Trajectory Tracking Control for a Spherical Parallel Manipulator

Results Chapter

By

Oskar Durham
Bachelor of Engineering (Robotics)

ENGR7710A
College of Science and Engineering
7/12/2025

TABLE OF CONTENTS

LIST OF SYMBOLS

Symbol/Function	**Definition**
$\omega\ (\omega_x, \omega_y, \omega_z)$	Angular velocity of platform around the fixed XYZ axes
$\theta\ (\theta_y, \theta_p, \theta_r)$	Angular position in yaw-pitch-roll (YPR)
$\phi\ (\phi_1, \phi_2, \phi_3)$	Co-axial actuator angular position
$v\ (v_1, v_2, v_3)$	Set of SPM platform joint unit vectors
$w\ (w_1, w_2, w_3)$	Set of SPM intermediate joint unit vectors
$R_x(a), R_y(a), R_z(a)$	Standard rotation matrix around the X/Y/Z axes by angle a
$R_{axis}(u, a)$	Standard rotation matrix around a unit vector u by angle a
u_x, u_y, u_z	Unit vectors pointing in the positive X/Y/Z axis
$unit(h)$	Normalise vector h to a unit length
$anglevec(h_1, h_2)$	Calculate minimum angle between two vectors h_1 and h_2
$arctan2(y, x)$	Inverse tangent function within a range of $\pm\pi$

LIST OF ABREVIATIONS

Abbreviation	**Definition**
SPM	spherical parallel manipulator
PID	proportional-integral-derivative
EKF	Extended Kalman Filter
6-DOF	6 degrees-of-freedom
IMU	Inertial measurement unit
YPR	Yaw-pitch-roll (Euler angles)

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

A spherical parallel manipulator (SPM) is a device which produces rotational motion, around a stationary point (often called spherical motion). This project involves designing a low-cost stepper driven SPM with trajectory tracking control. The controller aims to use accelerometer and gyroscope feedback to correct the trajectory for the real-world case where the kinematics of the manipulator are imperfect. In this way, the project responds to the research gap of real-world implementation of trajectory controlled spherical manipulators. The purpose of these results is to create a prototype for a motion simulator device, for testing of trailer brake controllers in partnership with REDARC Electronics.

2. METHODOLOGY

For this experiment, a mechanically robust platform needed to be manufactured, and then tested for its accuracy using an external measurement, to get a ground truth for angular position.

2.1 Design Methodology

The mechanical platform design aimed to improve of three key areas where the previous prototype fell short:

- The device needed better range of motion, needing to reach the required 40° slope outlines in the requirements.
- The slack in the entire system needed to be significantly reduced, if not eliminated, to be capable of reaching the performance requirements.
- The rigidity of the kinematic joints needed to be increased significantly to reduce resonance in the system at the highest frequency outlined by the performance requirements.

Other considerations for the new design included the centre of rotation, which was also required to be positioned 1cm above the platform surface to ensure minimal linear acceleration of the test object. And the platform diameter, which was set to a minimum of 200mm to hold the necessary test objects. The equipment used for creating the design prototype included 3D printing, laser cutting and assorted hand tools for assembly.

2.2 Experimental Set-up

To measure the angular position of the device over time, a VICON motion capture system was used for high-fidelity motion tracking, as shown in Figure 1. The markers were attached to 3D printed mounts which ensured consistent placement across testing sessions, and the mounds were arranges in a non-symmetrical pattern, to allow the VICON software to resolve the platform orientation. The reference trajectory was sent in real-time to the device over a USB to UART connection, and the device was powered with 12V DC.

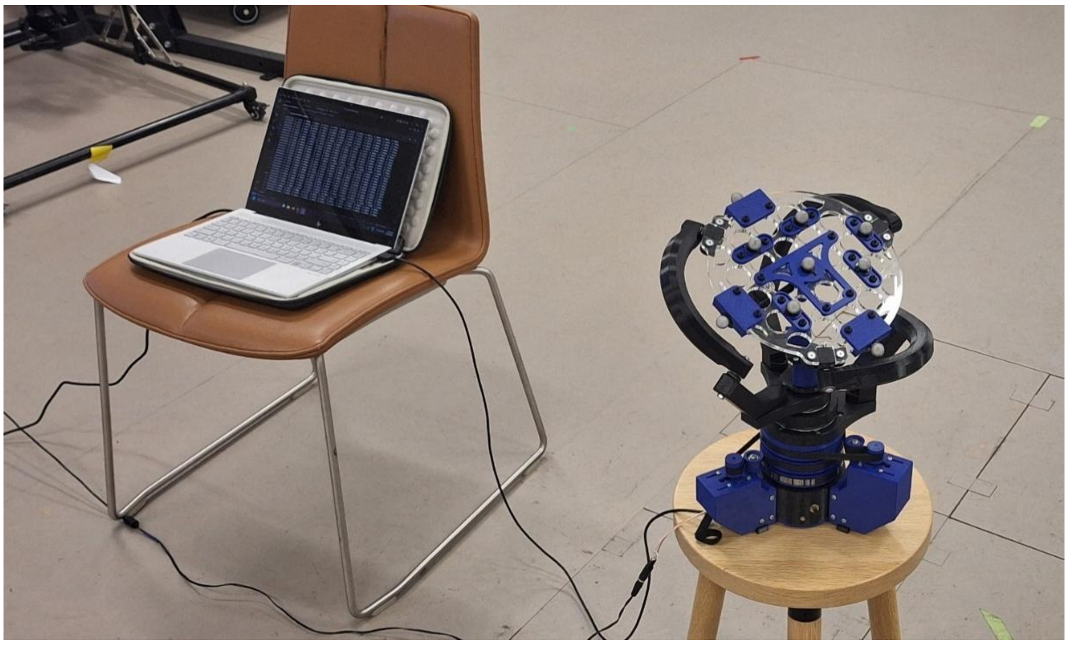

Figure 1. The SPM setup in the motion capture environment with motion tracking markers.

3. RESULTS

3.1 Design Outcomes

3.1.1 Mechanical Design

The kinematic paramaters of the device were changed from the typical $\alpha_1 = \frac{\pi}{4}, \alpha_2 = \frac{\pi}{2}, \beta = \frac{\pi}{2}$, to $\alpha_1 = 50°, \alpha_2 = 75°, \beta = 100°$. These values were chosen as they provided a minimum slope of ~43°, while keeping the length of the SPM arms at a reasonable length. With $\beta > 90°$, the platform joints were able to be places underneath the platform surface, making the platform completely flat while keeping the center of rotation above it, as shown in Figure 2.

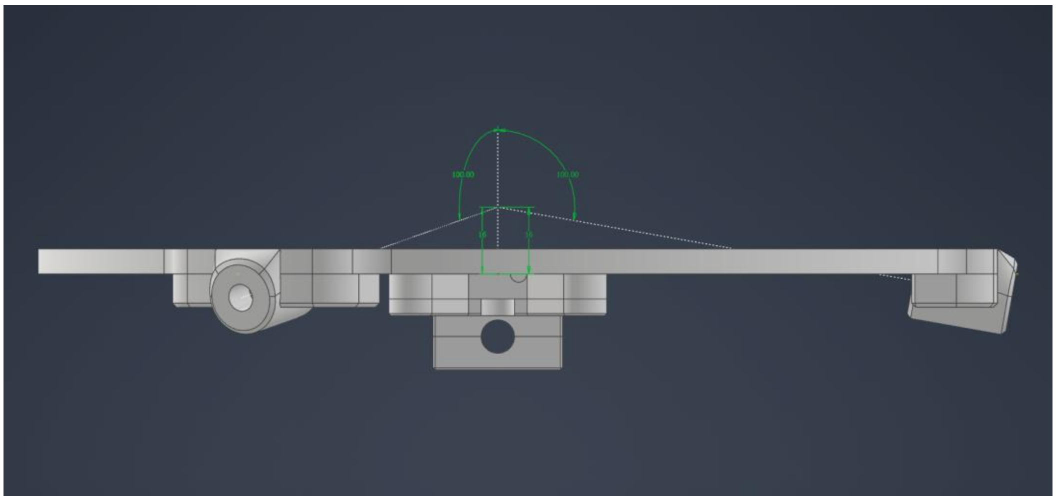

Figure 2. Manipulator platform, designed with adjusted kinematic parameters.

The device was manufactured using polycarbonate for the high-stiffness parts to ensure dimensional stability over the course of testing. While the original design used concentric planetary gearsets to achieve coaxial motion, this presented an issue of slack and wear, as there is no way to adjust planetary gears without replacing them. To eliminate slack in the system, the coaxial motion system was changed to a concentric pulley stack, driven by M3 belts. (See Figure 3).

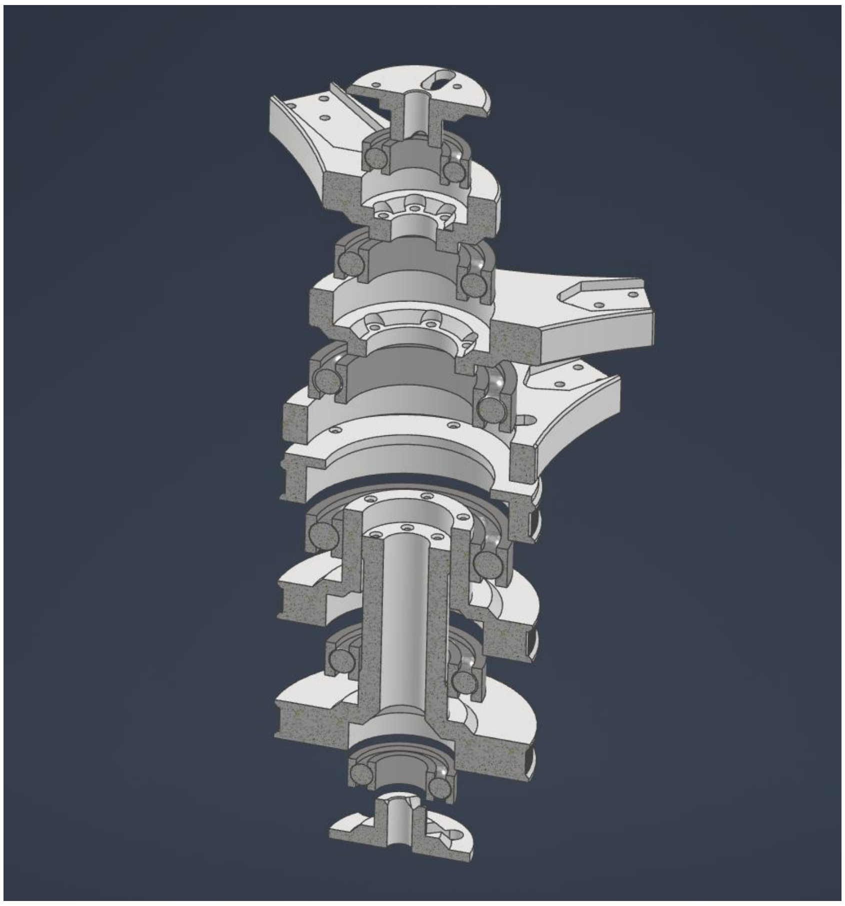

Figure 3. Cross section of the belt-driven concentric cylinder stack used for coaxial motion.

The weakest points for rigidity in the previous designs were identified as the rotating joints for the platform arms. Due to the parallel nature of the device, these joints experience both linear and rotational load. This can also be said for the new concentric pulleys which have three rotating joints inside one another. To ensure both rigidity in rotational and linear loads, as well as ensuring no backlash, they were designed as press-fit two-bearing joints. In all cases, a bolt is used to gently compress the joints, preloading the plastic bearing housing to further add stiffness, as shown in Figure 4.

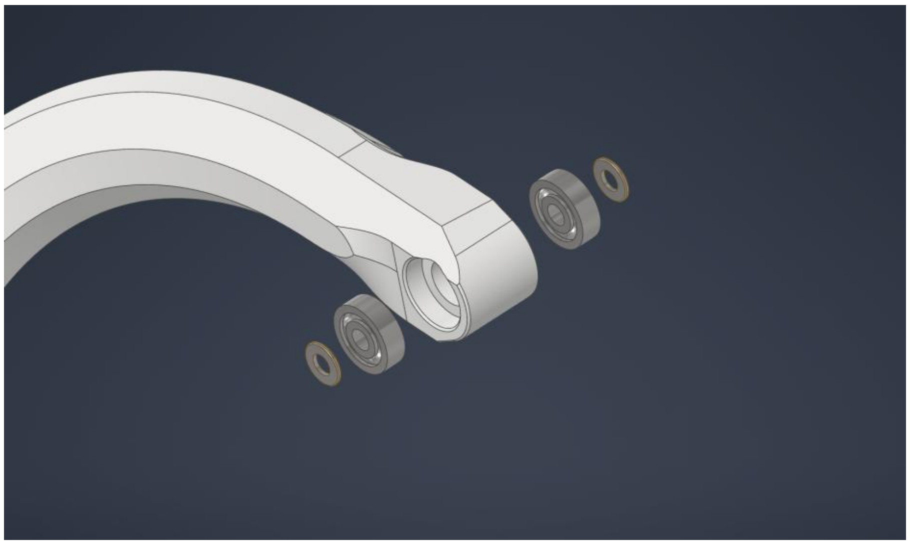

Figure 4. Construction of the rotational arm joints.

The final part cost includes all purchased parts, not including existing components such as the microcontroller, stepper motors and motor drivers, as well as not including small fasteners used for assembly. The parts list can be seen in Appendix 6.1.

3.1.2 Compensator Design

The trajectory controller follows the structure outlined in Figure 5. The controller requires a reference trajectory, which includes both position and velocity, as well as a measured position. Although these values are initially produced in Euler angle format, they were converted to the more conventional representations for robotics control. The reference and measured position are internally converted to quaternions, while the reference velocity and controller output are converted to rotation vector form.

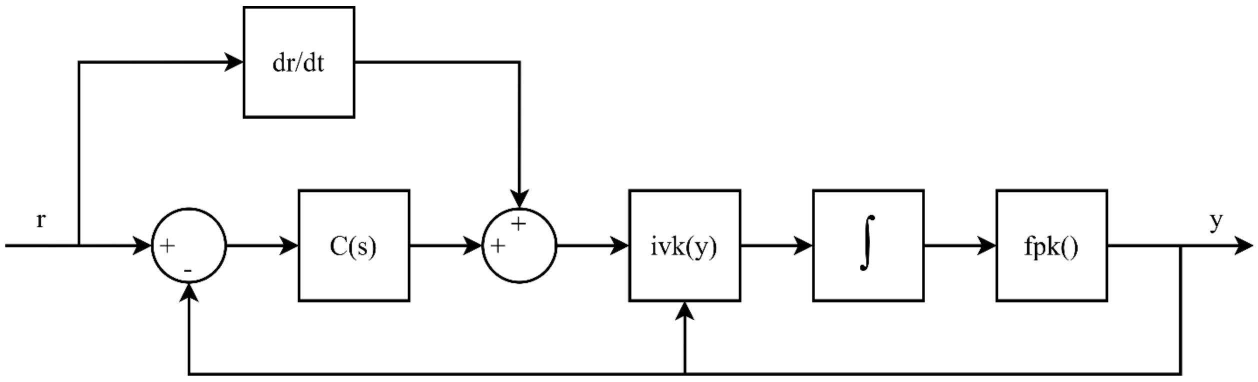

Figure 5. Closed-loop trajectory controller block diagram.

Two structures for the PID compensator were explored for trajectory tracking purposes. The first of which is a magnitude-only controller, applied to the angle component of the error in axis-angle representation. The second option is vector controller, which applies a compensator to each directional component of the error, represented as a rotation vector. These two were tested, and it was found that due to imperfect kinematics, the integral component was necessary for the compensator to rapidly eliminate the error in the system. This is not possible for an axis-angle magnitude controller, as the angle in this form is always positive, preventing the integral component from converging. For the open-loop trajectory controller, the simpler velocity-feedforward control structure in Figure 6 was used. Note that the plant is modelled as a pure integrator, with inverse-velocity and forward-positional kinematic gains described by ivk() and fpk() respectively.

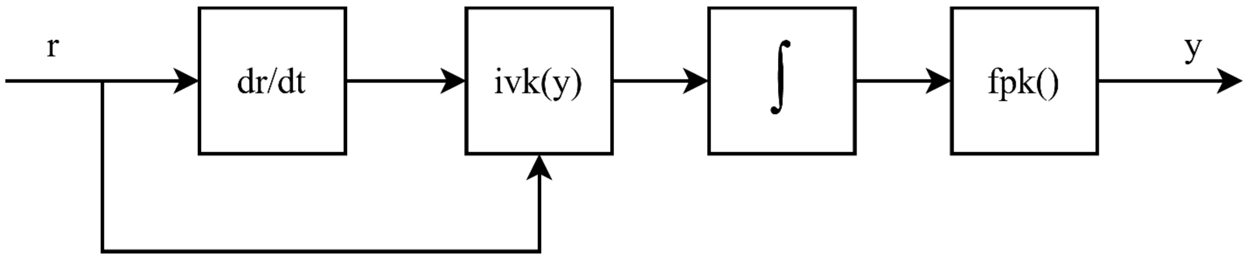

Figure 6. Open-Loop trajectory controller block diagram.

For homing and initialisation, a positional controller was also necessary to home the device and set it to the starting position before executing trajectory control. For simplicity, an axis-angle PID controller was used for this instance (See Figure 7), as the performance of the controller is not important. These controller functions can be found in Appendix 6.2.

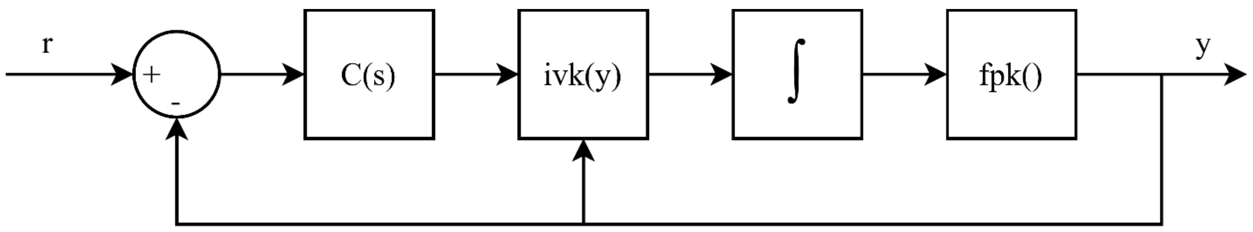

Figure 7. Closed-loop setpoint controller block diagram.

3.1.3 Communication

The update rate of the control system was chosen at 50Hz, significantly higher than the Nyquist frequency of the reference trajectory, and the signal was stored in a FIFO buffer to ensure accurate loop timing (See Appendix 6.3). To command the robot, a serial-based communication protocol was developed, which allows the user, or a program to change the device between the different control modes, command the motion or to perform other basic functions. The protocol followed a simple prefix-based command type scheme, where the first character determined the command, and any following characters can be followed by integers for device control. This protocol is outlined in Appendix 6.4

3.1.4 Localisation

For closed-loop control, the system was estimated using a 6-DOF IMU. The attitude of the system was calculated from the vector output of the accelerometer, while the body-frame rotational speed was calculated from the gyroscope. Due to the lack of magnetometer, the yaw axis estimate was obtained from the stepper motor position, passed through the forward-kinematic function. This problem can only be solved numerically [1], and so for computation speed a lookup table was generated, using a 2° spacing. Aside from this, the standard quaternion state-transition model was used for closed-loop orientation estimation [2]. The localisation functions can be found in Appendix 6.5.

For the kinematics, two contradictions were found in the literature, which had to be resolved to make localisation and control possible. The inverse positional kinematics assume in all cases that for a platform at the origin, the actuators positions would be $\theta = [0,0,0]^T$[1]. This was found to be only true for the common SPM parameters $\alpha_1 = \frac{\pi}{4}, \alpha_2 = \frac{\pi}{2}, \beta = \frac{\pi}{2}$. This was found to be untrue for any other non-trivial case, and for these configurations the origin value of the actuators had to be calculated using the closed-form solution upon initialisation. The literature also stated that for the numerical solution to the forward kinematic problem, the solution would be valid for all platform positions, provided the initial guess vector was set correctly. It was found that the proposed initial vector was only valid for a Z rotation of $\pm\frac{\pi}{2}$, and so a trial-and-error approach was taken for every initial vector combination to obtain the complete solution (See Appendix 6.6).

3.1.5 Motor Control

The stepper motors were dynamically controlled, with their velocity updated at every timestep. Although a hardware-mapped approach is the most robust way to implement stepper motor speed control, the microcontroller used (Arduino Nano 33 BLE) did not have enough free hardware timers to implement this. As a result, an interrupt-driven solution was implemented using the NRF timer library (See Appendix 6.7).

3.2 Experimental Results

The device orientation was recorded externally in the VICON laboratory, both in open-loop and in closed-loop for a variety of triangle and sine waves. The trials were abbreviated to the angles used (Y: yaw, P: pitch, R: roll), and the wave type (T: triangle, S: sinusoid). The maximum and average error between the measured orientation and the reference signal was aggregated and can be found in Table 1 – 3.

Table 1. Standalone Closed-Loop Test Error (Aggregated)

	Max Error (rad)			Av Error (rad)		
Trial	Y	P	R	Y	P	R
YT	0.066	0.333	0.042	0.004	0.006	0.003
YPT	0.031	0.033	0.036	0.004	0.007	0.005
PRT	0.023	0.032	0.047	0.003	0.007	0.006
YPRT	0.027	0.050	0.035	0.006	0.010	0.005
YS	0.026	0.020	0.026	0.026	0.003	0.003
YPS	0.313	0.027	0.059	0.007	0.004	0.005
PRS	0.018	0.029	0.017	0.004	0.007	0.005
YPRS	0.021	0.035	0.033	0.006	0.010	0.010
Max	0.313	0.333	0.059			
Average				0.008	0.007	0.005

Table 2. External Open-Loop Test Error (Aggregated)

	Max Error (rad)			Av Error (rad)		
Trial	Y	P	R	Y	P	R
YT	0.092	0.047	0.027	0.040	0.034	0.017
YPT	0.092	0.073	0.041	0.037	0.026	0.011
PRT	0.272	0.831	0.659	0.090	0.309	0.274
YPRT	0.160	0.078	0.152	0.042	0.027	0.060
YS	0.146	0.015	0.017	0.071	0.007	0.006
YPS	0.181	0.134	0.038	0.073	0.066	0.029
PRS	0.122	0.651	0.950	0.027	0.336	0.366
YPRS	0.787	0.528	0.900	0.153	0.117	0.140
Max	0.787	0.831	0.950			
Average				0.067	0.115	0.113

Table 3. External Closed-Loop Test Error (Aggregated)

	Max Error (rad)			Av Error (rad)		
Trial	Y	P	R	Y	P	R
YT	0.232	0.022	0.013	0.092	0.004	0.004
YPT	0.555	0.321	0.080	0.219	0.138	0.031
PRT	0.754	0.799	1.031	0.273	0.310	0.359
YPRT	1.459	1.459	1.459	1.459	1.459	1.459
YS	0.273	0.052	0.148	0.115	0.008	0.010
YPS	0.676	0.439	0.063	0.208	0.147	0.042
PRS	0.029	0.979	0.983	0.010	0.419	0.503
YPRS	0.773	0.416	0.489	0.242	0.160	0.175
Max	1.459	1.459	1.459			
Average				0.361	0.377	0.368

Figures 8-13 show the measured path and the angular error for the yaw-pitch-roll triangle waves, which repeatedly expose the device to the maximum ramp input on all axes. These results are aggregated and compared in Figure 14.

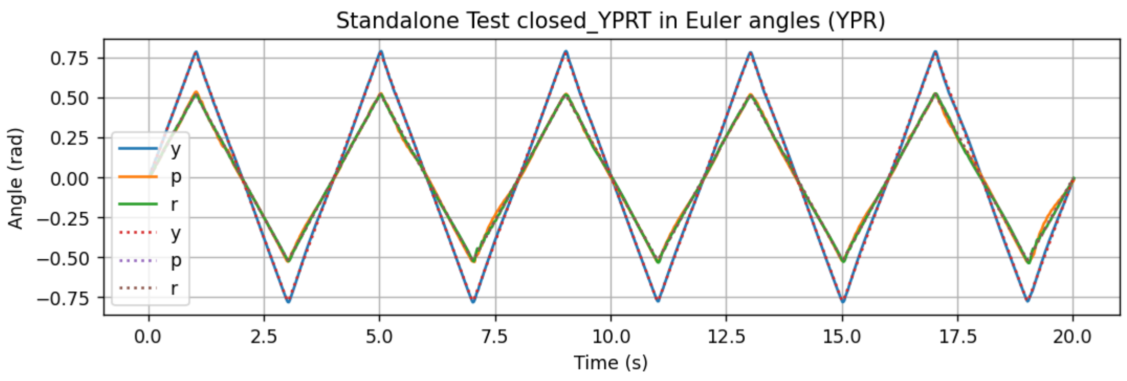

Figure 8. Reference signal (dotted line) and Measured signal (solid line) from standalone closed-loop test.

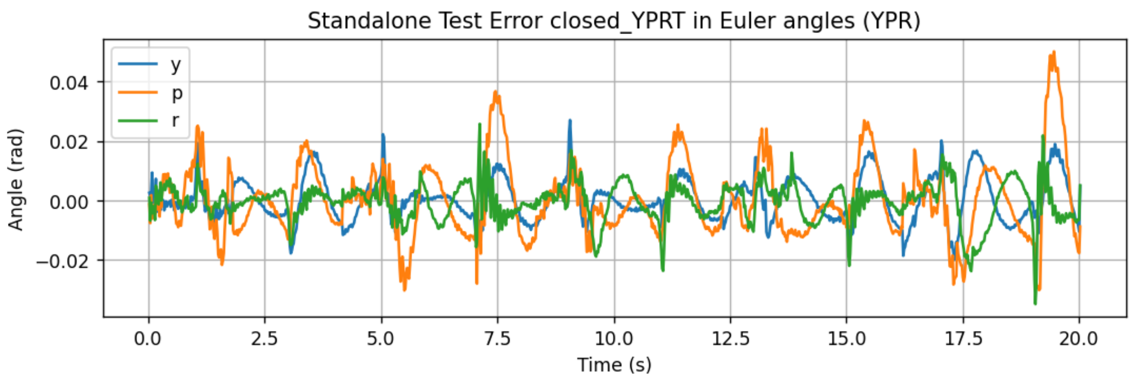

Figure 9. Error signal from standalone closed-loop test.

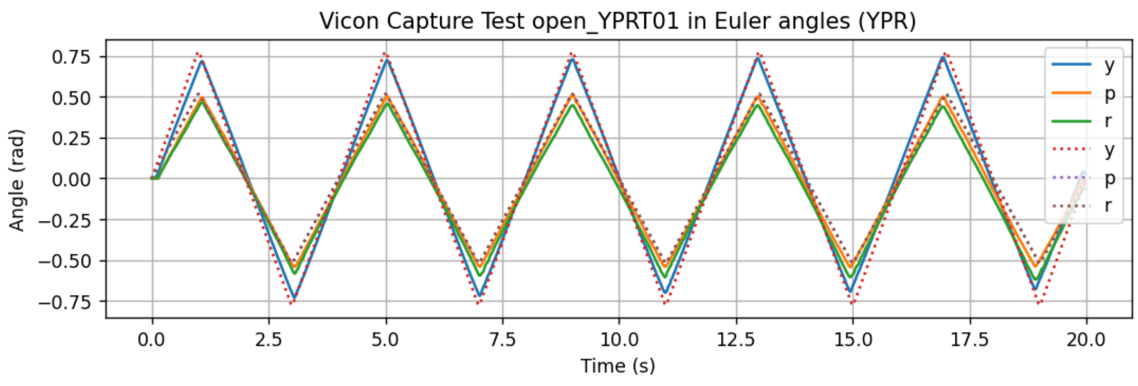

Figure 10. Reference signal (dotted line) and Measured signal (solid line) from VICON open-loop test.

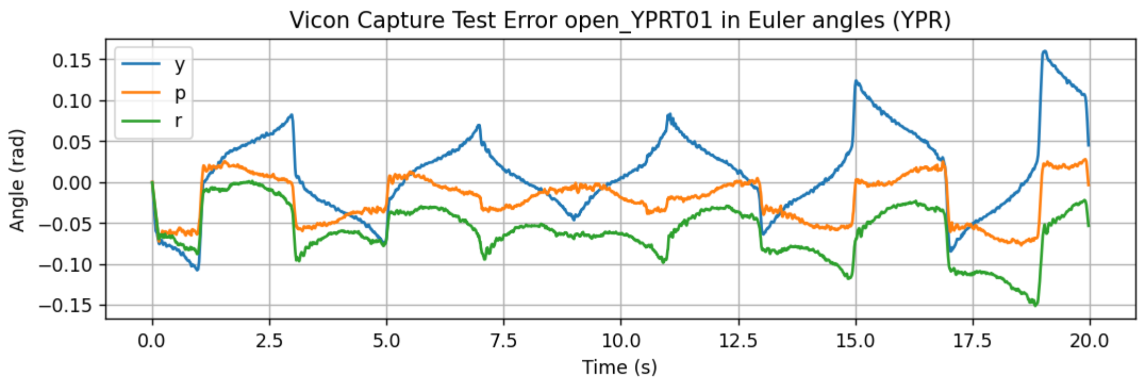

Figure 11. Error signal from external open-loop test.

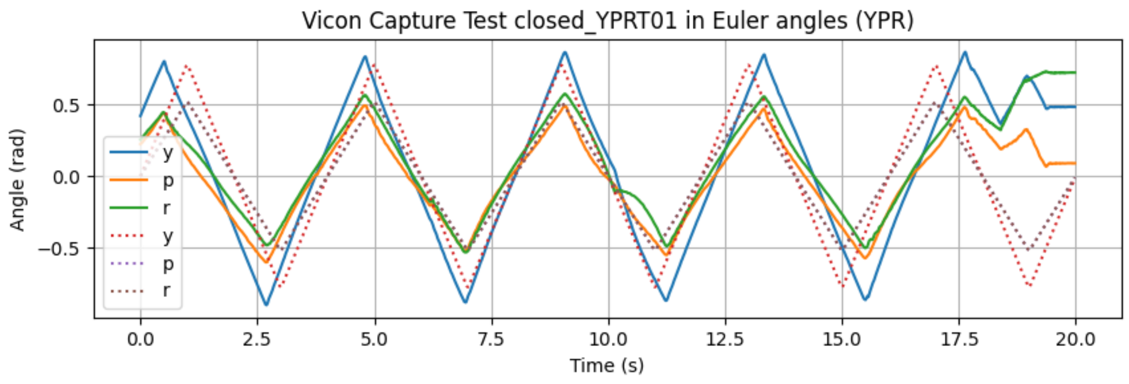

Figure 12. Reference signal (dotted line) and Measured signal (solid line) from external closed-loop test.

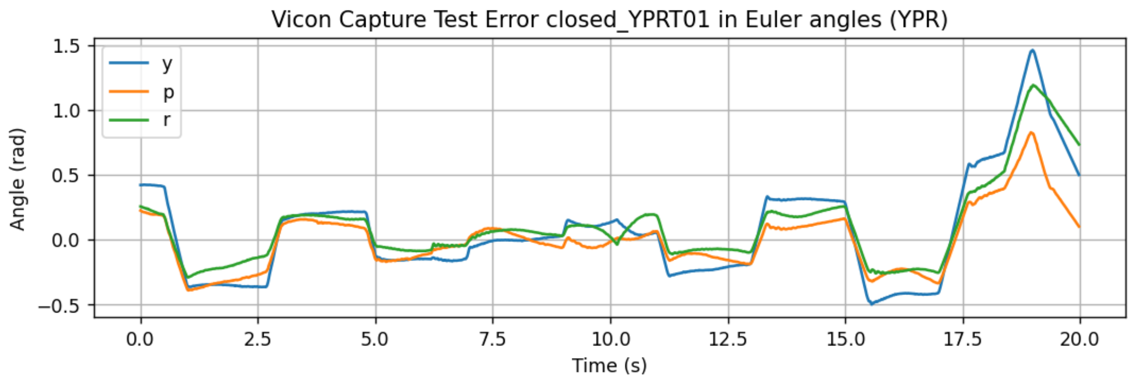

Figure 13. Error signal from external closed-loop test.

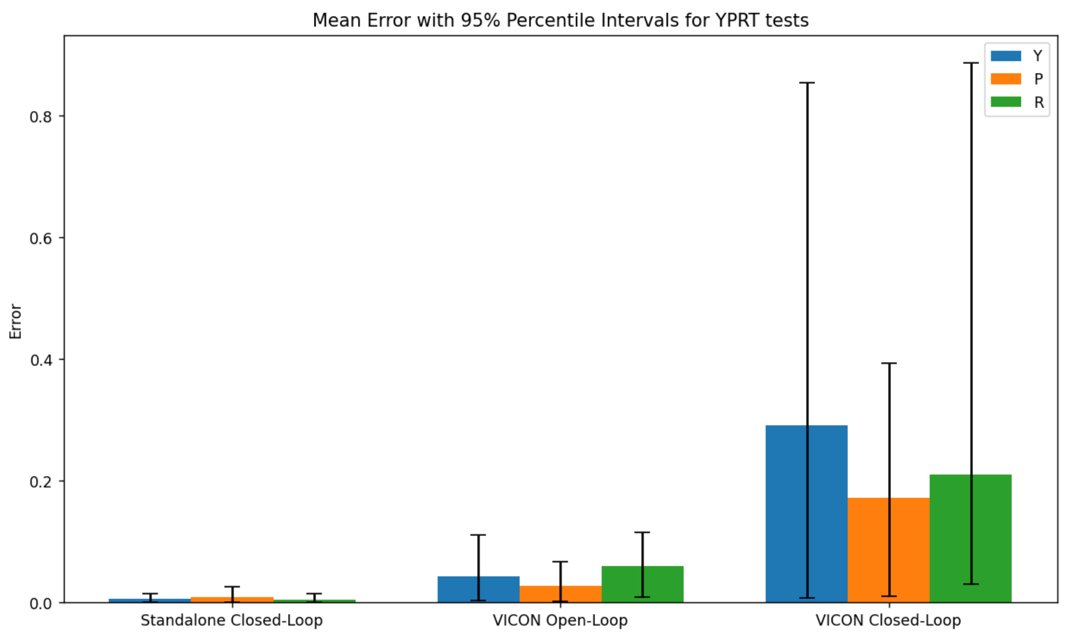

Figure 14. Bar Plot for error signals in all YPRT tests.

Table 4. Confidence Intervals for all YPRT tests.

Test	Standalone Closed-Loop (rad)			External Open-Loop (rad)			External Closed-Loop (rad)		
	Y	P	R	Y	P	R	Y	P	R
95%	0.0150	0.0259	0.0143	0.1108	0.0669	0.1161	0.8540	0.3927	0.8874
5%	0.0004	0.0007	0.0005	0.0030	0.0017	0.0089	0.0079	0.0112	0.0301

3.2.1 Standalone Closed-Loop Test Analysis

The results in Figure 8 show that in the standalone device testing, the trajectory-tracking controller performed well, following the triangle waves in all axes simultaneously, and correcting deviations in the platform orientation in real time. In Figure 9, the compensator is more evidently shown, with spikes in the error coming in regular intervals that line up with direction change. The compensator is shown to drive major spikes close to zero. In Table 4, the performance of the device in the standalone test is more clearly shown, with upper 95% confidence errors of less than 0.03 radians across all axes.

Table 1 shows the absolute maximum errors across all the experiments, which show substantial outliers in the system, specifically in the YPS and YT tests. In these tests, stepper motor skips were observed, however by manually adding additional resistance to the motor, it was shown that the motors were not working near their maximum torque threshold. This indicates a stepper control fault in the firmware.

3.2.2 External Open-Loop Test Analysis

The externally measured open-loop tests show good performance compared to the standalone test, with one noticeable caveat. Figure 10 shows the measured signal from the VICON tracking system, and while the shape and amplitude of the signal are remarkably accurate, given that the system is in open loop, the period of the signal is slightly different that that of the reference signal. While the difference is minor, it is enough to cause a much higher measured error, as shown in Figure 10. Table 2 also shows the increase in signal error, which appears to worser with the off-sync sinusoids, as in the YPRS test, the maximum and average errors are drastically higher than other tests. The horizontal dilation of the measured signal in this test indicates a loop timing error.

While this error could be caused by computation overload, the loop timing implemented in firmware relies on the global clock, and so this error should have been visible on the standalone test as well. Instead, it appears that the loop timing rate itself is incorrect, which indicates a hardware fault on the microcontroller. This is likely due to temperature, which has been shown to affect crystal oscillators [3]. This unforeseen issue highlights the need for ventilation within the device chassis, as well as spacing and thermal dissipation for the stepper drivers.

3.2.3 External Closed-Loop Test Analysis

Figures 12 and 13 show the closed-loop test with external measurement. While it was expected that the system would have performance like that of the standalone test, the graphs show that the performance has degraded significantly. As previously stated, the device suffered from a loop timing delay, which was more severe than the open-loop test, and resulted in significant errors. Additionally, the device experienced some artefacts indicative of instability with the EKF. To verify this, an open-loop standalone test was conducted, as shown in Figure 15. This confirmed that the localisation was unstable, leading to poor performance.

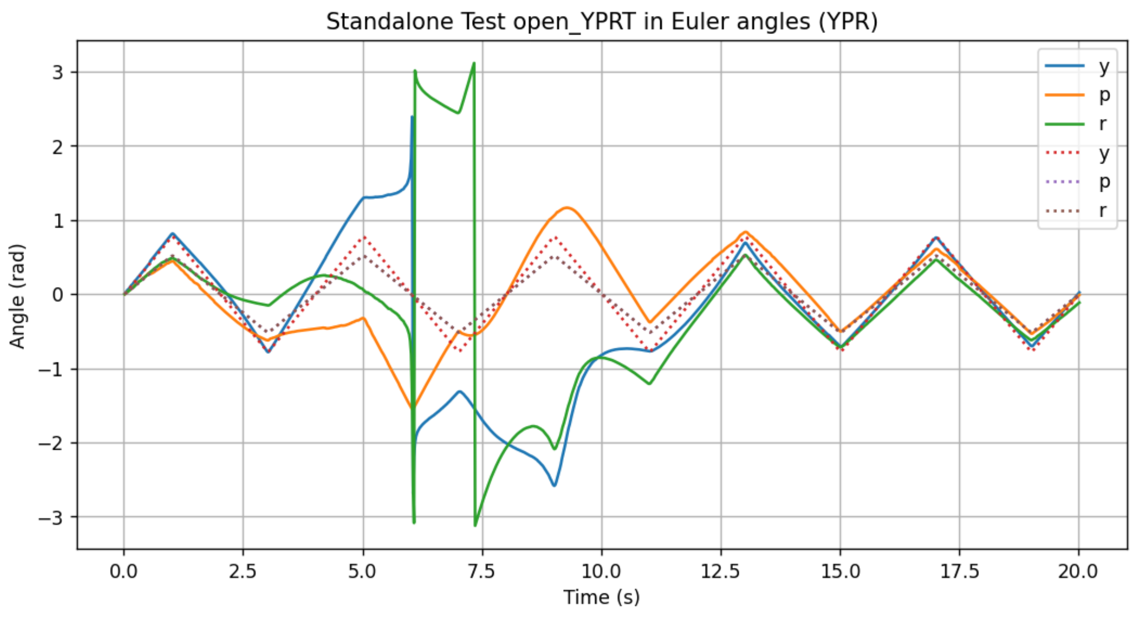

Figure 15. Open-loop standalone test with yaw-pitch-roll Triangle wave.

The exact cause of this instability was not determined, however it likely was exacerbated by the loop timing, as this directly affects the state transition model for the gyroscope in the EKF. The stepper control issues also may have played a part if the device was not correctly keeping track of the stepper position. While the average and upper confidence bound for the error in Figure 14 are drastically higher than any other test, the average errors for the single axis tests (YS and YT) show reasonable performance, indicating that this error happens randomly. All being said, this error massively skews the results, making them unusable for evaluating the performance of the device.

3.2.3 Performance Evaluation

While the performance of the device using external measurement is invalid, the performance of individual subsystems can be evaluated and validated for their functionality. The externally measured open-loop tests show that the inverse-velocity kinematics ares implemented correctly, and that a velocity feedforward controller is viable for trajectory tracking. The closed-loop standalone tests show that the system can reduce angular error to 0.026 radians (~1.5 degrees) with 95% confidence, under a ramp input of (45, 30, 30)°/s.

4. CONCLUSION

The design of the SPM was successful; with the main design objectives all being achieved. The result was a parallel manipulator device with high rigidity, no backlash and a high range of motion at a cost of just over $400. The device required only 3D printed and laser cut parts, making it replicable and cheap to fix or modify for future work. While the designed manipulator did not achieve real-world performance targets, the results demonstrate that a stepper-driven SPM is capable of being dynamically controlled to within 2° accuracy, using a velocity-feedforward compensator, provided the device localisation is sufficiently accurate. This project opens future work surrounding real-world manipulator localisation for closed-loop control.

5. REFERENCES

[1] A. T. Cruz-Reyes, M. Arias-Montiel, and R. Tapia-Herrera, “Kinematic Analysis of a Coaxial 3-RRR Spherical Parallel Manipulator Based on Screw Theory,” in *Mechanism Design for Robotics*, S. Zeghloul, M. A. Laribi, and M. Arsicault, Eds., Cham: Springer International Publishing, 2021, pp. 28–37. doi: 10.1007/978-3-030-75271-2_4

[2] “Quaternion State-Transition Model — Aceinna OpenIMU Developer Manual documentation.” Available: https://openimu.readthedocs.io/en/latest/algorithms/STM_Quaternion.html. [Accessed: Dec. 06, 2025]

[3] H. Zhou, “Frequency Accuracy & Stability Dependencies of Crystal Oscillators,” Carleton University, Ottawa, Ont., Canada, 200 AD. Available: https://kunz-pc.sce.carleton.ca/thesis/CrystalOscillators.pdf

6. APPENDICES

6.1 Manipulator Part Costing

Seller	**Name/Description**	**Unit Quantity**	**Unit Cost**	**Total Cost**
RS Components	50*80D 15W Bearing	2	$23.23	$46.46
RS Components	35*62D 14W Bearing	2	$14.89	$29.78
RS Components	20*42D 12W Bearing	2	$8.42	$16.84
RS Components	5*16D 5W Bearing	3	$3.89	$70.02
RS Components	Timing pulley 3M 12T 9W	3	$9.30	$27.90
RS Components	Timing Belt 3M 140T 9W 420mm	18	$16.52	$49.56
Core Electronics	SparkFun 6 Degrees of Freedom Breakout - LSM6DSO (Qwiic)	1	$27.75	$27.75
Core Electronics	Slip Ring - 12 Wire (2A)	1	$47.90	$47.90
Bambu Lab	PC (Black) 1kg Filament	1	$61.99	$61.99
Bambu Lab	PETG HF (Blue) 1kg Filament	1	$30.99	$30.99
			Subtotal	**$409.19**

6.2 PID Control Firmware

```
// Control functions
void position_control(Quaternionf& error, Quaternionf& meas) {
  Vector4f error_aa = q_to_aa(error);
  if (error_aa[0] < 0) {
    error_aa *= -1;  // ensure positive angular errors only
  }
  float        compensated_error        =        position_compensator.update(error_aa[0],
LOOP_TIMING_INTERVAL/1e6); // compensate for the angle and not direction
  error_aa[0] = compensated_error;
  Vector3f control = aa_to_xyz(error_aa);
  Matrix3f R = aa_to_R(q_to_aa(meas));
  Vector3f actuator_velocity = spm.solve_ivk(R, control);
  set_actuator_velocity(actuator_velocity);
}

void open_trajectory_control(Vector3f& ypr_ref, Vector3f& ypr_velocity_ref) {
  Matrix3f R_mat = spm.R_ypr(ypr_ref);

  Vector3f xyz_platform_velocity = spm.ypr_to_xyz_velocity(ypr_velocity_ref, ypr_ref);
  Vector3f actuator_velocity = spm.solve_ivk(R_mat, xyz_platform_velocity);
  set_actuator_velocity(actuator_velocity);
}

void  closed_trajectory_control(Quaternionf&  ref_q,  Quaternionf&  meas_q,  Vector3f&
ypr_velocity_ref) {
  Vector3f error_xyz = aa_to_xyz(q_to_aa((ref_q * meas_q.conjugate()).normalized()));
  // apply compensator to cartesian components
  float dt = LOOP_TIMING_INTERVAL/1e6;
  Vector3f control(traj_x_compensator.update(error_xyz.x(), dt),
                   traj_y_compensator.update(error_xyz.y(), dt),
                   traj_z_compensator.update(error_xyz.z(), dt)
                   );

  // velocity feedforward
  Vector3f xyz_vel_ref = spm.ypr_to_xyz_velocity(ypr_velocity_ref, q_to_ypr(ref_q));
  Matrix3f R_mat = aa_to_R(q_to_aa(meas_q));

  // combine open and closed loop signals
  Vector3f actuator_velocity = spm.solve_ivk(R_mat, xyz_vel_ref + control);
  set_actuator_velocity(actuator_velocity);
}
```

6.3 Loop timing and FIFO buffer code

```
void buffer_push(unsigned int length, char* items) {
  for (unsigned int i=0; i<length; i++) {
    command_buffer.push(items[i]);
  }
  command_buffer.push(DELIM); //add back the delimiter
}

char* buffer_pop() {
  static char command[COMMAND_MAX_SIZE];
  byte index = 0;
  clearCharArray(command,sizeof(command));
  // pop until the start character is reached
  while (!command_buffer.empty() && command_buffer.front() != DELIM) {
      command[index] = command_buffer.front();
      command_buffer.pop();
      index++;
  }
  if (!command_buffer.empty()) {
    command[index] = command_buffer.front();
    command_buffer.pop(); // pop the delimiter character as well
  }
  return command;
}

bool loop_timing_proc() {
  if (!loop_timing_enabled) {
    return true; // always allow loop function when loop timing is disabled
  } else {
    loop_time_elapsed = micros() - loop_start_time - LOOP_TIMING_INTERVAL; // elapsed time
since session start
    if (loop_time_elapsed - loop_time_proc > LOOP_TIMING_INTERVAL) {
      loop_time_proc += LOOP_TIMING_INTERVAL;
      #ifdef TRACE
      Serial.println("Loop timing proc");
      #endif
      return true; // enable buffer read when loop timing has started
    } else {
      return false; // prevent buffer read
    }
  }
}

void loop() {
  // Read serial data and place into buffer
  if (Serial.available()) {
    char command[COMMAND_MAX_SIZE];
    clearCharArray(command, sizeof(command));
    int command_length = Serial.readBytesUntil(DELIM, command, sizeof(command));
    buffer_push(command_length, command);
  }

  // application level flow control
  if (command_buffer.size() >= COMMAND_BUFFER_FULL_SIZE && flow_control_halt == false) {
    Serial.write(XOFF);
    flow_control_halt = true;
#ifdef TRACE
    Serial.println("XOFF sent");
#endif
  } else if (command_buffer.size() < COMMAND_BUFFER_EMPTY_SIZE && flow_control_halt ==
true) {
    Serial.write(XON);
    flow_control_halt = false;
#ifdef TRACE
    Serial.println("XON sent");
#endif
  }

  // Reference signals
  static Vector3f ypr_ref = Vector3f::Constant(0);
  static Vector3f ypr_velocity_ref = Vector3f::Constant(0);

  static bool movement_ongoing = false;
  char* command;
  if (loop_timing_proc()) {
#ifdef TRACE
    if (state != IDLE_STATE) {
      Serial.println("========== TRACE ==========");
    }
#endif
    // rest of main loop that runs every loop timing proc ...
}
```

6.4 Command Protocol Table

Each command has a single prefix character and may include additional parameters. Some commands are only available in certain states. Note that an outgoing command is one that is sent from the device back to the computer. All commands are delimited with a newline character.

Command / State Description	**State**	**Prefix**	**Parameters**	**Parameter Description**
Motor Enable Power	Any	E	None	None
Motor Disable Power	Any	D	None	None
Home Device	Any	H	None	None
Set Position State	Any	P	None	None
Set Closed-Loop Trajectory State	Any	C	None	None
Set Open-Loop Trajectory State	Any	O	None	None
Set Test State	Any	T	None	None
Set Idle State	Any	I	None	None
Movement Command	Position Control	M	Y<int> P<int> R<int>	Yaw, pitch and roll position (mrad)
Movement Command	Trajectory Control	M	Y<int> P<int> R<int> y<int> p<int> r<int>	Yaw, Pitch and Roll position (mrad), and velocity (mrad/s)
Test Motor (0, 1, 2)	Test	0/1/2	None	None
Print Accelerometer	Test	A	None	None
Print Gyroscope Value	Test	G	None	None
Print EKF Value	Test	K	None	None
Print current FPK table value	Test	F	None	None
Print FPK table value from index	Test	f	f<int> ,<int>	Index for forward-kinematic lookup table (unitless)
Outgoing Command	Position, Home	#	F	Indicates Movement Finished

6.5 Localisation Functions

```
Vector3f accel_ypr(const Vector3f& gyro, unsigned int samples) {
  float centrip_g = pow(gyro.norm(), 2) * 0.016 * 9.8067;
  Vector3f accel(0,0,0);
  // Get average accel reading
  for(uint8_t i=0; i<samples; i++) {
    accel[0] += platformIMU.readFloatAccelX();
    accel[1] += platformIMU.readFloatAccelY();
    accel[2] += platformIMU.readFloatAccelZ() - centrip_g;
  }
  if(samples > 1) {
    accel /= samples;
  }
  // unwind roll and pitch in the right-hand ENU frame
  Vector3f ypr;
  ypr[0] = 0; // do not set yaw
  ypr[1] = -atan2(accel[0], accel[2]);
  ypr[2] = atan2(accel[1], sqrt(pow(accel[0],2) + pow(accel[2],2)));
  return ypr;
}

float interp_yaw_fpk() {
  Vector3f act_pos = actuator_position();
  float offset = - act_pos[0] + spm.actuator_origin;
  Vector3f actuator_offset = act_pos + Vector3f::Constant(offset); // offset the actuator
position such that m0 is placed at the origin
  float yaw_offset = fpk_yaw_table.interp(actuator_offset[1], actuator_offset[2]); // fpk
table is in the m1/m2 domain
  if (yaw_offset < -PI || yaw_offset > PI ||
  abs(act_pos[0] - act_pos[1] + (2*PI/3)) < MIN_ACT_DIFF || abs(act_pos[1] - act_pos[2] +
(2*PI/3)) < MIN_ACT_DIFF || abs(act_pos[2] - act_pos[0] + (2*PI/3)) < MIN_ACT_DIFF) { //
if yaw value out-of-bounds (NAN)
    disable_motors();
    next_state = IDLE_STATE;
    Serial.println("Platform Out of Bounds! Ensure that position commands have less than
a 40 degree slope.");
    return FPK_NAN_CODE;
  }
  return wrap_rad(yaw_offset + offset); // add the offset back to obtain the true yaw value
}

Quaternionf estimate(bool include_yaw_fpk) {
  Vector3f gyro = gyro_xyz() - gyro_bias;
  Vector3f ypr_meas = accel_ypr(gyro);
  if (include_yaw_fpk) {
    ypr_meas[0] = interp_yaw_fpk();
  }
  Quaternionf q_meas = ypr_to_q(ypr_meas);

  kalman.predict(gyro, LOOP_TIMING_INTERVAL/1e6);
  kalman.correct(Vector4f(q_meas.w(),q_meas.x(), q_meas.y(), q_meas.z()));
  kalman.x /= kalman.x.norm();
  Quaternionf est = Quaternionf(kalman.x[0], kalman.x[1], kalman.x[2], kalman.x[3]);
  return est;
}
```

6.6 Complete Kinematic Solution Class Implementation

```
class Coaxial_SPM:
    def __init__(self, a1, a2, b): # gamma assumed to be zero for coaxial spm
        self.a1 = a1 # alpha 1
        self.a2 = a2 # alpha 2
        self.b = b # beta
        self.v = None
        self.w = None
        self.n = None
        self.u = np.array([0, 0, -1]).T # u vector face down for all joints
        self.i_range = range(0,3) # joint values are i = (0,1,2)
        self.v_origin = [self.v_i_origin(v_i) for v_i in self.i_range]
        self.n_origin = np.array([0, 0, 1]).T
        self.eta = [self.eta_i(i) for i in self.i_range]
        self.J = None
        self.w_fpk = None
        self.v_fpk = None
        self.v_fpk = None
        self.actuator_origin = self.solve_ipk(np.eye(3))[0]
        self.actuator_direction = -1
        self.angle_between_v = angle_between(self.v_origin[0], self.v_origin[1])

    # calculate rotation amtrix from yaw-pitch-roll input
    def R_ypr(self, angle):
        yaw = angle[0]
        pitch = angle[1]
        roll = angle[2]
        return R_z(yaw) @ R_y(pitch) @ R_x(roll)

    # eta defines how a function changes for the three limbs
    def eta_i(self, i):
        return 2*i*pi/3 # assume i = (0,1,2)

    # calculate the vector of an ith itermediary joint vector
    def w_i(self, in_i, i):
        e_i = self.eta_i(i)
        return   np.array([cos(e_i-in_i)*sin(self.a1),   sin(e_i-in_i)*sin(self.a1),   -
cos(self.a1)]).T

    # caluclate the home position of the ith platform joint vector
    def v_i_origin(self, i):
        e_i = self.eta_i(i)
        return np.array([-sin(e_i)*sin(self.b), cos(e_i)*sin(self.b), cos(self.b)]).T

    # calculate the ith A term in the inverse positional kinematic problem
    def A_i_ipk(self, v_i, i):
        e_i = self.eta_i(i)
        v_ix = v_i[0]
        v_iy = v_i[1]
        v_iz = v_i[2]
        return    -v_ix*cos(e_i)*sin(self.a1)    -    v_iy*sin(e_i)*sin(self.a1)    -
v_iz*cos(self.a1) - cos(self.a2)

    # calculate the ith A term in the inverse positional kinematic problem
    def B_i_ipk(self, v_i, i):
        e_i = self.eta_i(i)
        v_ix = v_i[0]
        v_iy = v_i[1]
        return v_ix*sin(e_i)*sin(self.a1) - v_iy*cos(e_i)*sin(self.a1)

    # calculate the ith A term in the inverse positional kinematic problem
    def C_i_ipk(self, v_i, i):
        e_i = self.eta_i(i)
        v_ix = v_i[0]
        v_iy = v_i[1]
        v_iz = v_i[2]
        return v_ix*cos(e_i)*sin(self.a1) + v_iy*sin(e_i)*sin(self.a1) - v_iz*cos(self.a1)
- cos(self.a2)

    # algebraically solves positional inverse kinematic problem
    def solve_ipk(self, r, exception=True, overwrite=True):
        temp_v = [r@self.v_origin[i] for i in self.i_range]
        temp_n = r@self.n_origin
        A = [self.A_i_ipk(temp_v[i], i) for i in self.i_range]
        B = [self.B_i_ipk(temp_v[i], i) for i in self.i_range]
        C = [self.C_i_ipk(temp_v[i], i) for i in self.i_range]
        T = [solve_quadratic(A[i], B[i]*2, C[i])[0].real for i in self.i_range] # we choose
the first solution [0] because it lines up with the orientation of our manipulator
        input_angle = np.array([2 * np.arctan(T[i]) for i in self.i_range])
        temp_w = [self.w_i(input_angle[i], i) for i in self.i_range] # intermediate joint
unit vector

        if overwrite==True:
            self.v = temp_v
            self.n = temp_n
            self.w = temp_w
        if self.verify_position(exception) == True:
            return input_angle
        else:
            return [np.nan, np.nan, np.nan]

    # use system paramaters to verify if the current state is valid
    def verify_position(self, exception=True):
        invalid_param = None
        for i in self.i_range:
            if isclose(angle_between(self.u, self.w[i]), self.a1) == False:
                invalid_param = "a1"
            elif isclose(angle_between(self.w[i], self.v[i]), self.a2) == False:
                invalid_param = "a2"
            elif isclose(angle_between(self.v[i], self.n), self.b) == False:
                invalid_param = "b"
            if invalid_param != None:
                if exception == True:
                    raise BaseException("Rotation invalid! paramater %s is incorrect" %
invalid_param)
                return False
            else:
                return True

    # algebraically solves inverse velocity kinematic problem using Jacobian matrix
    def solve_ivk(self, platform_angle, platform_velocity):
        self.solve_ipk(platform_angle) # solve for w and v unit vectors at operting point
        A = np.array([np.cross(self.w[i], self.v[i]).T for i in self.i_range])
        B = np.diag([np.dot(np.cross(self.u, self.w[i]), self.v[i]) for i in self.i_range])
        self.J = np.linalg.inv(B)@A
        input_velocity = self.J@platform_velocity.T
        return input_velocity

    # quadratic system to solve fpk problem. Has eight solutions so the initial guess must
be chosen carefully to ensure the correct solution is found
    def fpk_system(self, v_out):
        # Initialize residuals
        r = np.zeros(9)
        # rename variables for ease of use
        v0_x = v_out[0]
        v0_y = v_out[1]
        v0_z = v_out[2]
        v1_x = v_out[3]
        v1_y = v_out[4]
        v1_z = v_out[5]
        v2_x = v_out[6]
        v2_y = v_out[7]
        v2_z = v_out[8]
        # w[0][2] is w_0_y (first number is ith w vector, second number is xyz selection)
        w = self.w_fpk
        # angle to intermediate joint constraint
        r[0] = w[0,0]*v0_x + w[0,1]*v0_y + w[0,2]*v0_z - cos(self.a2)
        r[1] = w[1,0]*v1_x + w[1,1]*v1_y + w[1,2]*v1_z - cos(self.a2)
        r[2] = w[2,0]*v2_x + w[2,1]*v2_y + w[2,2]*v2_z - cos(self.a2)
        # vector angular constraint
        r[3] = v0_x*v1_x + v0_y*v1_y + v0_z*v1_z - cos(self.angle_between_v)
        r[4] = v1_x*v2_x + v1_y*v2_y + v1_z*v2_z - cos(self.angle_between_v)
        r[5] = v2_x*v0_x + v2_y*v0_y + v2_z*v0_z - cos(self.angle_between_v)
        # unit vector magnitude constraint
        r[6] = v0_x**2 + v0_y**2 + v0_z**2 - 1
        r[7] = v1_x**2 + v1_y**2 + v1_z**2 - 1
        r[8] = v2_x**2 + v2_y**2 + v2_z**2 - 1
        return r

    # numerically solves fpk problem and returns the rotation matrix
    def solve_fpk(self, input_angles, ignore_error=False, overwrite=False):
        # decouple yaw by subtracting first angle
        # also turn actuator angle/velocity positive (around the up Z axis instead of the
default down)
        # nonlinear canonical system of equations
        self.w_fpk = np.array([self.w_i(input_angles[i], i) for i in self.i_range])
        for i in range(512):
            init_v = [1 if bit == '1' else -1 for bit in f"{i:0{9}b}"]
            try:
                v_out = optimize.fsolve(func=self.fpk_system, x0=init_v)
                fpk_out = np.array([v_out[i*3:i*3+3] for i in self.i_range])
                ypr = self.unwind_ypr(fpk_out)
                ypr[0] = wrap_rad(ypr[0])
                ypr[1] = wrap_rad(ypr[1])
                ypr[2] = wrap_rad(ypr[2])
                check_act_angles  =  self.solve_ipk(self.R_ypr(ypr),  overwrite=overwrite,
exception=False)
                #print("check:", check_act_angles)
                if    isclose(check_act_angles[0],    input_angles[0],    1e-2)    and
isclose(check_act_angles[2],  input_angles[2],  1e-2)  and  isclose(check_act_angles[2],
input_angles[2], 1e-2):
                    self.v_fpk = fpk_out
                    return ypr
                else:
                    #print("no match")
                    pass
            except RuntimeWarning as warn:
                #print(warn)
                pass

        return [np.nan, np.nan, np.nan]

    # unwinds ypr angles from three-vector (v) platform representation. Angular range is
-180 <= a < 180
```

```
    def unwind_ypr(self, v):
        # normal, pitch and roll axes in the body frame
        roll_ax, pitch_ax, normal = self.get_orthonormals(v)
        normal_z = np.array([0,0,1])
        # using angle_between will not provide angles above 180, so the roll/pitch unwind
range is limited to +/- 90 degrees
        roll_projected_vector = unit_vector(project_to_plane(roll_ax, normal_z))
        roll = pi/2 - angle_between(roll_projected_vector, pitch_ax)
        pitch_ax = R_axis(roll_ax, -roll) @ pitch_ax
        pitch = -(pi/2 - angle_between(normal_z, roll_ax))
        roll_ax = R_axis(pitch_ax, -pitch) @ roll_ax
        # use trig values to convert unit vector to angle to avoid 180 degree constraint
        yaw = atan2(roll_ax[1], roll_ax[0])
        return np.array([yaw, pitch, roll])

    def center_vector(self, v1, v2, v3, theta):
        v1 = np.asarray(v1, float)
        v2 = np.asarray(v2, float)
        v3 = np.asarray(v3, float)
        M = np.column_stack([v1, v2, v3])
        b = np.full(3, np.cos(theta))
        # Try direct solve first (non-coplanar case)
        try:
            u = np.linalg.solve(M.T, b)
        except np.linalg.LinAlgError:
            # Coplanar case: find nullspace of M.T
            _, _, VT = np.linalg.svd(M.T)
            u = VT.T[:, -1]
            # the given theta should be ≈ 90°, so cos(theta)=0
            # enforce correct direction if desired
            if np.dot(u, v1 + v2 + v3) < 0:
                u = -u
        # Normalize result
        u /= np.linalg.norm(u)
        return u

    # get orthonormal vectors (xyz) relative to the body frame
    def get_orthonormals(self, v):
        z_v = self.center_vector(v[0], v[1], v[2], self.angle_between_v)
        x_v = unit_vector(np.cross(v[0], z_v))
        y_v = unit_vector(np.cross(z_v, x_v))
        return x_v, y_v, z_v
```

6.7 Stepper Control Firmware

```
StepperMotor::StepperMotor(uint8_t _stepPin, uint8_t _dirPin, uint8_t _sleepPin)
    : stepPin(_stepPin), dirPin(_dirPin), sleepPin(_sleepPin), direction(false),
        position(0), step_state(false), halted(true)
    {
        // Initialise pins
        pinMode(stepPin, OUTPUT);
        pinMode(dirPin, OUTPUT);
        pinMode(sleepPin, OUTPUT);
        setSpeed(0);
        set_dir(true);
        step();
    }

void StepperMotor::setSpeed(float speed) {
    if (abs(speed) < 1e-5) {
        if (!halted) { // set flag when speed is close to zero
            halted = true;
            timer->CC[ccIndex] = timer_current() - 1; // set compare reg behind
timer so it takes max time to step
        }
    } else {
        intervalTicks = round(abs(speed_scale/(speed*2)));
        if (halted) {
            halted = false;
            prev_cc = timer->CC[ccIndex];
            timer->CC[ccIndex] = timer_current() + 1; // initialise CC so the
interupt fires on the next tick
        } else {
            // if the new speed value is slow enough to be set immediately, then
set the corresponding cc value,
            // if it is too fast, set the cc to the next tick
            uint32_t new_cc = prev_cc + intervalTicks;
            uint32_t current_cc = timer_current();
            if (new_cc > current_cc) {
                prev_cc = timer->CC[ccIndex];
                timer->CC[ccIndex] = new_cc;
            } else {
                prev_cc = timer->CC[ccIndex];
                timer->CC[ccIndex] = current_cc + 1;
            }
        }
        // set direction
        if (speed < 0) { set_dir(false); }
        else { set_dir(true); }
    }
}

void StepperMotor::enable() {
    digitalWrite(sleepPin, HIGH);
    enable_interrupt(ccIndex);
    timer->CC[ccIndex] = timer_current() - 1;
    halted = true;
}

void StepperMotor::disable() {
    digitalWrite(sleepPin, LOW);
    disable_interrupt(ccIndex);

void StepperMotor::set_dir(bool dir) {
    if (direction != dir) {
        direction = dir;
        digitalWrite(dirPin, direction);
    }
}

void StepperMotor::step() {
    step_state = !step_state;
    digitalWrite(stepPin, step_state);
    if (step_state) {
        if (direction) position++;
        else position--;
    }
    prev_cc = timer->CC[ccIndex];
    timer->CC[ccIndex] += intervalTicks;
}

StepperDriver::StepperDriver(NRF_TIMER_Type*  _timer,  IRQn_Type  _irq,  uint8_t
_reg_max, uint8_t _prescaler)
    : timer(_timer), irq(_irq), motorCount(0), reg_max(_reg_max),
    prescaler(_prescaler), speed_scale(pow(2,(4-_prescaler))*1e6) {}

void StepperDriver::begin() {
    timer->TASKS_STOP  = 1;  // just in case it's running
    timer->TASKS_CLEAR = 1;
    timer->BITMODE = 3UL; // 32 bit
    timer->MODE = 0UL;    // timer, not counter
    timer->PRESCALER = prescaler; // freq = 16Mhz / 2^prescaler = 1Mhz
    timer->INTENSET = 0; // NRF_RTC_INT_COMPARE0_MASK | NRF_RTC_INT_COMPARE1_MASK
| NRF_RTC_INT_COMPARE2_MASK;
    NVIC_SetPriority(UARTE0_UART0_IRQn, 1);
    NVIC_SetPriority(irq, 2);
    NVIC_EnableIRQ(irq);
    timer->TASKS_START = 1;
}

void StepperDriver::add_motor(StepperMotor& m) {
    m.ccIndex = motorCount;
    motorCount ++;
    motors.push_back(&m);
    if (motors.size() >= reg_max) {
        Serial.println("Too many motors for this timer!");
    }
    m.timer = timer;
    m.reg_max = reg_max;
    timer->CC[m.ccIndex] = 0;
    m.speed_scale = speed_scale;
}

void StepperDriver::irq_handler() {
    for (uint8_t i=0; i<motorCount; i++) {
        if (timer->EVENTS_COMPARE[i] == 1) {
            timer->EVENTS_COMPARE[i] = 0;
            motors[i]->step();
        }
    }
}

uint32_t StepperMotor::timer_current() {
    timer->TASKS_CAPTURE[5] = 1;
    return timer->CC[5];
}

void StepperMotor::enable_interrupt(uint8_t index) {
    timer->EVENTS_COMPARE[index] = 0;
    timer->INTENSET = (1UL << 16 + index);
}

void StepperMotor::disable_interrupt(uint8_t index) {
    timer->INTENCLR = (1UL << 16 + index);
}
```