Lessons Learned - Otto Quick Trip Code Development

MPU6050 Gyroscope Integration

Calibration Process

- **Critical importance**: MPU6050 must be calibrated while completely stationary
- **Sample size**: 1500-3000 samples for accurate bias measurement
- **Offset calculation**: Average of samples becomes `gyroZ_offset` subtracted from all readings
- **Surface dependency**: Calibration must be done on actual competition surface
- **Verification needed**: Test readings after calibration should be near 0.0°/s when stationary

Sensor Direction Issues

- **Mounting orientation matters**: MPU6050 readings may need inversion based on physical mounting
- **Code fix**: Added negative sign `return -((gyroZ_raw / 131.0) - gyroZ_offset)` to correct direction
- **Convention**: Positive readings = right turns, negative = left turns

Core Code Architecture

Main Program Loop

```
void loop() {
  if (walking && current_step < TOTAL_STEPS) {
    updateHeading();              // Read gyroscope, integrate to heading
    current_speed = calculateSpeed(); // Handle acceleration/deceleration
    takeStepWithCorrection();     // Walk + correct based on heading
    current_step++;
```

```
 }
}
```

Heading Monitoring System

- **Integration method**: `heading += filtered_gyroZ * dt` (continuous integration)
- **Filter applied**: 70% previous + 30% new reading to reduce noise
- **Update rate**: 50Hz (every 20ms) for smooth tracking
- **Range management**: Keep heading between -180° and +180°

Step Calculation & Distance Control

- **Target distance**: 120 inches (10 feet)
- **Step length**: 1.5 inches per step (measured and calibrated)
- **Total steps**: 80 steps calculated (120 ÷ 1.5 = 80)
- **Speed control**: Acceleration phase (15 steps) → constant speed → deceleration (15 steps)

Heading Correction Logic

Error Detection

- **Target heading**: 0° (straight line)
- **Correction threshold**: ±2.5° (smaller = more precise, but more corrections)
- **Error calculation**: `heading_error = target_heading - heading`

Correction Commands - CRITICAL LESSON

Major Discovery: Otto's servo directions were backwards!

- **Problem**: When Otto drifted left, code sent LEFT correction, making it drift MORE left
- **Root cause**: `Otto.turn(LEFT)` and `Otto.turn(RIGHT)` commands opposite to expected
- **Solution**: Swapped correction commands in code

Correction Strategy Evolution

1. **Initial approach**: Walk + turn simultaneously (caused erratic behavior)
2. **Improved approach**: Pure turning when correction needed, followed by forward step
3. **Final approach**: Gentle corrections with settling delays

Critical Issues Discovered

Servo Direction Mapping

- **Symptom**: Otto turned in continuous left circles
- **Diagnosis**: Serial output showed correct heading detection, but wrong correction response
- **Root cause**: LEFT/RIGHT servo commands were opposite to code expectations
- **Fix**: Reversed correction logic in `takeStepWithCorrection()`

Gyroscope Drift vs. Walking Vibration

- **Challenge**: Distinguishing real rotation from walking-induced sensor noise
- **Attempted solutions**:
  - Filtering (70%/30% low-pass filter)
  - Thresholds (ignore readings below certain values)
  - Better calibration (more samples, stable surface)
- **Key insight**: Always integrate approach worked better than threshold filtering

Slippery Floor Compensation

- **Problem**: Competition floors often slippery, causing servo slipping
- **Solutions implemented**:
  - Slower walking speeds (800-1000ms per step vs 300-600ms)
  - Longer settling delays between movements
  - Gentler correction turns (slower turn speed)

Debug and Monitoring Techniques

Serial Output Strategy

- **Comprehensive logging**: Raw gyro, filtered gyro, heading, heading change, correction decisions
- **Memory optimization**: Used `F()` macro to store strings in program memory vs RAM
- **Serial plotter integration**: Formatted output for Arduino Serial Plotter visualization

Testing Methodology

- **Square walk test**: Validate mechanical function before competition code
- **Manual verification**: Physically rotate Otto to verify gyroscope tracking
- **Step-by-step analysis**: Monitor each correction decision and outcome

Key Programming Patterns

State Management

```
// Competition parameters
const int TOTAL_STEPS = 80;
int current_step = 0;
bool walking = false;
float heading = 0;
float target_heading = 0;
```

Timing and Integration

```
unsigned long current_time = millis();
float dt = (current_time - last_gyro_time) / 1000.0;
heading += filtered_gyroZ * dt;  // Critical integration step
```

Error Handling

- **MPU6050 communication**: Check `Wire.available()` before reading
- **Heading range**: Wrap angles between -180° and +180°
- **Graceful degradation**: Continue operation even with sensor issues

Competition Strategy Insights

Calibration is Everything

- **Surface matters**: Different floors require different calibration
- **Time sensitivity**: Calibration degrades over time, do it fresh
- **Verification critical**: Always test calibration with known movements

"Good Enough" vs Perfect

- **Engineering balance**: Adequate precision vs over-optimization
- **Time management**: Better to have working robot than perfect non-functioning one
- **Iterative improvement**: Version 1.0 that works, then enhance for Version 2.0

Lessons for Future Development

What Worked Well

- **Systematic debugging**: Serial output provided clear insight into problems
- **Modular testing**: Square walk test isolated mechanical issues from navigation issues
- **Documentation discipline**: Tracking changes and results proved invaluable

What Needs Improvement

- **Servo characterization**: Better understanding of actual vs commanded movements
- **Environmental adaptation**: Automatic adjustment to different floor surfaces
- **Robustness**: Handle sensor failures more gracefully

Next Competition Improvements

- **Servo feedback**: Consider encoders or position feedback
- **Multi-sensor fusion**: Combine gyroscope with other sensors
- **Adaptive algorithms**: Self-tuning parameters based on performance

This comprehensive debugging and development process revealed that robotics success depends as much on systematic problem-solving and testing methodology as it does on the actual code implementation.