Teaching JBC and Botball Movement Commands:
Extending the Sequence — Fusion, Calibration, and Waypoint Navigation

Part II: From Precision Driving to Autonomous Path Following

For Teachers and Robotics Mentors

Using the Botball and JBC Framework on the KIPR Wombat

Roger Clement

Abstract

This paper extends the scaffolded instructional sequence for teaching movement commands in the Botball and JBC framework on the KIPR Wombat introduced in the companion paper, "Teaching JBC and Botball Movement Commands: A Scope and Sequence from Beginner to Advanced." Where the original six stages carried students from a first "go-forward" command through calibrated, ramped straight-line driving, this paper introduces three additional stages that consolidate those skills into reusable functions, calibrate turning through repeated physical trials, and culminate in autonomous multi-point waypoint navigation. As with the original sequence, each stage is designed to teach a robotics capability while simultaneously reinforcing a computer science concept — here, function design and abstraction, iterative convergence and empirical calibration, and coordinate geometry and structured data.

I. Introduction

The original scope and sequence established a six-stage progression: direct motor control, velocity-based motion, tick-based precision driving, gyroscope-based turning, fused gyro-and-tick driving, and calibrated acceleration/deceleration. Taken together, those stages give students a single robot that can drive a measured distance in a straight line, smoothly and repeatably. This paper picks up exactly where that sequence left off and asks a natural next question: what happens when a classroom robot needs to do more than drive straight once — when it needs to reliably turn to an arbitrary heading, and then string many such moves together into a path? The three stages below answer that question, and in doing so mirror the structure many competitive Botball teams' own driving libraries converge on independently: a single fused drive function, a calibrated turn function, and a waypoint layer built on top of both.

II. A Note on Foundational Accuracy: Correcting TPI Calibration

Before extending the sequence, it is worth revisiting a foundational measurement that every stage from Stage 3 onward depends on: ticks-per-inch (TPI). A calibration approach sometimes seen in student and mentor libraries derives TPI indirectly — spinning a single wheel through a gyro-timed 90° arc and computing the expected arc length from wheel-base geometry. This is not the correct approach for a classroom setting, and it is worth stating plainly why: it entangles two independently error-prone systems (gyro timing accuracy and wheel-base geometry assumptions) instead of directly measuring the one quantity that matters.

The more direct fix — drive a known distance, count ticks, divide — has a hidden prerequisite that is easy to overlook: driving straight enough to trust the distance already depends on a calibrated gyro bias. This creates what looks like a chicken-and-egg problem, since Stage 7's fused drive() needs TPI to know when to stop, and TPI calibration needs a straight run to be meaningful. The dependency loop breaks cleanly, however, because bias calibration doesn't need TPI at all — it only requires the robot to sit motionless while gyro samples are averaged. Bias must always be calibrated first, before TPI, precisely because it has no dependency on distance or ticks.

Even with bias corrected, very short calibration runs proved unreliable in practice. Over a short distance, the total tick count is small enough that mechanical noise — wheel backlash, a single rounding tick, a moment of static friction breaking free — becomes a large fraction of the whole measurement rather than an insignificant one. In practice, any calibration distance over 12 inches produces a tick count large enough for that noise to average out safely; shorter runs are where the unreliability observed in testing actually came from.

A simple, repeatable physical setup solves both the “how do I get a long enough run” problem and the “how do I know my start and end points precisely” problem at once: position the back of the robot flush against a fixed PVC pipe (or wall) as the start line, and use an existing, already-marked line as the end line — the edge of the starting box or the game board's midline are natural choices, since students and teachers already know exactly where they are without measuring anything new. Clear the motor position counters, drive forward with bias-corrected straight driving, and call a dedicated calibration routine that uses the robot's IR reflectance sensors to detect that line as an objective stop condition — rather than a human judgment call — before reading tick counts to compute TPI:

```
// Stage 7 prerequisite: bias first, then TPI, using a PVC-to-reference-line run

int bias = compute_bias(20);          // robot motionless against the PVC pipe

// Robot's back is flush against the PVC pipe (start line).
// End line is an existing board reference (starting box edge or midline),
// measured once and reused — distance must be 12 inches or more.
calibrate_tpi_to_line(1000, 60.0);    // speed, expected_distance (inches)

void calibrate_tpi_to_line(int speed, double expected_distance) {
    if (expected_distance <= 0.0) {
        printf("[CALIBRATE_TPI_TO_LINE] ERROR: expected_distance must be > 0\n");
        return;
    }
    if (speed <= 0) speed = 1000;
    double old_left_tpi = left_tpi;
    double old_right_tpi = right_tpi;
    clear_motor_position_counter(LEFT_MOTOR);
    clear_motor_position_counter(RIGHT_MOTOR);

    // Phase 1: drive forward, gyro-assisted, until either IR sensor sees the line
    int theta = 0;
    while (analog(IR_LEFT) < SquareUpMidpoint && analog(IR_RIGHT) < SquareUpMidpoint)
{
        mav(LEFT_MOTOR, speed - (speed * (theta * 0.0001)));
        mav(RIGHT_MOTOR, speed + (speed * (theta * 0.0001)));
        msleep(10);
        theta += (gyro_z() - bias) * 2;
    }

    // Phase 2: squareup — bring whichever side hasn't reached the line up to it
    while (analog(IR_LEFT) < SquareUpMidpoint || analog(IR_RIGHT) < SquareUpMidpoint)
{
        mav(LEFT_MOTOR, (analog(IR_LEFT) > SquareUpMidpoint) ? -250 : 450);
        mav(RIGHT_MOTOR, (analog(IR_RIGHT) > SquareUpMidpoint) ? -250 : 450);
        msleep(1);
    }
    msleep(20);

    // Phase 3: pull back any side that overshot, so both sides settle evenly
    while (analog(IR_LEFT) > SquareUpMidpoint || analog(IR_RIGHT) > SquareUpMidpoint)
{
        mav(LEFT_MOTOR, (analog(IR_LEFT) > SquareUpMidpoint) ? -450 : 250);
        mav(RIGHT_MOTOR, (analog(IR_RIGHT) > SquareUpMidpoint) ? -450 : 250);
        msleep(1);
    }

    mav(LEFT_MOTOR, 0);
    mav(RIGHT_MOTOR, 0);
    msleep(30);

    int left_ticks = abs(gmpc(LEFT_MOTOR));
    int right_ticks = abs(gmpc(RIGHT_MOTOR));
    left_tpi = (double)left_ticks / expected_distance;
```

```
    right_tpi = (double)right_ticks / expected_distance;

    printf("[CALIBRATE_TPI_TO_LINE] new_left_tpi = %.4f  new_right_tpi = %.4f\n",
           left_tpi, right_tpi);

    double min_tpi = (left_tpi < right_tpi) ? left_tpi : right_tpi;
    double max_tpi = (left_tpi > right_tpi) ? left_tpi : right_tpi;
    if (((max_tpi - min_tpi) / max_tpi) > 0.10) {
        printf("MOTOR SWAP NEEDS TO BE CONSIDERED\n");
    }
}
```

A nice side benefit worth calling out to students: the built-in 10% mismatch check between left_tpi and right_tpi is itself a teachable moment about hardware variability — two "identical" motors rarely perform identically, and the code treats that as a real possibility to check for, not an edge case to ignore.

III. Extending the Sequence

Stage 7: The Unified Fused Drive Function

```
int compute_bias(int samples) {
    long total = 0;
    for (int i = 0; i < samples; i++) {
        total += gyro_z();
        msleep(5); // brief pause so each sample is a fresh reading, not the same
instant repeated
    }
    return total / samples;
}

void drive(int speed, double distance) {
    int bias = compute_bias(20);        // same calibration established in Section II
    cmpc(left); cmpc(right);

    int target_ticks = (int)(distance * left_tpi);  // left_tpi from
calibrate_tpi_to_line()
    int direction = (speed < 0) ? -1 : 1;
    int abs_speed = abs(speed);
    int theta = 0, correction = 0;
    int current_power = 0;

    // Acceleration phase (Stage 6 ramp, folded in here)
    while (current_power < abs_speed) {
        current_power += ACCEL_STEP;
        if (current_power > abs_speed) current_power = abs_speed;
        mav(left, direction * current_power);
        mav(right, direction * current_power);
        msleep(5);
    }

    // Main drive phase - gyro holds initial heading via accumulated theta
    while (abs(target_ticks - abs(get_motor_position_counter(left))) > 150) {
        theta += gyro_z() - bias;
        correction = (speed * theta) / 12000;
        mav(left, speed - correction);
        mav(right, speed + correction);
        msleep(5);
    }

    // Deceleration phase - power tapers proportionally to ticks remaining
    while (abs(target_ticks - abs(get_motor_position_counter(left))) > 0) {
```

```
        int ticks_left = abs(target_ticks - abs(get_motor_position_counter(left)));
        current_power = ticks_left * 30;
        if (current_power > abs_speed) current_power = abs_speed;

        theta += gyro_z() - bias;
        correction = (speed * theta) / 12000;
        mav(left, direction * (current_power - correction));
        mav(right, direction * (current_power + correction));
        msleep(5);
    }

    mav(left, 0);
    mav(right, 0);
    msleep(30);
}
```

Purpose: Consolidates the separate tick-loop, gyro-averaging line, and correction math that Stages 3 through 5 built by hand into a single reusable function. drive() is not introducing a new calibration concept — it is the first stage to depend on the bias and TPI values already established as prerequisites in Section II, and to combine acceleration, gyro-held heading, and proportional deceleration into one continuous maneuver.

Pedagogical Value:

- *What are the positives?*
  - Reinforces that calibration is a one-time setup step, not something recomputed on every call — a real distinction between configuration and runtime logic.
  - First real exercise in API design: students must decide what a caller needs to specify (speed, distance) versus what should already be known from setup (bias, left_tpi).
  - A genuine refactoring exercise — taking code students already wrote by hand in Stages 3–5 and packaging it as a single, tested function with three distinct phases (accelerate, hold heading, decelerate).
  - The proportional deceleration (ticks_left * 30, capped at abs_speed) gives a concrete, visible example of a value that scales continuously with distance remaining, rather than switching abruptly between states.
- *What are the negatives?*
  - If compute_bias() or calibrate_tpi_to_line() was skipped or run carelessly, drive() will fail silently — it has no way to know its inputs are bad, which is a good moment to discuss the limits of trusting upstream calibration.
  - Because the function now does more internally, a bug is harder to isolate; students should be encouraged to test compute_bias() and TPI calibration independently, printing their results, before trusting them inside drive().
  - The gyro correction holds the robot to its initial heading only — it has no way to turn toward a different target heading. That capability doesn't arrive until Stage 9.

Stage 8: Turn Calibration Through Repeated Convergence Testing

```
void turn(double angle) {
    int bias_local = compute_bias(20);   // same calibration pattern as drive()
    double accumulated_angle = 0.0;
    int direction = (angle >= 0) ? 1 : -1;

    const int FAST_SPEED = MAX_TURN_SPEED;
    const int SLOW_SPEED = 300;
    const double DECEL_ANGLE = 5.0; // degrees before target to start slowing

    // Fast phase: turn at full speed until within DECEL_ANGLE of target
    mav(left, -FAST_SPEED * direction);
    mav(right, FAST_SPEED * direction);
    while (fabs(accumulated_angle) < fabs(angle - (DECEL_ANGLE * direction))) {
        msleep(5);
        accumulated_angle += (gyro_z() - bias_local) * 0.005;
```

```
    }

    // Deceleration phase: turn at reduced speed for the final DECEL_ANGLE degrees
    mav(left, -SLOW_SPEED * direction);
    mav(right, SLOW_SPEED * direction);
    while (fabs(accumulated_angle) < fabs(angle)) {
        msleep(2);
        accumulated_angle += (gyro_z() - bias_local) * 0.002;
    }

    mav(left, 0);
    mav(right, 0);
    msleep(30);
}

void calibrate_turn(int trials) {
    for (int trial = 0; trial < trials; trial++) {   // outer: up to 16 full-circle
trials
        for (int leg = 0; leg < 4; leg++) {          // inner: four 90-degree legs =
one full circle
            turn(90);
        }
        msleep(500); // pause so drift from the starting orientation can be checked
by eye
    }
}
```

Purpose: Calibrates turn() through direct, visible evidence rather than a computed value. Since four consecutive 90° turns should return the robot to precisely its starting orientation, any visible drift after a full circle tells the student whether turn(90)'s internal constant is too aggressive or too weak. The loop can run up to 16 trials, but it is designed to be interrupted at any point — as soon as drift is visible, the student stops, adjusts the constant, and restarts the trial count from zero with the newly tuned value.

Pedagogical Value:

- *What are the positives?*
    - A clean example of nested iteration where each level has a genuinely different purpose — the inner loop performs one real physical maneuver (a full circle), while the outer loop repeats that maneuver so drift becomes visible rather than a one-off coincidence.
    - Extends the iterative-refinement mindset from Stage 7's bias averaging to an entire maneuver — here judged by direct observation instead of a computed value.
    - Mirrors genuine engineering practice: calibration constants tuned empirically from repeated physical trials rather than derived theoretically.
- *What are the negatives?*
    - Because the loop is meant to be watched and interrupted rather than run to completion, it doesn't produce a single clean number the way Stage 7's bias average does — the “result” is a judgment call the student makes mid-run, which is a different (and for some students, less comfortable) kind of evidence than a printed value.
    - Judging “too far” versus “too short” by eye takes some practice — worth a few guided run-throughs as a class before students calibrate independently.

Stage 9: Waypoint Driving — Autonomous Multi-Point Navigation

```
typedef struct { double x, y; } waypoint;

// Persistent robot pose - holds current position/heading between calls
double robot_x = 0.0;
double robot_y = 0.0;
double robot_heading = 0.0; // degrees
```

```
void drive_to_waypoint(waypoint target) {
    double dx = target.x - robot_x;
    double dy = target.y - robot_y;
    double target_heading = atan2(dy, dx) * (180.0 / M_PI);
    double turn_amount = target_heading - robot_heading;

    while (turn_amount > 180) turn_amount -= 360;   // normalize so the robot
    while (turn_amount < -180) turn_amount += 360;  // always takes the shorter turn

    turn(turn_amount);

    double distance = sqrt(dx * dx + dy * dy);
    drive(1000, distance);

    // Update pose from this leg's motion, then hold it for the next call
    robot_x = target.x;
    robot_y = target.y;
    robot_heading = target_heading;

    printf("[POSE] x: %.2f  y: %.2f  heading: %.2f\n", robot_x, robot_y,
robot_heading);
}

void follow_path(waypoint path[], int num_points) {
    for (int i = 0; i < num_points; i++) {
        drive_to_waypoint(path[i]); // pose carries forward automatically between
calls
    }
}
```

Purpose: Treats the Stage 7 drive() and Stage 8 turn() functions as a stable, trusted two-function API, and builds an autonomous path-follower on top of them — the payoff moment for everything the sequence has built toward. Because drive() only holds whatever heading it starts at (it cannot steer toward a new heading itself), all of the aiming work happens once, up front, in turn() — a clean division of labor between the two functions.

Pedagogical Value:

- *What are the positives?*
    - Makes trigonometry concrete and load-bearing: atan2, the distance formula, and angle normalization are no longer abstract math-class content — they are the only thing standing between a list of coordinates and a robot that visits them.
    - robot_x, robot_y, and robot_heading are the first persistent state students carry across multiple function calls in this sequence — a genuine, motivated introduction to the idea of a variable that outlives a single function call, printed each time so students can watch it accumulate.
    - The clearest possible demonstration of the value of abstraction: once drive() and turn() are trustworthy, the waypoint logic becomes almost declarative — a list of points in, a robot walking them out.
    - A direct on-ramp to Botball competition strategy, where a scoring run is, in essence, exactly a sequence of waypoints to be visited in order.
- *What are the negatives?*
    - There is no closed-loop position feedback (no camera or beacon), so error compounds: any leftover turn error from Stage 8's calibration, or any drift from drive()'s gyro-held heading, becomes part of the starting error for the next leg. This is a natural moment to introduce “dead reckoning drift” by name.
    - Structs and persistent global state both represent a real jump in complexity and should only be introduced here after students are comfortable with basic functions and variables elsewhere in the curriculum.
    - Waypoint coordinates only mean something once the class agrees on an origin and a zero-heading direction; a short, explicit class discussion establishing the field's coordinate convention is time well spent before any code is written.

IV. Conclusion

Stages 7 through 9 close the gap between a robot that can drive a single straight line and one that can autonomously walk a path across the field. Getting there required revisiting a foundation the earlier stages had glossed over — a properly sequenced bias-then-TPI calibration, verified against a real reference line rather than an arbitrary tape strip — before a single fused drive() function could be trusted at all. Stage 8 extended that same calibration instinct to turning, trading a computed constant for direct, observable evidence: watch the robot complete repeated full circles, and let visible drift tell you whether turn() is tuned. Only once both of those were solid did Stage 9's waypoint driving become possible, and the payoff was immediate — drive() and turn() disappear into the background, and a list of coordinates becomes a robot walking a path.

That said, follow_path() as written is trusting entirely to dead reckoning: robot_x, robot_y, and robot_heading are only ever computed forward from turn and drive calls, never checked against anything in the physical world. After ten or twenty waypoints in a row, how confident should we really be that those three printed numbers still match where the robot actually is? What would it take — in code, in sensors, or in the course of a match — to check that confidence periodically rather than assume it indefinitely?