Contents

Atticus Terminal

Analytical Documentation for VEX Robotics Autonomous Planning Enhancement

Austin McGrath, 15800A - Atticus

January, 2026

Abstract

This document defines the concepts, data structures, algorithms, and physical assumptions implemented by the **Atticus Terminal**, a VEX Robotics autonomous editor that compiles node-and-path intent into a deterministic execution timeline. This process is customized for the physics and geometry unique to the user's robot and programming library.

1. Introduction

1.1. What Atticus Terminal is

Atticus Terminal is a VEX Robotics autonomous editor that lets a user specify intent in *field geometry* (nodes, headings, paths, offsets, and action triggers), then *compiles* that intent into a deterministic *execution timeline*. The same timeline is used consistently by:

- **Simulation:** deterministic pose stepping with geometric overlays and collision checks.
- **Time estimation:** a conservative timing model intended to predict viability.
- **Export/code generation:** mapping segments and actions into any VEX library.

1.2. Purpose: editor geometry → deterministic execution

Atticus Terminal exists to make the *execution reality* of a VEX autonomous routine explicit and to enhance the routine creation process, shortening the time taken by weeks. It answers three practical questions that usually decide whether an auton wins or fails:

1. **Will it fit?** Clearance and legality under the current robot footprint (including *reshape* states).
2. **Will it be repeatable?** Drift, deadband, settle behavior, and safety buffers under battery sag.
3. **Will it finish in** 15 s**?** Segment-level timing estimates, timeouts, and cap (voltage) limits.

The editor is not the truth by itself, as the compiled *segment timeline* is the single source of truth that drives simulation, time estimation, and export.

1.3. What the Terminal produces: a compiled segment timeline

The Terminal compiles node-and-path intent into a time-ordered list of *segments*. A segment is the smallest unit that is meaningful to: (i) timing, (ii) collision sampling, (iii) export mapping, and (iv) debug logs.

Core segment families..

- **Drive:** translation along a straight chord or a sampled path segment.
- **Turn/Face:** heading control toward a target angle or target point.
- **Swing / Arcline:** simultaneous translation and rotation at approximately constant curvature.
- **Path-follow:** spline sampled into waypoints and consumed by a follower (e.g., pure pursuit).
- **Wait:** explicit pausing.
- **Buffer:** auto or semi-auto settle margin inserted for repeatability.
- **Marker action:** an in-motion mechanism trigger at a specific progress along a segment.
- **State toggles:** persistent flags such as reverse driving, cap changes, or *reshape* enable/disable.

Minimal routine schema (conceptual)..

```
{
  "nodes": [
    {"id": 0, "x": 0, "y": 0, "face_deg": 0},
    {"id": 1, "x": 48, "y": 0, "face_deg": 90,
    ↪  "actions": ["CLAMP_CLOSE"]}
  ],
```

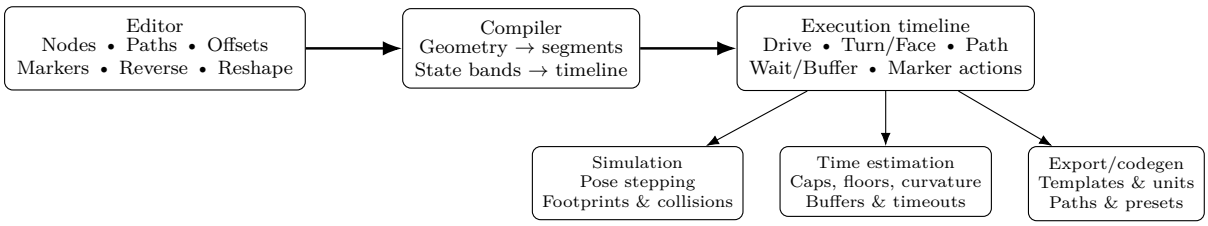

Figure 1: Editor-to-timeline pipeline: geometry and actions compile into a segment list used for simulation, estimation, and export.

```
  "edges": [
    {"from": 0, "to": 1, "type": "drive",
    ↪  "markers": [
      {"progress": 0.50, "action": "INTAKE_ON"}
    ]}
  ],
  "global": {"flip": "none", "default_cap": 0.80}
}
```

1.4. VEX constraints

A VEX autonomous routine is constrained by:

- **Hard time cap:** autonomous means the plan must include conservative timeouts and buffer margins.
- **Battery sag under load:** peak current drops available voltage; the same cap behaves differently at 12.6 V vs 11.2 V.
- **Deadband and static friction:** small commands may not move the robot at all, so minimum speed floors must exist in estimates.
- **Slip and drift:** lateral slip grows with curvature, cap, and omni usage. It affects both repeatability and legality.
- **Legality and clearance:** contact with forbidden zones or objects is a hard failure.
- **Library contracts:** exports must match the conventions of the target stack (units, heading mapping, async semantics).
- **Iteration speed:** the editor must make these constraints visible quickly (visualizers, overlays, and deterministic logs).

1.5. Applicable examples

1. **Mid-drive intake:** drive 48 in, trigger intake at 24 in, continue driving without stopping.
2. **Curved intake lane:** follow a path that stays aligned to game objects, slowing in tight curvature to reduce slip.
3. **Multi-point swing sweep:** chain two or three swing arcs to scoop field objects while keeping the intake face aligned.
4. **Matchload geometry change:** approach matchloads with *reshape* enabled (matchload down / intake down / extensions), then disable reshape after clearing the restricted region.
5. **Red/blue mirroring:** horizontally flip the routine so geometry, headings, paths, and swing directions remain consistent.

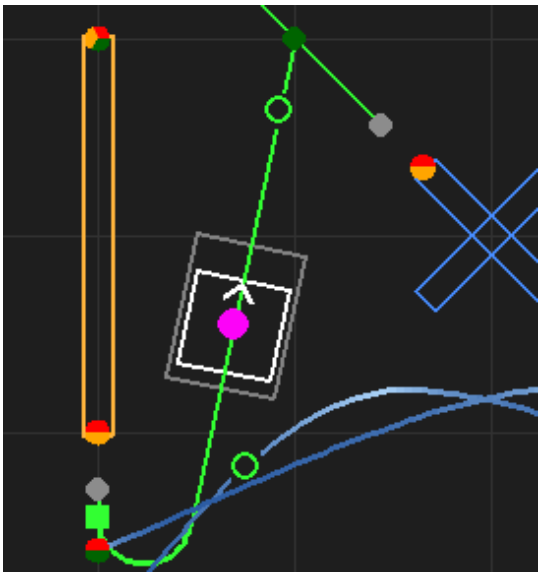

Figure 3: Canvas view showing grid, nodes, headings, footprints, restricted regions, and marker indicators.

2. Modeling inputs and coordinate conventions

2.1. Field geometry, coordinate frames, and pose

VEX autonomous planning is tile-natve: the field is naturally edited in 24 in squares. Atticus Terminal uses a *field frame* for editing and simulation and a *robot frame* for offsets and footprints. A robot pose is represented as

$$\mathbf{x} \stackrel{\text{def}}{=} (x,\ y,\ \theta), \tag{1}$$

where (x, y) is the robot reference point (chassis center or a configurable reference) and θ is heading.

Two frames are used throughout:

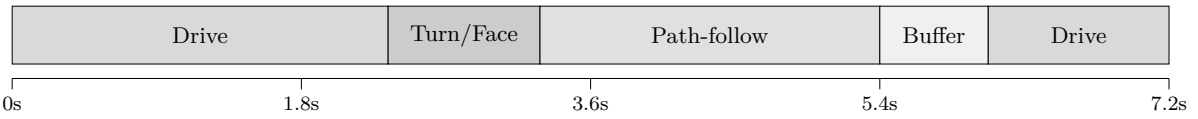

Figure 2: Compiled segment timeline schematic generated from compiler output and reused by simulation, time estimation, and export.

- **Field frame** F: fixed axes aligned to the field grid.
- **Robot frame** R: axes attached to the robot; local offsets are defined here.

By default, the editor reports *field-centric* coordinates: origin at field center, $+x$ forward (up-field), $+y$ left (driver's left). When field-centric is disabled, the origin is the initial pose for relative editing. The pixel-to-field conversion uses the screen center (c_x, c_y):

$$x = -\frac{y_{\text{px}} - c_y}{\text{PPI}}, \qquad y = -\frac{x_{\text{px}} - c_x}{\text{PPI}}.$$

The rotation matrix mapping robot-local vectors into the field frame is

$$\mathbf{R}(\theta) \stackrel{\text{def}}{=} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{2}$$

2.2. Heading conventions and mapping

A common source of mismatch between *what the driver sees on the field* and *what the code executes* is the heading convention. The Terminal therefore treats heading conversion as an explicit mapping between: (i) the editor display angle θ_{ui}, and (ii) the internal mathematical angle θ used in rotation matrices and kinematics.

A generic mapping can be written as

$$\theta = s\,(\theta_{\text{ui}} - \theta_0), \tag{3}$$

where θ_0 encodes the editor's $\theta_{\text{ui}} = 0$ direction and $s \in \{+1, -1\}$ accounts for CW/CCW sign conventions. This mapping is applied consistently across node face headings, moveToPose targets, exported angle units, and flip/mirror transformations.

In Atticus Terminal, the display convention is 0° = left and 90° = up (CW positive), while the internal math frame uses 0° = right and CCW positive. The implemented conversion is:

$$\theta = (180^\circ - \theta_{\text{ui}}) \bmod 360^\circ. \tag{4}$$

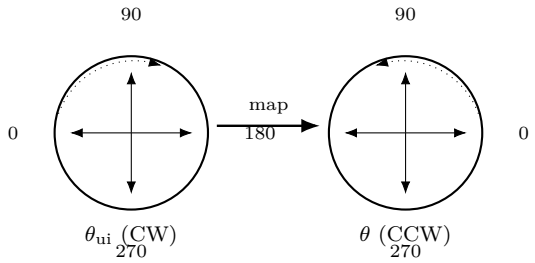

Figure 4: Heading mapping: θ_{ui} (0 left, 90 up) to θ.

```
# Example shape of a convention mapper (conceptual)
def heading_to_math(theta_ui_deg: float) -> float:
    return s * math.radians(theta_ui_deg -
    ↪  theta0_deg)
```

2.3. Unit modes and conversions used by export

The editor operates in field units (tiles and inches), but exports commonly require other unit systems, such as wheel rotations, motor degrees, or encoder ticks. The Terminal therefore anchors conversions to a small set of measured drivetrain constants.

Let D be wheel diameter and C be circumference:

$$C = \pi D. \tag{5}$$

For travel length L,

$$\text{wheel\_rot} = \frac{L}{C}, \tag{6}$$

$$\text{motor\_rot} = \text{wheel\_rot} \cdot G, \tag{7}$$

$$\text{motor\_deg} = 360 \cdot \text{motor\_rot}, \tag{8}$$

$$\text{ticks} = \text{motor\_rot} \cdot N_{\text{tpr}}, \tag{9}$$

where G is the motor-to-wheel gear ratio and N_{tpr} is ticks per motor revolution for the chosen sensor convention.

Worked sanity example.. If $D = 3.25\,\text{in}$ then $C \approx 10.21\,\text{in}$. A $48\,\text{in}$ drive corresponds to wheel_rot $\approx 48/10.21 \approx 4.70$.

Command units and the $12\,\text{V}$ *cap..* VEX motor commands are typically issued as signed units in $[-u_{\max}, u_{\max}]$ with $u_{\max} = 127$, or as voltage (mV) in PROS. The Terminal stores caps in volts (0 to $12\,\text{V}$) and converts to command magnitudes via

$$u_{\text{cmd}} = \text{clamp}\left(\frac{V_{\max}}{12}\,u_{\max},\, 0,\, u_{\max}\right),$$

with sign applied by segment direction (reverse/heading) rather than the cap itself.

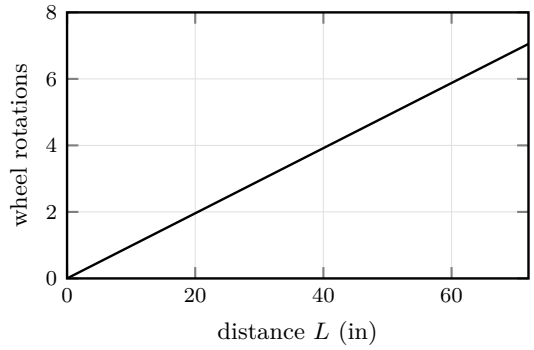

Figure 5: Distance-to-rotation conversion (sanity plot): L vs wheel rotations L/C.

2.4. Physical constants menu: measured robot parameters

Timing and legality depend on the *actual* robot. The Physical Constants menu is the surface where a team anchors the model to measurements that are stable across routines.

Drivetrain geometry.. Wheel diameter D, gear ratio G, track width w, and (optionally) a reference-point offset between the chassis center and the effective center of rotation.

Motor and voltage behavior.. A nominal battery voltage V_{oc}, a sag proxy R_{int}, cartridge RPM (effective free speed), and cap presets that clamp segment effort.

Slip and drift.. A friction/traction proxy, omni scaling, and a horizontal-drift constant that inflates expected lateral error during curved motion.

Settling and safety.. Settle error thresholds, settle time windows, default buffer times, timeout padding multipliers, and minimum timeout floors.

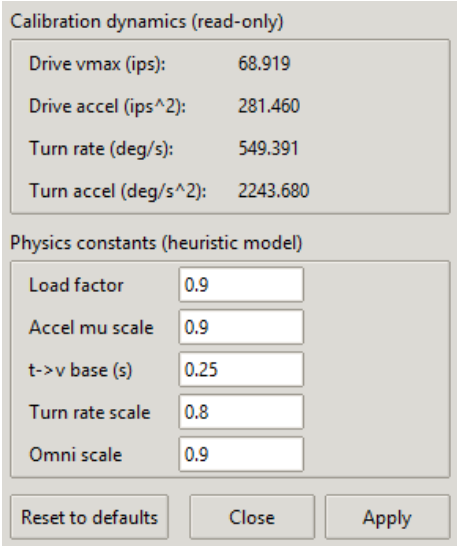

Figure 6: Physics constants window showing calibrated drivetrain limits and heuristic timing constants used by the estimator.

Path-follow tuning is visualized on-canvas so drivers can see how the lookahead target is chosen (see Figure 12 for a combined path-edit and lookahead view).

2.5. Spline curves from control points (path edges)

Spline edges are defined by *control points* the user edits in the path editor. The Terminal evaluates the curve, samples it into waypoints, and then uses those waypoints for three coupled behaviors:

1. **Visualization:** drawing the curve and its local tangent/curvature cues.
2. **Simulation / collision checks:** stepping pose samples and testing the robot footprint along the path.
3. **Time estimation and export:** computing an arc-length parameterization and follower-friendly waypoints.

A common representation in the Terminal is a *Catmull–Rom spline* (a cardinal spline). This is a good fit for VEX-style routine authoring because the points the user places on the field are *interpolation points*: the curve passes through them, so the path "goes where you clicked" without requiring separate handle points.

For uniform parameter spacing, the Terminal evaluates each spline segment as a *uniform Catmull–Rom* cubic using four consecutive control points $\mathbf{P}_{k-1}, \mathbf{P}_k, \mathbf{P}_{k+1}, \mathbf{P}_{k+2}$ and parameter $t \in [0, 1]$. In

compact matrix form,

$$\mathbf{p}(t) = \tfrac{1}{2}\,\mathbf{T}(t)\,M_{\mathrm{CR}} \begin{bmatrix} \mathbf{P}_{k-1} \\ \mathbf{P}_k \\ \mathbf{P}_{k+1} \\ \mathbf{P}_{k+2} \end{bmatrix},$$

$$\mathbf{T}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}.$$

$$M_{\mathrm{CR}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}.$$

Equivalently (and this is the form most convenient for direct evaluation in code), the same segment can be written as a weighted sum of the same four points:

$$\mathbf{p}(t) = \tfrac{1}{2} \sum_{i=0}^{3} a_i(t)\,\mathbf{P}_{k-1+i},$$

$$a_0(t) = -t^3 + 2t^2 - t, \qquad a_1(t) = 3t^3 - 5t^2 + 2,$$
$$a_2(t) = -3t^3 + 4t^2 + t, \quad a_3(t) = t^3 - t^2.$$

(These coefficients are exactly the row weights produced by $\mathbf{T}(t)M_{\mathrm{CR}}$.)

Catmull–Rom parameterization: uniform vs centripetal.. The uniform form above uses evenly spaced knots. A more general parameterization defines

$$t_{i+1} = t_i + \|\mathbf{P}_{i+1} - \mathbf{P}_i\|^{\alpha},$$

with $\alpha = 0$ (uniform), $\alpha = \frac{1}{2}$ (centripetal), and $\alpha = 1$ (chordal). Tangents become

$$\mathbf{m}_k = \frac{\mathbf{P}_{k+1} - \mathbf{P}_{k-1}}{t_{k+1} - t_{k-1}},$$

and the segment between $\mathbf{P}_k$ and $\mathbf{P}_{k+1}$ is evaluated with $u = (t - t_k)/(t_{k+1} - t_k)$ via the Hermite form:

$$\begin{aligned} \mathbf{p}(u) = {} & (2u^3 - 3u^2 + 1)\mathbf{P}_k \\ & + (u^3 - 2u^2 + u)(t_{k+1} - t_k)\mathbf{m}_k \\ & + (-2u^3 + 3u^2)\mathbf{P}_{k+1} \\ & + (u^3 - u^2)(t_{k+1} - t_k)\mathbf{m}_{k+1}. \end{aligned}$$

Centripetal spacing reduces overshoot and self-intersections when control points are unevenly spaced, which yields more stable curvature for path planning and follower tuning. Uniform spacing is faster to compute and works well when points are roughly evenly spaced and a more aggressive, "snappy" curvature is acceptable. The Terminal exposes a spline-parameterization toggle (uniform vs. centripetal) so teams can trade speed of computation for tighter curvature control when planning VEX paths near objects or matchloads.

Why this matters in the Terminal.. Control-point edits modify the geometry matrix $G = \begin{bmatrix} \mathbf{P}_{k-1} & \mathbf{P}_k & \mathbf{P}_{k+1} & \mathbf{P}_{k+2} \end{bmatrix}^{\mathsf{T}}$, which changes $\mathbf{p}(t)$ and therefore the *resampled* waypoint list. The Terminal's curvature estimate is computed from these sampled points and feeds the curvature-gain speed cap, so spline edits affect both path *shape* and *estimated time*.

Uniform arc-length resampling.. After evaluating the spline, the Terminal resamples the polyline at a uniform spacing Δs set by the point density $\Delta s = 1/$`point_density_per_in`. Given cumulative distances $s_i = \sum_{k<i} \|\mathbf{p}_{k+1} - \mathbf{p}_k\|$, each resampled point is placed by linear interpolation:

$$\mathbf{p}(s) = \mathbf{p}_i + \lambda(\mathbf{p}_{i+1} - \mathbf{p}_i), \qquad \lambda = \frac{s - s_i}{s_{i+1} - s_i}.$$

Discrete curvature estimate.. For three consecutive samples $\mathbf{p}_{i-1}, \mathbf{p}_i, \mathbf{p}_{i+1}$, the signed curvature proxy is

$$\kappa_i \approx \frac{\Delta\mathbf{p}_{i-1} \times \Delta\mathbf{p}_i}{\|\Delta\mathbf{p}_{i-1}\|\,\|\Delta\mathbf{p}_i\|\,\|\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\|}, \tag{10}$$

where $\Delta\mathbf{p}_{i-1} = \mathbf{p}_i - \mathbf{p}_{i-1}$ and the 2D cross product is $\Delta\mathbf{p}_{i-1} \times \Delta\mathbf{p}_i = x_{i-1}y_i - y_{i-1}x_i$. Curvature computed in pixel space is converted to 1/in by multiplying with PPI.

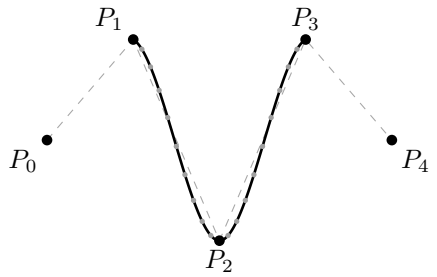

Figure 7: Catmull–Rom spline through control points. The Terminal samples the curve into waypoints, then uses those samples for curvature estimation (curvature gain), collision sampling, time estimation, and exported path assets.

2.6. Offsets, footprints, reshape, and collision constraints

Atticus Terminal treats many goals as *mechanism-first* rather than *chassis-center* targets. If the user places a node at a field location p (e.g., a goal, matchload point, or contact surface), the robot should often place a mechanism feature (intake mouth, clamp face, extension tip) at that point. This makes goal alignment easy and repeatable.

A robot-local offset $d_R = (d_x,\ d_y)$ is rotated into the field frame by the current heading θ:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \qquad p_{\text{eff}} = p_{\text{ref}} + R(\theta)\, d_R, \tag{11}$$

where p_{ref} is the chosen robot reference point (often the chassis center, but configurable).

Figure 8: Offset targeting in the Terminal: the user edits a field target p, but the robot executes by placing an effective point p_{eff} using a rotated robot-local offset.

Footprints, reshape, and legality.. Collision/legality checks operate on the robot *footprint* (hitbox) projected into the field. A common approximation is an oriented rectangle with width w and length ℓ. Let the robot-local corner set be $q_i \in \{(\pm\ell/2, \pm w/2)\}$. The field-space corners at pose (x, y, θ) are

$$Q_i = \begin{bmatrix} x \\ y \end{bmatrix} + R(\theta)\, q_i. \tag{12}$$

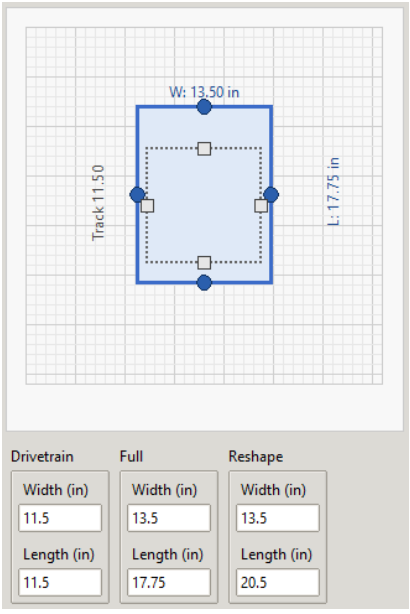

Figure 9: Oriented footprint geometry used by the Terminal for overlays and collision checks. "Reshape" swaps the footprint parameters to represent matchload-down, intake-down, or extension-deployed configurations.

Applicable advanced uses.. In VEX, offset targets and reshape previews make certain maneuvers practical: sweeping an intake along a wall without clipping, threading a clamp past a goal while reshaped, or chaining a short swing into a contact point that would be risky to script blind. The live footprint overlay is what makes these geometry-first moves repeatable.

Horizontal drift (lateral slip).. Curved motion in VEX is dominated by *slip*: the robot does not exactly follow the commanded curvature, especially with omniwheel drive or high caps. The Terminal models this via a tunable horizontal-drift constant that increases effective tracking error and motivates conservative speed limiting. In documentation plots, drift should be interpreted as a reason to (i) lower cap in high curvature, (ii) increase buffers near alignment-critical actions, and (iii) avoid aggressive chaining on tight approaches.

2.7. Voltage, resistance, and why cap changes timing

VEX drivetrains are voltage-limited and current-limited in practice (battery sag, motor heating, friction). Atticus Terminal uses *cap* $c \in (0, 1]$ as a user-facing proxy for "how much of the available drive authority is allowed" for a segment.

A minimal DC motor model (sufficient for conservative timing) is:

$$V = I R_m + k_e\, \omega, \qquad \tau = k_t\, I, \tag{13}$$

where R_m is motor winding resistance, k_e is the back-EMF constant, k_t is the torque constant, ω is shaft speed, and I is current. Eliminating I gives a linear torque–speed relation:

$$\tau(\omega) = \frac{k_t}{R_m}\,(V - k_e\,\omega) = \tau_{\text{stall}}\left(1 - \frac{\omega}{\omega_{\text{free}}}\right), \tag{14}$$

with $\omega_{\text{free}} = V/k_e$ and $\tau_{\text{stall}} = k_t V/R_m$.

Battery voltage is not constant under load. A practical approximation uses an internal resistance R_{int}:

$$V_{\text{load}} = V_{\text{oc}} - I\, R_{\text{int}}. \tag{15}$$

The Terminal's cap c conceptually scales the available drive voltage for that segment:

$$V_{\text{cmd}} = c\, V_{\text{load}}. \tag{16}$$

This is why increasing cap generally lowers estimated time. In the simplest regime $v_{\max} \propto V_{\text{cmd}}$, but higher cap also increases slip and settling, so buffers may need to increase for repeatability.

These relations are not used as a full dynamics engine; instead they justify the Terminal's tunable parameters and the shape of timing curves in §4 (cap scaling, deadband floors, curvature-gain speed caps).

3. Representation and editor workflow

3.1. Routine data model: intent → compiled timeline

The user edits *intent* (nodes, paths, actions) while the compiler produces a *timeline* (segments). This separation is what enables consistent simulation, time estimation, and export.

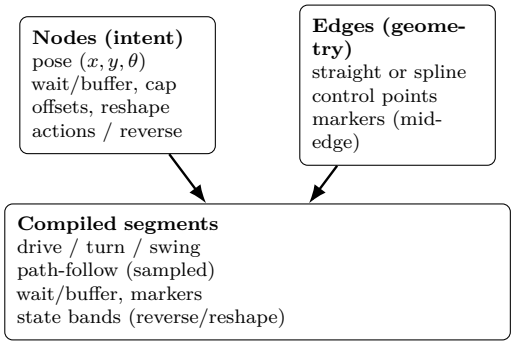

Figure 10: Routine representation: nodes (intent) and edges (geometry) compile into a segment timeline consumed by simulation, estimation, and export.

3.2. Editor visual language (what the user sees)

●	Standard node (segment boundary / anchor)
○	Hollow marker node (mid-edge action trigger)
□	Square chain node (motion chaining junction)
⟶	Path color gradient: low speed → high speed

Atticus Terminal uses consistent on-canvas semantics so that geometry, intent, and timeline structure are visible without reading menus. The symbol legend above is the quick reference used throughout the section.

- **Nodes:** boundary intent anchored to field coordinates. Includes pose, face, wait, reshape, and cap changes.
- **Hollow circles:** shift-insert previews for *mid-edge* placement and intermediate mechanism triggers.
- **Square junctions:** chain connectors indicating a non-stop transition where end-speed is carried into the next segment.
- **Path gradient:** a visual cue for speed variation (e.g., curvature gain / profile shaping), not merely decoration.
- **Field objects:** independently toggleable overlays for visibility and collision enforcement ("show" vs "collide").

These semantics are referenced later (e.g., export marker mapping in §5.1, and chaining parameters in §4.13).

3.3. Micro example: canvas intent → compiled segments → emitted code

This is a compact end-to-end example of how a visual routine becomes runnable code.

Visual intent (declarative).. A simple routine: drive forward 48 in, start the intake halfway through, then face 90 deg.

```
{
  "nodes": [
    {"id": 0, "x_in": 0,  "y_in": 0, "face_deg":
    ↪  0},
    {"id": 1, "x_in": 48, "y_in": 0, "face_deg":
    ↪  90,
     "node_actions": ["CLAMP_CLOSE"]}
  ],
  "edges": [
    {"from": 0, "to": 1, "motion": "drive",
     "cap": 0.80,
     "markers": [
       {"progress": 0.50, "action": "INTAKE_ON"}
     ]}
  ],
  "globals": {
    "default_buffer_s": 0.10,
    "timeout_pad": 1.25,
    "min_timeout_ms": 250
  }
}
```

Compiled timeline (imperative).. The compiler turns this into a deterministic segment list:

```
1) DriveSegment: L=48 in, cap=0.80, reverse=false,
↪  reshape=intake_down
   - Marker @ progress=0.50: preset=intake_on
2) FaceSegment: target=90 deg, cap=0.60,
↪  settle/buffer applied
```

One possible emitted output (library-style)..

```
chassis.moveToPoint({48, 0}, 2000,
↪  {.maxSpeed=0.80});
chassis.waitUntil(24);
intake.move_voltage(12000);
chassis.waitUntilDone();
chassis.turnToHeading(90, 900, {.maxSpeed=0.60});
```

The key idea is that the *same* segment list drives simulation and time estimation before it ever touches an export template.

3.4. Markers and intermediate mechanisms (in-motion actions)

Efficient VEX autonomous routines run mechanisms while moving. Atticus Terminal supports *mid-edge markers* that trigger actions at a specified progress along a motion segment (distance fraction, arc-length fraction, or time fraction depending on the export target).

A marker is compiled into a deterministic condition such as "trigger when segment progress $p \geq p^*$" where progress is measured along the segment arc-length s:

$$p \stackrel{\text{def}}{=} \frac{s}{s_{\text{seg}}} \in [0, 1]. \tag{17}$$

Markers do not inherently add time, as they are instead scheduled alongside the motion. When exporting to libraries that require synchronization, the exporter may emit a lightweight "waitUntil(progress)" guard (still inside the same segment window).

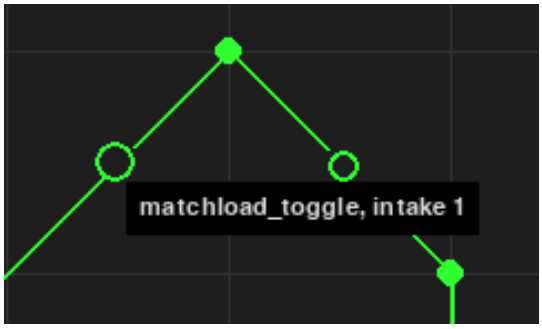

Figure 11: Marker placement UI along an edge: markers positioned at progress p^* with an action payload (e.g., INTAKE_ON, reshape enable).

3.5. Path edit mode and control-point editing

Path-follow edges are edited in a dedicated mode (e.g., the `P` hotkey) where control points, tangents, and handles are manipulated directly. To preserve spatial context, non-active paths remain visible but de-emphasized so alignment between multiple routes can be checked quickly.

Spline edges can be evaluated with either uniform or centripetal Catmull–Rom parameterization (see §2.5 for the math). Uniform spacing is the fast default; centripetal spacing reduces overshoot and produces more stable curvature when points are unevenly spaced, which is often preferable for VEX path planning near field objects and matchloads. The parameterization choice is exposed as a per-path setting in the editor, so teams can trade drafting speed for tighter curvature control on demanding routes. The visual editor also enables maneuvers that are risky to author blindly, such as 2- and 3-point swing turns that sweep intakes around field objects while preserving alignment. The moveToPose boomerang preview in Section 4.9 builds on the same path-edit workflow.

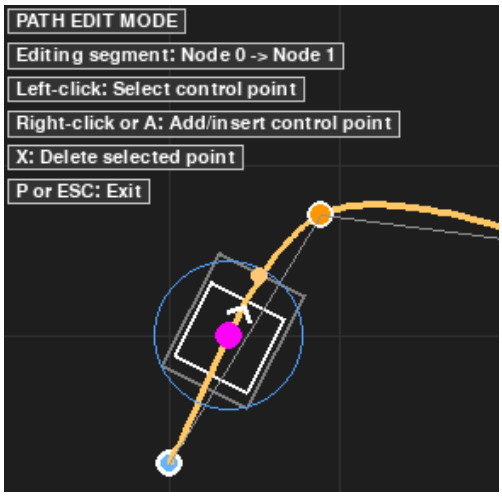

Figure 12: Path edit mode with a selected control point; the highlighted path shows the lookahead circle and pure-pursuit aim over the curve.

3.6. Undo/redo via full-state snapshots

Editor actions mutate nested structures (control points, marker lists, per-node settings, template selections). Undo/redo is implemented by storing full-state snapshots so that reversing an action restores a coherent routine without partial drift.

```
# Snapshot-centric undo model (conceptual)
def push_snapshot(state, undo_stack, redo_stack):
    undo_stack.append(deepcopy(state))
    redo_stack.clear()

def undo(state, undo_stack, redo_stack):
    if not undo_stack: return state
    redo_stack.append(deepcopy(state))
    return undo_stack.pop()
```

3.7. Nodes: discrete intent points

A node represents a location on the field and a bundle of intent that applies at that point or immediately after it. In VEX use, nodes are where you decide alignment and mechanism state choice.

Typical per-node settings (user-facing):

- **Pose:** x, y and an optional target heading θ ("face" behavior).
- **Wait:** explicit pause for sequencing (e.g., allow a clamp to finish).
- **Cap / speed bucket:** limits effort for the outgoing move.
- **Reverse toggle:** flips the implied forward direction for subsequent movement.
- **Reshape toggle:** changes robot geometry (matchload down, intake down, extensions, etc.).

- **Offsets:** shifts the effective target point to match the mechanism contact point (see §2.6).
- **Mechanism actions:** trigger presets at the node boundary (intake on/off, clamp, extension).

Offset geometry and the footprint visualizer are defined in §2.6 (see Figures 8 and 9).

3.8. Field objects toggles (visibility vs collision enforcement)

Field objects are treated as two independent concerns:

- **Visibility toggle:** whether objects/regions are rendered (planning clarity).
- **Collision toggle:** whether those same objects participate in legality checks.

This supports common workflows: temporarily hide clutter while still enforcing legality, or visually inspect an object without enforcing collision if you are drafting geometry.

Geometry menu (units and overlays).. The Geometry tab centralizes field units (distance, angle), the routine's initial heading, and overlay toggles used during alignment and legality checks. It is also where users flip a routine across the field and set reshape labels that drive matchload/intake configurations.

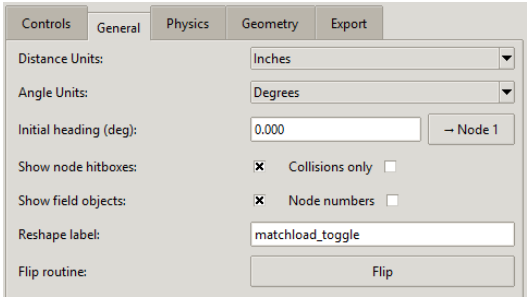

Figure 13: Geometry menu showing units, initial heading, overlay toggles, reshape label, and routine flip controls.

3.9. Calibration, mechanism presets, and templates

The Physical Constants menu is introduced in §2 (Figure 6); this section focuses on calibration and export-facing UI.

Calibration wizard and log analysis (VEX).. The Terminal includes a three-step calibration workflow: (i) export a calibration plan, (ii) run it on the robot, and (iii) paste JSON logs back into the analyzer. The wizard can export/import calibration JSON and generate a VEX auton routine that sweeps caps and profiles; when enabled, calibrated dynamics and error/time scales override nominal constants in timing and settling. The output window shows compile logs and per-segment measurements so teams can tie estimated time, drift, and settle windows to real VEX field data. Full Physics and calibration wizard screens are included in the appendix for reference (see Figures A.41 and A.42).

Mechanism preset menu (action abstraction).. Mechanism presets are named action blocks mapped to export code, e.g., `INTAKE_ON` and `CLAMP_CLOSE`. Presets are attached to nodes or mid-edge markers so the timeline can schedule them deterministically without hardcoding library calls.

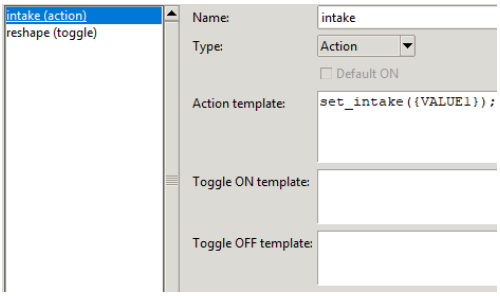

Figure 14: Mechanism preset editor surface.

Template menu (segment → code mapping).. Templates define how each compiled segment is emitted into a chosen library (LemLib/JAR/custom). The same timeline can map to different call signatures by token replacement and conditional blocks.

The full template editor surface is included in the appendix for reference (see Figure A.43).

3.10. Controls, hotkeys, and interaction affordances

Atticus Terminal prioritizes fast iteration:

- **Placement/alignment:** left-click selects or creates nodes (snap toggle `Q`); `Ctrl+Drag` constrains to 8 global directions; `Shift+Click` inserts a node on a straight segment.
- **Node commands/offsets:** right-click opens command entry (turn/wait/offset/reshape); `F` cycles offset presets; `R/G` toggle reverse/reshape.

- **Paths and markers:** `P` toggles path edit mode for control points; `M` adds/edits edge markers; `T` toggles lookahead simulation; right-click a curved segment to set per-path min/max cmd and lookahead overrides.
- **Run/log:** `Space` start/pause; `Ctrl+Space` reset. Output log: `O`.
- **I/O:** `S`/`L` save/load routines; `C` exports generated code.
- **Visual aids:** overlay toggles for hitboxes, conflicts-only view, field objects, and node numbers; hover highlights, tangent handles, heading arrows, and ghost targets.

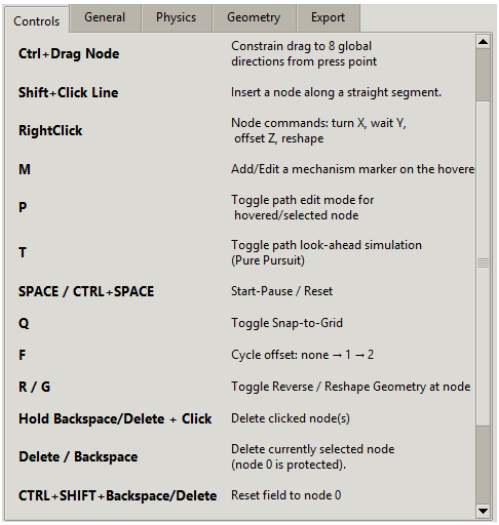

Figure 15: Controls tab listing hotkeys and interaction commands for nodes, paths, markers, and transforms.

4. Motion primitives and timing model

This section documents how the Terminal interprets motion *geometrically* and how it estimates time in a VEX-relevant way. The goal is a model that is consistent, conservative, and tunable rather than a full dynamic simulation.

Terminal-only perks are called out inline in the relevant motion sections where they are used in practice.

4.1. Differential drive kinematics (translation + yaw)

Most VEX drivetrains used with Atticus Terminal can be approximated as a differential drive at the level needed for timing, turning authority, and curvature-limited speed.

Let v_l and v_r denote left/right wheel linear speeds at the ground, and let w be effective track width. Then the chassis forward speed v and yaw rate ω satisfy:

$$v = \frac{v_r + v_l}{2}, \qquad \omega = \frac{v_r - v_l}{w}. \tag{18}$$

These identities connect directly to three Terminal behaviors: (i) face/turn segments (where ω is primary), (ii) swing/arcline segments (where v and ω co-exist), and (iii) curvature gain (where the path curvature κ implies a relationship $|\omega| \approx |v|\,|\kappa|$.

From the same geometry, the estimator derives turn limits as

$$\omega_{\max} \approx \frac{2v_{\max}}{w}, \qquad \alpha_{\max} \approx \frac{2a}{w},$$

converts them to deg/s and deg/s^2, then scales by the configured turn-voltage factor (`volts_turn/12`). The rate is multiplied by `turn_rate_scale`. It is then clamped to `turn_rate_min/max` and `TURN_ACCEL_MIN/MAX`.

Minimum command floors.. Because real robots have deadband and static friction, the estimator clamps computed speed to a floor:

$$v(s) = \max\big(v_{\text{floor}},\, v_{\text{model}}(s)\big), \tag{19}$$

so that the estimate does not assume unrealistically slow motion near zero command.

The Terminal models this as a minimum-command threshold u_0: below u_0 the drivetrain does not move; above u_0 speed rises approximately linearly.

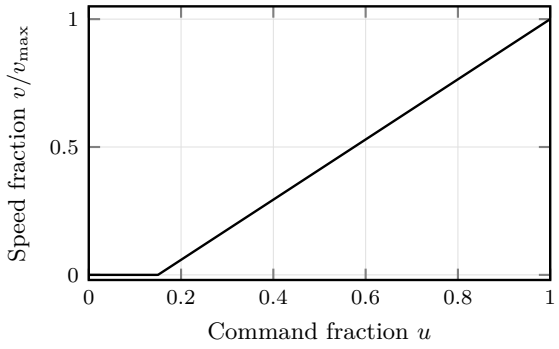

Figure 16: Illustrative deadband model: command below u_0 produces zero motion; above u_0 maps linearly to speed. This prevents the estimator from assuming arbitrarily small speeds near segment ends and improves chaining realism.

Command units and voltage cap.. Terminal command inputs are signed in $[-127, 127]$ and map to a voltage fraction; the hardware cap is 12 V:

$$V_{\text{cmd}} = 12 \cdot \text{clamp}\left(\frac{|u|}{127},\, 0,\, 1\right), \qquad u \in [-127, 127]. \tag{20}$$

The estimator uses the magnitude (via $u_{\max}/127$) to scale $v_{\max}$ and a_{est}; direction is carried by the segment's reverse/heading state rather than the cap itself.

4.2. Distances and time profiles

Two distance computations dominate the entire system:

Node-to-node distance (straight segments).. For consecutive nodes at $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$, the straight-line travel is

$$L_{\text{line}} = \|\mathbf{p}_1 - \mathbf{p}_0\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}.$$

Polyline / sampled-path arc length (paths and boomerang previews).. When a curve is sampled into points $\{\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_N\}$, the traveled distance is

$$L_{\text{path}} \approx \sum_{k=0}^{N-1} \|\mathbf{q}_{k+1} - \mathbf{q}_k\|.$$

This exact quantity is exported as `path_length_in` and drives estimator timing, collision sampling, marker triggers, and profile selection.

Time estimation assumes bounded acceleration: accelerate to a max speed, optionally cruise, then decelerate. Short segments become triangular; longer segments become trapezoidal.

Given entry/exit speeds v_0, v_1, cap $v_{\max}$, and acceleration a, the profile distances are

$$d_{\text{acc}} = \frac{v_{\max}^2 - v_0^2}{2a}, \qquad d_{\text{dec}} = \frac{v_{\max}^2 - v_1^2}{2a}. \tag{21}$$

If $d_{\text{acc}} + d_{\text{dec}} \le L$ the profile is trapezoidal with $t_{\text{acc}} = (v_{\max} - v_0)/a$, $t_{\text{dec}} = (v_{\max} - v_1)/a$, and $t_{\text{flat}} = (L - d_{\text{acc}} - d_{\text{dec}})/v_{\max}$. Otherwise it is triangular with

$$v_{\text{peak}} = \sqrt{\frac{2aL + v_0^2 + v_1^2}{2}}. \tag{22}$$

When motion chaining is enabled, segments can carry nonzero entry/exit speeds v_0, v_1; the profile solver clamps them to $v_{\max}$ and shortens or eliminates the cruise phase accordingly. The implementation also enforces minimum durations: straight moves are clamped to about 0.017 s and turns to 0.14 s to avoid zero-time segments in exports and animation.

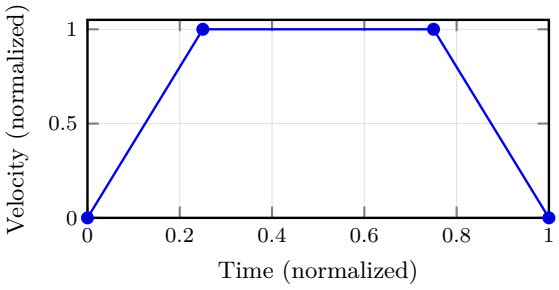

Figure 17: Trapezoidal velocity profile $v(t)$: accelerate, cruise, decelerate.

Short moves that cannot reach $v_{\max}$ follow the triangular profile below.

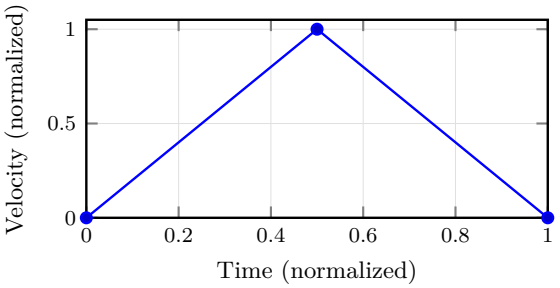

Figure 18: Triangular velocity profile $v(t)$ for short moves.

4.3. Acceleration and physical bounds

The estimator uses acceleration to decide whether a motion is triangular (never reaches $v_{\max}$) or trapezoidal (accelerate, cruise, decelerate).

Traction-limited acceleration.. A conservative physical upper bound for planar acceleration is

$$a_{\max} \le \mu g,$$

where μ is an effective traction coefficient and $g = 386.09$ in/s^2 in inch-units. In practice, μ is a calibrated proxy that folds in wheel type, load, and surface.

Motor/free-speed-based speed limit (first-order).. With wheel diameter D and wheel RPM RPM_w, the free speed proxy is

$$v_{\text{free}} \approx \pi D \cdot \frac{\text{RPM}_w}{60}.$$

If the configuration stores motor RPM and a motor-to-wheel gear ratio G (motor rotations per wheel rotation), then $\text{RPM}_w \approx \text{RPM}_m / G$.

The Terminal folds load and command scaling into a capped straight-line speed:

$$v_{\max} = \text{clamp}\big(v_{\min}, v_{\max}^{\text{cfg}}, \tag{23}$$

$$v_{\text{free}} \cdot \lambda \cdot (V_{\text{drive}}/12) \cdot (u_{\max}/127)\big), \tag{24}$$

$$t_v = \text{clamp}(t_{\min}, t_{\max}, t_0\sqrt{W/15}), \tag{25}$$

$$a_{\text{est}} = \text{clamp}(a_{\min}, a_{\max}, \tag{26}$$

$$\min(\mu g\, s_\mu, v_{\max}/t_v)). \tag{27}$$

Here λ is a load factor, $u_{\max}$ is the command ceiling, W is robot weight (lb), t_0 is the base time-to-speed constant, and s_μ is a traction scaling constant. If all-omni drive is enabled, a_{est} is scaled by $s_{\text{omni}} < 1$.

Profile multipliers and command floors.. Motion profiles scale the straight-line cap by a profile multiplier s_p (precise/normal/fast/slam). The effective cap is $v_{\max,\text{eff}} = s_p\, v_{\max}$ with $s_p \in \{0.75, 1.0, 1.15, 1.30\}$ by default (custom falls back to 1.0). For path following, command limits $u_{\min}, u_{\max}$ are interpreted in command units and converted to ips (in-per-sec) floors:

$$v_{\max,\text{path}} = v_{\max,\text{eff}} \frac{u_{\max}}{127}, \qquad v_{\min} = v_{\max,\text{eff}} \frac{u_{\min}}{127}.$$

Direction is carried by the reverse flag and caps apply to magnitude.

If torque-limited (i.e., the speed-based acceleration limit is tighter than the traction limit), the model scales a_{est} by a gear-ratio factor $\tau_g = \text{clamp}(0.5, 2.0, G)$. When calibration is enabled, measured dynamics can override $(v_{\max}, a_{\text{est}}, \omega_{\max}, \alpha_{\max})$ to anchor the model to the reality.

These constants live in the Physical Constants menu so timing, caps, and legality sampling are anchored to the *actual robot*.

4.4. Cap, voltage, resistance, and conservative speed limits

Cap c is the primary user-facing speed/authority limiter. In the Terminal it acts as a knob that scales: (i) the effective max speed used by motion profiles, and (ii) the aggressiveness that drives slip and settling.

A minimal DC motor model motivates why voltage scaling changes achievable speed and torque:

$$V = I R_m + k_e\, \omega, \qquad \tau = k_t\, I, \tag{28}$$

which implies the linear torque–speed relation

$$\tau(\omega) = \tau_{\text{stall}} \left(1 - \frac{\omega}{\omega_{\text{free}}}\right). \tag{29}$$

Battery sag can be modeled with internal resistance:

$$V_{\text{load}} = V_{\text{oc}} - I\, R_{\text{int}}, \tag{30}$$

so high-load segments can run at a lower effective voltage than the nominal 12 V.

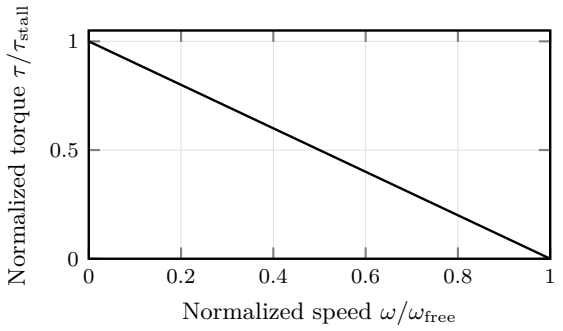

Figure 19: Motor torque–speed curve (normalized). This motivates conservative accel limits and explains why aggressive cap settings can trade raw speed for slip and settling.

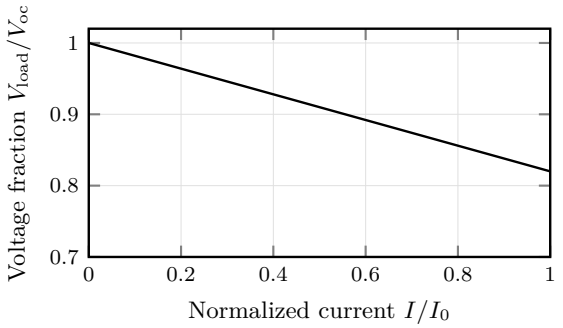

Figure 20: Battery sag under load (normalized). High-load segments can run below nominal voltage, so caps and buffers remain conservative.

Derived limits used by the estimator.. In code, the Terminal derives a conservative straight-line limit from drivetrain parameters:

$$v_{\max} \approx \frac{\pi D\,\text{RPM}}{60} \cdot \ell_f \cdot \frac{V_{\text{straight}}}{12} \cdot \frac{u_{\max}}{127}, \tag{31}$$

$$a_{\text{trac}} \approx k_\mu \cdot \mu \cdot 386.09, \tag{32}$$

$$t_{\text{to}\,v} \approx \min\Big(t_{\max}, \max\big(t_{\min},\, t_{\text{base}}\sqrt{W/15}\big)\Big), \tag{33}$$

$$a_{\text{motor}} \approx \frac{v_{\max}}{t_{\text{to}\,v}}, \tag{34}$$

$$a \approx \min(a_{\text{trac}}, a_{\text{motor}}) \cdot \gamma_g. \tag{35}$$

Here ℓ_f is the load_factor, $u_{\max}$ is max_cmd, and k_μ = accel_mu_scale from the configuration. Command values are signed motor commands in $[-127, 127]$; the magnitude scales voltage to the motors, while direction is handled by sign or the reverse flag in the timeline. In implementation, $v_{\max}$ is clamped between `vmax_min` and `vmax_max`, and $t_{\text{to}\,v}$ is computed as `t_to_v_base`$\sqrt{W/15}$ (weight scaling) then clamped to `t_to_v_min/max`. The traction-limited

acceleration is also clamped by `accel_min/max`. The estimator uses $a = \min(a_{\text{trac}}, a_{\text{motor}})$; if motor-limited, it scales a by a bounded gear-ratio factor, with $\gamma_g = \text{clamp}(G, 0.5, 2.0)$ applied only in that case. All-omni drivetrains apply an additional `omni_scale`. When calibration is enabled, constants like `load_factor`, `accel_mu_scale`, `t_to_v_base`, and `turn_rate_scale` can be overridden, and dynamic calibration can directly override $v_{\max}$, a, $\omega_{\max}$, and $\alpha_{\max}$.

Direct connection to the estimator.. These plots justify the Terminal's timing design: cap scales $v_{\max}$, deadband imposes $v_{\min}$, curvature gain lowers $v(s)$ in tight turns, and buffers account for settling.

4.5. Turns / face segments: heading wrap and bounded angular rate

Turns are treated as an angular analog of straight motion: bounded angular acceleration up to $\omega_{\max}$, optional cruise, then deceleration. The correct "short way around" must be handled with a wrap function:

$$\Delta\theta = \text{wrap}(\theta_{\text{target}} - \theta_{\text{now}}). \tag{36}$$

The angular profile mirrors the linear case:

$$t_{\text{acc}} = \frac{\omega_{\max}}{\alpha}, \qquad \theta_{\text{acc}} = \tfrac{1}{2}\alpha t_{\text{acc}}^2.$$

If $2\theta_{\text{acc}} \geq |\Delta\theta|$, the turn is triangular with $t = 2\sqrt{|\Delta\theta|/\alpha}$; otherwise it is trapezoidal with $t = 2t_{\text{acc}} + \big(|\Delta\theta| - 2\theta_{\text{acc}}\big)/\omega_{\max}$.

Turn-rate bounds from track width.. With track width w, the yaw-rate and yaw-accel limits are derived from straight-line caps:

$$\omega_{\max} \approx \text{clamp}\big(\omega_{\min}, \omega_{\max}^{\text{cfg}}, \tag{37}$$

$$\frac{2v_{\max}}{w} \cdot \frac{V_{\text{turn}}}{12} \cdot s_\omega\big), \tag{38}$$

$$\alpha_{\max} \approx \text{clamp}\big(\alpha_{\min}, \alpha_{\max}^{\text{cfg}}, \tag{39}$$

$$\frac{2a_{\text{est}}}{w} \cdot \frac{V_{\text{turn}}}{12} \cdot s_\omega\big). \tag{40}$$

4.6. Swing turns and arclines

An **arcline** is a constant-curvature arc segment. It is the geometric model behind swing turns: the robot moves forward while turning at (approximately) constant curvature. If the arc has radius R and heading change $\Delta\theta$, arc length is

$$s = R\,|\Delta\theta|, \tag{41}$$

and curvature is

$$\kappa \stackrel{\text{def}}{=} \frac{1}{R}. \tag{42}$$

Atticus Terminal treats CW/CCW selection as a first-class choice because VEX routines often require a non-shortest turn to maintain clearance.

In practice, chained two- and three-arc swing sweeps are used to swoop along field objects while keeping an intake or clamp aligned. The swing preview and collision overlays make these sequences feasible without trial-and-error on the field.

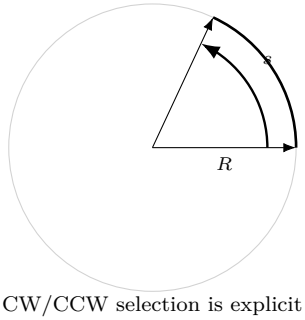

Figure 21: Swing/arcline schematic: show radius R, arc length s, and explicit CW/CCW selection.

Swing arc geometry.. A swing segment uses a fixed radius $R = \frac{1}{2}w$ (half the track width). The pivot center is offset from the start point by R at $\pm 90°$ from the start heading θ_0, depending on CW/CCW side:

$$\mathbf{c} = \mathbf{p}_0 + R\begin{bmatrix}\cos(\theta_0 \pm 90°)\\ \sin(\theta_0 \pm 90°)\end{bmatrix}. \tag{43}$$

For a candidate heading change $\Delta\theta$, the arc end position is

$$\mathbf{p}_{\text{end}} = \mathbf{c} + \mathbf{R}(\Delta\theta)\,(\mathbf{p}_0 - \mathbf{c}), \tag{44}$$

and the arc length is $s = R|\Delta\theta|$. The solver iterates $\Delta\theta$ so the final heading $\theta_0 + \Delta\theta$ aims at the target (or the moveToPose lead-in "carrot"), while respecting the requested swing direction and reverse mode.

Auto swing direction and solver.. When the swing direction is set to `auto`, the sign of $\Delta\theta$ is chosen from the initial heading error; reverse mode flips the geometric direction. The solver refines the turn by iterating

$$\begin{aligned}\Delta\theta_{k+1} = \text{sgn}(\text{dir}) \cdot \Big|\Delta\theta_k \\ + \text{wrap}\big(\theta_{\text{des}} - (\theta_0 + \Delta\theta_k)\big)\Big|,\end{aligned} \tag{45}$$

typically for a small fixed number of iterations (up to eight) until the heading error converges.

4.7. Spline paths, curvature, and curvature-gain speed limiting

Spline edges are sampled into waypoints for simulation and export. A path is parameterized by arc-length s or a normalized progress variable. A

key physical observation in VEX is that tight turns increase slip risk, so speed should be reduced when curvature is high. In practice, threaded spline approaches that hug obstacles or rely on reshape footprints are only safe to author with the on-canvas geometry overlay and path gradient showing where caps will bite.

The curve is resampled at uniform arc-length spacing $\Delta s = 1/\rho$, where ρ is the configured point density (points per inch). Curvature at sample i is estimated from the local triplet:

$$\kappa_i \approx \frac{(\mathbf{p}_i - \mathbf{p}_{i-1}) \times (\mathbf{p}_{i+1} - \mathbf{p}_i)}{\|\mathbf{p}_i - \mathbf{p}_{i-1}\| \, \|\mathbf{p}_{i+1} - \mathbf{p}_i\| \, \|\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\|}, \tag{46}$$

where $a \times b = a_x b_y - a_y b_x$. Curvature is converted to 1/in via $\kappa_{\text{in}} = |\kappa| \, \text{PPI}$.

Curvature along a path is treated as a function $\kappa(s)$. Time along a path is estimated by

$$T = \int \frac{\mathrm{d}s}{v(s)}. \tag{47}$$

Physical upper bound (traction).. Even with perfect control, a drivetrain cannot exceed the friction-limited lateral acceleration. Using $a_{\text{lat}} = v^2|\kappa|$ and $a_{\text{lat}} \lesssim \mu g$ gives

$$v_{\max}(\kappa) \approx \sqrt{\frac{\mu g}{|\kappa|}}. \tag{48}$$

The Terminal's *curvature gain* k_κ is a user-friendly proxy for this limit, absorbing unknowns like wheel type, omni slip, and load distribution.

A practical curvature-gain limiter is modeled as a speed cap of the form

$$v(s) = \min\big(v_{\text{cap}}, \; \frac{v_0}{1 + k_\kappa \, |\kappa(s)|}\big), \tag{49}$$

with $v_{\text{cap}} = v_{\max,\text{path}}$ from the command ceiling and a floor $v_{\min}$ from the command minimum.

In implementation, curvature from resampled points is converted to 1/in and scaled by a tuning factor:

$$\begin{aligned} v_{\text{cap}}(\kappa) &= \max\left(v_{\min}, \; \frac{v_{\max}}{1 + k_\kappa \, |\kappa| \, 120}\right), \\ v_{\text{cap}} &\le \frac{\omega_{\max}}{|\kappa|}. \end{aligned} \tag{50}$$

Time along the path is computed by summing per-segment $\Delta s / v_{\text{cap}}$, then taking $T = \max(T_{\text{base}}, T_{\text{profile}})$ so acceleration limits are respected.

The curvature profile and resulting cap are kept together so the path-parameterized signal and its speed limits are read as a unit (see Figures 22 and 23).

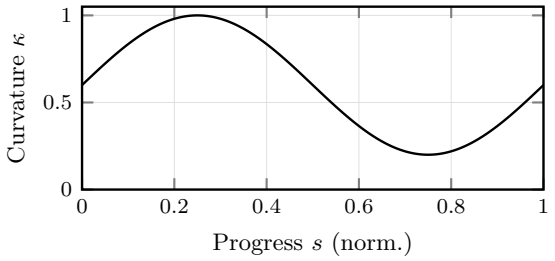

Figure 22: Curvature profile $\kappa(s)$ along a normalized path.

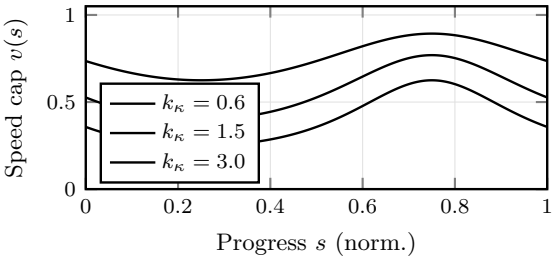

Figure 23: Resulting speed cap $v(s)$ for several curvature-gain settings k_κ.

The plot below shows the curvature-only envelope implied by the traction bound, independent of path progress.

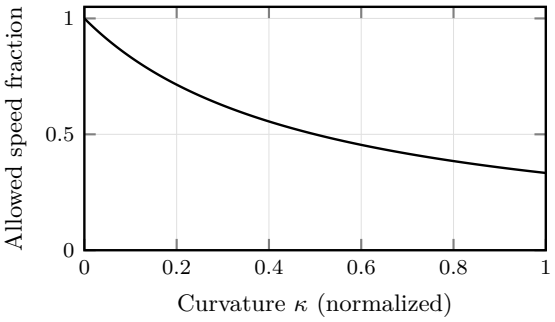

Figure 24: Illustrative curvature-based max-speed cap $v_{\max}(\kappa)$. The Terminal's *curvature gain* setting reduces allowed speed in high-curvature regions to improve tracking and reduce drift; this cap feeds both time estimation and follower setpoints.

4.8. Moving to a pose

moveToPose targets (x, y, θ) simultaneously by shaping the approach so heading error is reduced near arrival. This often improves repeatability and efficiency compared to "drive then turn" when the final heading matters for scoring.

4.9. moveToPose Boomerang controller preview and mathematics

The Terminal's move-to-pose preview path is constructed as a **quadratic Bézier** curve from the

current pose position to the target pose position, using a lead ("carrot") point placed forward along the desired final heading.

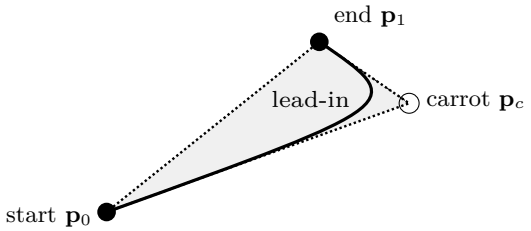

Figure 25: Boomerang moveToPose schematic: a quadratic Bézier curve constrained by the dotted triangle formed by start, end, and lead/carrot.

Lead point (carrot).. Let the target pose be $(\mathbf{p}_1, \theta_{\text{target}})$ and the start position be $\mathbf{p}_0$. Define the unit vector along target heading

$$\hat{u}(\theta_{\text{target}}) = (\cos\theta_{\text{target}},\ \sin\theta_{\text{target}}).$$

Let $d = \|\mathbf{p}_1 - \mathbf{p}_0\|$ and let `lead_in` be the user-controlled scalar. The control point is

$$\mathbf{p}_c = \mathbf{p}_1 + (\texttt{lead\_in})\ d\ \hat{u}(\theta_{\text{target}}).$$

Quadratic Bézier curve.. For $t \in [0, 1]$, the curve is

$$\mathbf{B}(t) = (1-t)^2\mathbf{p}_0 + 2(1-t)t\mathbf{p}_c + t^2\mathbf{p}_1.$$

Distance traveled (used by timing, markers, and sampling).. The curve is sampled into points $\{\mathbf{q}_k\}$ and its length is approximated by

$$L_{\text{boom}} \approx \sum_{k=0}^{N-1} \|\mathbf{q}_{k+1} - \mathbf{q}_k\|.$$

This is the quantity the system uses for (i) time estimation via $T = \int ds/v(s)$ (discretized), (ii) marker triggers expressed in distance, and (iii) consistent collision sampling density along the approach.

4.10. Pure pursuit: lookahead and corner cutting tradeoff

Pure pursuit selects a target point on the path at a lookahead distance L_d from the robot. Increasing L_d typically smooths control but increases corner cutting; decreasing L_d improves adherence but can become twitchy. The Terminal's live lookahead overlay makes this tradeoff visible during tuning, which is difficult to do with code-only workflows. A representative overlay is shown in Figure 12.

Given the current position $\mathbf{c}$, the lookahead point $\mathbf{p}_L$ is chosen as the first intersection between the path polyline and the circle $\|\mathbf{p}_L - \mathbf{c}\| = L_d$ ahead along the path; if none exists, the fallback is $\mathbf{p}_L = \mathbf{p}(s + L_d)$ in arc-length coordinates. The steering aim uses the tangent toward the lookahead point:

$$\theta_{\text{aim}} = \text{atan2}\big(-(y_L - y_c),\ x_L - x_c\big).$$

When auto-lookahead is enabled, the default in-code heuristic is $L_d = \text{clamp}(0.9w,\ 8,\ 24)$ inches, where w is effective track width. The simulator can either (i) step a pure-pursuit follower toward the lookahead point, or (ii) sample the path by time and rate-limit heading changes; the `simulate_pursuit` toggle selects between these modes.

Auto-lookahead scaling (implementation).. For curved segments, the base lookahead $L_{d,0}$ is scaled by curvature and average speed:

$$\bar{\kappa} = \frac{1}{N}\sum_i |\kappa_i|, \tag{51}$$

$$\kappa_{\max} = \max_i |\kappa_i|, \tag{52}$$

$$s_\kappa = \text{clamp}\big(0.2, 1, \tag{53}$$

$$1/(1 + 8\max(1.5\bar{\kappa},\ 0.75\kappa_{\max}))\big), \tag{54}$$

$$s_v = \text{clamp}\left(0.7, 1.25,\ 0.75 + 0.45\,\frac{\bar{v} - 60}{50}\right), \tag{55}$$

$$L_d = \text{clamp}(6, 60, L_{d,0}\ s_\kappa\ s_v). \tag{56}$$

Here $\bar{v}$ is the segment's average speed (ips) and L_d is clamped in inches.

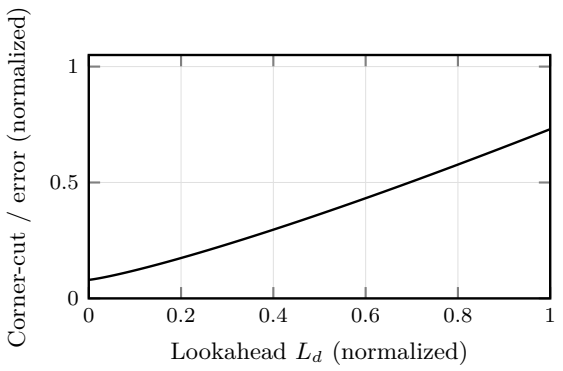

Figure 26: Illustrative lookahead L_d vs corner-cutting/tracking error trend. In the Terminal, L_d is a user-facing tuning surface: increasing L_d generally smooths steering but can cut corners; decreasing L_d tracks tighter but may oscillate.

The lookahead curve is intentionally monotone: larger L_d reduces steering effort but trades away corner fidelity.

4.11. Voltage caps, max speed, and estimated time scaling

VEX motor commands are abstracted as a cap fraction $c \in [0, 1]$ that limits effective effort and max

speed. At the simplest level, higher cap lowers estimated time; however, higher cap can increase slip and settling, so it may increase buffers and reduce repeatability. Because VEX motors are typically capped at 12 V, computed caps are clamped to $[0, 12]$ volts and then normalized as $c = V_{\max}/12$.

Profile selection (size-based).. Let m be move magnitude (inches for drive/path, degrees for turn/swing). The Terminal chooses a profile by rules:

$$p(m) = \begin{cases} \text{precise}, & m < m_p, \\ \text{slam}, & m \geq m_s \text{ (if set)}, \\ \text{fast}, & m > m_f, \\ \text{normal}, & \text{otherwise.} \end{cases}$$

Defaults are $m_p = 12$ in (drive) or 25° (turn/swing). The fast threshold is $m_f = 36$ in or 120°. Per-segment profile overrides bypass these rules.

Voltage-shape caps and heading scaling.. For a move of magnitude m, the exporter maps m into a voltage cap using a three-point shape (v_s, v_m, v_l) and breakpoints (x_1, x_2):

$$V_{\max}(m) = \begin{cases} v_s + (v_m - v_s)\frac{m}{x_1}, & 0 \leq m < x_1, \\ v_m + (v_l - v_m)\frac{m-x_1}{x_2-x_1}, & x_1 \leq m < x_2, \\ v_l, & m \geq x_2. \end{cases} \tag{57}$$

Drive uses $x_1 = 6$ in and $x_2 = 48$ in; turn/swing use $x_1 = 15°$ and $x_2 = 90°$. Heading caps are derived as a profile fraction of drive cap and intentionally kept below the 12 V hardware max: $V_{\text{head}} = \text{clamp}(f_p V_{\max}, 3\,\text{V}, 9\,\text{V})$ with $f_p \in \{0.50, 0.60, 0.65, 0.70\}$.

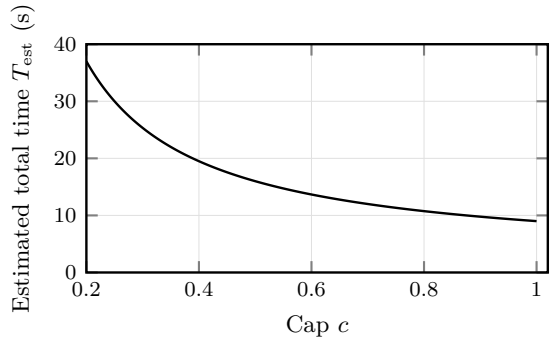

Figure 27: Illustrative total-time scaling vs cap (effort/voltage fraction). In the Terminal, cap changes the derived $v_{\max}$ used by motion profiles and the curvature-limited $v(s)$ used on splines; higher cap reduces raw motion time but can increase drift and settling, which the user accounts for via buffers and tuned minimum command floors.

Because cap scales both linear and angular limits, it implicitly changes $v_{\max}$, $\omega_{\max}$, and the curvature cap in Equation (50).

4.12. Buffers, settling, and timeouts

A major difference between a "path length" estimate and a VEX-relevant estimate is **settling**. After a high-speed move, the robot may still oscillate or drift; a buffer models this as explicit time.

Total routine time is treated as a sum of segment family totals:

$$\sum T_{\text{tot}} = T_{\text{drive}} + T_{\text{turn}} + T_{\text{path}} + T_{\text{wait}} + T_{\text{buffer}}. \tag{58}$$

Markers schedule actions along a segment but do not add time unless they also introduce an explicit wait/buffer.

Settle windows are profile-shaped from magnitude and cap. For a drive of length m inches (or a turn of $|\Delta\theta|$ degrees), the internal heuristic uses normalized terms

$$M = \text{clamp}(m/48, 0, 1),$$
$$V = \text{clamp}(V_{\text{cap}}/12, 0, 1),$$

then linearly interpolates between a profile's base ranges:

$$\text{err}_{\text{settle}} = \text{lerp}(\text{err}_{\min}, \text{err}_{\max}, 0.9M + 0.1V), \tag{59}$$
$$T_{\text{settle}} = \text{lerp}(t_{\min}, t_{\max}, 0.85M + 0.15V). \tag{60}$$

Here $\text{lerp}(a, b, t) = a + t(b - a)$.

Calibration and sensor weighting.. When calibration is enabled, the intent settle window is made more conservative using measured buckets:

$$\text{err}_{\text{settle}} \leftarrow \max\big(\text{err}_{\text{settle}},\ s_e\,\text{err}_{p90},\ k_{\text{noise}}\text{noise}\big), \tag{61}$$

$$T_{\text{settle}} \leftarrow \max\big(T_{\text{settle}},\ s_t\,T_{p90}\big), \tag{62}$$

where s_e and s_t are calibration scales and k_{noise} inflates a noise floor. If advanced motion is enabled, a further sensor scale is applied based on tracking wheels (0/1/2) and all-omni: $(\text{err}_{\text{settle}}, T_{\text{settle}}) \leftarrow (s^{(e)}_{\text{sens}}\text{err}_{\text{settle}},\ s^{(t)}_{\text{sens}}T_{\text{settle}})$.

Timeouts used in exports are derived from estimated time with padding:

$$t_{\text{timeout}} = \max\big(t_{\min},\ \alpha\,T_{\text{est}}\big), \tag{63}$$

where $\alpha > 1$ is a padding multiplier and $t_{\min}$ is a minimum timeout floor.

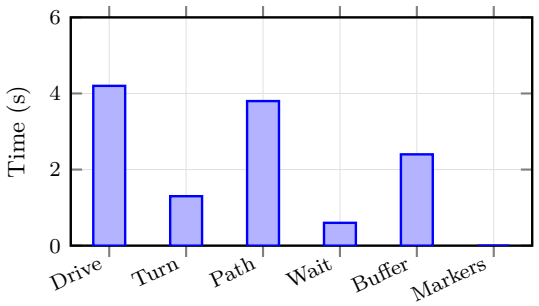

Figure 28: Illustrative breakdown of time contributors in a 15 s routine. In the Terminal, the total comes from the compiled segment list: straight moves, turns/faces, spline following, explicit waits, buffers (settle margin), and marker/action synchronization (markers themselves do not add time unless paired with a wait/buffer).

The bar breakdown is a quick diagnostic, where the tallest bar indicates where tuning or routine refactors yield the largest time savings.

4.13. Motion chaining: smooth transitions between segments

Motion chaining is the feature set that reduces unnecessary stops at intermediate nodes. Instead of enforcing a full settle at each node, the Terminal uses:

- minimum speed floors,
- early-exit ranges (accept looser error sooner),
- a user-controlled looseness parameter (precision vs speed).

Chaining is appropriate when the next action does not require precise alignment. It is risky when a mechanism interaction is tolerance-tight.

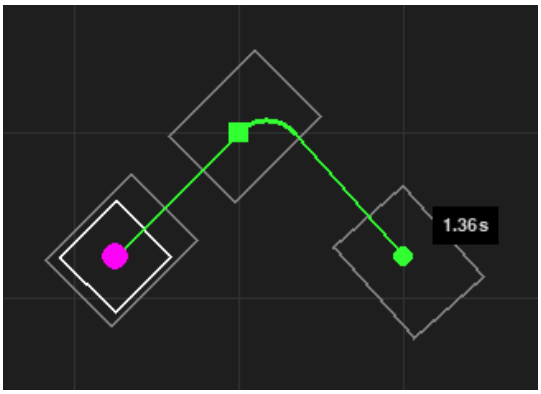

Figure 29: Motion chaining: a junction node connecting two moves with carry-through speed indicated.

5. Export, constraints, and configuration surfaces

5.1. Export panel and codegen surfaces

The Export / Codegen panel is the boundary between a compiled timeline and library-specific code. It exposes four user-editable surfaces that must be documented because they directly change generated output:

1. **Style selection:** LemLib (primary), JAR (advanced motion variables), etc. (Customizable).
2. **Path assets:** export folder and *path column format editor* (how per-point rows are written).
3. **Templates:** tokenized strings mapping each segment type into custom function calls.
4. **Mechanism presets:** named actions (e.g., INTAKE_ON) that templates reference.

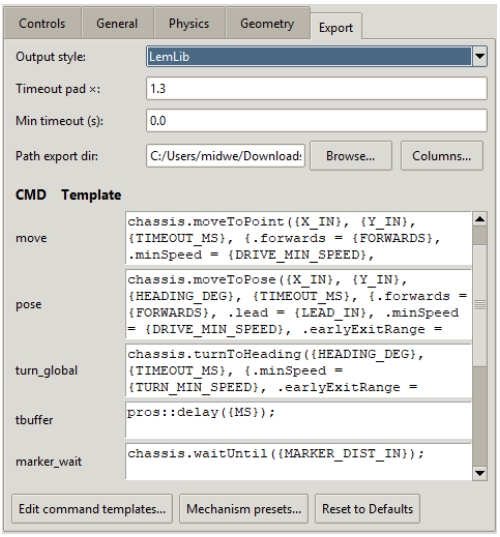

Figure 30: Export / Codegen panel showing style selection, path assets, templates, and mechanism presets.

5.1.1. Token system (syntax and optional fields)

Templates use **single-brace** tokens: `{TOKEN}`. Tokens are substituted with compiled values (units already converted by the chosen export mode).

Optional tokens are handled by a sentinel (conceptually `__OMIT__`): if a token is omitted for a segment (e.g., no lead-in on a non-pose move), the exporter removes the optional fragment rather than emitting nonsense. A template may also contain `||` to split into multiple output lines.

5.1.2. Markers and waits (correct Terminal semantics)

Atticus Terminal does **not** model marker actions as tasks. Marker actions are emitted as an *ordering fence* around an in-motion wait:

- `marker_wait`: "wait until distance/progress is reached"
- emit the mechanism action line(s)
- `marker_wait_done`: "ensure motion completion before boundary actions if required"

For LemLib this maps to `waitUntil(...)` and `waitUntilDone()`. For JAR in this revision, per option (A), marker waits are omitted (template entries blank), so intermediate mechanisms are only exported for compatible styles or by configuring asynchronous calls.

5.1.3. Path column format editor

Spline paths are exported as point files. The user-editable *path columns* string controls how each row is written using per-point tokens for position, heading, and command. This editor exists because different runtimes want different columns (and different separators). The same sampled curve can be exported to different formats without changing geometry.

Example row format.. If the column format is:

```
{X}, {Y}, {HEADING}, {COMMAND}
```

then each path point is written as a CSV-like row using the point's exported units and any per-point command tags.

The `{COMMAND}` token is a command-unit speed computed from the physics cap. For path files it is exported as a magnitude in $[0, u_{\max}]$, with direction handled by the segment's reverse flag or the caller's direction parameter:

$$u_{\text{cmd}} = \text{clamp}\left(\frac{v_{\text{ips}}}{v_{\max}} \cdot u_{\max},\ 0,\ u_{\max}\right),$$
$$u_{\max} = \texttt{max\_cmd} \quad \text{(default 127)}.$$

If a target library accepts signed commands, the sign follows segment direction, while the magnitude remains capped. Voltage-based targets use $V_{\max}$ directly (bounded by 12 V); cmd-based targets use $u_{\max} = 127$ with the same $V_{\max}/12$ scaling. The optional `{HEADING}` token uses the UI convention via $\theta_{\text{ui}} = (180^\circ - \theta) \bmod 360^\circ$.

5.1.4. Template + filled-output examples (Lemlib)

Below, the *template* is shown first, followed by the *filled output* for a concrete segment.

Example segment intent (from the compiled timeline).. Drive to $(x, y) = (48, 0)$, forward, timeout 2000 ms, min speed 40, early-exit 3.50 in, and trigger `INTAKE_ON` at 24 in during the move.

LemLib templates (tokenized)..

```
move:       chassis.moveToPoint({X_IN}, {Y_IN},
↪ {TIMEOUT_MS},
            {.forwards = {FORWARDS}, .minSpeed =
            ↪ {DRIVE_MIN_SPEED}, .earlyExitRange
            ↪  = {DRIVE_EARLY_EXIT}});

marker_wait:
↪  chassis.waitUntil({MARKER_DIST_IN});
marker_wait_done: chassis.waitUntilDone();

preset(INTAKE_ON): intake.move_voltage(12000);
```

Filled LemLib output (what the export preview shows)..

```
chassis.moveToPoint(48, 0, 2000, {.forwards = true,
↪  .minSpeed = 40, .earlyExitRange = 3.50});
chassis.waitUntil(24.0);
intake.move_voltage(12000);
chassis.waitUntilDone();
```

5.1.5. Template + filled-output examples (JAR advanced motion variables)

JAR-style exports optionally include **advanced motion variables**: drive/heading max voltage, settle error/time, and profile-shaped caps. This document shows them because the Terminal computes them from the motion profile rules and the voltage/settle shape tables.

JAR template (tokenized, motion variables explicit)..

```
move: chassis.drive_distance({DIST_IN},
↪  {HEADING_DEG},
      {DRIVE_MAX_V}, {HEADING_MAX_V},
      {DRIVE_SETTLE_ERR}, {DRIVE_SETTLE_TIME},
      {TIMEOUT_MS});
```

How the Terminal computes {DRIVE_MAX_V} (piecewise).. For a drive move of magnitude m inches, the exporter maps m into a voltage cap using a three-point shape (v_s, v_m, v_l) and two breakpoints (x_1, x_2):

$$V(m) = \begin{cases} v_s + (v_m - v_s)\frac{m}{x_1}, & 0 \le m < x_1, \\ v_m + (v_l - v_m)\frac{m - x_1}{x_2 - x_1}, & x_1 \le m < x_2, \\ v_l, & m \ge x_2. \end{cases} \tag{64}$$

The first segment ramps to v_m at x_1; the second ramps to v_l at x_2. In VEX, caps are clamped to 12 V before being normalized to a fraction for templates that use $V/12$. The chosen $(v_{\text{small}}, v_{\text{mid}}, v_{\text{large}})$ depends on the selected profile (`precise`, `normal`, `fast`, or `slam`). The profile itself is chosen from move size rules (short → precise, long → fast/slam). In code, the drive breakpoints are $x_1 = 6$ in and $x_2 = 48$ in; for turn/swing they are $x_1 = 15^\circ$ and $x_2 = 90^\circ$. Heading-voltage caps are derived as a fraction of drive cap (0.50–0.70 by profile) and then clamped to 3 V–9 V, staying under the 12 V hardware ceiling.

Filled JAR output (example).. For a long drive (48 in) that selects a fast profile, the export preview emits concrete values:

```
chassis.drive_distance(48.0, 0.0, 12.0, 7.80, 0.35,
↪  200, 2000);
```

Marker note (option A).. In this revision, JAR marker templates are blank by design, so intermediate mechanisms (mid-edge markers) are exported for LemLib and action-list styles, not JAR.

The template menu is the user-facing surface where:

- output units (inches/rotations/degrees/ticks) are chosen,
- timeout strategy is selected,
- path file naming and directory layout are set,
- per-segment caps and follower parameters are mapped into target library arguments.

5.2. Export architecture: templates, tokens, mechanism presets, and async

Export is where intent becomes code. The Terminal exports the same compiled segment timeline through *templates* that map segment types and actions into target-library calls (PROS/LemLib/JAR/custom/action-list).

Token-driven templates.. Templates replace tokens such as distance, pose targets, headings, caps, timeouts, and file paths. Token categories the user can customize include: geometry (L, x, y, θ, R), timing (timeout, buffer, settle thresholds), control (cap, min speed, curvature gain), state (reverse, reshape), and assets (path filenames / folder layout).

```
# Token replacement skeleton (conceptual)
def render(template: str, values: dict) -> str:
    for k, v in values.items():
        template = template.replace("{{"+k+"}}",
        ↪  str(v))
    return template
```

Mechanism presets.. Mechanism presets are named action blocks (e.g., `INTAKE_ON`, `CLAMP_CLOSE`, `MATCHLOAD_DEPLOY`) that compile to a segment-aligned action timeline. Presets improve reuse and portability, as the same preset name can emit different code depending on the template stack.

Async semantics.. In many VEX libraries, a motion call can be asynchronous: it begins motion and returns immediately. This enables overlap such as "start driving; enable intake at 50%; wait until done; then score." The exporter must therefore manage synchronization points explicitly (e.g., a wait-until-done call) when the segment boundary requires it.

```
// Example of what an exported "marker + async
↪  move" pattern can look like in
↪  PROS/LemLib-style code.
//
// Visual intent (Terminal):
//   Segment: Drive 48 in forward (cap 0.80,
↪  async=true)
//   Marker:  At 24 in (50% progress), start intake
//   Next:    Immediately begin next segment once
↪  the async move completes

int mainAuton() {
  // Start the drive segment asynchronously.
  chassis.moveToPoint({48, 0}, /*timeout_ms=*/2000,
  ↪  {
    .maxSpeed = 0.80,
    .async = true
  });

  // Marker action: trigger intake at 50% progress
  ↪  along the move.
  pros::Task markerTask([&] {
    while (chassis.getPathProgress() < 0.50)
    ↪  pros::delay(10);
    intake.move_voltage(12000);
  });

  // Optional: wait for the move to finish before
  ↪  chaining the next segment.
  chassis.waitUntilDone();

  // Continue with the next compiled segment...
  chassis.turnToHeading(90, /*timeout_ms=*/900,
  ↪  {.maxSpeed = 0.60});
  return 0;
}
```

The full template editor surface is included in the appendix for reference (see Figure A.43). The mechanism preset editor is shown in Figure 14.

```
// Example: visual routine intent vs exported calls
↪  (illustrative)
// Drive 48 in @ cap=0.80, trigger intake at
↪  halfway, then clamp.
chassis.moveDistance(48_in, {.maxVoltage=0.80,
↪  .async=true});
waitUntilDistance(24_in);  // marker sync (if
↪  needed by the target library)
setIntake(true);
chassis.waitUntilDone();
setClamp(true);
```

5.3. Collision and constraint checking

The Terminal can represent obstacles, restricted regions, and "no-go" polygons. Collision checks are primarily geometric: a robot footprint (or reshaped footprint) is tested against polygons.

If the robot footprint is modeled as an oriented rectangle centered at $\mathbf{p}$ with half-extents (a, b) in robot frame, its corner points in field frame are

$$\mathbf{c}_i = \mathbf{p} + \mathbf{R}(\theta)\,\mathbf{u}_i, \qquad \mathbf{u}_i \in \{(\pm a, \pm b)\}. \tag{65}$$

A typical overlap test uses the Separating Axis Theorem (SAT): two convex polygons do not intersect iff there exists an axis along which their projections are disjoint. The Terminal uses these checks to support:

- out-of-bounds warnings,
- restricted-zone highlighting,
- "will this reshaped footprint clip this object" validation.

Separating Axis Theorem (SAT).. The SAT schematic below summarizes the projection test used for convex footprints and field polygons.

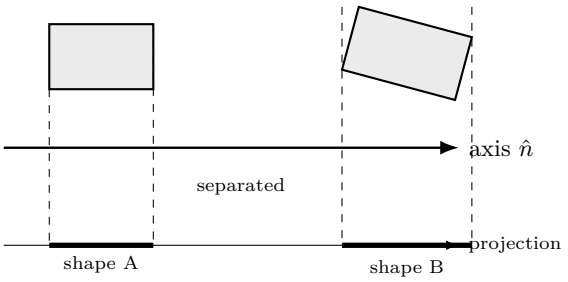

Figure 31: Separating Axis Theorem (SAT) schematic for collision/legality checks.

Clearance margins and padding.. The geometry panel includes margin inputs for normal and reshaped footprints (left/right/front/back) plus wall padding. These margins expand the effective footprint used for legality checks without changing the drawn chassis, letting VEX teams budget for intake overhang, sensor standoffs, or field tolerances:

$$w_{\text{eff}} = w + m_L + m_R, \qquad \ell_{\text{eff}} = \ell + m_F + m_B.$$

Reshape has its own margin set so matchload and intake-down states can be validated independently.

Forbidden regions and object overlays.. The editor can represent immovable structures and no-go zones as polygons on the field plane. At runtime, the simulator samples poses along each motion segment and tests intersection against these polygons.

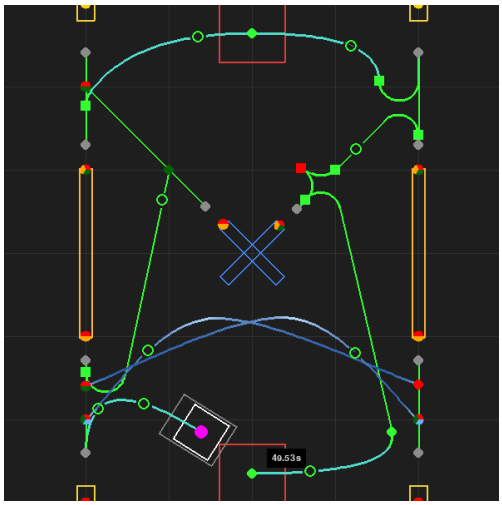

Figure 32: Field overlay showing forbidden regions and static objects used for legality checks.

Conflicts-only mode.. Dense routines can become visually unreadable if every sample is shown. Conflicts-only mode filters the display to show only poses (or segment portions) that violate constraints, so the user can iterate quickly.

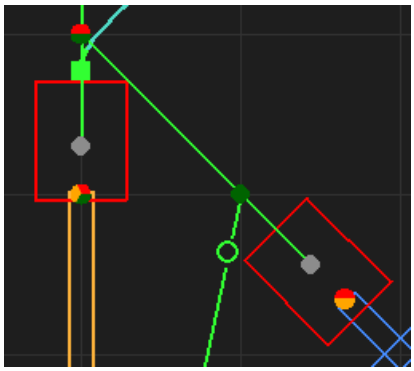

Figure 33: Conflicts-only visualization: only overlapping poses/segments are highlighted, with the overlap region emphasized.

5.4. Transformations, persistence, and packaging

Horizontal flip (red/blue mirroring).. A correct mirror must transform positions, headings, and spline control points. If the field mirror axis is $x = x_m$, then a point (x, y) maps to $(2x_m - x,\ y)$. Heading transforms depend on the convention mapping in Eq. (3). The mirrored overlay is essential for VEX because swing direction and clearance can invert; the visualizer lets teams verify CW/CCW intent and contact geometry before export.

$$(x', y') = (2x_m - x,\ y). \tag{66}$$

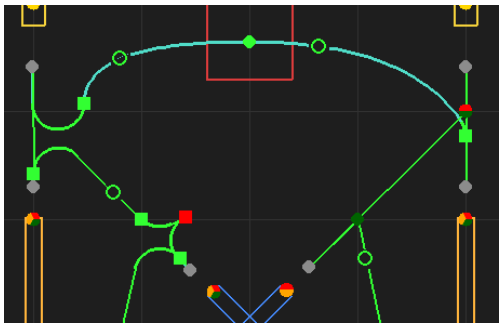

Figure 34: Routine before/after horizontal flip: nodes, headings, and spline control points mirrored consistently about the field centerline.

Initial heading helper.. To reduce start-state mismatch, the initial heading can be inferred from the first segment direction (e.g., node0 → node1 chord) and applied as a convenience toggle.

Persistence and migration.. Routines and global settings persist to disk. When new features are added (new per-node fields, new template tokens), a migration step preserves backwards compatibility so older routine files remain usable.

Atticus Terminal uses persistent configuration so that:

- physical constants survive restarts,
- templates/export targets remain consistent,
- UI toggles (hitboxes, overlays) do not reset,
- calibration/tuning values can be iterated over time.

In VEX practice, this reduces the "it worked yesterday" failure mode caused by hidden setting drift.

Packaging.. Packaging as a single executable enables non-programmers on a VEX team to author, review, and export autons consistently.

```
ATTICUS_TERMINAL/
  ATTICUS_TERMINAL.exe
  ATTICUS.png
  config.json
  main.py
  __init__.py
  README.md
  LICENSE
  requirements.txt
  setup.py
  test-91.json
  mod/
    config.json
    config.py
    codegen.py
    draw.py
    geom.py
    pathing.py
    sim.py
    storage.py
    ui.py
    util.py
  export/
    build_exe.bat
    build_exe.ps1
    paths/
      routine_seg0_path.txt
  atticus_terminal.spec
  dist/ (built exe output)
  build/ (PyInstaller work)
  .venv/ (optional)
  __pycache__/ (optional)
```

Figure 35: Packaged build layout (schematic).

Build and distribution.. Atticus Terminal is intended to be usable by a VEX team where not every member runs a Python development environment. Published as an open-source repository on GitHub, it therefore includes build metadata (e.g., a spec file and a build script) to package the editor into a distributable executable.

- **Why this matters:** consistent versions across laptops, fewer dependency failures, and easier iteration on the field.
- **Documentation tie-in:** exported templates, presets, and configuration files should be versioned alongside builds so that "same routine" implies "same emitted code and timing assumptions."

Atticus Terminal: Localization

Customized Probabilistic Localization System with Automatic Tuning

Abstract

Monte Carlo Localization (MCL) + Extended Kalman Filter (EKF). This section documents Atticus Terminal's PROS-side localization stack, that being a map-anchored particle filter with an EKF layer for smooth fused pose output, all of which is entirely customizable to the user's robot and library. It ties the explicit models (motion, sensor likelihoods, gating, and confidence) to the configuration surfaces and a microSD `.mcllog` automatic tuning loop.

Key Points

- Localization challenges in VEX robotics: short autonomous, limited sensors, limited compute, and frequent slip/collisions.
- The exported runtime dataflow (`ProsMCL`), including how to integrate, what to feed in, and which pose to use for decisions.
- MCL internals: motion and sensor models, particle filter mechanics, resampling, adaptive particle count, and recovery tools.
- EKF fusion: smoothing, confidence-weighted updates, and the "fused pose" contract used by autonomous/pathing.
- The tuning and validation loop: microSD sessions, CRC-verified `.mcllog` import/reporting, and regression checks.

6. Atticus Terminal: Customized Localization

This section describes the design and implementation of Atticus Terminal's customized localization system, that being map-anchored Monte Carlo Localization (MCL) combined with an Extended Kalman Filter (EKF) for smooth fused pose output. The intent is to be both rigorous and deployable on real VEX robots, where the math is tied directly to configuration fields, sensor setups, and on-field tuning workflows.

6.1. Localization challenges in VEX

Competitions impose a set of conditions that make reliable localization both critical and difficult.

Short time horizons and fast convergence.. Match autonomous is typically 15 s, while skills routes are often up to 60 s. Any localization system must converge quickly and correct drift in real time without lengthy initialization.

Small field, known geometry, limited sensors.. The field is 144 in × 144 in with a defined perimeter and season-dependent fixed objects. This creates an opportunity for map-based localization, but most VEX robots have a limited sensor setup:

- wheel odometry (motor encoders) for displacement,
- an IMU for yaw/heading,
- a small number of distance sensors for semi-absolute wall/object references.

These sensors are noisy and imperfect (IMU bias, slip/odometry error, distance sensor occlusion), so the estimator must gate implausible observations and avoid brittle "snap" behavior.

Compute constraints and robustness requirements.. The VEX V5 has limited compute compared to a laptop (∼200 MHz class CPU), so the localization update loop must be lightweight. At the same time, VEX robots are routinely bumped, dragged, or collide with field elements, and skills runs often include these bumped movements for alignment and efficiency. A practical estimator must both run fast and recover from unmodeled disturbances.

Why localization?. Standard odometry accumulates error; even a few inches of drift can be the difference between scoring and missing an interaction. Map-anchored updates let the robot periodically re-center its belief using stable geometry.

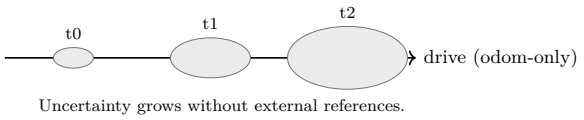

Figure 36: Particle dispersion under drift (placeholder). As the robot drives without external corrections, pose uncertainty expands.

6.2. What the Terminal exports (and what it is for)

Atticus Terminal can generate an optional PROS-side localization package intended for long autonomous routes (especially skills), where encoder+IMU odometry alone accumulates drift. The exported stack is designed around a single idea, to *feed incremental motion and sensor observations in, then read a stable fused pose out.*

Export surface (ProsMCL package).. The generated output is organized as:

- **`include/` and `src/`:** MCL core, runtime wrapper, configuration, and map data.
- **`docs/` and `examples/`:** integration templates and tuning workflow references.
- **Tuning helpers:** `.mcllog` binary logging and a microSD marker API for repeatable tuning sessions.

The system is *not* a substitute for stable drivetrain control: it assumes the robot can follow commands repeatably. Localization is therefore tuned only after PID and basic geometry calibration are correct (see Section 6.12).

6.3. Coordinate and unit conventions)

The localization stack uses the same pose words as the rest of the document:

$$\mathbf{x} \stackrel{\text{def}}{=} (x,\ y,\ \theta).$$

Field frame.. As in Section 2, the exported runtime uses $+x$ forward (from alliance side), $+y$ left, and heading θ in degrees with Atticus's UI convention: 0° = left, 90° = up-field, and **clockwise-positive**.

Odometry deltas (robot frame).. When using manual odometry feeding, the runtime expects incremental deltas in the robot's local frame: $(dx,\ dy,\ d\theta)$ with $dx > 0$ forward, $dy > 0$ left, and $d\theta > 0$ for a clockwise turn. For a differential drive with left/right travel dL, dR and track width w,

$$\begin{aligned} dx &= \tfrac{1}{2}(dL + dR), \\ dy &= 0, \\ d\theta_{\text{cw}} &= \frac{dL - dR}{w} \cdot \frac{180}{\pi}. \end{aligned} \tag{67}$$

Sensor units.. Distance sensors are modeled in millimeters internally (for gate thresholds and noise parameters), while poses are in inches and degrees. Consistency (signs, units, and port ordering) is a hard requirement when many "MCL is broken" failures are convention mismatches.

6.4. Mathematical and physics models

This section writes the system in "equations-first" form, transitioning between the kinematics that move pose, the probabilistic assumptions behind drift and sensor noise, and the exact update equations implemented by MCL and EKF. The intent is not to be abstract, as these are the same models you end up tuning on a the robot when something feels "off."

State, control, and belief.. Let the localization state be robot pose in the field frame: $\mathbf{x}_t \stackrel{\text{def}}{=} (x_t,\ y_t,\ \theta_t)$, with x, y in inches and heading θ in radians when used inside trigonometric functions. The control/odometry input for a tick is the incremental motion in the robot frame: $\mathbf{u}_t \stackrel{\text{def}}{=} (dx_t,\ dy_t,\ d\theta_t)$. The Bayesian belief is a probability density over pose: $\text{bel}(\mathbf{x}_t) \equiv p(\mathbf{x}_t \mid \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$, where $\mathbf{z}_{1:t}$ denotes all sensor observations up to time t.

Heading convention and trig-safe angle.. The exported stack may store headings in the Atticus UI convention (θ_{ui} in degrees, clockwise-positive, 0° = left), but any use of $\sin(\cdot), \cos(\cdot)$ must be performed in a consistent mathematical angle θ (radians, CCW-positive). The conversion used across the document is Eq. (4):

$$\theta = \text{wrap}\big(\pi - \theta_{\text{ui}} \tfrac{\pi}{180}\big)\,.$$

Here $\text{wrap}(\cdot)$ maps angles into a chosen interval (e.g., $(-\pi, \pi]$). This looks pedantic, but it is the difference between sensors correcting drift and fight odometry.

Rigid-body pose update (robot frame → field frame).. Given a local-frame delta (dx, dy) and current heading θ, the corresponding field-frame delta is:

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \mathbf{R}(\theta) \begin{bmatrix} dx \\ dy \end{bmatrix}, \qquad \mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{68}$$

and the discrete update is $x_t = x_{t-1} + \Delta x,\ y_t = y_{t-1} + \Delta y,\ \theta_t = \text{wrap}(\theta_{t-1} + d\theta)$.

If the runtime uses clockwise-positive $d\theta_{\text{cw}}$ (degrees), then $d\theta = -d\theta_{\text{cw}} \frac{\pi}{180}$. This is simply "rotate the robot-frame into the field frame, then add it," and it is exactly what every particle and the EKF do each tick.

Differential drive kinematics (physics link).. For a differential drive, encoder-measured left/right wheel travel dL, dR (in) and track width w (in) produce:

$$\Delta s = \tfrac{1}{2}(dL + dR), \qquad d\theta_{\text{ccw}} = \frac{dR - dL}{w}, \quad (69)$$

with $d\theta_{\text{cw}} = -d\theta_{\text{ccw}} \frac{180}{\pi}$ in the UI convention. For small increments, a curvature-consistent local delta can be written using the midpoint heading:

$$dx \approx \Delta s \cos\!\left(\tfrac{1}{2} d\theta_{\text{ccw}}\right), \qquad dy \approx \Delta s \sin\!\left(\tfrac{1}{2} d\theta_{\text{ccw}}\right). \quad (70)$$

This is the physics bridge from encoders and track width calibration into the odometry deltas consumed by the filter. In practice, track width w is the main physics knob. If it is wrong, heading drift becomes systematic and the estimator is forced to correct constantly.

Bayes filter.. The recursive belief update is:

$$\text{bel}(\mathbf{x}_t) = \eta\, p(\mathbf{z}_t \mid \mathbf{x}_t) \int p(\mathbf{x}_t \mid \mathbf{u}_t, \mathbf{x}_{t-1}) \cdot \text{bel}(\mathbf{x}_{t-1})\, d\mathbf{x}_{t-1}, \quad (71)$$

where η is a normalization constant. The two key models are the *motion model* $p(\mathbf{x}_t \mid \mathbf{u}_t, \mathbf{x}_{t-1})$ and the *sensor model* $p(\mathbf{z}_t \mid \mathbf{x}_t)$. The motion model is "how far could we really have moved" (slip, bias, compliance), and the sensor model is "how surprising is this reading if we were at this pose on the map."

Motion noise as a physical uncertainty model.. Wheel slip, carpet/foam compliance, and IMU bias appear as uncertainty in $\mathbf{u}_t$. One practical model is additive Gaussian noise in the local delta:

$$\tilde{dx} = dx + \epsilon_x, \quad \tilde{dy} = dy + \epsilon_y, \quad \tilde{d\theta} = d\theta + \epsilon_\theta, \quad (72)$$

with $\epsilon_x \sim \mathcal{N}(0, \sigma_x^2)$, $\epsilon_y \sim \mathcal{N}(0, \sigma_y^2)$, $\epsilon_\theta \sim \mathcal{N}(0, \sigma_\theta^2)$. Alternatively, the standard α-model makes variance scale with motion magnitude:

$$\sigma^2_{\Delta s} = \alpha_1 \Delta s^2 + \alpha_2 d\theta^2, \qquad \sigma^2_{d\theta} = \alpha_3 d\theta^2 + \alpha_4 \Delta s^2. \quad (73)$$

Atticus exposes these as configuration parameters because they encode robot-specific drivetrain physics. Bigger moves and sharper turns tend to slip more, so the filter must "spread out" more on those ticks.

6.5. Runtime architecture: `ProsMCL`

The exported runtime provides a single high-level API (see `mcl_runtime.h`) and runs updates in a background task. Cadences are configured in the Terminal (typical targets are 20 ms motion ticks and 50 ms sensor ticks).

Two supported odometry feed modes..

1. **Manual deltas:** call `setOdomDelta(...)` each control cycle (≈10 ms).
2. **Pose provider:** register a callback via `setFieldPoseProvider(...)` (useful for select libraries), and the runtime computes deltas.

Which pose to use.. `getPose()` returns the raw MCL estimate. `getFusedPose()` returns the EKF-smoothed pose (recommended for control). If EKF is disabled, the fused pose falls back to the raw estimate.

Minimal integration sketch (manual deltas)..

```
ProsMCL localizer(IMU_PORT, {DIST1, DIST2, DIST3});

void initialize() {
  localizer.startEasy((int)pros::millis(),
  ↪  start_heading_deg, start_x_in, start_y_in,
  ↪  start_theta_deg);
}

void autonomous() {
  while (pros::competition::is_autonomous()) {
    localizer.setOdomDelta(dx_in, dy_in,
    ↪  dtheta_deg_cw);   // every ~10 ms
    localizer.updateVision(vx, vy, vtheta, vconf);
    ↪  // optional trusted fixes
    const MCLPose fused = localizer.getFusedPose();
    ↪  // use for decisions
    pros::delay(10);
  }
}
```

6.6. Monte Carlo Localization (MCL) in Atticus Terminal

MCL represents belief over pose as a set of weighted particles $\{(\mathbf{x}_i, w_i)\}_{i=1}^N$. Think of each particle as a "plausible robot" on the field. Each sample has a pose hypothesis and a weight that says how believable it currently is. Each sensor update asks the same question for every particle: *if the robot were here, would these readings make sense on this map?*

Principles of MCL (predict → update → resample).. Instead of maintaining a single pose guess, MCL maintains a cloud of many hypotheses. **Predict** moves every particle by odometry (plus noise) so drift is represented. **Update** scores particles using sensors against the map so geometry can pull the estimate back toward reality. **Resample** throws away low-weight hypotheses and duplicates high-weight ones so limited compute is spent on the poses that still fit. This loop is why MCL can converge quickly on a small VEX field even when the initial pose is slightly wrong.

Adapting MCL to VEX constraints.. Classic MCL can use thousands of particles; Atticus targets hundreds. A typical default is $N \approx 300$ with bounds ($N_{\min}$, $N_{\max}$) and optional KLD adaptive sampling so the particle count expands when belief becomes complex and contracts when belief is tight. Adaptive N exists to keep the V5 system real-time, where we pay extra particles only when the robot is genuinely ambiguous, and save CPU when belief is tight.

Efficient map scoring.. Atticus represents the field as perimeter segments plus optional object segments. Distance sensors can be scored with either a *beam model* (raycast + mixture weights) or a faster *likelihood field* (precomputed nearest-obstacle distance grid). The likelihood field approach is a good default for VEX, as it replaces many expensive raycasts with a cheap lookup so sensor update time does not steal cycles from drive control.

Particle contents.. Each particle stores x (in), y (in), θ (deg), and a nonnegative weight w. After weight normalization $\sum_i w_i = 1$, the estimate uses a weighted mean for x, y and a circular mean for heading:

$$\hat{x} = \sum_i w_i x_i, \qquad \hat{y} = \sum_i w_i y_i,$$
$$\hat{\theta} = \operatorname{atan2}\left(\sum_i w_i \sin\theta_i,\ \sum_i w_i \cos\theta_i \right). \tag{74}$$

Motion update (prediction).. On each motion tick, particles are advanced by the odometry delta and injected with noise. Noise can be configured as:

- **Additive sigmas:** σ_x, σ_y (in) and σ_θ (deg) applied every tick, or
- **Alpha model:** motion-dependent noise scaled by translation/rotation magnitude ($\alpha_1 \dots \alpha_4$).

This is how the filter admits that encoders/IMU are not perfect. The cloud spreads just enough that the correct pose remains represented even after slip. This step is also the only way the filter moves. If odometry signs are wrong, no amount of sensor tuning will save it.

Distance sensor likelihood (scoring).. Each enabled distance sensor has a pose on the robot (x_s, y_s, ϕ_s) and produces a range reading. For a particle $\mathbf{x}_i$, the expected range $\hat{z}$ is computed from field map geometry (Eq. (76)):

- **Perimeter mode:** score rays only against the outer boundary.
- **Objects mode:** score only against configured obstacles (goals, braces, matchloaders. NOT reccomended).
- **Both:** perimeter + selected obstacles (default for most setups).

Two likelihood models are supported: **beam** (mixture model with hit/random/short/max terms) and **likelihood field** (endpoint distance-to-nearest-obstacle scoring). Gates reject or downweight implausible innovations to prevent "snapping" to the wrong wall when a sensor is occluded. The key intuition is that sensors are not steering the robot, they are steering the *belief* by rewarding poses that would have seen the same wall.

Sensor ray model (geometry, explicit).. This is the geometric "wiring harness" between a real sensor mount and the field map: it turns a particle pose into a ray that can be intersected with walls/objects. For a particle pose $\mathbf{x}_i = (x_i, y_i, \theta_i)$ and sensor offset (x_s, y_s) with sensor-facing angle ϕ_s (robot frame), the sensor origin in field coordinates is

$$\mathbf{p}_s(\mathbf{x}_i) = \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \mathbf{R}(\theta_i) \begin{bmatrix} x_s \\ y_s \end{bmatrix}, \qquad \theta_s = \theta_i + \phi_s, \tag{75}$$

and the idealized beam ray is $\mathbf{r}(\lambda) = \mathbf{p}_s + \lambda[\cos\theta_s,\ \sin\theta_s]^\mathsf{T}$, $\lambda \geq 0$. Given a map $\mathcal{M}$ represented as line segments, the expected range (in inches) is

$$z^\star(\mathbf{x}_i) = \min_{\lambda \geq 0} \left\{ \lambda \ :\ \mathbf{r}(\lambda) \in \partial\mathcal{M} \right\}, \tag{76}$$

clamped by the sensor max range. The measured sensor value is z in millimeters; conversion is $z_{\text{in}} = z/25.4$. In practice, sensor mounting thickness and beam width are absorbed into the map hitbox inflation and sensor noise.

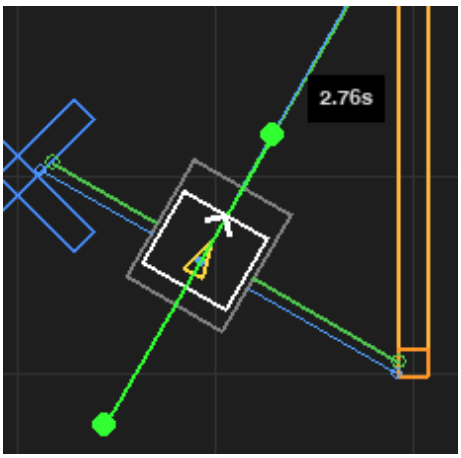

Figure 37: MCL visual simulation overlay showing the estimated pose and distance-sensor raycasts against the configured map geometry. The ray intersections define the expected range $z^\star$ in Eq. (76).

Conditional independence (multi-sensor update).. With multiple sensors $z^{(1)}, \ldots, z^{(m)}$, the common assumption is conditional independence given pose:

$$p(\mathbf{z} \mid \mathbf{x}_i) = \prod_{j=1}^{m} p\Big(z^{(j)} \mid \mathbf{x}_i\Big). \tag{77}$$

The runtime typically accumulates log-likelihoods for numerical stability, then exponentiates and renormalizes. Each sensor is casting a vote, where multiplying likelihoods means agreement strengthens belief, and disagreement weakens it.

Particle weight update.. For particle i with prior weight w_i, the multiplicative update for one sensor is

$$w_i \leftarrow w_i \; p(z \mid \mathbf{x}_i), \tag{78}$$

and with multiple conditionally independent sensors the product in Eq. (77) is applied (often in log-space). After scoring, weights are normalized: $w_i \leftarrow w_i / \sum_j w_j$. This is the Bayes-filter idea in code form, where particles that explain the measurement get more probability mass.

Beam model mixture weights.. In the beam model, the likelihood is commonly expressed as a mixture: "hit" (Gaussian around expected range), "random" (uniform failures), and optional "short"/"max" components. Atticus exposes mixture weights such as w_{hit} and w_{rand} (often the dominant pair in VEX), plus optional short/max weights and decay terms. In practice, teams usually start with a strong hit weight and a small random weight, then adjust gates before making the model more complex.

Beam model likelihood (math, explicit).. Let $z^\star$ be the raycast expected range and $z_{\max}$ the sensor max range (in millimeters). A standard mixture model is:

$$\begin{aligned} p(z \mid \mathbf{x}) = {} & w_{\text{hit}} \, \mathcal{N}\Big(z;\, z^\star, \sigma_{\text{hit}}^2\Big) \\ & + w_{\text{short}} \; \lambda_{\text{short}} e^{-\lambda_{\text{short}} z} \; \mathbb{I}[0 \le z \le z^\star] \\ & + w_{\max} \; \mathbb{I}[z = z_{\max}] \\ & + w_{\text{rand}} \; \frac{1}{z_{\max}} \; \mathbb{I}[0 \le z < z_{\max}], \end{aligned} \tag{79}$$

with weights summing to 1 (100%). In VEX practice, $(w_{\text{hit}}, w_{\text{rand}})$ usually dominates; the short/max terms can be disabled unless the sensor often returns clipped or near-zero values. The meaning behind the mixture is that sensors usually behave, but sometimes they fail, and you want those rare failures to reduce confidence, not catastrophically delete the correct hypothesis.

Likelihood field scoring.. The likelihood-field approximation replaces repeated raycasting with a precomputed distance-to-obstacle field $D(\mathbf{p})$ over the map. If a particle predicts an endpoint $\mathbf{p}_{\text{end}} = \mathbf{p}_s + z\,[\cos\theta_s, \sin\theta_s]^{\mathsf{T}}$, then the likelihood can be scored using the nearest-obstacle distance:

$$p(z \mid \mathbf{x}) \propto \exp\left(-\frac{D(\mathbf{p}_{\text{end}})^2}{2\sigma_D^2}\right). \tag{80}$$

Equivalently, some configurations score the residual $r = z - z^\star$ with a Gaussian $\exp\big(-r^2/(2\sigma_{\text{hit}}^2)\big)$. Both serve the purpose of rewarding poses whose predicted geometry explain the measurement.

Innovation gates and robust rejection.. Distance residuals are gated by thresholds such as `gate_mm` (hard/soft rejection) and `innovation_gate_mm` (optional extra gate). Hard-gate mode rejects implausible updates outright; soft-gate mode applies a penalty factor so a single suspicious reading cannot dominate the belief. The runtime can also apply a short median window to suppress single-frame spikes.

Gating as a statistical test.. Let the scalar innovation be $r = z - z^\star$. Under a Gaussian hit model with variance σ_{hit}^2, the normalized innovation squared is

$$\nu = \frac{r^2}{\sigma_{\text{hit}}^2}. \tag{81}$$

A hard gate is $\nu \le \gamma^2$ (equivalently $|r| \le \gamma\sigma_{\text{hit}}$); a soft gate multiplies the weight by a tapering penalty once ν exceeds a threshold. The mm-valued gates in config are an engineering surface for this statistical rejection. In a VEX match, this is what stops one bad wall reading (angled surface, opponent robot, sensor dropout) from yanking the pose estimate across the field.

Sensor setups and object visibility masks.. Teams typically mount 2–3 distance sensors (front/left/right). Atticus allows per-sensor map modes and per-sensor object visibility masks so each sensor scores only against geometry it can realistically observe, reducing symmetric-map confusion (a common problem in MCL) and occlusion-driven false matches.

IMU and vision updates (optional anchors).. IMU heading can be incorporated as a heading-likelihood term with a configured σ_θ. An optional vision/landmark pose fix may be injected via `updateVision(x,y,theta,confidence)`; low-confidence fixes should be rejected in the adapter so they do not corrupt the belief.

Resampling, N_{eff}, and adaptive particle count.. After weights are normalized, the runtime computes effective sample size $N_{\text{eff}} \approx 1/\sum_i w_i^2$ and resamples when N_{eff} falls below a threshold fraction of N. Systematic/stratified/multinomial resampling are supported. For efficiency, Atticus can also use KLD (Kullback-Leibler Distance) adaptive sampling to choose N based on posterior complexity, clamped to $[N_{\min}, N_{\max}]$. Resampling is re-centering the compute budget. When only a few particles still explain the sensors, we duplicate those and stop wasting time updating obviously-wrong hypotheses.

Confidence from weights.. Many equivalent concentration metrics exist (entropy, weight variance, N_{eff}). A simple normalized measure is:

$$c \stackrel{\text{def}}{=} \operatorname{clamp}\left(1 - \frac{N_{\text{eff}}}{N}, 0, 1\right). \tag{82}$$

Here $c \approx 0$ means weights are diffuse (filter uncertain), while $c \approx 1$ means a small subset dominates (high confidence). This is why confidence is useful as a safety signal, as it is measuring how many particles still matter.

KLD-sampling bound (why adaptive N is principled).. If the posterior is approximated by bins with k occupied bins, KLD-sampling chooses N large enough that the KL divergence between the true distribution and its particle approximation is below ϵ with probability $1-\delta$. A common bound is:

$$\begin{aligned} \eta(k,\delta) &\stackrel{\text{def}}{=} 1 - \frac{2}{9(k-1)} \\ &\quad + z_{1-\delta}\sqrt{\frac{2}{9(k-1)}}, \\ N &\geq \frac{k-1}{2\epsilon}\,\eta(k,\delta)^3, \end{aligned} \tag{83}$$

where $z_{1-\delta}$ is the $(1-\delta)$-quantile of the standard normal distribution. This is the mathematical basis for "more particles when belief is complex, fewer when belief is tight." You do *not* want to spend "worst-case particles" all the time. Instead, you want extra particles only during moments that are ambiguous (symmetry, long open-field drives, kidnapping), because those are the moments where additional hypotheses change the outcome.

Kidnapping recovery.. To recover from large unmodeled displacements, the stack supports: (i) random particle injection, (ii) augmented MCL (fast/slow average likelihood tracking), and (iii) explicit re-localization requests. The exported runtime exposes this as `requestRelocalize()` to broaden the belief.

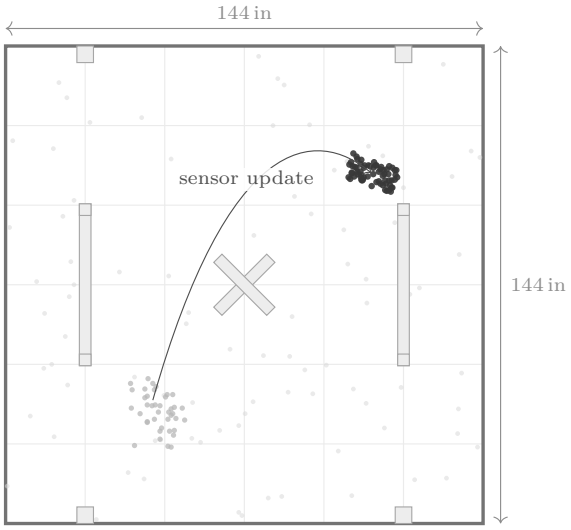

Figure 38: MCL particle filter in action. Particles inconsistent with sensor readings are downweighted; resampling concentrates belief near map-consistent geometry.

Update loop..

```
# Motion tick (fast): propagate particles and EKF.
particles = apply_odom_delta_with_noise(particles,
↪  dx, dy, dtheta)
ekf.predict(dx, dy, dtheta)

# Sensor tick (slower): score sensors -> normalize
↪  -> resample if needed.
for sensor in enabled_sensors:
    particles = score_particles(particles, sensor,
    ↪  field_map)
normalize(particles)
if Neff(particles) < thresh * N:
    particles = resample(particles, method)

# Estimate + confidence, then smooth with EKF when
↪  confidence is adequate.
pose_mcl = weighted_mean_pose(particles)
confidence =
↪  confidence_from_weight_concentration(particles)
ekf.update_from_mcl(pose_mcl, confidence)
pose_fused = ekf.pose()
```

6.7. Extended Kalman Filter (EKF) and pose fusion

While MCL is powerful for global robustness, its instantaneous estimate can be noisy or jumpy (especially when resampling occurs). Atticus therefore layers an EKF on top of MCL to provide a control-friendly *fused pose* along with an uncertainty representation.

EKF intent.. The EKF predicts pose forward using odometry deltas, then accepts MCL as a measurement only when MCL confidence exceeds a threshold. Measurement noise is scaled as a function of confidence, where higher confidence results in smaller assumed measurement sigmas, allowing the

EKF to pull the fused pose closer to the particle estimate. The key VEX-specific reason for this layer is cost-to-benefit. A 3×3 EKF update is cheap on the V5, but it turns occasional chunky particle corrections into a smooth signal that path followers and PID loops can tolerate.

EKF state and prediction model.. The EKF maintains a Gaussian approximation $\mathbf{x} \sim \mathcal{N}(\hat{\mathbf{x}}, P)$ over pose, where $\hat{\mathbf{x}} = [x \ y \ \theta]^\mathsf{T}$ and $P \in \mathbb{R}^{3\times3}$ is the covariance. Using the local delta input $\mathbf{u} = [dx \ dy \ d\theta]^\mathsf{T}$ (with $d\theta$ in radians, CCW-positive), the nonlinear prediction model matches the rigid update in Eq. (68):

$$f(\hat{\mathbf{x}}, \mathbf{u}) = \begin{bmatrix} x \\ y \end{bmatrix} + \mathbf{R}(\theta) \begin{bmatrix} dx \\ dy \end{bmatrix}, \qquad \theta' = \mathrm{wrap}(\theta + d\theta). \tag{84}$$

Linearizing about the current estimate yields the Jacobian $F = \partial f / \partial \mathbf{x}$:

$$F = \begin{bmatrix} 1 & 0 & f_{x\theta} \\ 0 & 1 & f_{y\theta} \\ 0 & 0 & 1 \end{bmatrix}. \tag{85}$$

$$\begin{aligned} f_{x\theta} &= -\sin\theta\, dx - \cos\theta\, dy, \\ f_{y\theta} &= \ \ \cos\theta\, dx - \sin\theta\, dy. \end{aligned} \tag{86}$$

With a local-delta process-noise covariance $Q = \mathrm{diag}(\sigma_x^2, \sigma_y^2, \sigma_\theta^2)$, the EKF prediction is:

$$\hat{\mathbf{x}}^- = f(\hat{\mathbf{x}}, \mathbf{u}), \qquad P^- = FPF^\mathsf{T} + GQG^\mathsf{T}, \tag{87}$$

where G maps local input noise into state space (often $G \approx \mathrm{diag}(1,1,1)$ when σ is already expressed in state units, or $G = \partial f / \partial \mathbf{u}$ if modeling noise in the control space explicitly).

Measurement update (pose, IMU, and vision).. Each measurement type defines a function $h(\mathbf{x})$ and Jacobian $H = \partial h / \partial \mathbf{x}$. For a full pose measurement $\mathbf{z} = [x \ y \ \theta]^\mathsf{T}$ (e.g., MCL pose or vision pose), $h(\mathbf{x}) = \mathbf{x}$ and $H = I$. For an IMU yaw measurement z_θ, $h(\mathbf{x}) = \theta$ and $H = [0\ 0\ 1]$. The EKF update uses:

$$\begin{aligned} \mathbf{y} &= \mathbf{z} - h(\hat{\mathbf{x}}^-), \\ y_\theta &= \mathrm{wrap}(z_\theta - \hat{\theta}^-), \\ S &= HP^-H^\mathsf{T} + R, \\ K &= P^-H^\mathsf{T}S^{-1}, \\ \hat{\mathbf{x}}^+ &= \hat{\mathbf{x}}^- + K\mathbf{y}, \\ P^+ &= (I - KH)P^-(I - KH)^\mathsf{T} \\ &\quad + KRK^\mathsf{T} \ \ \text{(Joseph form)}. \end{aligned} \tag{88}$$

The measurement covariance R is the tunable trust knob. In Atticus, R for MCL-driven pose updates is commonly scaled by confidence so high-confidence particle convergence yields smaller assumed measurement noise.

Confidence → measurement covariance.. Let $c \in [0,1]$ be a confidence metric derived from particle weight concentration. A simple continuous mapping is:

$$\begin{aligned} \sigma_{\mathrm{mcl}}(c) &= \sigma_{\max}(1 - c) + \sigma_{\min} c, \\ R_{\mathrm{mcl}}(c) &= \mathrm{diag}(\sigma_x^2(c), \sigma_y^2(c), \sigma_\theta^2(c)). \end{aligned} \tag{89}$$

This makes the EKF mathematically conservative when MCL is uncertain and aggressive only when the particle cloud has genuinely converged.

EKF gating.. As with distance-sensor gating, EKF updates can be rejected using the Mahalanobis distance:

$$d^2 = \mathbf{y}^\mathsf{T} S^{-1} \mathbf{y}. \tag{90}$$

Rejecting measurements with d^2 above a threshold prevents rare corrupt pose fixes (bad distance returns, false vision locks) from destabilizing the fused pose.

Covariance and Kalman gain.. The EKF maintains a covariance matrix P that represents uncertainty in x, y, θ. When P is large (the filter is uncertain), trusted measurements pull the fused pose strongly. When P is small (the filter is confident), measurements are blended more conservatively unless their assumed noise is also small.

Confidence gating and startup behavior.. At startup, the estimator should not blindly trust a weak particle set. The exported runtime can gate EKF until MCL confidence passes a minimum threshold (or the user explicitly sets pose), preventing early wrong locks when the robot has not yet observed enough map geometry.

IMU and vision fusion.. The EKF can also incorporate IMU yaw as a heading measurement (useful for stabilizing θ under slip), and it can accept occasional absolute pose fixes (vision/landmarks) via `updateVision(...)`. If an external odometry framework already uses IMU for heading, disable duplicate IMU updates in the localization config to avoid double-counting.

Output contract.. Use `getFusedPose()` for autonomous/pathing decisions (not raw odometry pose). If the EKF is disabled, `getFusedPose()` returns the raw MCL estimate so the API contract remains stable.

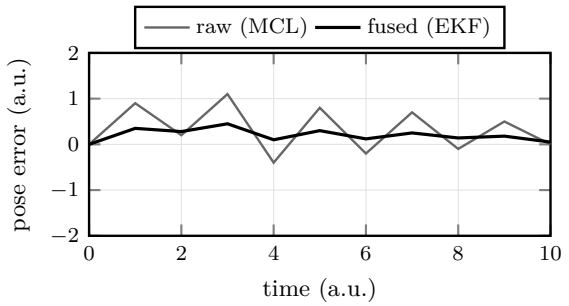

Figure 39: EKF smoothing of pose updates (placeholder). The fused estimate filters noisy particle-based corrections into a stable signal.

6.8. Using MCL to correct other odometry (Library bridge)

Many teams already run an odometry framework (e.g., LemLib) and want MCL to "intervene" only when drift is observed. The runtime provides `applyOdomCorrection(...)` to blend an odom pose toward the fused estimate with a safe alpha selected from configured bounds ($\alpha_{\min}$ to $\alpha_{\max}$). This avoids hard snaps that can destabilize pathing.

Blend equation.. Let $\hat{\mathbf{x}}_{\text{odom}}$ be the external odometry pose and $\hat{\mathbf{x}}_{\text{fused}}$ be the EKF fused pose. The corrected pose written back into the odometry system is:

$$\begin{aligned} x_{\text{odom}} &\leftarrow x_{\text{odom}} + \alpha\,(x_{\text{fused}} - x_{\text{odom}})\,, \\ y_{\text{odom}} &\leftarrow y_{\text{odom}} + \alpha\,(y_{\text{fused}} - y_{\text{odom}})\,, \\ \theta_{\text{odom}} &\leftarrow \text{wrap}\big(\theta_{\text{odom}} + \alpha \cdot \text{wrap}(\theta_{\text{fused}} - \theta_{\text{odom}})\big)\,, \end{aligned} \tag{91}$$

where $\alpha \in [0, 1]$ is a small correction gain.

Alpha selection.. If $c \in [0, 1]$ is confidence (Eq. (82)), a simple monotone mapping is:

$$\alpha(c) = \text{clamp}\Big(\alpha_{\min} + (\alpha_{\max} - \alpha_{\min})c, \\ \alpha_{\min},\ \alpha_{\max}\Big), \tag{92}$$

optionally forced to $\alpha = 0$ if c is below a minimum-confidence gate.

```
MCLPose odom_pose =
↪  lemlib_to_mcl(chassis.getPose(false, false));
double alpha = 0.0;
if (localizer.applyOdomCorrection(odom_pose,
↪  &alpha)) {
  chassis.setPose(odom_pose.x, odom_pose.y,
  ↪  odom_pose.theta);
}
```

6.9. Internal models and parameters

Atticus exposes the localization internals as explicit configuration so teams can tune for their robot, sensors, and driving style. Each parameter corresponds to an assumption about reality (drift, sensor noise, gating strictness, and correction aggressiveness).

Motion model.. The motion model controls how fast the particle cloud expands under dead-reckoning:

- If noise is too low, the filter can stick to a wrong pose after a collision.
- If noise is too high, the estimate can become jittery and slow to converge.

Teams should tune translational/rotational sigmas (or alpha-model coefficients) using a mixed straight+turn route before enabling strong corrections.

Distance sensor model.. Distance sensors are the primary semi-absolute anchor in most VEX setups. The most important parameters are:

- σ_{hit}: expected measurement noise (mm),
- gate thresholds (hard/soft innovation gates),
- mixture weights in the beam model (hit vs random return probability).

If false snaps occur, tighten gates. If perimeter/walls are ignored, loosen them and verify sensor geometry.

Resampling and convergence.. Resampling method and the N_{eff} threshold determine when the filter focuses onto a few hypotheses. Particle count behavior (fixed vs. KLD adaptive, Eq. (83)) trades CPU headroom against robustness when belief becomes multi-modal.

Confidence, correction, and EKF gates.. Atticus maintains a confidence-like measure from particle weight concentration and uses it to: (i) decide when EKF should trust MCL updates, and (ii) choose a safe correction alpha for blending odometry toward the fused pose. This is the main mechanism for preventing unstable over-correction when belief is weak.

6.10. Map geometry, segment constraints, and pose estimation in context

The localization map reuses the same field geometry assumptions used for overlays and collision checks, that being perimeter walls plus selected fixed objects. This makes tuning interpretable: when the robot snaps, it is snapping to something visible in the map configuration.

Segment bands (optional corridor constraints).. Atticus can optionally apply a *segment band*: a corridor around the expected route segment that discourages off-route hypotheses. This is useful when the map is symmetric and the robot has not yet observed a distinctive landmark, but it must be used sparingly. Overly tight bands can block recovery after real collisions. In the exported runtime, bands are exposed via `setSegmentBand(...)` / `clearSegmentBand()`, with a corridor radius parameter in inches.

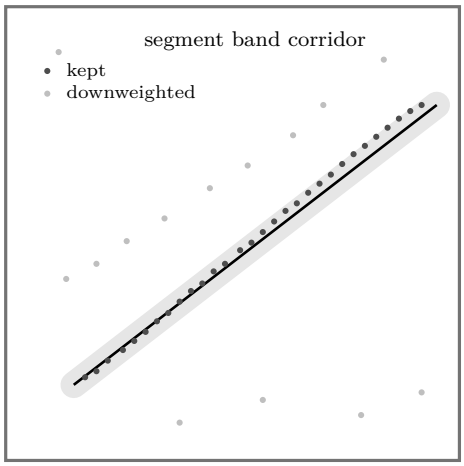

Figure 40: Segment band constraint. A temporary corridor biases particles toward the planned route while still allowing escape.

6.11. Configuring and using localization in Atticus Terminal

From a user perspective, deploying the localization stack is: configure → export → integrate → tune. The MCL tab provides the configuration surface for sensor geometry, map objects, particle/filter settings, EKF behavior, and tuning mode.

Integration checklist..

1. Implement one long-lived `ProsMCL` object with IMU port and distance sensor ports in the configured order.
2. Call `startEasy(...)` once from `initialize()` so IMU calibration does not consume autonomous time.
3. Feed odometry either manually via `setOdomDelta` or via a pose provider using `setFieldPoseProvider(...)` — not both.
4. Use `getFusedPose()` for decisions and pathing.
5. If bridging into a library, run `applyOdomCorrection` in a parallel task to gently nudge library pose toward fused pose.

Config fingerprinting and log linkage.. Tuning logs are stamped with a config fingerprint so the Terminal can reject sessions generated by a different export. This prevents wrong config tuning mistakes when iterating quickly.

Minimal config shape..

```
{
  "mcl": {
    "enabled": 1,
    "motion_ms": 20,
    "sensor_ms": 50,
    "particles": { "n": 300, "n_min": 200, "n_max":
    ↪  400 },
    "motion": { "sigma_x_in": 0.12, "sigma_y_in":
    ↪  0.12, "sigma_theta_deg": 1.0 },
    "sensors": { "distance": { "model":
    ↪  "likelihood_field" }, "imu": { "enabled": 1
    ↪  } },
    "correction": { "alpha_min": 0.03, "alpha_max":
    ↪  0.12, "min_confidence": 0.6 },
    "tuning": { "enabled": 0, "log_rate_hz": 20 }
  }
}
```

6.12. Tuning and validation workflow.

The Terminal includes a practical checklist in the MCL tab. The required order is:

1. **PID first:** tune drive/turn/path control with localization corrections effectively off.
2. **Geometry and sign checks:** IMU calibration behavior, distance port ordering, $dx/dy/d\theta$ sign conventions, and track width.
3. **Motion noise:** adjust $\sigma_x, \sigma_y, \sigma_\theta$ (or alpha model) until the filter neither jitters nor sticks after bumps.
4. **Sensor gating:** tighten/loosen innovation gates to prevent false snaps while still accepting walls/objects.
5. **Particle count and resampling:** increase N only if recovery is too slow, decrease if CPU load is high.
6. **Correction alpha last:** raise/lower $\alpha_{\min}, \alpha_{\max}$ to trade off recovery speed vs oscillation risk.
7. **Vision integration (optional):** accept only high-confidence fixes and keep update frequency conservative.

PID-first acceptance targets.. Localization cannot compensate for unstable base motion. Before tuning MCL/EKF corrections, a reasonable target is:

- Drive 72 in straight, 5 reps: end error ≤ 2 in.
- Turn 90 deg, 5 reps: end error ≤ 2 deg.
- No sustained oscillation after settling.

If these do not pass, stop and tune drive/turn/path control first.

Track width calibration (diff drive).. Run a 360 deg in-place turn and compare measured vs expected heading. If under-rotating, decrease track width by 2–5%; if over-rotating, increase by 2–5%. Repeat until the 360 deg error is within ≈3 deg.

6.13. No-USB tuning loop: microSD `.mcllog` sessions

To make tuning easy and repeatable for teams without a debugging laptop on-field, the exported package can log compact telemetry frames to microSD (`.mcllog`). A simple "wizard" marker API inserts step boundaries and kidnapping events so the Terminal can analyze the session.

CRC-verified sessions and config matching.. Each log includes a config fingerprint (`config_hash32`) so the Terminal can reject sessions generated by a different export, and the frame payload can be protected by a CRC32 footer so corrupted logs are detected before analysis. Imported sessions are summarized into a report by replaying the time-ordered frame stream (coverage, residual statistics, confidence percentiles, and kidnapped recovery timing) and can be saved for review.

Robot-side flow (wizard session)..

1. Enable tuning mode in the Terminal (`mcl.tuning.enabled=1`) and export/build.
2. Run a scripted session that includes stillness, straight motion, turning, and a kidnapped pick+place.
3. Mark phases with `markStep()`, and kidnapping with `markKidnappedStart()` / `markKidnappedPlaced()`.
4. End with `end()` and confirm the printed saved log path.

Terminal-side flow (import → recommend → re-export)..

1. Import the latest `.mcllog` file in the MCL tab.
2. Review PASS/FAIL checks (duration, log CRC, step markers, sensor coverage, residual quality, and kidnapped recovery).
3. Apply recommended parameter updates, then re-export and rerun the same session.

This creates a closed loop where tuning changes are justified by repeatable telemetry rather than subjective driving feel.

Regression checks.. Before collecting tuning runs, run the provided regression check script (`mod/mcl_regression_checks.py`) and ensure it passes. These checks validate codegen/runtime invariants (including log CRC gating behavior and snapshot/event-flag semantics) so tuning sessions are trustworthy.

Practical acceptance targets.. Repeat the tune loop until these criteria are met on repeated runs:

- Kidnapped recovery ≤ 8 s.
- Distance dropout fraction < 0.25 (sensors provide enough usable updates).
- No repeated "wrong-wall" snaps in the same route.
- Autonomous endpoints within team tolerance on 3 consecutive runs.

6.14. Automatic tuning model: `.mcllog` replay and recommendations

The microSD tuning wizard produces a binary `.mcllog` session that already contains both the measured distance readings and the robot-side expected distances under the current map and pose. The Terminal imports this time-ordered frame stream, verifies integrity (config fingerprint + optional CRC), and summarizes the data into residual and stability statistics which map to recommended configuration updates.

Frame stream model.. Let the log contain K frames with m distance sensors. Each frame k can be modeled as:

$$f_k = \left(t_k, \hat{\mathbf{x}}_k^{\text{odom}}, \hat{\mathbf{x}}_k^{\text{mcl}}, \hat{\mathbf{x}}_k^{\text{fused}}, P_k, c_k, \mathbf{z}_k, \mathbf{z}_k^\star, \mathbf{u}_k, \boldsymbol{\phi}_k\right), \tag{93}$$

where $\hat{\mathbf{x}} = [x\ y\ \theta]^\mathsf{T}$ poses are in inches/degrees, P_k is the EKF covariance, $c_k \in [0,1]$ is logged MCL confidence, $\mathbf{z}_k \in \mathbb{R}^m$ are measured ranges (mm), $\mathbf{z}_k^\star \in \mathbb{R}^m$ are expected ranges (mm), $\mathbf{u}_k \in \{0,1\}^m$ is the robot-side "used by filter" mask, and $\boldsymbol{\phi}_k$ are wizard event flags (step marks, recovery, corrections applied).

Residual stream, dropout, and outliers.. For sensor $s \in \{1, \ldots, m\}$, the residual is

$$r_{k,s} \stackrel{\text{def}}{=} z_{k,s} - z_{k,s}^\star. \tag{94}$$

Let $v_{k,s} = \mathbb{I}[z_{k,s} \geq 0 \wedge z_{k,s}^\star \geq 0]$ denote validity (dropouts/unknown expectations are encoded as negative values). Define the residual sets

$$\begin{aligned} \mathcal{R}_{\text{used}} &\stackrel{\text{def}}{=} \{r_{k,s} \mid v_{k,s} = 1 \wedge u_{k,s} = 1\}, \\ \mathcal{R}_{\text{all}} &\stackrel{\text{def}}{=} \{r_{k,s} \mid v_{k,s} = 1\}, \end{aligned} \tag{95}$$

and choose $\mathcal{R} = \mathcal{R}_{\text{used}}$ if non-empty, else $\mathcal{R} = \mathcal{R}_{\text{all}}$. Coverage is summarized by the dropout fraction:

$$f_{\text{drop}} \stackrel{\text{def}}{=} 1 - \frac{\sum_{k=1}^{K} \sum_{s=1}^{m} v_{k,s}}{Km}. \tag{96}$$

Residual spread is summarized by

$$\mu_r \overset{\text{def}}{=} \frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}} r, \qquad \sigma_r \overset{\text{def}}{=} \sqrt{\frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}}(r-\mu_r)^2}, \tag{97}$$

and outliers are counted relative to a conservative threshold

$$\begin{aligned} r_{\text{thr}} &\overset{\text{def}}{=} \max\big(40,\ 3\max(1,\sigma_r)\big)\ \text{mm}, \\ f_{\text{out}} &\overset{\text{def}}{=} \frac{1}{|\mathcal{R}|}\sum_{r\in\mathcal{R}} \mathbb{I}[|r| > r_{\text{thr}}]. \end{aligned} \tag{98}$$

Mapping statistics → recommended distance-model parameters.. The report treats the residual spread as the effective hit-model noise:

$$\sigma_{\text{hit}} \leftarrow \text{clamp}\big(\max(4,\sigma_r),\ 4,\ 35\big)\ \text{mm}, \tag{99}$$

then inflates the "random return" mixture weight when coverage/outliers are poor:

$$\begin{aligned} w_{\text{rand}} &= \begin{cases} 0.12, & f_{\text{drop}} > 0.20 \text{ or } f_{\text{out}} > 0.25, \\ 0.08, & f_{\text{drop}} > 0.10 \text{ or } f_{\text{out}} > 0.15, \\ 0.05, & \text{otherwise}, \end{cases} \\ w_{\text{hit}} &\leftarrow \text{clamp}(1 - w_{\text{rand}},\ 0.70,\ 0.97), \end{aligned} \tag{100}$$

and derives gates as multiples of σ_{hit}:

$$\begin{aligned} g_{\text{mm}} &\leftarrow \text{clamp}\big(\max(80,\ 4.5\sigma_{\text{hit}}),\ 80,\ 260\big)\,, \\ g_{\text{innov}} &\leftarrow \begin{cases} \text{clamp}(3\sigma_{\text{hit}},\ 0,\ 200), & \substack{c_{\text{med}}\geq 0.70 \\ f_{\text{drop}}<0.20, \\ f_{\text{out}}<0.15} \\ 0, & \text{otherwise}, \end{cases} \end{aligned} \tag{101}$$

where c_{med} is the median logged confidence. This keeps the innovation gate disabled unless the run is stable, reducing the risk of over-gating when the robot is still converging or the session is sensor-poor.

IMU noise from stillness.. The wizard includes a still segment. If $\sigma_{\theta,\text{still}}$ is the circular standard deviation of still headings (deg), a conservative IMU heading sigma is:

$$\sigma_{\text{imu}} \leftarrow \text{clamp}\big(\max(0.6,\ 3\sigma_{\theta,\text{still}}),\ 0.6,\ 3.0\big)\ \text{deg}. \tag{102}$$

EKF defaults from odom–fused disagreement.. To choose safe default process-noise scales in Eq. (87), the report summarizes odom vs. fused disagreement $e_{k,xy}^{\text{odom}\to\text{fused}}$ and $e_{k,\theta}^{\text{odom}\to\text{fused}}$ over the log and maps percentiles to sigmas:

$$\begin{aligned} \sigma_{dx} &\leftarrow \text{clamp}\big(0.08 + 0.03\, q_{75}(e_{xy}),\ 0.08,\ 0.60\big), \\ \sigma_{dy} &\leftarrow \text{clamp}\big(0.08 + 0.03\, q_{75}(e_{xy}),\ 0.08,\ 0.60\big), \\ \sigma_{d\theta} &\leftarrow \text{clamp}\big(0.70 + 0.06\, q_{75}(e_{\theta}),\ 0.60,\ 4.00\big), \end{aligned} \tag{103}$$

where $q_{75}(\cdot)$ denotes the 75th percentile across frames, and e_{xy}, e_θ use units of inches and degrees respectively.

6.15. Conclusion and practical considerations

Atticus Terminal's MCL+EKF system is designed to make VEX localization both powerful and deployable, where MCL provides global robustness using field geometry, while the EKF provides a smooth fused pose for control loops. In practice, the system performs best when teams follow a few discipline rules:

- **Initialize deliberately:** always start from a correct pose and allow IMU calibration to complete before movement.
- **Validate sensors before tuning:** confirm port order, mounting angles, and raw readings. Wrong geometry creates impossible residuals.
- **Keep compute headroom:** add particles and sensor rate only when needed, as starving the drive loop will hurt more than it helps.
- **Design for exposure:** route plans that periodically see walls/objects will maintain higher confidence than long open-field drives.
- **Treat confidence as a safety signal:** gate risky actions on confidence when appropriate, and request re-localize after known displacement events.

Acknowledgements

This program could not have been possible without multiple important sources of support. First, I want to thank my team, 15800A Atticus, of which this invention is named after, for their knowledge on the hardware behind robotics and patience in the creation of this program over a full calendar year. For spline creation, I would like to thank Freya Holmér's "The Continuity of Splines" documentary. For Monte Carlo Localization, I would like to thank Cyrill Stachniss for his public videos, Dr. Tabor of Worcester Polytechnic Institute for conceptual guidance in particle filtering, and D. Fox et al. for the "Probabalistic Robotics" textbook. My mathematical and physical understanding required for this project would not have been possible without several professors and a series of advanced courses taken through the University of Nebraska system as well as the Halliday-Krane-Resnick textbook used in preparation for the National Physics Olympiad.

Appendix A. Reference Screens

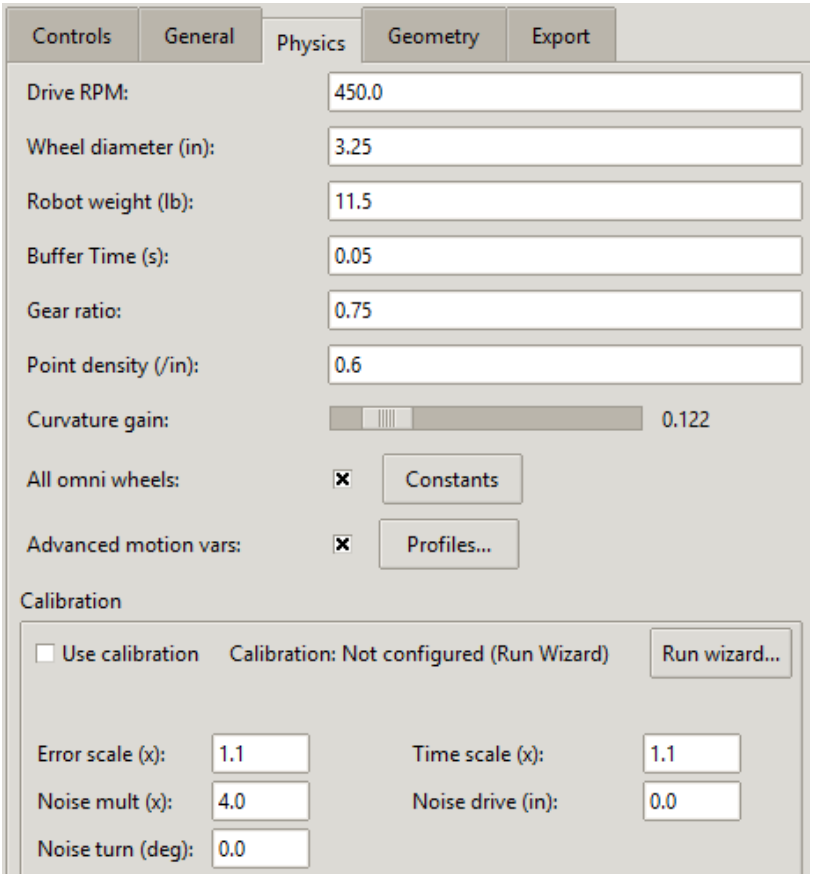

Figure A.41: Full Physics tab showing drivetrain constants and calibration controls.

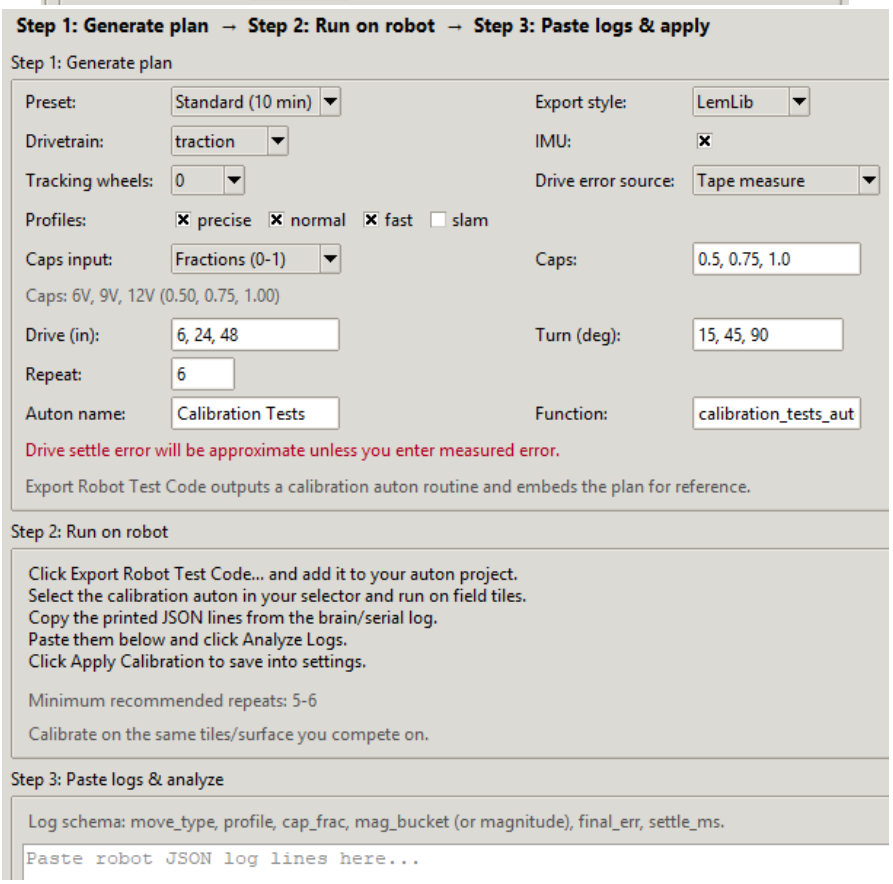

Figure A.42: Calibration wizard for plan generation, on-robot tests, and log import.

Figure A.43: Template editor surface with token glossary, render rules, and code preview.

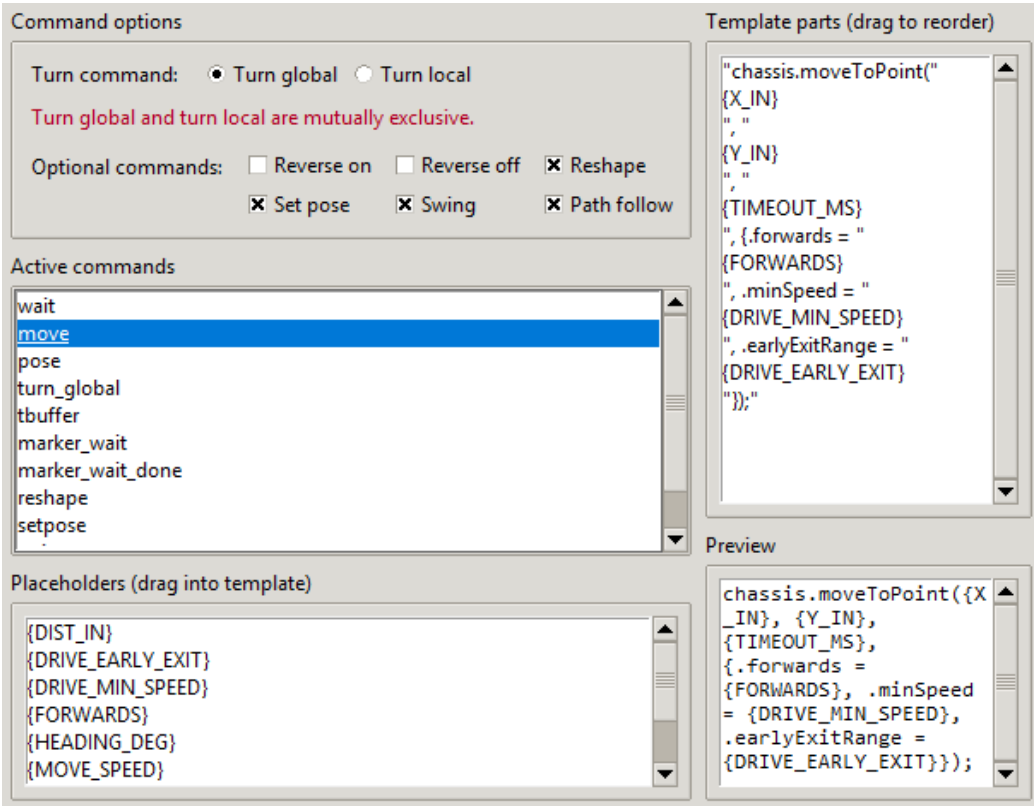

Figure A.44: A partial snippet of the MCL tab showing distance sensor geometry configuration and map object selection.