EN 530.421: Mechatronics

Final Lab Report

Github:

https://github.com/ZSchweyk/MechatronicsFinalProject

Demo:

https://www.youtube.com/watch?v=mmgVBFrsvv8&ab_channel=ThePianoBoys

Yash Permalla, Alexander Shen, Zeyn Schweyk, Alan You

Johns Hopkins University

Spring 2025

5/8/2025

Part 1: Design and Strategy

Our original design involved using a Pixy, IMU, ZigBee, and one servo motor with its arm embedded in a rotatable 3D-printed gate/wall to wind a torsional spring for our shooting mechanism. Our goal was to approach the puck as quickly as possible using our Pixy for puck detection and IMU for orientation correction via a PID controller, and use the spring-loaded servo to launch the puck into the goal, using basic angle calculations from our current ZigBee coordinates.

While initial testing seemed promising, we needed a way to release the holding torque of the winding servo motor (connected to the torsional spring) for launching. As discussed in below sections, we attempted to release the holding torque using Arduino's Servo.h `.detach()` method, but we ran into physical specification constraints that prevented us from easily doing so.
Instead, we used a MOSFET to conditionally connect our servo to its required voltage, which essentially enabled us to release holding torque by simply adjusting the terminal voltages of the transistor. However, even after testing this, we realized that our winding servo had too much internal friction while disconnected from power to efficiently and quickly release the spring for shooting. We iterated on our mechanical design to have our winding servo arm simply push our 3D-printed gate/wall from the outside to wind it, and added a second release servo motor to hold and release our spring's potential energy at the right moment. Using these two servos together, we were able to overcome our initial issues with holding torque and improve our shooting mechanism.

While testing our shooting mechanism, we found ourselves having to continuously iterate and improve on the mechanical structures of our shooting mechanism, due to mechanical fatigue of our 3D-printed gate/wall. To combat this, we adjusted our print settings to lay filament perpendicular to how force was applied on our components, which drastically improved the

lifetime of our mechanism before failure. Additionally, use of ABS over PLA helped with shock loads.

Part 2: System Overview

(i) Mechanical

The primary scoring strategy of our robot is a 5cm-long spring-loaded lever arm that accelerates the puck into the goal. During testing, the scoring method has sufficient power to fire the puck directly into the goal from across the entirety of the playing field. Additionally, the mechanism has the ability to bounce the puck off a wall and still make it into the goal from across the playing field. The robot was designed around this primary mechanism and mounting wholes on the top and bottom integrate the mechanism as both scorer and superstructure. The shooter is made of three parts, all printed with ABS for shock absorption of the shock loading that comes from the spring. A top-mounted servo motor winds the shooter back while the side-mounted servo motor locks the shooting arm back. Once locked, the top-mounted servo motor can move back to start position. To fire, the release servo motor is simply moved back to home and the arm swings forward, propelled by the energy stored in the torsion spring.

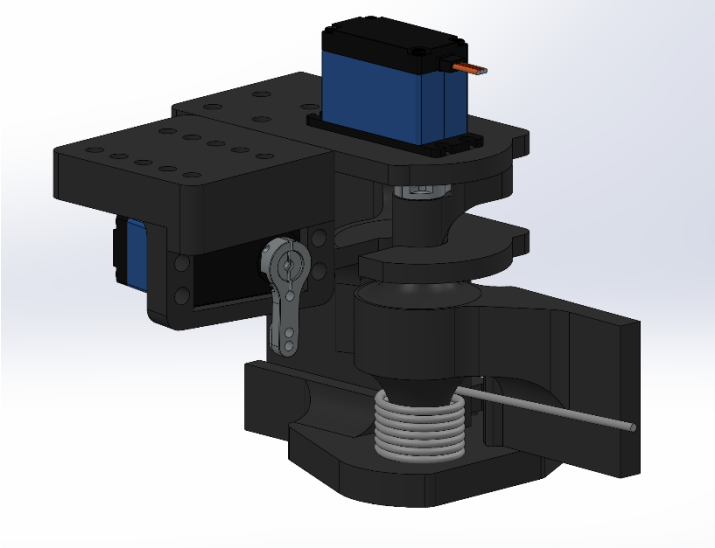


Figure. Shooter Assembly

The robot was designed to maximize the width restriction of 219.24mm, allowing for the best defense possible (block slightly less than half of the width of our goal). We have a frame perimeter of 717.67mm and a maximum height of 189.31mm (when adjustable pixy mount is vertical). The puck is controlled with a static arm and a cutout in the robot base. The arm is specifically mounted so that half and all of the puck is accessible from the side and the top, respectively.

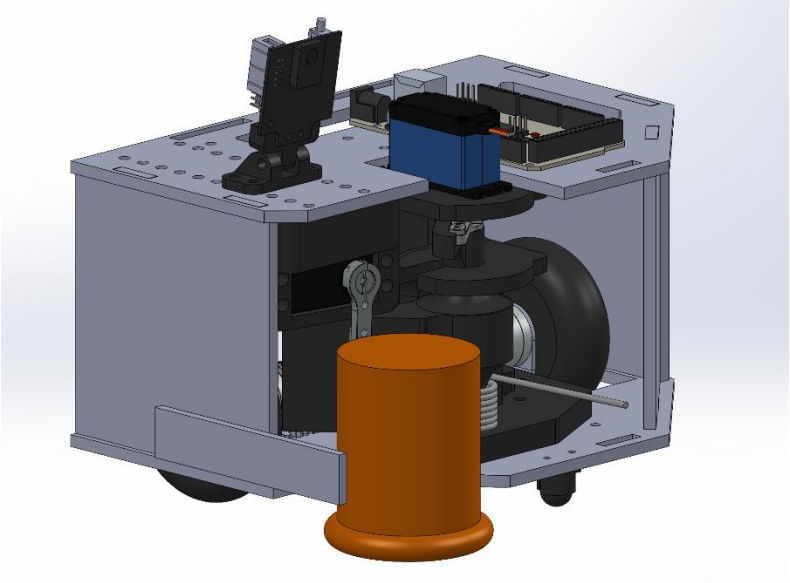
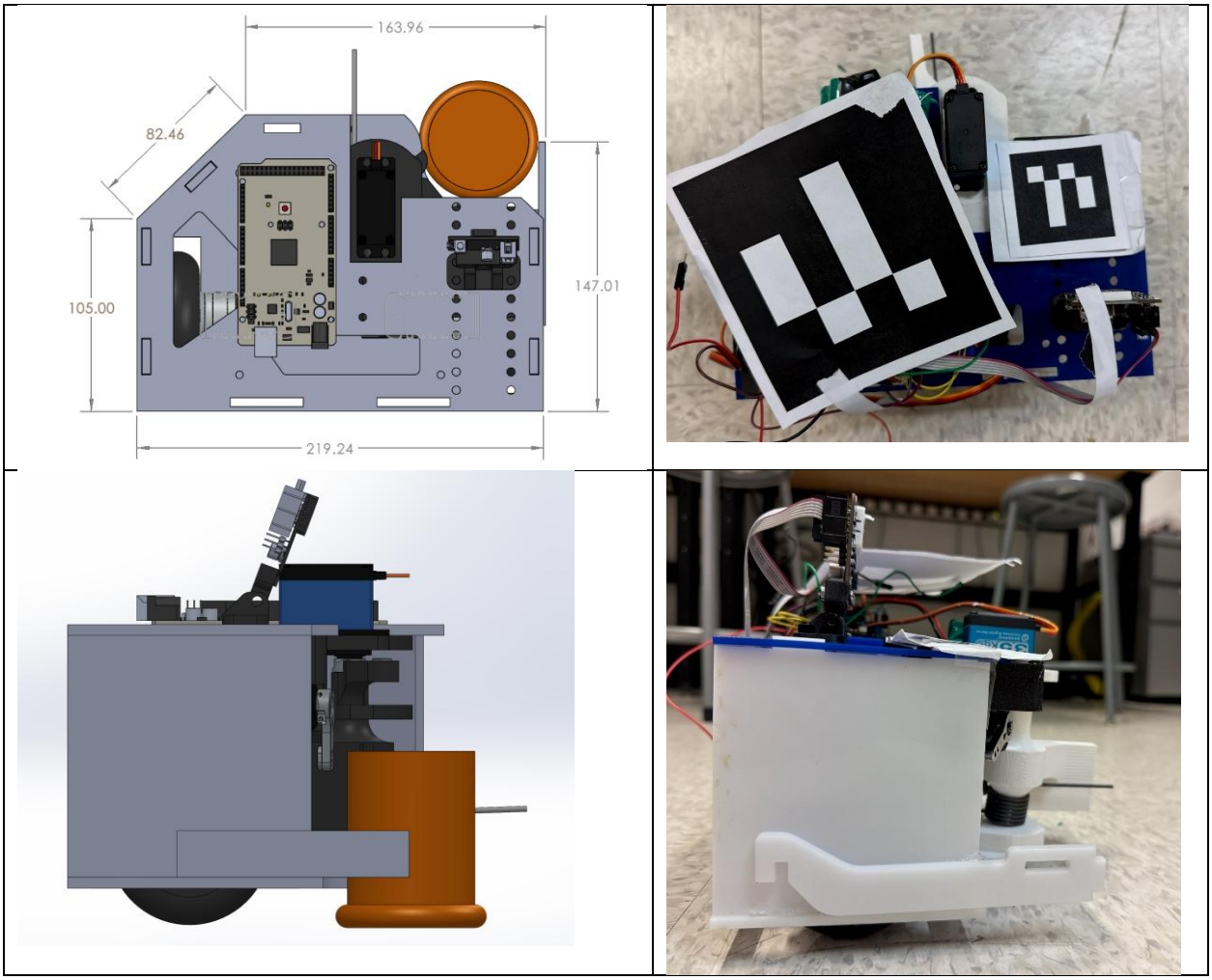


Figure. Isometric view of robot CAD



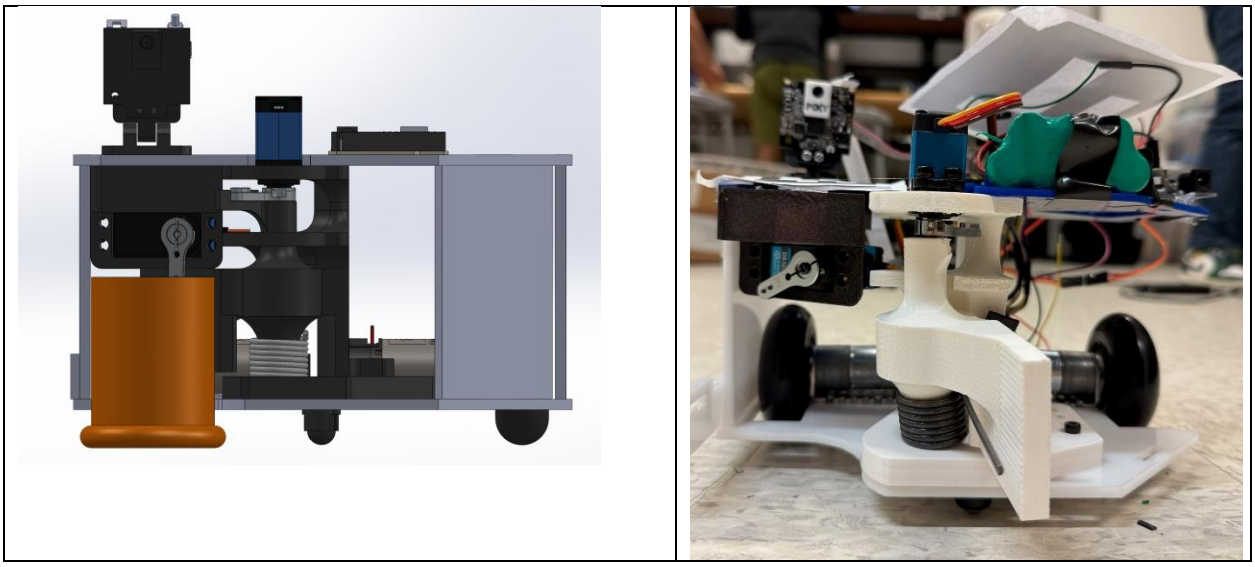


Figure. Top, Side, Front views of robot CAD and real-life.

(ii) Electrical

Our robot utilized a combination of sensors, motors, and communication modules to effectively execute puck tracking, navigation, and shooting functions. The Pixy2 vision sensor was employed to detect and track the orange puck. The Adafruit BNO055 IMU provided orientation and tilt data, enabling precise turning maneuvers and trajectory adjustments. The DualMAX14870MotorShield controlled the drive motors, delivering proportional speed control necessary for accurate puck approach and realignment. Additionally, the Zigbee module facilitated wireless communication, receiving the start byte. Collectively, these components ensured the robot maintained consistent performance across all operational states, from puck acquisition to goal shooting.

We used a servo motor for our shooting mechanism. To enable and disable our servomotor, we chose to use a MOSFET transistor, which engages and disengages the servo using a digital pin. This allows the servo to naturally reset once a puck is launched. We weren't able to fully implement retractability in the code due to emergent hardware concerns (which we can resolve in future iterations), but the MOSFET was successfully able to maintain and relax holding torque on demand, allowing for future design iterations. The wiring diagram is shown below (pins 5 and 6 can be replaced to any PWM and digital pin respectively).

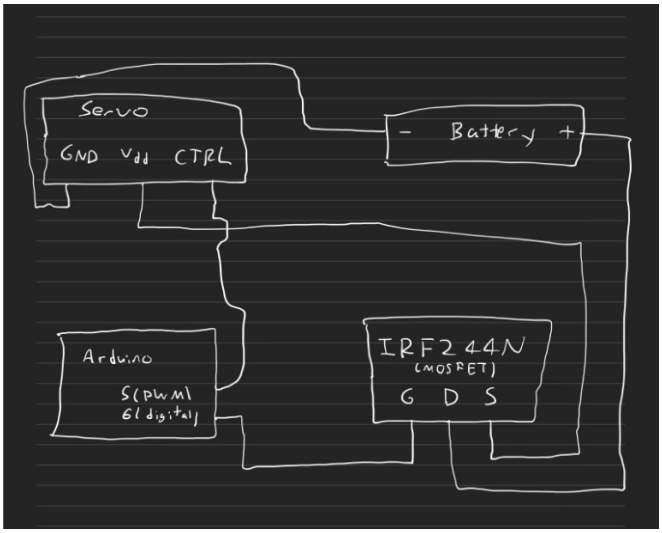


Figure. Robot wiring schematic.

(iii) Software

Our code makes use of a finite state machine, with 8 states. The main recurrent logical structure in our code is in handling losing and regaining vision of the puck, where we set up timed intervals for when a puck is lost (to avoid frame delay errors), and keep track of which side of the camera a puck is lost for more efficient capture (determining which direction to spin).

In terms of actually moving, we use PID, both for driving towards the puck (using the Pixy block coordinates to create our error signal), and for spinning to specific headings to launch based on our initialization (using the IMU to get the current heading and comparing against a target heading). We rarely need to drive forward when the puck is acquired, since our launching mechanism allows us to score from anywhere on the arena. The actual servo launching is just a matter of setting the servo angle to be high, releasing the wound-up spring.

Here is a high-level depiction of pseudo-code for pertinent parts of our code:

In our loop, we call detectPuck(), which serves as our FSM update rule. Each state has its own actuating function defined for performing the action. A few states are described here (the rest below): SHOOTING triggers forward motion and servo release for shooting, GOAL_APPROACH performs a quick motion towards a goal to provide slight momentum before shooting, SHUTTLING handles puck control before shooting by turning until the heading faces the correct half of the arena, PRESHUTTLING handles forward control to ensure the puck is captured.

ORIENTATION_CONTROL is a transition period that we used for PID testing but didn't take out of our code, so used as a rest state.

The servo, motor and sensor inputs and readings (Pixy, IMU) are the typical implementations (with one interesting Pixy processing technique being searching among blocks for the one with largest area for a given color code). The FSM progression at a high level is as follows (we also added a lot of time buffers to avoid getting stuck in infinite loops or failed control settings):

Start in START_SIGNAL, wait for Zigbee signal. Look for puck. If puck is visible, transition to PUCK_TRACKING (after a brief wait in ORIENTATION). Otherwise, check if we were already tracking or not. If not, check if we're preparing to shuttle, approach or shoot. If none of these apply, enter SEARCHING. If we were tracking, then we either lost the puck, or we have captured it, based on where it left the Pixy FOV. Based on this, we go back to SEARCHING, or progress to approaching and shooting (which looks like PRE_SHUTTLING -> SHUTTLING -> GOAL_APPROACH -> SHOOTING, all described in functionality above).

In SEARCHING, we use global information about where and when we last saw our PUCK, and spin until we see a puck again. If we see a puck, we return to PUCK_TRACKING and repeat.

PUCK_TRACKING just uses PID to track the puck, assuming it's in the Pixy FOV, driving forward and keeping it centered, before going out of FOV, at which point we transition to shuttling or go back to searching, handled in our detectPuck() function.

Part 3: Testing Results

During testing, the hockey robot successfully detected the orange puck with an average detection accuracy of 96.7**%**, based on 30 trials. The robot's approach maneuver achieved perfect positional accuracy of from the puck's centroid. The shooting mechanism, controlled by a servo motor, demonstrated a shot accuracy of 85%, successfully directing the puck toward the goal in 17 out of 20 attempts. However, accuracy decreased to **7**0% when the puck was positioned at wider angles relative to the goal.

We ran into a few issues. One was inconsistent puck detection under different lighting. We fixed this by adjusting Pixy2 signature settings to fine-tune the orange hue range, reducing false positives. We also have some overshooting during approach and fixed this by Implementing and fine tuning our PID control for precise angle adjustments.

On the final competition day, our shooter broke the night before during testing and we were not able to get a new shooter printed on time.

Part 4: Further Improvement

Improvements to our design include a mechanism for retracting the shooter mechanism. We currently preload the spring, and release using a servo, but the spring cannot be rewound during the match without human interference, requiring success on the first capture of the puck. This is a flaw in our mechanical design due to an excess amount of force and energy needed to wind the spring, so we would likely need a strong retraction mechanism, which would require rebalancing our energy and financial budget.

We also don't use the ZigBee coordinates in our code at all. We only use the ZigBee for match start, and just use the IMU for position initialization, and orient towards the goal based on heading, but this doesn't take into account translational movement over the course of the match, which changes the ideal heading. If we kept track of the robot coordinates over a match, we could calculate precise headings for launching, using the IMU to make precise turns. This would let us shoot from virtually anywhere on the field consistently, without dealing with noise in the Pixy input or other sensors.

Moreover, we could improve our puck intake. The robot can only recover the puck by turning in one direction, and the puck immediately falls out when turning in the other direction, making it inefficient to correct orientation towards a specific setpoint. This is especially inconvenient when using PID if we overshoot. Precise PID makes this problem negligible, but for robustness we should ideally have bidirectional recovery.

Part 5: References

- https://docs.arduino.cc/libraries/servo/
- https://www.pololu.com/product/2519
- https://github.com/eliwoods/arduino/blob/master/libraries/Adafruit_BNO055_t3/utility/imumaths.h
- https://www.adafruit.com/product/2472
- https://pixycam.com/pixy2/
- https://www.arrow.com/en/research-and-events/articles/pid-controller-basics-and-tutorial-pid-implementation-in-arduino
- Canvas PID_Explanation.pdf
- https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

- BNO055 IMU: https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/arduino-code

Part 6: Appendix

Material List

Component	Quantity	Purpose	Specifications/Details
Pixy2 Vision Sensor	1	Puck and goal detection	Color signature recognition, SPI/I2C/UART interface
Adafruit BNO055 IMU	1	Orientation and tilt measurement	9-axis absolute orientation sensor
DualMAX14870 Motor Shield	1	Motor control	Dual H-bridge, 5A peak per channel
Zigbee Module	1	Wireless communication	2.4 GHz, UART interface
Servo Motors	2	Puck handling and shooting mechanism	180° rotation, 5V operating voltage
DC Motors	2	Drive motors	12V, 200 RPM, geared
Motor Mounts and Wheels	2	Mobility and stability	Rubber wheels, 7 cm diameter
Battery Pack (12V, 3000mAh)	1	Power supply	Rechargeable lithium-ion
Microcontroller (Arduino Mega)	1	Main control unit	54 digital I/O, 16 analog inputs

Wiring and Connectors	Various	Circuit connections	Dupont wires, terminal blocks
Mounting Hardware	Various	Assembly	Screws, bolts, nuts, spacers
Chassis Frame	1	Structural support	Acrylic, laser-cut
Shooting Assembly	1	Shooter	ABS, PLA

Code

```
/*
* Pipeline:
* 1. Maintain orientation control
* 2. Detect an orange puck (signature 1)
* 3. Approach and follow the puck when detected
* 4. Return to orientation maintenance when puck is not visible
* 5. Search for the puck
* 6. When ready to shoot, check if in front half of initialization orientation
* If yes: shoot directly
* If no: turn LEFT to shuttle puck using left-mounted arm
* 7. During shuttling, when facing close to the initial orientation:
* Drive straight forward for a short duration
* Activate the shooter
*/

#include <Arduino.h>
#include <DualMAX14870MotorShield.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <Pixy2.h>
#include <Servo.h>



DualMAX14870MotorShield motors;
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28); // 0x28 is I2C address
Pixy2 pixy; // Create Pixy2 object
Servo shooter; // Create servo object for shooter
Servo shuttleArm; // Create servo object for shuttle arm

// Serial communication settings
#define USB_BAUD 115200

// Motor control variables
```

```
const int MAX_TURN_SPEED = 400; // Maximum speed for rotation correction
const int MIN_TURN_SPEED = 150; // Minimum speed for rotation
const int MAX_FORWARD_SPEED = 300; // Maximum speed for forward movement
int BASE_SPEED = 0; // Variable forward speed
const long interval = 500;
// Search speed configuration
const int SEARCH_SPEED = 150; // Search rotation speed
unsigned long previousMillis = 0;
// Shooter servo configuration
const int SERVO_PIN = 3; // Digital pin for shooter servo
const int SERVO_HOLD = 90; // Servo position to hold shooter (default)
const int SERVO_RELEASE = 0; // Servo position to release shooter

// Shuttle arm servo configuration
const int SHUTTLE_ARM_PIN = 5; // Digital pin for shuttle arm servo
const int ARM_EXTENDED = 180; // Position for extended arm
const int ARM_RETRACTED = 0; // Position for retracted arm

// PID constants for orientation
double Kp = 1.0; // Proportional gain
double Ki = 0.01; // Integral gain
double Kd = 0.1; // Derivative gain

// PID constants for puck tracking
double Kp_track = 1.5; // Proportional gain for puck tracking
double Kd_track = 0.3; // Derivative gain for puck tracking

// Orientation variables
double setpoint_yaw = 0.0; // Target heading
double current_yaw = 0.0; // Current heading
double error = 0.0; // Difference between setpoint and input
double last_error = 0.0; // Previous error
double integral = 0.0; // Accumulated error
double derivative = 0.0; // Rate of change of error
const double ERROR_THRESHOLD = 1.0; // Error threshold in degrees

// Shuttling variables
const int SHUTTLE_TURN_SPEED = 200; // Speed to use when shuttling puck with the static left-side arm
const unsigned long SHUTTLE_DURATION = 10000; // How long to shuttle (ms)
const double GOAL_ANGLE_THRESHOLD = 20.0; // Threshold in degrees to determine if facing goal
const int PRE_SHUTTLE_SPEED = 200; // Speed to use for pre-shuttle forward movement
const unsigned long PRE_SHUTTLE_DURATION = 300; // Duration of pre-shuttle forward movement (ms)

// Goal shooting variables
const int GOAL_APPROACH_SPEED = 250; // Speed to use for the straight forward movement before shooting
const unsigned long GOAL_APPROACH_DURATION = 800; // Duration of straight movement in ms

// Puck detection variables
const uint8_t PUCK_SIGNATURE = 1; // Orange puck signature
int puck_x = 0; // X position of puck in Pixy's fov
int puck_y = 0; // Y position of puck in Pixy's fov
```

```
int puck_width = 0; // Width of puck in Pixy's fov
int last_puck_x = 0; // Previous X position for derivative calculation
boolean puck_visible = false; // Whether puck is currently visible
unsigned long last_detection = 0; // Time of last puck detection


// FSM
enum RobotState {
START_SIGNAL,
ORIENTATION_MAINTAIN, // Maintain orientation, no puck visible
PUCK_TRACKING, // Track and follow puck
SHOOTING, // Puck disappeared while moving forward (shooting)
SEARCHING, // Puck lost while rotating (search mode)
PRE_SHUTTLING, // Brief forward movement before shuttling
SHUTTLING, // Puck ready for shuttle (not in front half)
GOAL_APPROACH // Straight forward movement before shooting after shuttle
};
RobotState currentState = START_SIGNAL; //start waiting for zigbee


// Control timing
unsigned long lastTime = 0;
const unsigned long REFRESH_RATE = 10; // Update time in milliseconds
const unsigned long SHOOTING_DURATION = 1500; // How long to continue forward after losing puck (ms)



unsigned long stateStartTime = 0; // When the current state was entered
int searchDirection = 1; // Direction to search (1=clockwise, -1=counterclockwise)


// read the current yaw from IMU
double readYawInput() {
sensors_event_t event;
bno.getEvent(&event);
// Get the x-axis orientation / yaw
double yaw = event.orientation.x;
// Convert to -180 to +180 range
if (yaw > 180) {
yaw -= 360;
}
return yaw;
}


// Function to check if current orientation is in the front half
// relative to the initial orientation
bool isInFrontHalf() {
double current = readYawInput();
double diff = abs(current - setpoint_yaw);
// Handle angle wrap-around
if (diff > 180) {
diff = 360 - diff;
}
// If difference is less than 90 degrees, we're in the front half
return (diff <= 90);
}
```

```
bool isFacingGoal() {
double current = readYawInput();
// Calculate target angle: 10 degrees to the left of initial orientation
double targetAngle = setpoint_yaw - 7.0; // Subtract for counter-clockwise (left)
// Handle angle wrap-around for target angle
if (targetAngle < -180.0) {
targetAngle += 360.0;
}
// Calculate difference between current orientation and target angle
double diff = abs(current - targetAngle);
// Handle angle wrap-around for difference calculation
if (diff > 180.0) {
diff = 360.0 - diff;
}
// Check if within +/- 3 degrees of the target angle
const double GOAL_THRESHOLD = 3.0;

// facing goal if within threshold of target angle
return (diff <= GOAL_THRESHOLD);
}



void stopMotors() {
motors.setM1Speed(0);
motors.setM2Speed(0);
}



void detectPuck() {
// Get blocks from Pixy
pixy.ccc.getBlocks();
// Check if any blocks detected
bool previously_visible = puck_visible;
puck_visible = false;
if (pixy.ccc.numBlocks) {
// Look for the largest block with our puck signature
int largest_area = 0;
int largest_index = -1;
for (int i = 0; i < pixy.ccc.numBlocks; i++) {
if (pixy.ccc.blocks[i].m_signature == PUCK_SIGNATURE) {
int block_area = pixy.ccc.blocks[i].m_width * pixy.ccc.blocks[i].m_height;
if (block_area > largest_area) {
largest_area = block_area;
largest_index = i;
}
}
}
// If we found a block with our signature
if (largest_index >= 0) {
// Store the last detection position
last_puck_x = puck_x;
```

```
// Update puck information
puck_x = pixy.ccc.blocks[largest_index].m_x;
puck_y = pixy.ccc.blocks[largest_index].m_y;
puck_width = pixy.ccc.blocks[largest_index].m_width;
puck_visible = true;
last_detection = millis();
// Print puck information
// Check if we were in SEARCHING state and found the puck
if (currentState == SEARCHING) {
currentState = PUCK_TRACKING;
stateStartTime = millis();
}
// Check if we need to transition from ORIENTATION_MAINTAIN to TRACKING
else if (currentState == ORIENTATION_MAINTAIN) {
currentState = PUCK_TRACKING;
stateStartTime = millis();
}
}
}
// If puck was not detected in this frame
if (!puck_visible) {
// Check what state we're in to determine appropriate action
if (previously_visible && currentState == PUCK_TRACKING) {
// We just lost sight of the puck while tracking it
// Calculate if we were moving forward based on recent motor speeds
// Check the y position
if (puck_y > (pixy.frameHeight * 0.7)) {
// Puck was near the bottom of the frame, probably in front of robot
// Check if we're in the front half to decide between shooting and shuttling
if (isInFrontHalf()) {
currentState = SHOOTING;
} else {
currentState = PRE_SHUTTLING;
}
stateStartTime = millis();
} else {
// Puck was not near bottom, probably gone from view
currentState = SEARCHING;
stateStartTime = millis();
// Decide search direction based on last puck position
if (puck_x < pixy.frameWidth / 2) {
searchDirection = -1; // Turn counterclockwise if puck was on left
} else {
searchDirection = 1; // Turn clockwise if puck was on right
}
}
}
else if (currentState == SHOOTING) {
// Already in shooting mode, check if time is up
if (millis() - stateStartTime > SHOOTING_DURATION) {
currentState = SEARCHING;
stateStartTime = millis();
// Pick a random search direction after shooting
searchDirection = (random(2) == 0) ? -1 : 1;
```

```
}
}
else if (currentState == PRE_SHUTTLING) {
// Check if pre-shuttling time is up
if (millis() - stateStartTime > PRE_SHUTTLE_DURATION) {
currentState = SHUTTLING;
stateStartTime = millis();
// Extend the shuttle arm when entering shuttling mode
shuttleArm.write(ARM_EXTENDED);
}
}
else if (currentState == SHUTTLING) {
// Check if shuttling time is up or if we need to transition to goal approach
if (isFacingGoal()) {
currentState = GOAL_APPROACH;
stateStartTime = millis();
return;
}
if (millis() - stateStartTime > SHUTTLE_DURATION) {
currentState = SEARCHING;
stateStartTime = millis();
// Retract the shuttle arm after shuttling is complete
shuttleArm.write(ARM_RETRACTED);
// Always search counter-clockwise after shuttling
searchDirection = -1;
}
}
else if (currentState == GOAL_APPROACH) {
// Check if goal approach time is up
if (millis() - stateStartTime > GOAL_APPROACH_DURATION) {
currentState = SHOOTING;
stateStartTime = millis();
// Retract the shuttle arm before shooting
shuttleArm.write(ARM_RETRACTED);
}
}
else if (currentState == SEARCHING) {
// In search mode, we continue searching indefinitely
}
}
}


// PID-controlled orientation maintenance/correction
void applyOrientationControl() {
// Read current orientation
current_yaw = readYawInput();
// Calculate error (-180 to +180 degrees)
error = setpoint_yaw - current_yaw;
// Handle angle wrap-around
if (error > 180) {
error -= 360;
} else if (error < -180) {
error += 360;
}
```

```
// Calculate time delta
unsigned long currentTime = millis();
double timeChange = (currentTime - lastTime) / 1000.0; // Convert to seconds
// Calculate integral term with anti-windup
integral += error * timeChange;
// Limit integral term to prevent windup
if (integral > 50) integral = 50;
if (integral < -50) integral = -50;
// Calculate derivative term (rate of change of error)
derivative = (error - last_error) / timeChange;
// Save current error for next iteration
last_error = error;
lastTime = currentTime;
// Calculate PID output
double p_term = Kp * error;
double i_term = Ki * integral;
double d_term = Kd * derivative;
double pid_output = p_term + i_term + d_term;
// Pure rotation without forward movement when in orientation mode
int leftSpeed, rightSpeed;
if (pid_output > 0) {
// Need to turn clockwise
leftSpeed = pid_output;
rightSpeed = -pid_output;
} else {
// Need to turn counterclockwise
leftSpeed = pid_output;
rightSpeed = -pid_output;
}
// Ensure minimum turn speed to overcome friction when error is significant
if (abs(error) > ERROR_THRESHOLD) {
if (leftSpeed > 0 && leftSpeed < MIN_TURN_SPEED) leftSpeed = MIN_TURN_SPEED;
if (leftSpeed < 0 && leftSpeed > -MIN_TURN_SPEED) leftSpeed = -MIN_TURN_SPEED;
if (rightSpeed > 0 && rightSpeed < MIN_TURN_SPEED) rightSpeed = MIN_TURN_SPEED;
if (rightSpeed < 0 && rightSpeed > -MIN_TURN_SPEED) rightSpeed = -MIN_TURN_SPEED;
} else {
// If error is very small, stop motors to avoid jitter
leftSpeed = 0;
rightSpeed = 0;
}
// Ensure speeds are within valid range
leftSpeed = constrain(leftSpeed, -MAX_TURN_SPEED, MAX_TURN_SPEED);
rightSpeed = constrain(rightSpeed, -MAX_TURN_SPEED, MAX_TURN_SPEED);
// Set motor speeds
motors.setM1Speed(-leftSpeed);
motors.setM2Speed(rightSpeed);
}

void trackPuck() {
// Calculate puck position error
// Pixy's x-coordinate range is 0-315, with center at 158
int center_x = pixy.frameWidth / 2;
int track_error = center_x - puck_x;
// Calculate puck tracking derivative term
```

```
int track_derivative = track_error - (center_x - last_puck_x);
// Calculate proportional and derivative terms
double p_term_track = Kp_track * track_error;
double d_term_track = Kd_track * track_derivative;
// Calculate turning adjustment
double turn_adjustment = p_term_track + d_term_track;
// Calculate speed
int distance_factor = map(puck_y, 0, pixy.frameHeight, MAX_FORWARD_SPEED, MAX_FORWARD_SPEED/3);
// slow down
int width_factor = map(puck_width, 10, 100, MAX_FORWARD_SPEED, MAX_FORWARD_SPEED/4);
width_factor = constrain(width_factor, 0, MAX_FORWARD_SPEED);
BASE_SPEED = min(distance_factor, width_factor);
// Calculate left and right motor speeds
int leftSpeed = BASE_SPEED - turn_adjustment;
int rightSpeed = BASE_SPEED + turn_adjustment;
// Ensure speeds are within valid range
leftSpeed = constrain(leftSpeed, -MAX_FORWARD_SPEED, MAX_FORWARD_SPEED);
rightSpeed = constrain(rightSpeed, -MAX_FORWARD_SPEED, MAX_FORWARD_SPEED);
// Set motor speeds (note: might need to flip signs based on your motor configuration)
motors.setM1Speed(-leftSpeed);
motors.setM2Speed(rightSpeed);
}

// Function to wait for IMU calibration
void waitForCalibration() {
uint8_t system_cal, gyro_cal, accel_cal, mag_cal;
digitalWrite(LED_BUILTIN, HIGH); // Turn on LED during calibration
// Wait for gyro to be calibrated (we only need gyro for yaw)
int calibrationAttempts = 0;
do {
bno.getCalibration(&system_cal, &gyro_cal, &accel_cal, &mag_cal);
// Flash LED to indicate calibration in progress
digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
delay(500);
calibrationAttempts++;
// After 20 attempts (10 seconds), proceed if gyro is at least partially calibrated
if (calibrationAttempts > 20 && gyro_cal >= 1) {
Serial.println("Proceeding with partial gyro calibration.");
break;
}
} while (gyro_cal < 3); // Only checking for gyro calibration
digitalWrite(LED_BUILTIN, LOW); // Turn off LED when calibrated
}

// Function to execute pre-shuttle behavior (brief forward movement)
void executePreShuttle() {
// Calculate elapsed time in pre-shuttle state
unsigned long elapsedTime = millis() - stateStartTime;
// Move forward at the pre-shuttle speed to position the puck better
motors.setM1Speed(-PRE_SHUTTLE_SPEED);
motors.setM2Speed(PRE_SHUTTLE_SPEED);
}
```

```
// Function to execute shuttle behavior
void executeShuttle() {
// Calculate elapsed time in shuttle state
unsigned long elapsedTime = millis() - stateStartTime;
// Check if we're now facing the goal during shuttling
if (isFacingGoal()) {
// We're facing the goal, time to transition to goal approach
currentState = GOAL_APPROACH;
stateStartTime = millis();
return;
}
motors.setM1Speed(SHUTTLE_TURN_SPEED);
motors.setM2Speed(SHUTTLE_TURN_SPEED);
}

// Function to execute goal approach behavior (straight movement before shooting)
void executeGoalApproach() {
// Calculate elapsed time in goal approach state
unsigned long elapsedTime = millis() - stateStartTime;
// Move forward in a straight line at the goal approach speed
motors.setM1Speed(-GOAL_APPROACH_SPEED);
motors.setM2Speed(GOAL_APPROACH_SPEED);
}

// Function to execute shooting behavior
void executeShoot() {
// Continue forward at full speed for a brief time
int shootSpeed = MAX_FORWARD_SPEED;
// Set motor speeds for straight forward motion
motors.setM1Speed(-shootSpeed);
motors.setM2Speed(shootSpeed);
// Trigger the shooter if we're within the first 100ms of the shooting state
if ((millis() - stateStartTime) < 100) {
// Activate the shooter by moving servo to release position
shooter.write(SERVO_RELEASE);
Serial.println("SHOOTER ACTIVATED!");
}
// Reset the servo to hold position when shooting duration is almost complete
if ((millis() - stateStartTime) > (SHOOTING_DURATION - 100)) {
shooter.write(SERVO_HOLD);
Serial.println("Shooter reset to hold position");
}
}

// Function to search for the puck
void searchForPuck() {
// Rotate in place to search for the puck using the configurable search speed
// Set motor speeds for rotation in the search direction
if (searchDirection > 0) {
// Clockwise rotation
motors.setM1Speed(-SEARCH_SPEED);
motors.setM2Speed(-SEARCH_SPEED);
} else {
```

```
// Counter-clockwise rotation
motors.setM1Speed(SEARCH_SPEED);
motors.setM2Speed(SEARCH_SPEED);
}

}

void setup() {
// Initialize serial communication
Serial.begin(USB_BAUD);
// Initialize LED for status
pinMode(LED_BUILTIN, OUTPUT);
// Initialize motor drivers
motors.enableDrivers();
delay(100);
// Initialize servo for shooter
shooter.attach(SERVO_PIN);
shooter.write(SERVO_HOLD); // Set to holding position by default
// Initialize servo for shuttle arm
shuttleArm.attach(SHUTTLE_ARM_PIN);
shuttleArm.write(ARM_RETRACTED); // Start with arm retracted
delay(100);
// Initialize IMU
if (!bno.begin()) {
while (1) {} // Don't proceed if IMU not working
}
// Use external crystal for better accuracy
bno.setExtCrystalUse(true);
// Initialize Pixy2 camera
pixy.init();
// Set brightness if needed
pixy.setLamp(0, 0); // Turn off lamp initially
pixy.setCameraBrightness(80); // Set brightness (0-255)
// Wait for IMU calibration
waitForCalibration();
// Set initial target orientation
setpoint_yaw = readYawInput();
// Initialize PID variables
last_error = 0;
integral = 0;
lastTime = millis();
// Initialize random number generator for search direction
randomSeed(analogRead(0));

// Start in search mode
currentState = SEARCHING;
stateStartTime = millis();
// Pick random initial search direction
searchDirection = (random(2) == 0) ? -1 : 1;
}

void loop() {
// Detect puck using Pixy2
```

```
detectPuck();
// State machine for robot behavior
switch (currentState) {
case START_SIGNAL:
// Send data from the serial monitor to XBee module
if (millis() - previousMillis >= interval) {
previousMillis = millis();
Serial1.print('?'); // Send request to XBee
}

// Receive data from the XBee module and parse coordinates
if (Serial1.available()) {
String receivedData = Serial1.readStringUntil('\n'); // Read until newline
receivedData.trim(); // Remove any leading/trailing whitespace

// Check if the received data is in the expected format
int firstComma = receivedData.indexOf(',');
if (firstComma != -1) {
String matchbyte = receivedData.substring(0, firstComma);

if (matchbyte == "1") {
currentState = ORIENTATION_MAINTAIN;
}
} else {
Serial.println("Error parsing ZigBee data.");
}
}
break;
case ORIENTATION_MAINTAIN:
// When no puck is visible, maintain orientation
applyOrientationControl();
break;
case PUCK_TRACKING:
// When puck is visible, track and follow it
trackPuck();
break;
case SHOOTING:
// When in front half, shoot the puck forward
executeShoot();
break;
case PRE_SHUTTLING:
// Brief forward movement before shuttling
executePreShuttle();
break;
case SHUTTLING:
// When not in front half, shuttle the puck using left turn
executeShuttle();
break;
case GOAL_APPROACH:
// Straight forward movement before shooting after shuttle
executeGoalApproach();
break;
case SEARCHING:
```

```
// When the puck is lost, search for it
searchForPuck();
break;
}
// Check for serial commands (for manual tuning and debugging)
if (Serial.available()) {
// Add code here for processing serial commands if needed
}

delay(10);
}
```

Budget

Item	Source	Quantity	Unit Price
Servo Motor	Amazon	2	$28.99
Assorted M4 Screw Kit	Amazon	1	$8.99
Heat set inserts	Amazon	1	$9.99
Shooter Assembly	AML: Stratasys	1	$46.03
Other Prints	AML: Prusa	1	$5.69
Spare/Broken/Test Shooter Parts	AML: Prusa	1	$20.80
		Total:	**$149.48**