to extract data from the sim workspace.

"Default for underspecified data type" was set to "double" in "Configuration Parameters" -> "Math and Data Types", and here "Simulation behaviour for denormal numbers" was set to "Gradual Underflow".

Models were developed on a Windows 11 OS, Intel x86-64 (Windows64) PC, and should work on Mac and Linux, though this has not been validated.

Common warnings include:

- "No supported compiler found": This is only needed for code generation, not simulation. Click "Don't show again". It could also imply that the Simulink model is in Accelerator mode.
- "SLCC Exception" warning: Model is set to Accelerator mode.

Fix for Accelerator mode, in MATLAB CLI run:

```
set_param('robotic_sorting_system', 'SimulationMode', 'normal')
save_system('robotic_sorting_system') % change prefix to "dpid" for that model
```

5. Understanding Scope Outputs

Table 1: Summary of all scope outputs in each model.

Signal	Range/Values	Description
current_state	0-8	State machine: 0=IDLE, 1=SELECT_DISH, 2=MOVE_TO_PICK, 3=READ_LABEL, 4=DETERMINE_DEST, 5=PICK, 6=MOVE_TO_PLACE, 7=PLACE, 8=COMPLETE
dish_idx	1-65	Current dish (65 = all 64 complete)
destination_area	1,2,3	1=Sorting (Area 1, 16 slots), 2=Disposal (Area 2, 7 slots), 3=Buffer (Area 3, 16 slots)
reason_code	1-5	1=Previously completed, 2=Matching type, 3=Just completed, 4=New type, 5=No slots
buffer_count	0-64	Cumulative dishes to buffer
disposal_count	0-64	Cumulative dishes to disposal
target_position	3 traces	X (0-1300 mm), Y (0-800 mm), Z(0-300 mm) target positions inside workspace
current_position	3 traces	Actual robot position (follows target with lag, error calculated from difference in mm)
position_reached	0,1	0=moving,1=at target (within tolerance)
gripper_cmd	0,1	0=open, 1=closed
mean_error	mm	Running average tracking error. TS: ~2.4 mm, DPID: ~58.2 mm from testing
success_rate	%	Placement accuracy. TS: 100%, DPID: ~91%
tr_error X/Y/Z	mm	Instantaneous tracking error per axis

The workspace configuration is summarised accordingly:

- X: 0-200 mm (Input), 200-900 mm (Sorting/Buffer), 1100-1300 mm (Disposal)
- Y: 0-800 mm (Input/Disposal), 0-400 mm (Sorting), 400-800 mm (Buffer)
- Z: 50 mm (Placement), 100 mm (Pick/transport), 250 mm (Safe transit)

Example screenshots of the scopes can be found in Appendix A, with accompanying interpretation and insight from the examples.

6. Database Management

Each simulation generates data to the workspace, and `process_after_simulation.m` processes this data into the `database_state.mat` file needed to visualise and check for completed types between simulation sessions.

`database_state.mat` has the following contents:

- `Area1_Types`: [1x16] vector – Dish type in each slot (0 = empty)
- `Area1_Counts`: [1x16] vector – Number of dishes in each slot (maximum of 10)
- `Completed_Types`: [1x100] vector – History of completed types (only a size of 27 is needed)
- `cumulative_buffer`: Total dishes sent to buffer (all sessions)
- `cumulative_disposal`: Total dishes sent to disposal (all sessions)
- `session_count`: Number of simulation runs completed
- `total_removed`: Dishes manually processed (completed types)

The database is created upon the first call of the `process_after_simulation.m` script, and persists between runs where further simulation calls have their data added to the existing database once the processing script is run again. The user has the option to select 'y' to manually process dish types that have reached a maximum of 10 counts per type, and that information is preserved in the database where said dish type is recategorised to a 'completed' type. Selecting 'n' when prompted by the processing script leaves the completed type in the slot unprocessed, and will remain there for subsequent runs of the simulation until the user manually processes it. There are 27 types in total for the given workspace, numbers ranging from 111 to 333 where each digit can vary from 1 to 3. Future instances of completed types (whether or not they have been removed from the workspace) are sent to disposal by the sorting system (according to the Sorting Logic).

7. Performance Analysis

Key metrics used in the performance analysis conducted using the scripts `compare_systems_versions.m`, `measure_performance.m` and `generate_performance_report.m` are:

- Success rate: Percentage of dishes placed within tolerance.
- Mean error: Average Euclidean distance between target and actual position during motion.
- RMS error: Squares errors before averaging to emphasise errors more than mean to detect systems with occasional large spikes rather than consistently having small errors.
- Maximum error: Worst-case tracking error during the entire session to identify peak deviation for setting safety margins.
- Standard deviation: Measures the variability or consistency in tracking error. A low stdev equates to consistent performance and a higher one shows erratic behaviour.

- 95th Percentile error: Presents the typical worst case, where only 5% of errors exceed the result. Mainly used to determine quality control thresholds.
- 99th Percentile error: A tighter measure to determine 1% failures, and used for conservative safety margins or analysis of rare events.

Analysis and discussion of the system's findings can be found in the accompanying report.

Benchmarks from model development and testing:

```
================================================================================
DETAILED PERFORMANCE COMPARISON
================================================================================

ACCURACY METRICS:
Metric                              DPID     Traj Smoothing        Improvement
------------------------------ ------------------ ------------------ ------------------
Mean Error                       58.15 mm            2.37 mm            95.9%
RMS Error                        31.16 mm           10.53 mm            66.2%
Maximum Error                   850.07 mm          850.03 mm             0.0%
Standard Deviation               30.72 mm           10.50 mm                -
95th Percentile Error            25.00 mm            0.78 mm                -
99th Percentile Error           105.76 mm           13.35 mm                -

OPERATIONAL METRICS:
Metric                              DPID     Traj Smoothing             Change
------------------------------ ------------------ ------------------ ------------------
Success Rate                      91.2%              100.0%               8.8%
Cycle Time                        72.7 sec           255.9 sec          183.2 sec
Time per Dish                     1.14 s/d           4.00 s/d           2.86 s/d
Dishes Completed                  64/64              64/64                 -
Completion Rate                   100.0%             100.0%                -
```

Figure 1: Output into `Performance_Report.txt`, with cycle time and time per dish manually corrected for the actual time needed to sort 64 dishes for each model. Bayesian-optimised parameters were used in `compare_system_version.m` before this information was computed.

8. Parameter Optimisation

Bayesian optimisation of PID parameters and damping were performed and the workflow can be done by using the scripts `optimise_bayesian.m`, `resume_bayesian_optimisation.m` and `analyse_bayesian_results.m`. The method builds a surrogate of the objective function using a Gaussian process with acquisition function Expected Improvement Plus (EI+). This algorithm was built with inspiration from Optuna's TPE algorithm for neural network hyperparameter optimisation.

The parameter space is summarised as such (12 in total):

- Base X-axis (4): damping_x: [40, 150] N*s/m, Kp_x: [50, 400], Ki_x: [50, 600], Kd_x: [2, 25]
- Axis ratios (8):
    - Y/X ratios: damping_y_ratio [0.4, 0.8], Kp_y_ratio [0.8, 1.2], Ki_y_ratio [0.8, 1.2] and Kd_y_ratio [0.8, 1.2].
    - Z/X ratios: damping_z_ratio [0.2, 0.6], Kp_z_ratio [1.0, 1.6], Ki_z_ratio [1.2, 2.0] and Kd_z_ratio [1.2, 2.0].

Ratios were used to maintain physically meaningful axis relationships and to help in faster convergence. A scoring function was used to give incomplete runs a penalty of 5000, and poor success rate (<50%) a penalty of 2000, with a lower number in the objective function being better:

$$score = mean\_error\ + (100 - success\_rate) * 3 + penalty$$

Running the optimisation, in MATLAB CLI run `optimise_bayesian.m`, which runs for an extended period of time as 50 trials (in the script's default state) are completed, each time running the simulation. Since it can take so long to obtain results, the script has functionality to save checkpoints such that if an error occurs mid-run, the `resume_bayesian_optimisation.m` script may resume from the checkpoint .mat file that is generated by the first script. Output from the optimisation script is a set of 12 optimised parameters, which can then be input into the respective model's initialisation scripts or comparison scripts (or manually reconfigure the PID blocks in Simulink) to run simulations with the optimised parameters. To change the optimiser to work on the DPID model after optimising the TS model, for example, there is instructional commented code at the top of the script to manually reconfigure the script to work on the other model:

```
%% optimise_bayesian.m
% Bayesian optimisation with axis ratio parameters
% Similar to Optuna's TPE algorithm but using MATLAB's bayesopt and EI+

% Manually reconfigure this script to optimise parameters for either dpid_
%   or "non-dpid" models by changing prefix before init and sim calls
% Lines 17, 150, 237 and 238 are where such prefix changes would occur
```

Figure 2: Screenshot of the parameter optimiser script mentioning where to manually change lines of code to work between different models.

Optimised parameters are saved in database `bayesian_optimisation_results.mat`, which may be loaded into the workspace to set `test_*` variables to match the optimised ones, as exemplified in the following CLI code:

```
% Load results
load('bayesian_optimisation_results.mat')
% Set test parameters
test_damping_x = optimal_params_bayesian.damping_x;
test_damping_y = optimal_params_bayesian.damping_y;
test_damping_z = optimal_params_bayesian.damping_z;
test_Kp_x = optimal_params_bayesian.Kp_x;
test_Kp_y = optimal_params_bayesian.Kp_y;
test_Kp_z = optimal_params_bayesian.Kp_z;
test_Ki_x = optimal_params_bayesian.Ki_x;
test_Ki_y = optimal_params_bayesian.Ki_y;
test_Ki_z = optimal_params_bayesian.Ki_z;
```

```
test_Kd_x = optimal_params_bayesian.Kd_x;
test_Kd_y = optimal_params_bayesian.Kd_y;
test_Kd_z = optimal_params_bayesian.Kd_z;
% Initialise and run
init_robotic_sorting
sim('robotic_sorting_system', 'StopTime', '300')
```

The optimised parameters as determined during development and testing of the Simulink models are distributed for convenience, though it is recommended that new users of this package run the script themselves to determine best parameters in case they prove to be system-dependent. Some variance will occur in performance metrics resultant from the presented optimised parameters due to the random nature of input dishes being generated by the Vision System in the Simulink models.

9. Troubleshooting

If models don't run, after opening MATLAB R2025a and changing to the relevant working directory, run `configure_model.m` to ensure Simulink workspace **settings** are set up properly (ode4 Solver, fixed step of 0.001).

An error encountered during development was in MATLAB when running the **processing** script, "Simulation data not found". This occurs when `process_after_simulation.m` is run without a simulation. To fix, ensure the model in question is initialised according to its init script, and run it before being able to process the data to `database_state.mat`.

Path shadowing occurs when model files of the same name are saved in different directories off from the working directory. If there are multiple paths to the model file you are working with (after checking with the `which` command on your model file in MATLAB CLI, addressed earlier in this document), do `restoredefaultpath`, then `addpath` to the correct directory.

Red block indicators in Simulink means that the init script for said model has not been run yet. Ensure the initialisation script for the matching model has been run before the simulation is run.

If the **simulation doesn't complete 64 dishes**, this could mean the simulation timed out before getting to `dish_idx = 65`. Increase `StopTime`, and if that doesn't work, check if the robot gets stuck by checking the State Machine during a simulation (if it stops at a certain block then the issue can be diagnosed further), or check for oscillation in the `current_position` scope. Increasing damping should decrease oscillations, and increasing Kp helps the system if it's too slow (simplified explanation). A lot of these nuances should be dealt with by running the simulation with optimised parameters, however. If the State Machine gets stuck, enable debugging mode with:

```
set_param('robotic_sorting_system', 'SimulationCommand', 'start', 'SimulationMode', 'normal', 'Debug', 'on')
```

just before running the simulation (or put into the init script).

10. Appendices

Appendix A:

Example scope block screenshots (all from the first run of the TS model on optimised parameters):

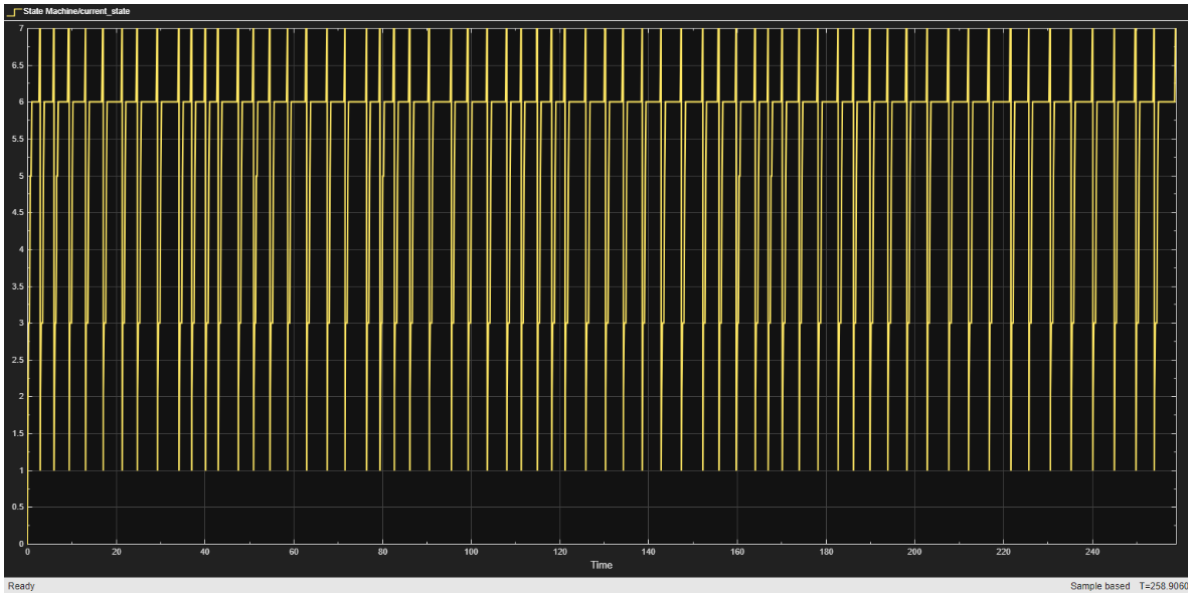


Figure A.1: current_state scope displays the state the robot is in as per their definitions in the state machine as the simulation runs. Note: current_state: 8 for completion is missing from the scope as the sim ends before the scope has time to display it.

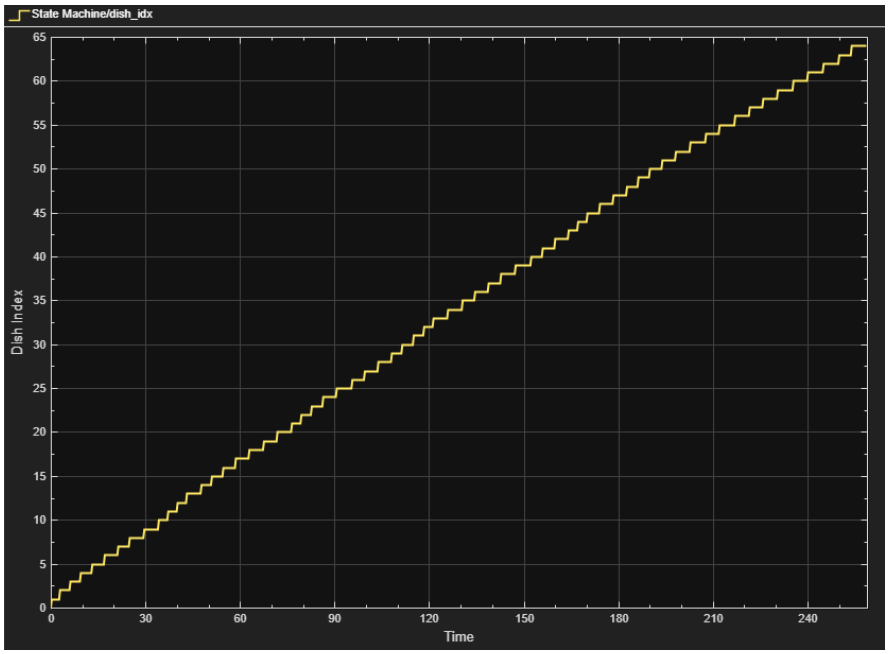


Figure A.2: dish_idx, range from 1 to 65. Increments by one dish to signify the one being processed (idx 65 = all 64 complete). Shorter steps signify a nearby destination from the input stack, and longer steps signify a longer distance. Variance in step length is expected, as is represented here.

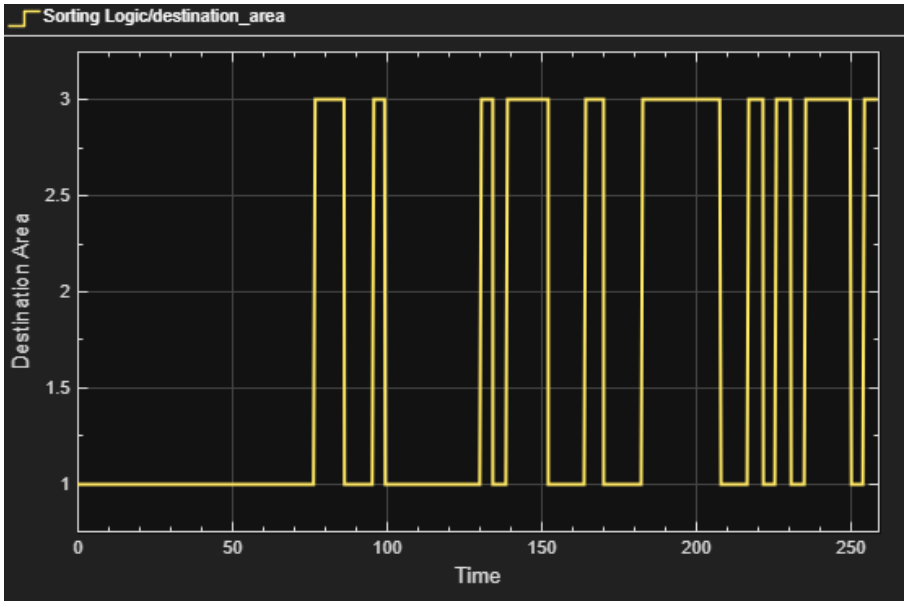


Figure A.3: destination_area: Screenshot from the first session shows no dishes going to Area 2 (disposal), which is expected. Area 1 = sorting = 1, and Area 3 = buffer = 3.

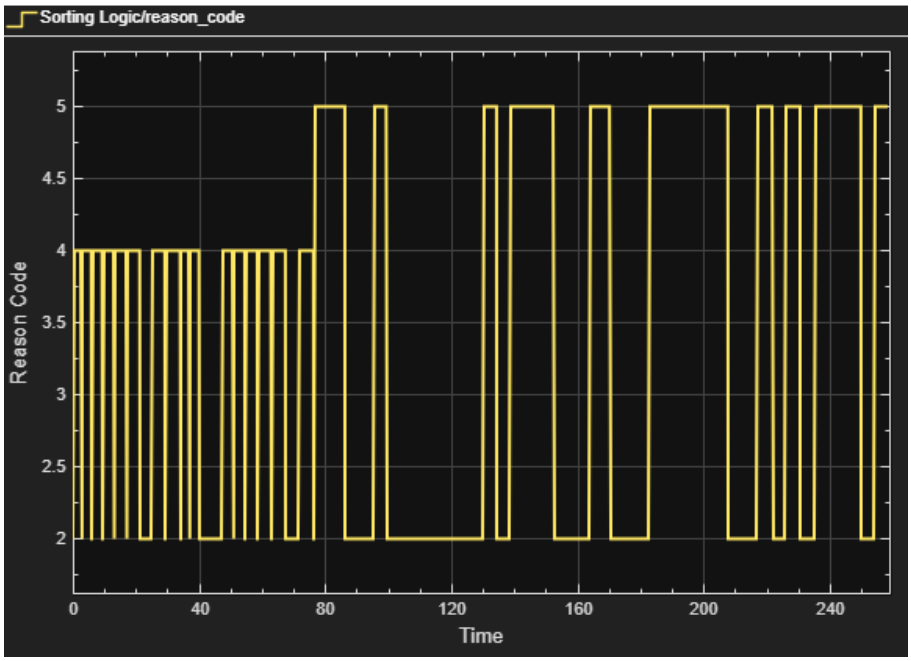


Figure A.4: reason_code: 1=send to disposal, 2=add to existing slot, 3=send to disposal, 4=assign to new slot in area 1, 5=no slots available, send to buffer.

Figure A.5: Cumulative buffer_count increments each time a dish goes to Area 3.

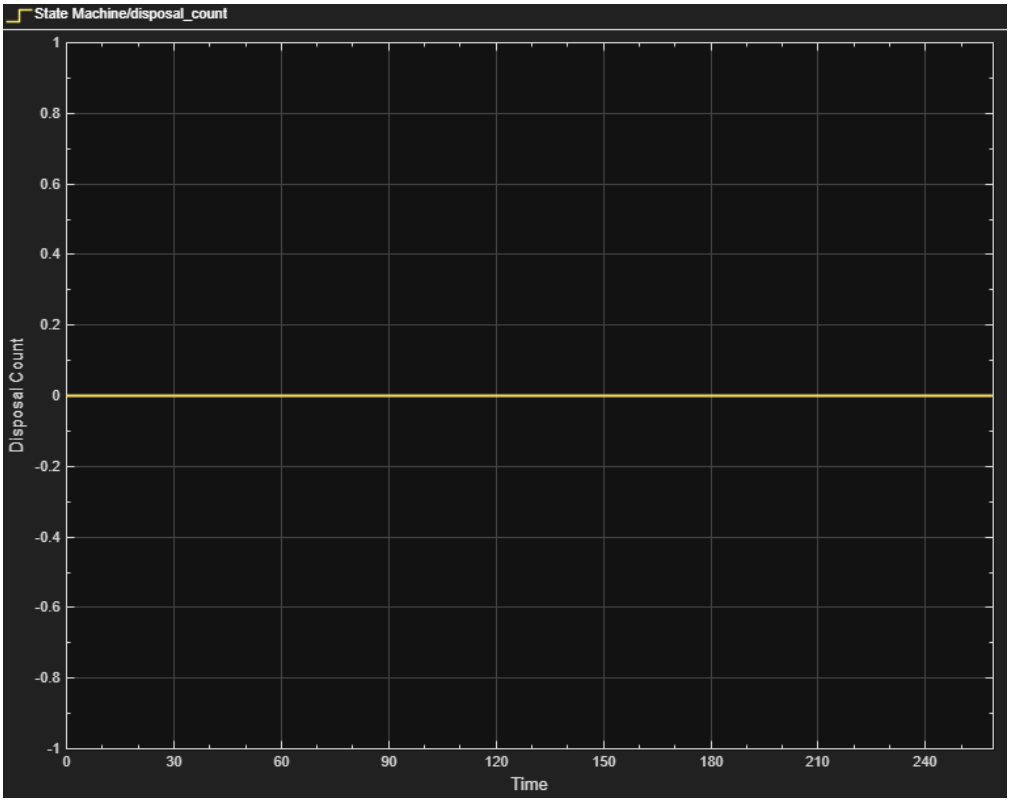


Figure A.6: Cumulative disposal_count increments each time a dish goes to Area 2 (expected to be 0 on the first session).

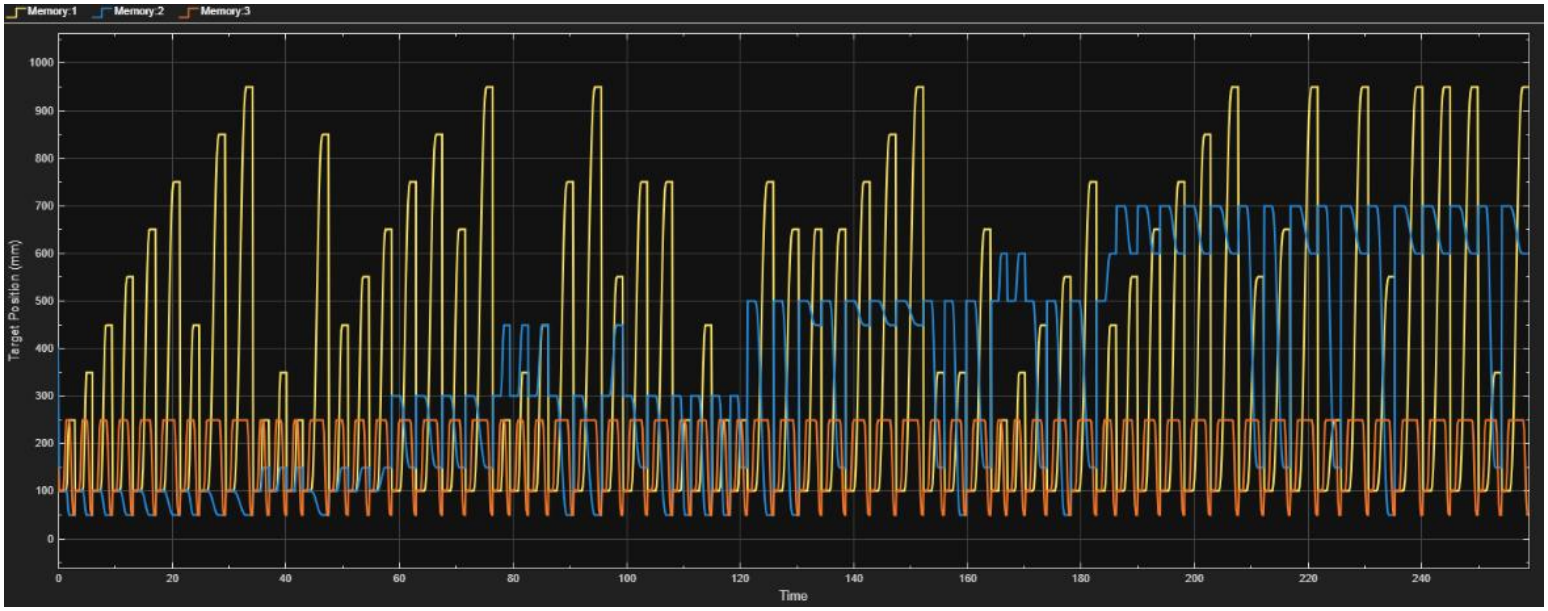


Figure A.7: target_position (memory signals, 3 traces), X is yellow, Y is blue and Z is orange. X: 0-200 mm is input (Area 0), 200-900 mm for Areas 1 and 3, and 1100-1300 mm for Area 2. Y: 0-800 mm correspond to Areas 0 and 2, 0-400 mm for Area 1, and 400-800 mm for Area 3. Z: 0-300 (up/down), 50 mm placement height, 100 mm pick/transport height, 250 mm, safe transit height. Further work needs to be done to fine-tune picking up variable heights within an input stack as the level dishes are at would realistically go down as more are collected by the robot arm.

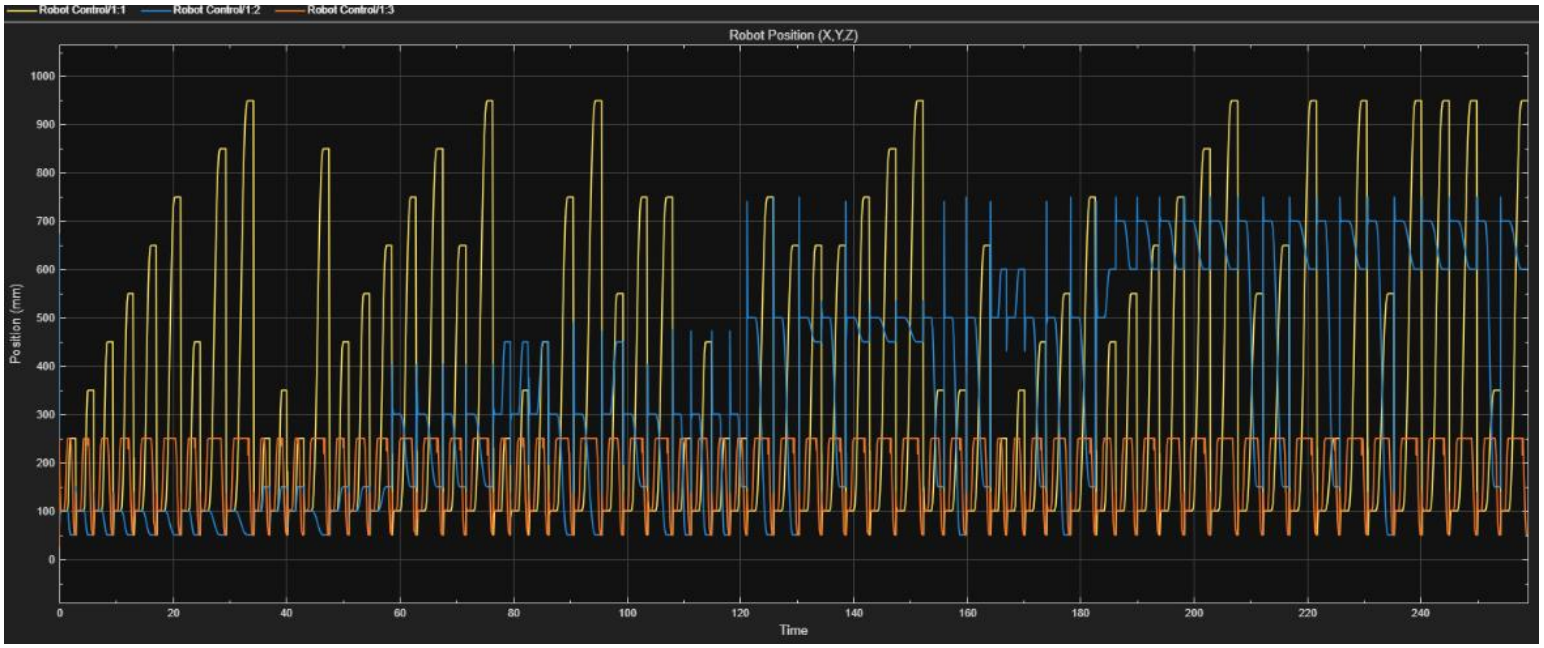


Figure A.8: current_position: follows target with small lag. Here, shows some overshoot in Y, which could be from suboptimal parameters (e.g., Kd), or resultant from difficult constraints for the robot to work with.

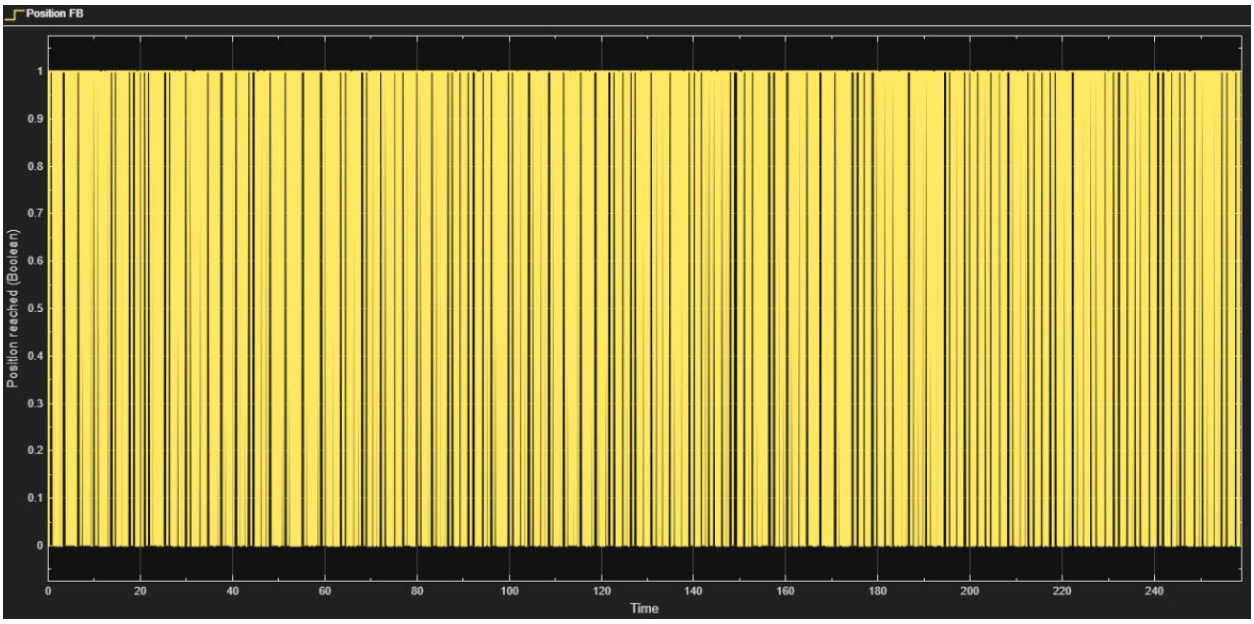


Figure A.9: Position FB (feedback): Tells us when the robot has actually reached the target position, used in feedback control of the robot control system and also to tell the system when to finish the session. the robot needs to have reached the target position at the end of the run before the session finishes as managed by Session Manager.

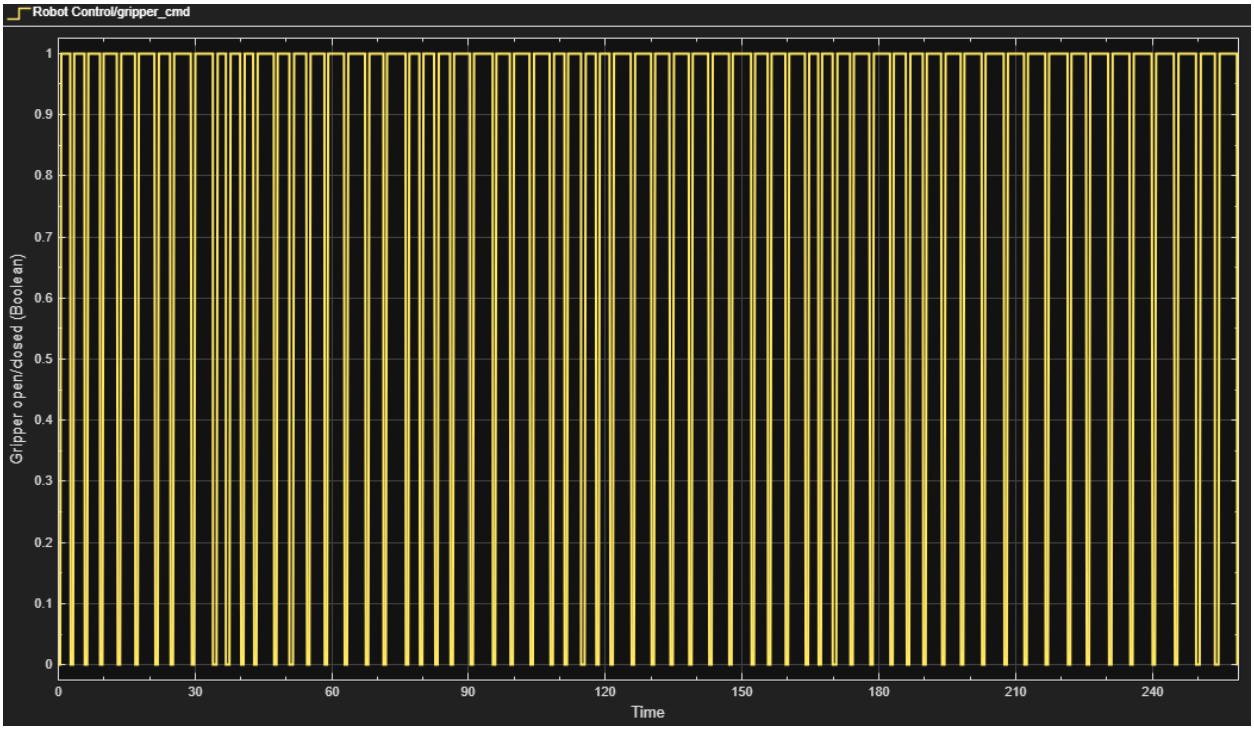


Figure A.10: Opens and closes according to state machine computation depending on where the robot is and whether it has reached the correct position for the sorted dish as computed by the Sorting Logic subsystem.

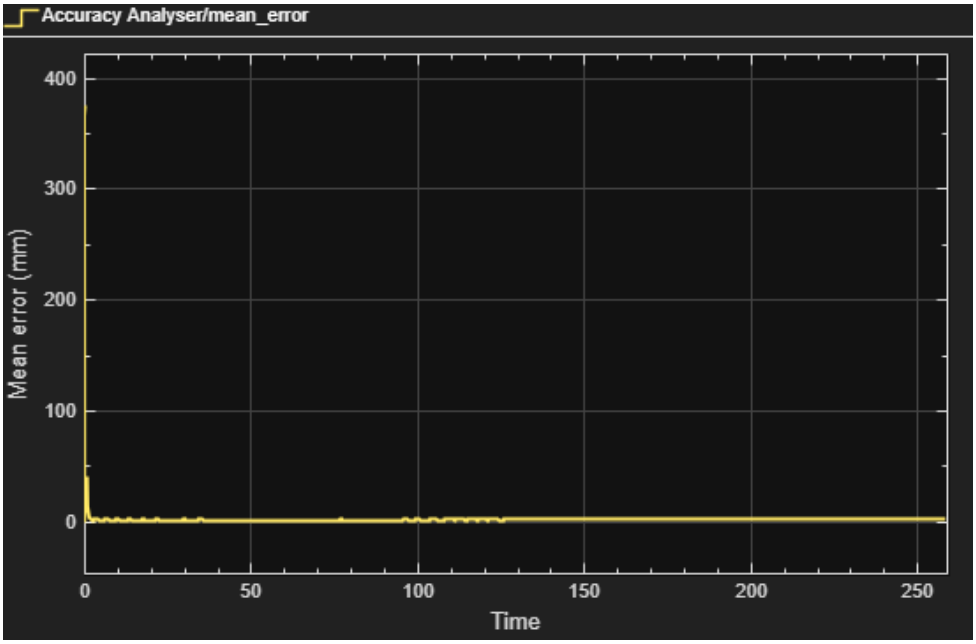


Figure A.11: Scope showing the mean error: running average tracking error of the robot's motion.

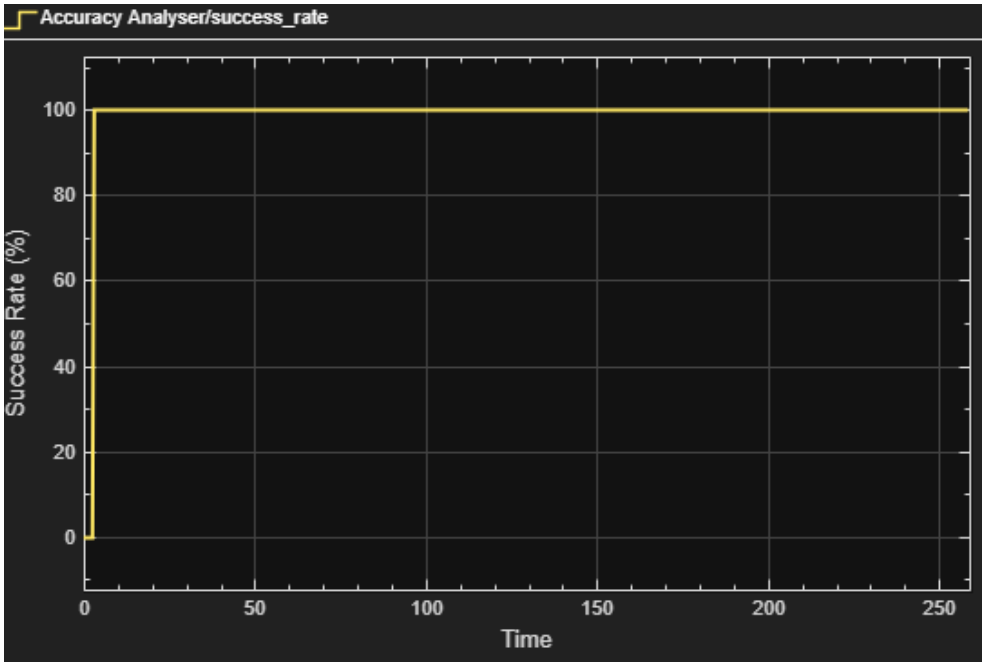


Figure A.12: Success rate: Running percentage of successful placements.

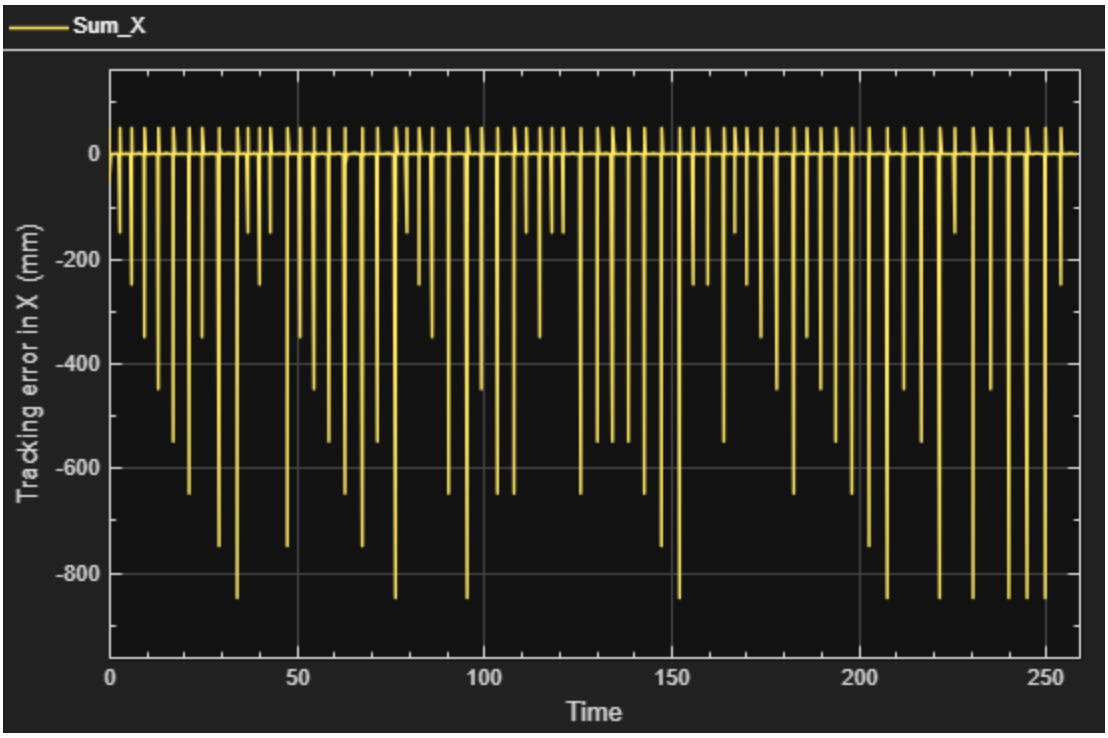


Figure A.13: Tracking error in X (mm): Real-time error that feeds into PID controller. Shows large initial error of 850 mm, which occurs no matter the control strategy. TS model accounts for this error better than the reactive DPID strategy, as seen by the comparative performance metrics.

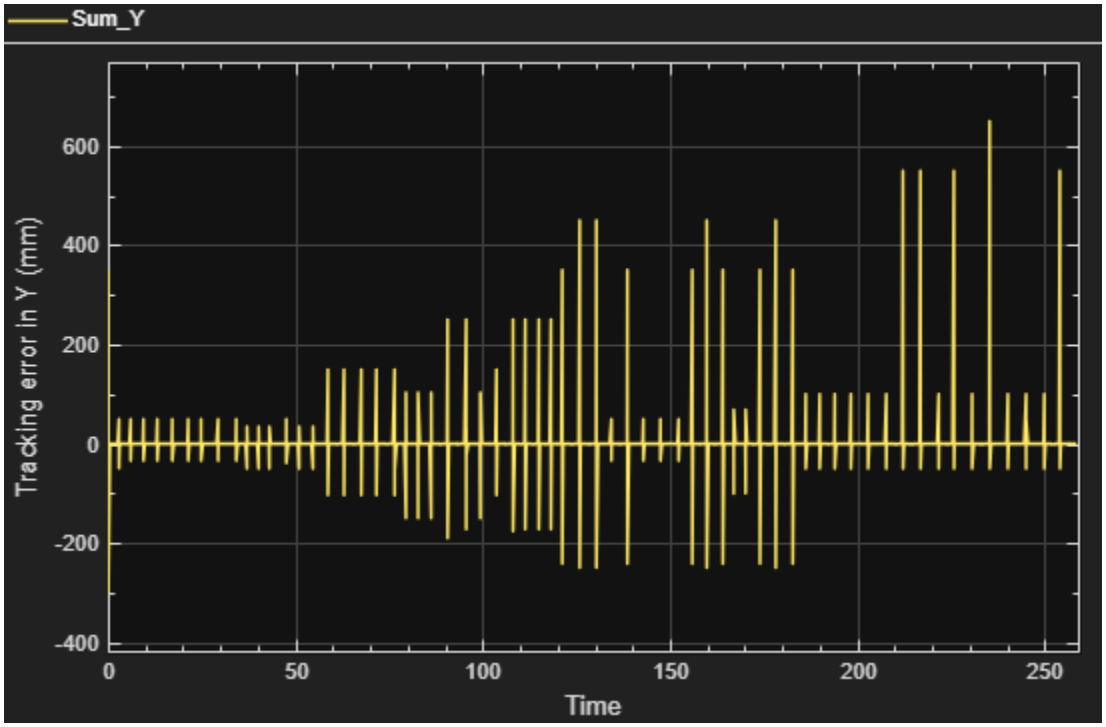


Figure A.14: Tracking error in Y (mm).

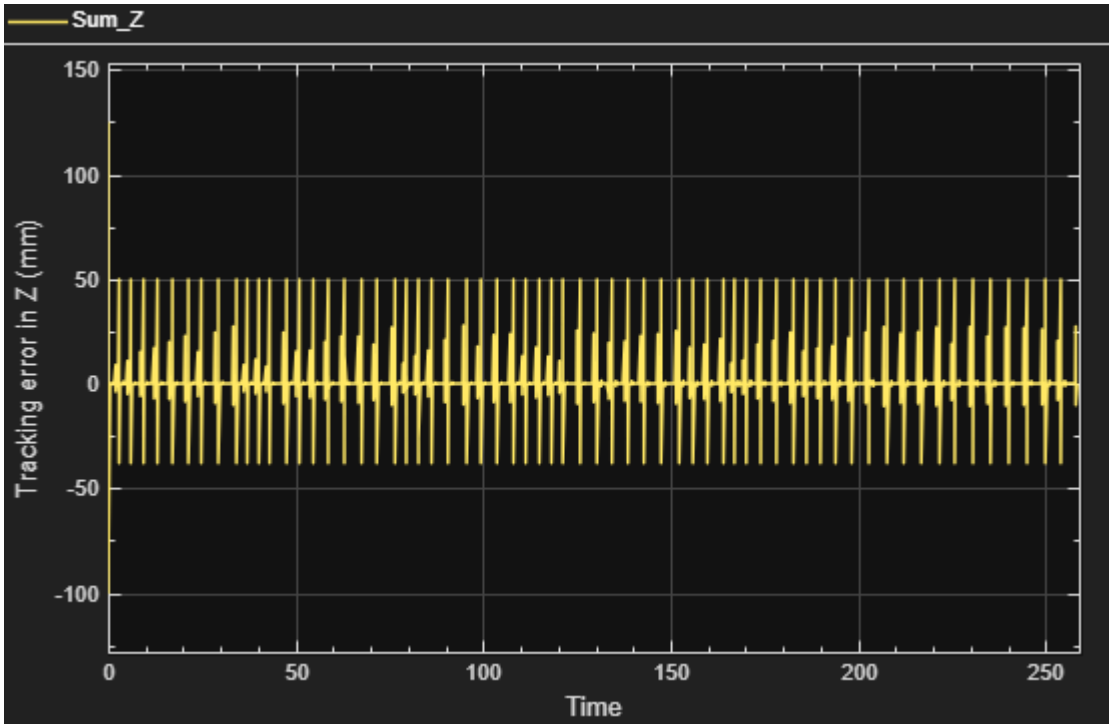
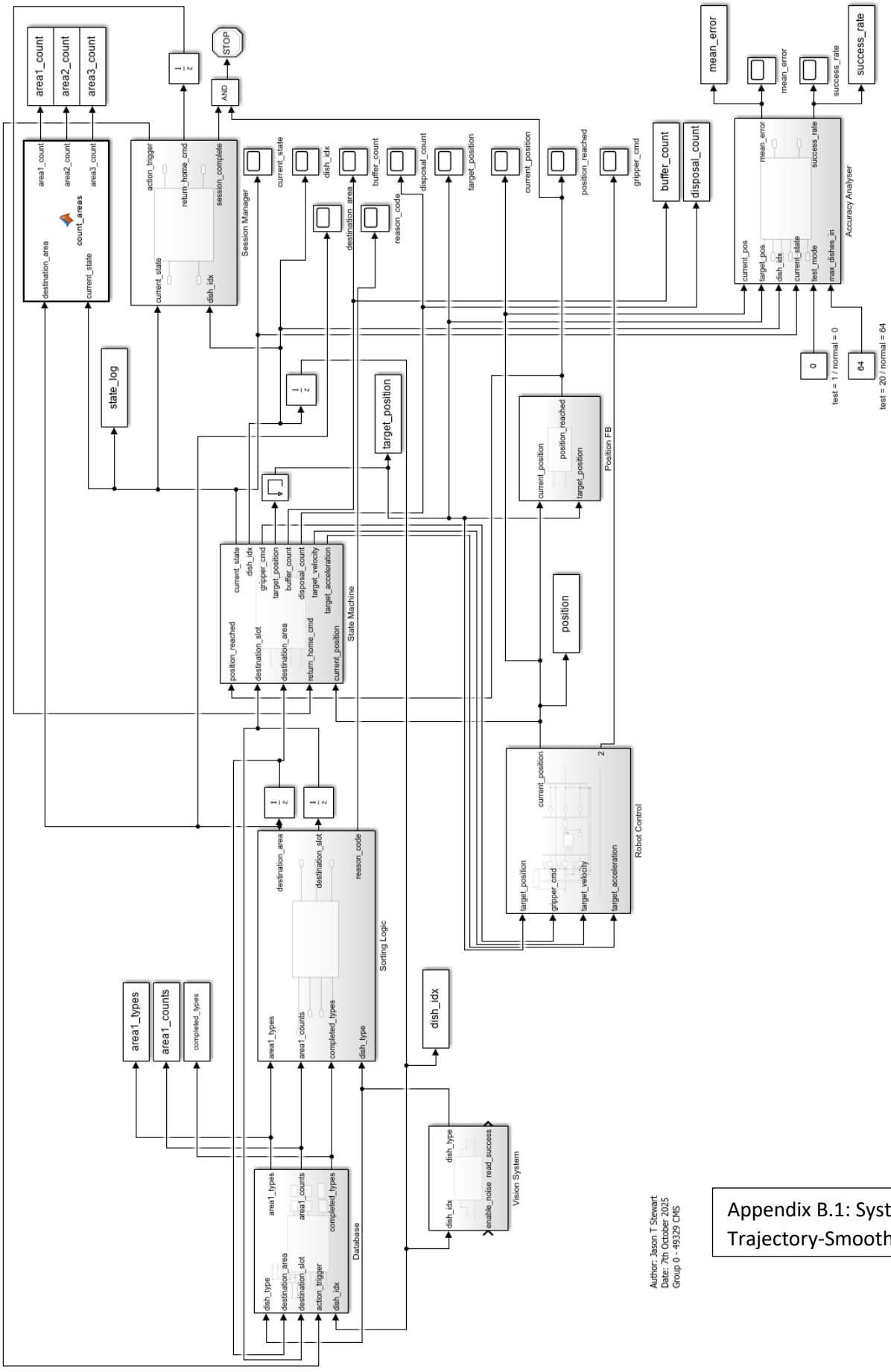
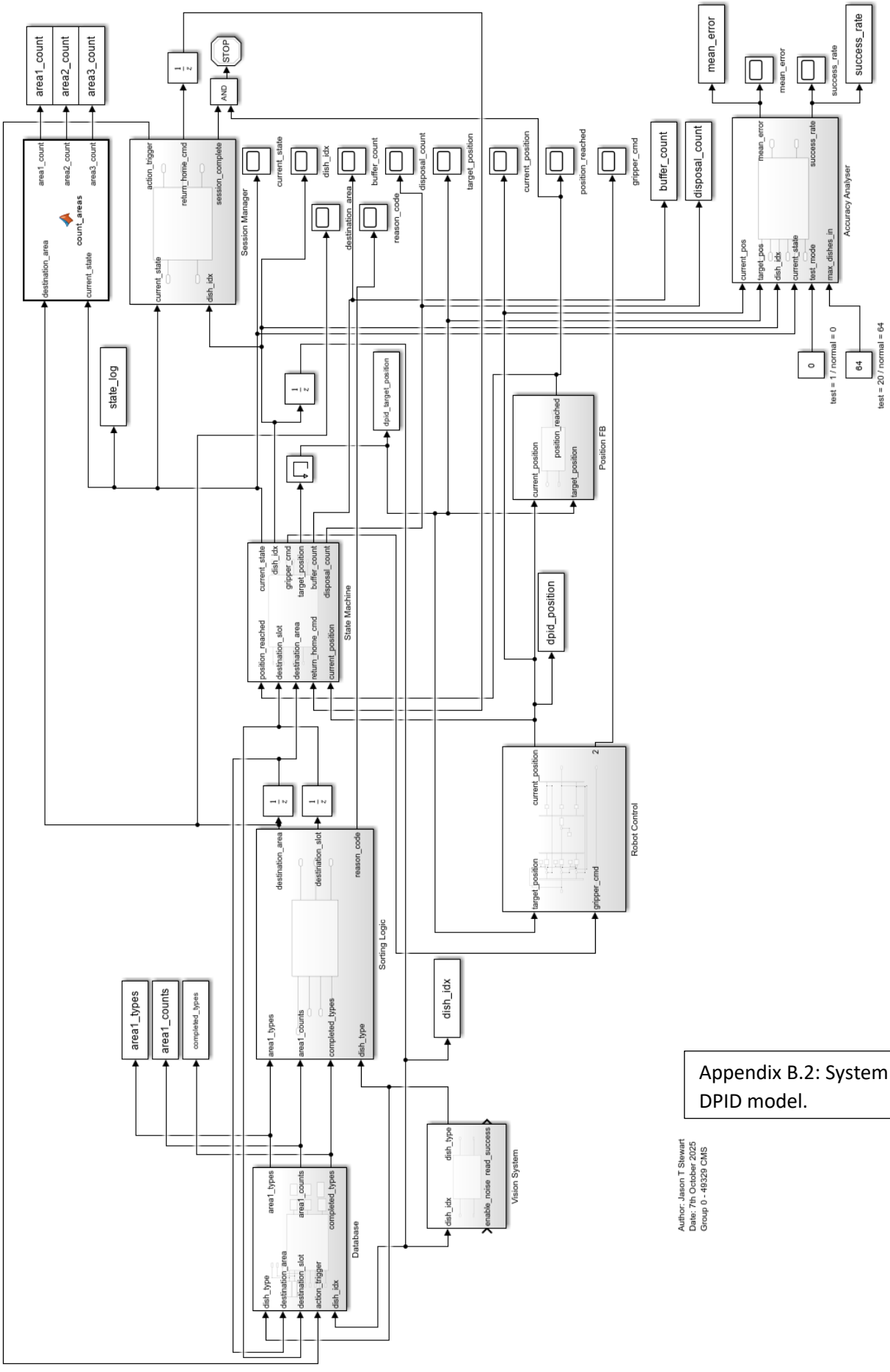


Figure A.15: Tracking error in Z (mm).

Appendix B (following page): Full system block diagrams from Simulink of each model.



Appendix B.1: System diagram of Trajectory-Smoothing (TS) model.



Appendix B.2: System diagram of DPID model.

Author: Jason T Stewart
Date: 7th October 2025
Group 0 - 49329 CMS