Snapshot Raycasting Pose Correction

Analytical Documentation for VEX Snapshot Localization

Austin McGrath, 15800A - Atticus[a]

[a] *Internal project documentation for VEX Robotics autonomous localization and odometry correction.*

December, 2025

Abstract

This document defines the concepts, geometry model, solver method, and integration workflow implemented by **Snapshot Pose**, a deterministic one-shot localization corrector for PROS projects. The module uses V5 distance sensors, per-sensor collision masks, and robust sampling/gating to infer a field translation (x, y) at known heading. The design target is practical autonomous reliability: low tuning overhead, portable runtime callbacks for multiple odometry libraries, and conservative acceptance checks that avoid unsafe pose snaps.

1. Introduction

1.1. What Snapshot Pose is

Snapshot Pose is a one-shot localization correction module for VEX autonomous routines. It takes a short burst of distance-sensor readings, raycasts those readings against a tagged field map, solves for a consistent translation (x, y) at fixed heading θ, and applies pose through a runtime callback.

1.2. Purpose: stop-and-snap correction, not continuous tracking

The module is designed for moments where the robot can pause briefly and view stable landmarks (usually perimeter walls). It is not a continuous probabilistic estimator: no particle filter, no recursive Bayesian state update, and no online heading solve. Heading is assumed to come from IMU/odometry and translation is corrected in one deterministic solve.

1.3. VEX constraints that shape the design

The V5 Distance Sensor operates in roughly 20 mm to 2000 mm range, with behavior that is less reliable on weak or ambiguous targets. That drives three implementation choices:

- emphasize large, stable geometry (`MAP_PERIMETER`) as the default map mask;
- sample multiple readings and use median filtering with gates;
- score candidate poses conservatively before applying any correction.

1.4. Pipeline overview

At fixed heading, each sensor contributes a ray-distance constraint. The solver then:

1. samples and gates sensor measurements;
2. raycasts with per-sensor mask filtering;
3. enumerates and scores candidate segment combinations;
4. applies pose only when validation gates pass.

2. Modeling Inputs and Coordinate Conventions

2.1. Field frame, robot frame, and pose state

A robot pose is represented as

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

where (x, y) is robot-center position in field coordinates and θ is heading.

Snapshot Pose uses a right/forward robot frame: $+x_R$ is robot-right and $+y_R$ is robot-forward. Heading convention is 0° along field $+Y$, clockwise positive. Define field-frame unit vectors:

$$\hat{\mathbf{f}}(\theta) \stackrel{\text{def}}{=} (\sin\theta,\ \cos\theta), \tag{2}$$

$$\hat{\mathbf{r}}(\theta) \stackrel{\text{def}}{=} (\cos\theta,\ -\sin\theta). \tag{3}$$

Any robot-frame offset $\mathbf{v}^R = (v_{\text{right}}, v_{\text{fwd}})$ maps into field frame as

$$\mathbf{v}^F = v_{\text{right}}\,\hat{\mathbf{r}}(\theta) + v_{\text{fwd}}\,\hat{\mathbf{f}}(\theta). \tag{4}$$

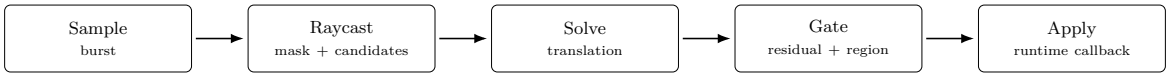

Figure 1: Snapshot pose pipeline from sensor burst to validated pose application.

2.2. Distance sensor mount and ray model

For sensor i, let:

- mount offset in robot frame be $\mathbf{o}_i^R = (x_{i,\text{right}}, y_{i,\text{fwd}})$ in inches;
- relative sensor angle be ϕ_i, where 0° is robot-forward and $+90^\circ$ is robot-right.

Sensor direction in robot frame is:

$$\hat{\mathbf{d}}_i^R(\phi_i) \stackrel{\text{def}}{=} (\sin\phi_i,\ \cos\phi_i). \tag{5}$$

At pose (x, y, θ), sensor origin and direction in field frame are

$$\mathbf{o}_i^F = (x, y) + x_{i,\text{right}}\,\hat{\mathbf{r}}(\theta) + y_{i,\text{fwd}}\,\hat{\mathbf{f}}(\theta), \tag{6}$$

$$\hat{\mathbf{d}}_i^F = (\sin\phi_i)\,\hat{\mathbf{r}}(\theta) + (\cos\phi_i)\,\hat{\mathbf{f}}(\theta). \tag{7}$$

A distance reading z_i is interpreted along ray

$$\mathbf{r}_i(t) = \mathbf{o}_i^F + t\,\hat{\mathbf{d}}_i^F, \qquad t \geq 0, \tag{8}$$

with t in inches.

2.3. Tagged collision-map model

Field geometry is represented as tagged line segments:

$$\mathcal{S} = \{(\mathbf{a}_j, \mathbf{b}_j, m_j)\}_{j=1}^N, \tag{9}$$

where m_j is a bitmask category.

Mask categories include:

- `MAP_PERIMETER`: perimeter walls (default reliable landmarks);
- `MAP_LONG_GOALS`: long-goal body segments;
- `MAP_LONG_GOAL_BRACES`: long-goal brace segments;
- `MAP_LONG_GOALS_ALL`: union alias for long-goal body and braces;
- `MAP_CENTER_GOAL_POS45`, `MAP_CENTER_GOAL_NEG45`, `MAP_CENTER_GOALS`: center-goal categories;
- `MAP_MATCHLOADERS`: matchloader segments;
- `MAP_PARK_ZONES`: park-zone boundaries;
- `MAP_ALL`: union mask.

Per-sensor overrides allow heterogeneous sensing: one sensor can raycast only perimeter walls while another includes `MAP_LONG_GOALS_ALL`.

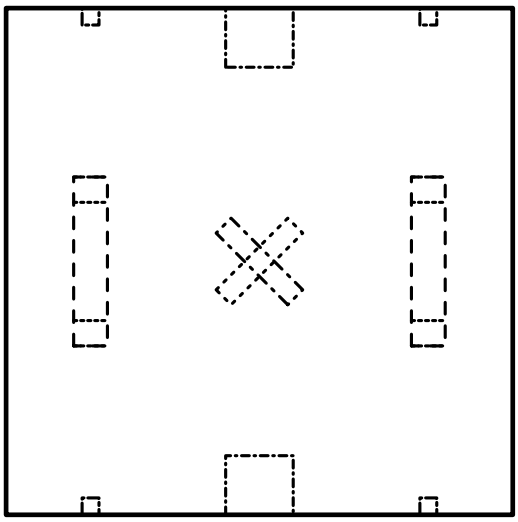

Figure 2: Tagged collision-map categories used for per-sensor mask filtering.

3. Method: Deterministic Raycast Solve and Validation

3.1. Ray-segment intersection test

Each map segment j is parameterized as

$$\mathbf{s}_j(u) = \mathbf{a}_j + u(\mathbf{b}_j - \mathbf{a}_j), \qquad u \in [0, 1]. \tag{10}$$

Ray intersection solves

$$\mathbf{o}_i^F + t\,\hat{\mathbf{d}}_i^F = \mathbf{s}_j(u), \tag{11}$$

with validity conditions

$$t \geq 0, \quad 0 \leq u \leq 1, \quad t \leq t_{\max}. \tag{12}$$

Implementation uses a 2D cross-product solve and rejects parallel/invalid cases.

3.2. Mask filtering and candidate hit generation

For sensor i, effective mask is

$$m_i = \begin{cases} m_i^{\text{override}}, & m_i^{\text{override}} \neq 0, \\ m^{\text{global}}, & \text{otherwise.} \end{cases} \tag{13}$$

Segment j is eligible only if $(m_i \,\&\, m_j) \neq 0$. Raycast hits are sorted by distance t, and up to K nearest hits are retained as hypotheses per sensor. This handles ambiguity when a ray could plausibly hit either `MAP_LONG_GOAL_BRACES` or `MAP_PERIMETER` from the current guess.

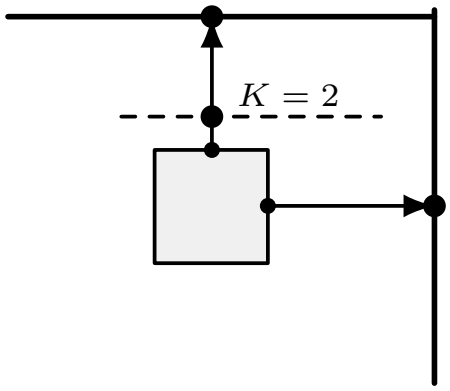

Figure 3: Local raycast candidate generation with one sensor retaining two plausible forward hits.

3.3. Measurement model and robust sampling

Each sensor is sampled n times and filtered by gates (range, confidence, and optional velocity/object-size). The snapshot measurement uses median:

$$z_i = \text{median}\{z_{i,1}, \ldots, z_{i,n}\}. \tag{14}$$

This suppresses transient spikes without introducing filter lag or extra state.

3.4. Fixed-heading translation solve

Heading is treated as known; unknowns are (x, y). For candidate segment selection, residual for sensor i is

$$r_i(x, y) = z_i - \hat{z}_i(x, y), \tag{15}$$

where $\hat{z}_i$ is predicted raycast distance.

Noise scale follows a practical sensor model:

$$\sigma(z) = \begin{cases} 15 \text{ mm}, & z < 200 \text{ mm}, \\ 0.05z, & z \geq 200 \text{ mm}. \end{cases} \tag{16}$$

The solver score is

$$\chi^2(x, y) = \sum_{i \in \mathcal{I}} \left(\frac{r_i(x, y)}{\sigma(z_i)} \right)^2, \tag{17}$$

where $\mathcal{I}$ is the set of gated-in sensors.

Locus-segment construction.. Instead of gradient descent, the implementation converts each candidate hit into a locus segment of feasible robot centers. Pairwise closest points between loci produce midpoint hypotheses; component-wise medians produce a deterministic pose estimate.

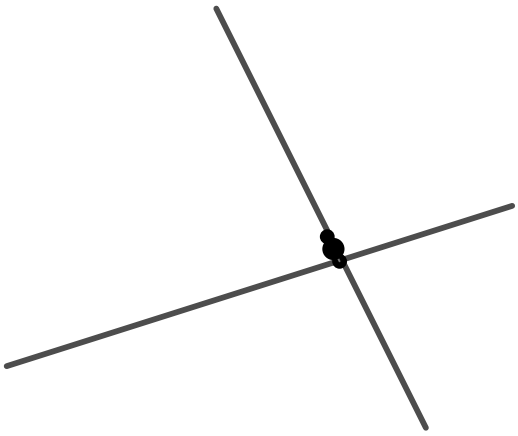

Figure 4: Pairwise locus closest-points and midpoint aggregation used by the deterministic translation solve.

3.5. Quadrant prior and acceptance gates

Quadrant options are `BL`, `BR`, `TL`, `TR`, and `ANY`. In current bindings, quadrant acts as a pre-solve guess bias with margin $m =$ `quadrant_margin_in`:

$$\texttt{TR}: \; x \leftarrow \max(x, 72 + m), \; y \leftarrow \max(y, 72 + m), \tag{18}$$

with analogous rules for other quadrants.

Final acceptance requires:

- at least two contributing sensors;
- per-sensor residual consistency under effective masks;
- total χ^2 competitiveness among tested combinations.

If gates fail, result is `ok=false` and no pose update is applied.

4. Implementation and Integration Notes

4.1. Library-agnostic runtime interface

Core raycasting and solve logic (Section 3) is library-agnostic. Bindings expose a runtime callback surface (`snapshot_bindings.hpp`) for:

- reading heading and guess pose;
- optionally reading tracker distances;
- applying the solved pose.

This lets one solver instance integrate with LemLib, JAR, EZ-Template, or custom odometry.

4.2. One-call autonomous entry point

Autonomous usage remains non-intrusive:

```
snapshot_setpose_quadrant(q);
```

On success, bindings call:

```
runtime.apply_pose(user_data, x_in, y_in,
↪  heading_deg,
                   fwdTrackerIn, sideTrackerIn);
```

If solve/validation fails, no pose change is applied.

4.3. Configuration surface in `snapshot_bindings.cpp`

Default user edits are centralized in one file:

- sensor ports or extern device hooks;
- sensor geometry $(x_{\text{right}}, y_{\text{fwd}}, \phi)$;
- global mask and per-sensor mask overrides;
- runtime callback wiring.

Runtime override helpers are also available:

```
snapshot_bindings_set_sensors(...)
snapshot_bindings_update_sensor(...)
snapshot_bindings_set_sensor_mask(...)
```

4.4. Runtime wiring pattern

```
SnapshotPoseRuntime runtime{};
runtime.get_heading_deg = cb_heading;
runtime.get_guess_x_in = cb_guess_x;
runtime.get_guess_y_in = cb_guess_y;
runtime.get_forward_tracker_in = cb_tracker_fwd;
↪  // optional
runtime.get_sideways_tracker_in = cb_tracker_side;
↪  // optional
runtime.apply_pose = cb_apply_pose;
snapshot_bindings_set_runtime(runtime);
```

Only callback bodies change across odometry libraries.

4.5. Operating constraints

Robot must be stationary during sample burst.. Sampling assumes one static pose. If the robot moves, readings represent different states and constraints become inconsistent.

Heading convention must be consistent end-to-end.. Heading source and apply-pose callback must use the same convention ($0^\circ = +Y$, clockwise positive).

Masks should start conservative.. Begin with `MAP_PERIMETER` only, then add interior masks only after sensor geometry and heading wiring are verified.

4.6. Recommended defaults

Table 1 lists safe defaults that are stable across most VEX drivetrains.

5. Operational Workflow and Deployment

5.1. Setup workflow

1. **Add snapshot module files.**
   Include `collision_map`, `raycast`, `snapshot_pose`, and `snapshot_bindings`.
2. **Wire runtime callbacks in initialize.**
   Set heading, guess-pose, and apply-pose callbacks; set tracker callbacks only if needed.
3. **Measure each sensor mount pose.**
   Enter x_{right}, y_{fwd}, and relative angle ϕ in inches/degrees.
4. **Set per-sensor collidable objects.**
   Use `field_mask_override` per sensor, or `0` to inherit global mask.
5. **Run controlled validation snapshots.**
   Robot stopped at known field points, compare solved (x, y) to tape measurements.

5.2. Per-sensor mask strategy

Recommended progression:

- start all sensors on `MAP_PERIMETER`;
- add `MAP_LONG_GOALS_ALL` only for sensors that reliably see long-goal geometry;
- add `MAP_CENTER_GOALS`, `MAP_MATCHLOADERS`, or `MAP_PARK_ZONES` only after repeated field validation.

Runtime edits without rebuilding defaults:

```
snapshot_bindings_update_sensor(0, frontCfg);
snapshot_bindings_set_sensor_mask(1, MAP_PERIMETER
↪  | MAP_LONG_GOALS_ALL);
```

5.3. Library-specific callback mappings

```
// LemLib-style mapping
runtime.get_heading_deg = [](void*) { return
↪  chassis.getPose(false, false).theta; };
runtime.get_guess_x_in  = [](void*) { return
↪  chassis.getPose(false, false).x; };
runtime.get_guess_y_in  = [](void*) { return
↪  chassis.getPose(false, false).y; };
runtime.apply_pose      = [](void*, float x, float
↪  y, float h, float, float) {
  chassis.setPose(x, y, h);
};
```

5.4. Autonomous call pattern

Use a conservative call site:

1. brake drivetrain and wait 150-250 ms;
2. choose a known quadrant (`BL`, `BR`, `TL`, `TR`) when available;
3. call `snapshot_setpose_quadrant(q)`;
4. continue routine only after checking `ok`.

5.5. Debug and regression checklist

- verify at least three known field poses per start side;
- monitor `used_sensors` and χ^2 for outliers;
- if snaps fail, first reduce masks to `MAP_PERIMETER` and re-validate geometry;
- only increase K or interior masks after repeatable baseline performance.

Table 1: Recommended default parameters for snapshot pose.

Parameter (default)	Rationale
Global fallback geometry mask (`MAP_PERIMETER`)	Walls are the most stable and least ambiguous landmarks. Start perimeter-only until wiring and geometry are verified.
Per-sensor mask override (`0` = inherit)	Sensors at different heights may see different objects. Override only where needed (e.g., low sensor: walls only; higher side sensor: walls + `MAP_LONG_GOALS_ALL`).
Samples per sensor ($n = 5$)	Median-of-5 removes transient spikes without a long acquisition window.
Inter-sample delay (35 ms)	Reduces repeated reads from the same internal frame and improves median robustness.
Valid range window (20 mm to 2000 mm)	Aligns with sensor limits and rejects clipped/no-object values.
Confidence gating (enabled; $c_{\min} = 35$ for $z > 200$ mm)	Rejects weak long-range reflections with minimal tuning burden.
Candidates per sensor ($K = 1$ perimeter; $K = 2$ with interior masks)	Higher K allows ambiguity resolution when non-perimeter geometry is enabled.
Quadrant margin (2 in)	Maintains a useful quadrant prior near center boundaries while avoiding over-constraining starts.
Per-sensor residual cap ($\chi^2_{\max} = 9$)	Rejects outlier-dominated solutions before pose application.
Minimum sensors used (2)	At least two independent constraints are required for a 2D translation solve.