Control Systems and Algorithms
BaseBot FTC Robot - Technical Reference

Team Documentation

January 15, 2026

Contents

1 Ballistic Solver Algorithm

The ballistic solver computes the required launch velocity for a projectile to reach a target at a known distance and height, given a fixed launch angle.

1.1 Problem Setup

We model the projectile as a point mass under constant gravitational acceleration, neglecting air resistance. The launcher is positioned at the origin with:

- Fixed launch angle θ above horizontal
- Target at horizontal distance x (feet)
- Target at vertical height y above the launcher (feet)
- Gravitational acceleration $g = 32.2\,\mathrm{ft\,s^{-2}}$

1.2 Derivation of Launch Velocity

1.2.1 Equations of Motion

The position of the projectile at time t is given by the kinematic equations:

$$x(t) = v_0 \cos(\theta) \cdot t \tag{1}$$

$$y(t) = v_0 \sin(\theta) \cdot t - \frac{1}{2} g t^2 \tag{2}$$

where v_0 is the initial launch velocity (magnitude).

1.2.2 Eliminating Time

From equation (1), we solve for time:

$$t = \frac{x}{v_0 \cos(\theta)} \tag{3}$$

Substituting (3) into (2):

$$y = v_0 \sin(\theta) \cdot \frac{x}{v_0 \cos(\theta)} - \frac{1}{2} g \left(\frac{x}{v_0 \cos(\theta)} \right)^2 \tag{4}$$

$$y = x \tan(\theta) - \frac{g x^2}{2 v_0^2 \cos^2(\theta)} \tag{5}$$

Equation (5) is the **trajectory equation** relating the position (x, y) to the launch parameters.

1.2.3 Solving for Launch Velocity

Rearranging equation (5) to isolate v_0^2:

$$y = x\tan(\theta) - \frac{gx^2}{2v_0^2\cos^2(\theta)} \tag{6}$$

$$\frac{gx^2}{2v_0^2\cos^2(\theta)} = x\tan(\theta) - y \tag{7}$$

$$v_0^2 = \frac{gx^2}{2\cos^2(\theta)\,(x\tan(\theta) - y)} \tag{8}$$

Taking the positive square root (physical solution):

$$\boxed{v_0 = \sqrt{\frac{gx^2}{2\cos^2(\theta)\,(x\tan(\theta) - y)}}} \tag{9}$$

1.2.4 Validity Conditions

The solution is valid when the denominator is positive:

$$x\tan(\theta) - y > 0 \quad \Rightarrow \quad x\tan(\theta) > y \tag{10}$$

This means the target must be *below* the line defined by the launch angle. If this condition is not met, the target is unreachable at the given angle.

1.3 Conversion to Motor RPM

The launch velocity must be converted to motor RPM for the shooter wheels.

1.3.1 Surface Velocity Relationship

For a wheel of diameter d, the surface velocity v relates to angular velocity ω by:

$$v = \frac{\omega \cdot d}{2} = \frac{\pi d \cdot n}{60} \tag{11}$$

where n is the rotational speed in RPM. Solving for n:

$$n = \frac{60 \cdot v}{\pi d} = \frac{v}{\pi d} \cdot 60 \tag{12}$$

1.3.2 Empirical Correction Factor

Due to ball compression, slip, and other real-world effects, we apply an empirical correction factor k_{emp}:

$$n_{\text{motor}} = \frac{k_{\text{magic}} \cdot v_0}{\pi \cdot d_{\text{wheel}}} \cdot k_{\text{emp}} \tag{13}$$

where k_{magic} is a gearing/conversion constant.

1.3.3 Conversion to Encoder Ticks

The motor velocity in ticks per second is:

$$\dot{\theta}_{\text{ticks}} = n_{\text{motor}} \cdot \frac{T_{\text{rev}}}{60} \tag{14}$$

where T_{rev} is the encoder ticks per revolution:

$$T_{\text{rev}} = C_{\text{motor}} \cdot G_r \tag{15}$$

with C_{motor} being the counts per motor revolution and G_r the gear ratio.

1.4 Tuned Parameter Values

Tuned Values

$$\theta = 48\,^{\circ} = 0.838\,\text{rad} \tag{Launch Angle}$$

$$y_{\text{target}} = 3.875\,\text{ft} \tag{Target Height}$$

$$g = 32.2\,\text{ft}\,\text{s}^{-2} \tag{Gravity}$$

$$d_{\text{wheel}} = 0.315\,\text{ft} \tag{Shooter Wheel Diameter}$$

$$k_{\text{emp}} = 1.2 \tag{Empirical Factor}$$

$$k_{\text{magic}} = 120.0 \tag{Gearing Constant}$$

$$C_{\text{motor}} = 28 \tag{Counts per Motor Rev}$$

$$G_r = \frac{30}{24} = 1.25 \tag{Gear Ratio}$$

$$T_{\text{rev}} = 28 \times 1.25 = 35 \tag{Ticks per Rev}$$

1.5 Complete Algorithm

Algorithm

```
Require: Robot pose $(x_r, y_r)$, Tower position $(x_t, y_t)$
Ensure: Motor velocity in ticks/second
1: Step 1: Compute distance to target
       $$x_{\text{dist}} = \frac{\sqrt{(x_r - x_t)^2 + (y_r - y_t)^2}}{12} \quad \text{(convert inches to feet)}$$
2: Step 2: Compute required launch velocity
       $$v_0 = \sqrt{\frac{g \cdot x_{\text{dist}}^2}{2\cos^2(\theta)\,(x_{\text{dist}}\tan(\theta) - y_{\text{target}})}}$$
3: Step 3: Convert to motor RPM
       $$n_{\text{motor}} = \frac{k_{\text{magic}} \cdot v_0}{\pi \cdot d_{\text{wheel}}} \cdot k_{\text{emp}}$$
4: Step 4: Convert to ticks per second
       $$\dot{\theta}_{\text{ticks}} = n_{\text{motor}} \cdot \frac{T_{\text{rev}}}{60}$$
5: return $\dot{\theta}_{\text{ticks}}$
```

2 Limelight Targeting System

The Limelight vision system provides target tracking data used for both steering assistance and shooter speed estimation.

2.1 Limelight Output Variables

The Limelight camera provides:

- t_x: Horizontal offset from crosshair to target (degrees, range $\pm 27.25°$)
- t_y: Vertical offset from crosshair to target (degrees)
- t_a: Target area as percentage of image (0–100%)

2.2 Steering Assist Algorithm

The steering assist modifies the robot's rotational velocity to center the target in the camera's field of view.

2.2.1 Proportional Control

The rotation modifier is computed as a proportional controller:

$$r_{\text{mod}} = \frac{t_x}{t_{x,\max}} \cdot k_{\text{steer}} \tag{16}$$

where:

- $t_{x,\max} = 27.25°$ is the maximum horizontal FOV
- k_{steer} is the steering gain coefficient

2.2.2 Combined Rotation Command

The total rotation command combines driver input with vision assist:

$$\omega_{\text{total}} = \omega_{\text{driver}} + r_{\text{mod}} \tag{17}$$

where:

$$\omega_{\text{driver}} = -\text{right\_stick}_x \cdot k_{\text{turn}} \tag{18}$$

2.3 Tuned Parameter Values

Tuned Values

$$t_{x,\max} = 27.25° \qquad \text{(Max Horizontal FOV)}$$

$$k_{\text{steer}} = \begin{cases} 0.8 & \text{(Single Driver Mode)} \\ 0.4 & \text{(Dual Driver Mode)} \end{cases} \qquad \text{(Steering Gain)}$$

$$k_{\text{turn}} = 0.6 \qquad \text{(Base Turn Speed Multiplier)}$$

2.4 Empirical Motor Speed from Target Area

An alternative (fallback) method estimates shooter speed directly from target area using an empirical exponential curve fit.

2.4.1 Exponential Model

The relationship between target area and motor speed follows:

$$v_{\text{motor}}(t_a) = A \cdot e^{B \cdot t_a} + C \tag{19}$$

This models the inverse relationship between apparent target size and distance—larger targets are closer and require less shooter power.

2.4.2 Derivation of Exponential Fit

The exponential form arises from the relationship between target area and distance:

$$t_a \propto \frac{1}{d^2} \tag{20}$$

Combined with the ballistic velocity requirement (which increases with distance), an exponential decay in motor speed versus target area is expected.

2.5 Tuned Empirical Curve Parameters

Tuned Values

$$A = 0.2273 \qquad \text{(Amplitude)}$$

$$B = -0.8680 \qquad \text{(Decay Rate)}$$

$$C = 0.49 \qquad \text{(Offset/Baseline)}$$

The complete empirical formula:

$$\boxed{v_{\text{motor}}(t_a) = 0.2273 \cdot e^{-0.868 \cdot t_a} + 0.49} \tag{21}$$

3 PID and PIDF Control Theory

This section provides background on PID control theory before discussing RoadRunner's specific implementation.

3.1 General PID Controller

A **Proportional-Integral-Derivative (PID)** controller computes a control signal based on the error between a desired setpoint and measured process variable.

3.1.1 Error Definition

$$e(t) = r(t) - y(t) \tag{22}$$

where:

- $r(t)$ is the reference (setpoint)
- $y(t)$ is the measured output
- $e(t)$ is the error

3.1.2 PID Control Law

The continuous-time PID control signal is:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{23}$$

3.1.3 Component Analysis

1. **Proportional (P):** $K_p e(t)$
   - Provides immediate response proportional to current error
   - Higher K_p increases response speed but may cause overshoot
   - Cannot eliminate steady-state error alone
2. **Integral (I):** $K_i \int_0^t e(\tau)\, d\tau$
   - Accumulates past errors over time
   - Eliminates steady-state error
   - Can cause overshoot and oscillation if too high
   - Susceptible to "integral windup"
3. **Derivative (D):** $K_d \frac{de(t)}{dt}$
   - Responds to rate of change of error
   - Provides damping, reducing overshoot
   - Sensitive to noise in measurements

3.1.4 Discrete-Time Implementation

For digital systems with sample period Δt:

$$u[k] = K_p e[k] + K_i \Delta t \sum_{j=0}^{k} e[j] + K_d \frac{e[k] - e[k-1]}{\Delta t} \tag{24}$$

3.2 PIDF Controller (Feedforward Extension)

A **PIDF controller** adds a feedforward term to the standard PID:

$$u(t) = \underbrace{K_p e(t) + K_i \int e\,dt + K_d \frac{de}{dt}}_{\text{Feedback (PID)}} + \underbrace{K_f \cdot r(t)}_{\text{Feedforward}} \tag{25}$$

The feedforward term $K_f \cdot r(t)$ provides an open-loop estimate of the required control effort, reducing the burden on the feedback loop.

3.3 Feedforward Control

Feedforward control predicts the required control effort based on system dynamics, without waiting for error to accumulate.

3.3.1 Velocity-Based Feedforward

For motor control, a common feedforward model is:

$$u_{\text{ff}} = k_S \cdot \text{sign}(v) + k_V \cdot v + k_A \cdot a \tag{26}$$

where:

- k_S: Static friction compensation (minimum power to move)
- k_V: Velocity feedforward gain (power per unit velocity)
- k_A: Acceleration feedforward gain (power per unit acceleration)
- v: Desired velocity
- a: Desired acceleration

3.3.2 Physical Interpretation

1. **Static Friction (k_S):** Overcomes stiction—the motor requires a minimum voltage to begin moving.
2. **Velocity (k_V):** Counteracts back-EMF and viscous friction, both proportional to velocity.
3. **Acceleration (k_A):** Provides torque for changing velocity (Newton's second law: $F = ma$).

3.4 Feedforward + Feedback Architecture

Modern motion control combines feedforward with feedback:

$$u_{\text{total}} = u_{\text{ff}}(v_{\text{ref}}, a_{\text{ref}}) + u_{\text{fb}}(e_{\text{pos}}, e_{\text{vel}}) \tag{27}$$

- Feedforward handles the *expected* dynamics
- Feedback corrects for *disturbances* and *model errors*

This architecture provides faster response than pure feedback control.

4 RoadRunner Control Implementation

RoadRunner uses a feedforward + proportional feedback architecture for trajectory following on mecanum and tank drive robots.

4.1 System Architecture

1. **Trajectory Generation:** Creates time-parameterized paths with position, velocity, and acceleration profiles
2. **Holonomic Controller:** Computes velocity corrections based on pose error
3. **Inverse Kinematics:** Converts robot velocity to individual wheel velocities
4. **Motor Feedforward:** Computes motor power from wheel velocity commands
5. **Voltage Compensation:** Normalizes for battery voltage variation

4.2 Motor Feedforward

RoadRunner uses the velocity-based feedforward model:

$$\boxed{P_{\text{motor}} = \frac{k_S \cdot \text{sign}(v) + k_V \cdot v + k_A \cdot a}{V_{\text{battery}}}} \tag{28}$$

4.2.1 Tuned Feedforward Parameters

Tuned Values

$$k_S = 0.7277143118850069 \qquad \text{(Static Friction)}$$

$$k_V = 0.0005548815688021238 \qquad \text{(Velocity Gain (tick units))}$$

$$k_A = 0.000055 \qquad \text{(Acceleration Gain (tick units))}$$

Note: k_V and k_A are internally converted using `inPerTick`:

$$k'_V = \frac{k_V}{\text{inPerTick}} \tag{29}$$

$$k'_A = \frac{k_A}{\text{inPerTick}} \tag{30}$$

4.3 Holonomic Position Controller

For mecanum drives, RoadRunner uses a **Holonomic Controller** that applies proportional feedback independently to each degree of freedom.

4.3.1 Pose Error

The pose error between target and actual pose:

$$e_x = x_{\text{target}} - x_{\text{actual}} \qquad \text{(Axial Error)}$$

$$e_y = y_{\text{target}} - y_{\text{actual}} \qquad \text{(Lateral Error)}$$

$$e_\theta = \theta_{\text{target}} - \theta_{\text{actual}} \qquad \text{(Heading Error)}$$

4.3.2 Velocity Error (Optional)

$$e_{\dot{x}} = \dot{x}_{\text{target}} - \dot{x}_{\text{actual}} \tag{31}$$
$$e_{\dot{y}} = \dot{y}_{\text{target}} - \dot{y}_{\text{actual}} \tag{32}$$
$$e_{\dot{\theta}} = \dot{\theta}_{\text{target}} - \dot{\theta}_{\text{actual}} \tag{33}$$

4.3.3 Control Law

The velocity correction command is:

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}_{\text{correction}} = \begin{bmatrix} K_{\text{axial}} & 0 & 0 \\ 0 & K_{\text{lateral}} & 0 \\ 0 & 0 & K_{\text{heading}} \end{bmatrix} \begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} + \begin{bmatrix} K_{\text{axialVel}} & 0 & 0 \\ 0 & K_{\text{lateralVel}} & 0 \\ 0 & 0 & K_{\text{headingVel}} \end{bmatrix} \begin{bmatrix} e_{\dot{x}} \\ e_{\dot{y}} \\ e_{\dot{\theta}} \end{bmatrix} \tag{34}$$

This is equivalent to three independent PD controllers (or P controllers if velocity gains are zero).

4.3.4 Tuned Controller Gains

Tuned Values

Position Gains (Proportional):

$$K_{\text{axial}} = 5.0 \qquad \text{(Forward/Backward)}$$
$$K_{\text{lateral}} = 7.0 \qquad \text{(Left/Right)}$$
$$K_{\text{heading}} = 6.0 \qquad \text{(Rotation)}$$

Velocity Gains (Derivative):

$$K_{\text{axialVel}} = 0.0 \tag{35}$$
$$K_{\text{lateralVel}} = 0.0 \tag{36}$$
$$K_{\text{headingVel}} = 0.0 \tag{37}$$

Note: Velocity gains are set to zero, making this a pure proportional controller.

4.4 Mecanum Inverse Kinematics

The robot velocity command is converted to individual wheel velocities:

$$\begin{bmatrix} v_{FL} \\ v_{BL} \\ v_{BR} \\ v_{FR} \end{bmatrix} = \begin{bmatrix} 1 & -1 & -w \\ 1 & 1 & -w \\ 1 & -1 & w \\ 1 & 1 & w \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} \tag{38}$$

where w is related to the track width and wheelbase geometry.

4.5 Motion Profile Constraints

Trajectories are generated respecting kinematic limits:

Tuned Values

$$v_{\text{max,wheel}} = 50\,\text{in}\,\text{s}^{-1} \qquad \text{(Max Wheel Velocity)}$$

$$a_{\text{min}} = -30\,\text{in}\,\text{s}^{-2} \qquad \text{(Max Deceleration)}$$

$$a_{\text{max}} = 50\,\text{in}\,\text{s}^{-2} \qquad \text{(Max Acceleration)}$$

$$\omega_{\text{max}} = \pi\ \text{rad}\,\text{s}^{-1} \qquad \text{(Max Angular Velocity)}$$

$$\alpha_{\text{max}} = \pi\ \text{rad}\,\text{s}^{-2} \qquad \text{(Max Angular Acceleration)}$$

4.6 Drive Model Parameters

Tuned Values

$$\text{inPerTick} = 0.002958762251124946 \qquad \text{(Inches per Encoder Tick)}$$

$$\text{lateralInPerTick} = 0.00237698919596079 \qquad \text{(Lateral Inches per Tick)}$$

$$\text{trackWidthTicks} = 3651.249759136032 \qquad \text{(Track Width in Ticks)}$$

4.7 Localization: Two Dead Wheel Odometry

The robot uses two “dead wheel” encoders (passive wheels that roll along the ground) for position tracking:

Tuned Values

$$\text{parYTicks} = -953.3323765076996 \qquad \text{(Parallel Encoder Y Position)}$$

$$\text{perpXTicks} = -580.0203235462603 \qquad \text{(Perpendicular Encoder X Position)}$$

These values represent the encoder positions relative to the robot center, in encoder tick units.

4.8 Complete Control Loop

RoadRunner Control Loop (per cycle)

```
Require: Trajectory $\tau(t)$, current time $t$, battery voltage $V$
Ensure: Motor powers $P_{FL}, P_{BL}, P_{BR}, P_{FR}$
1: Step 1: Get target pose and velocities from trajectory
       $(x_t, y_t, \theta_t), (\dot{x}_t, \dot{y}_t, \dot{\theta}_t), (\ddot{x}_t, \ddot{y}_t, \ddot{\theta}_t) \leftarrow \tau(t)$
2: Step 2: Get current pose from localizer
       $(x_c, y_c, \theta_c) \leftarrow \text{localizer.getPose()}$
3: Step 3: Compute pose error
       $\mathbf{e} = (x_t - x_c,\ y_t - y_c,\ \theta_t - \theta_c)$
4: Step 4: Apply holonomic controller
       $\mathbf{v}_{\text{cmd}} = \mathbf{v}_{\text{target}} + K \cdot \mathbf{e}$
5: Step 5: Inverse kinematics to wheel velocities
       $(v_{FL}, v_{BL}, v_{BR}, v_{FR}) \leftarrow \text{MecanumKinematics.inverse}(\mathbf{v}_{\text{cmd}})$
6: Step 6: Compute motor powers via feedforward
7: for each wheel $i$ do
       $P_i = \frac{k_S \cdot \text{sign}(v_i) + k_V \cdot v_i + k_A \cdot a_i}{V}$
8: end for
9: return $(P_{FL}, P_{BL}, P_{BR}, P_{FR})$
```

A Symbol Reference

Symbol	Description	Units
v_0	Launch velocity	ft/s
θ	Launch angle	radians
g	Gravitational acceleration	ft/s^2
x	Horizontal distance to target	ft
y	Vertical height of target	ft
t_x, t_y, t_a	Limelight outputs	degrees, degrees, %
K_p, K_i, K_d	PID gains	varies
k_S, k_V, k_A	Feedforward coefficients	power units
$e(t)$	Error signal	varies
$u(t)$	Control signal	varies

Table 1: Symbol definitions

B Tuning Procedures

B.1 Feedforward Tuning

1. Run `ForwardRampLogger` to collect velocity vs. power data
2. Fit linear regression: $P = k_S \cdot \text{sign}(v) + k_V \cdot v$
3. Run acceleration tests to determine k_A

B.2 Feedback Tuning

1. Start with low gains: $K_{\text{axial}} = K_{\text{lateral}} = K_{\text{heading}} = 1.0$
2. Increase gains until oscillation begins
3. Back off by 20–30%
4. Add velocity gains if needed for damping