RBE 550 — MOTION PLANNING

Assignment 4: Valet

Liam Jennings

April 2026

0.1 Introduction

This project implements a motion planning simulator for four vehicle types of increasing kinematic complexity: a holonomic point robot, a differential drive, an Ackermann car, and a car towing a trailer. Each vehicle navigates a randomly generated obstacle field from a fixed start pose to a goal pose.

The planner uses a python implementation of the Hybrid A* search algorithm, a variant of A* that searches over a continuous (x, y, θ) state space while using a discretised grid for duplicate detection. The planner explores this grid by generating circular arc motion primitives, verifying that they do not collide with any obstacles. When within a set range of the goal, the planner occasionally attempts to plan a direct path while exploring nodes. If this path is verified to be collision free, the planner reconstructs a kinematically correct path by traversing its tree back to the start pose.

Collision detection is built on shapely polygon geometry. Shapely is slow for repeated small queries, so a broad-phase occupancy-grid filter and a pre-rotated heading cache were implemented to keep query cost low. After successfully finding a raw path to the goal pose, it is post-processed by probabilistic shortcutting and then resampled to a uniform velocity for smooth playback.

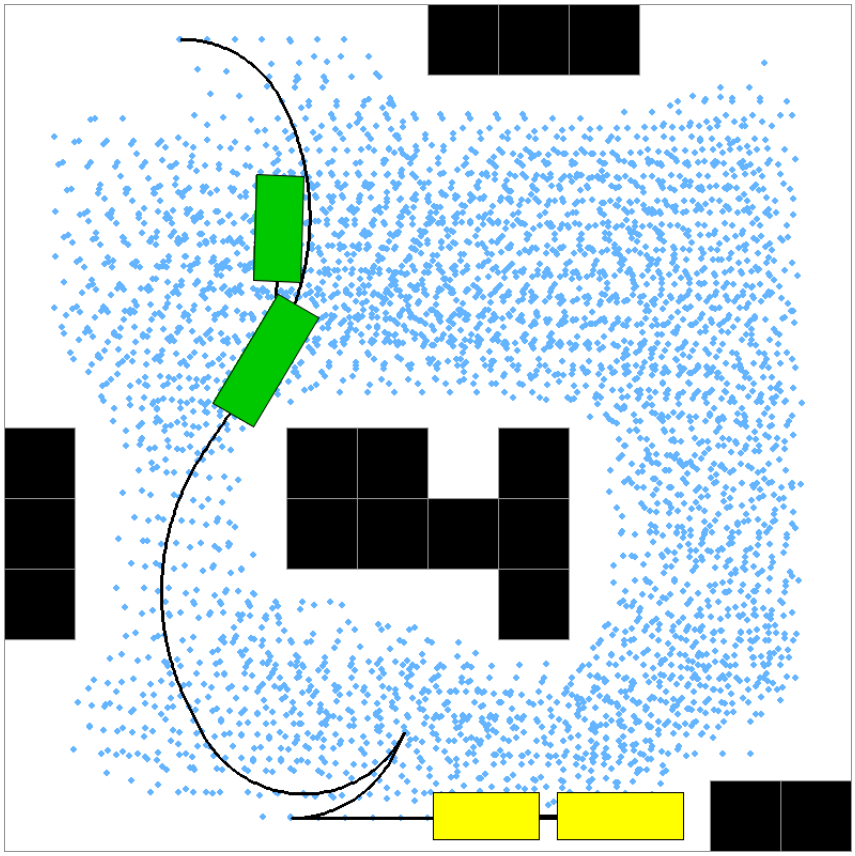

Figure 1: Trailer simulation mid-playback. The traced path follows the center of the rear axle for Ackermann and trailer vehicles, and the geometric center for differential drive and point robots. Blue dots mark the (x, y) position of every state expanded during the planning phase.

0.2 Setup

0.2.1 Installation

The project was originally built with **Python 3.12**. The `reeds_shepp` dependency (`deps/pyReedsShepp`) also includes a C++ extension that requires the Boost headers at build time. Install them before setting up the environment:

- **Ubuntu/Debian:** `sudo apt install libboost-dev`
- **Arch:** `sudo pacman -S boost`
- **macOS:** `brew install boost`

The recommended way to install dependencies is with uv:

```
# install uv (once, system-wide)
curl -LsSf https://astral.sh/uv/install.sh | sh

uv sync          # installs Python 3.12 + all dependencies
uv run runsim    # run the simulator
```

Alternatively, a `Makefile` is provided for manual setup if Python 3.12 is already available. It also provides additional tools for managing the project:

```
make pip-install   # create .venv and install all dependencies via pip
make venv          # create .venv if needed, then print the activation command
make clean         # remove .venv, caches, build artifacts, and any recorded mp4s
```

0.2.2 Usage

After activating the environment, the simulator is launched with the `runsim` entry point:

```
runsim [bot_type] [options]
```

`bot_type` is one of `point`, `diff`, `car`, or `trailer` (default: `diff`). Available options:

```
-m / --manual    drive the bot manually with arrow keys instead of planning
-r / --record    save the simulation to recording.mp4
-s / --seed N    fix the RNG seed for a reproducible obstacle layout
```

While the simulation is running, pressing `s` saves a screenshot to the current directory (distinct from the `-s` / `--seed` launch flag).

For example, to run the trailer bot with a fixed seed:

```
runsim trailer -s 42
```

When no seed is provided, a random one is chosen and printed to the terminal at startup, allowing any run to be reproduced with `-s`. The terminal also prints planner progress during the search - node expansion count, open set size, current cost and position - as well as a summary on completion including the total number of expansions, shortcuts applied during smoothing, and final path length after resampling.

0.2.3 Dependencies

The project uses the following notable external libraries:

- **Pygame** — provides the real-time 2-D rendering window, keyboard event handling for manual drive mode, and the simulation loop.
- **Shapely** — computational geometry library used to represent robot footprints as polygons and query them against obstacle geometry via an STRtree[1] spatial index.

- **imageio / imageio-ffmpeg** — used together to encode simulation recordings to MP4 when the `--record` flag is passed; imageio-ffmpeg supplies the FFmpeg backend.
- **reeds_shepp (pyReedsShepp)** [1] — Python bindings to a C implementation of Reeds-Shepp path length and curve computations. The original library [2] targets an older Python version; a personal fork (stored as a git submodule at `deps/pyReedsShepp`) was created to update the Cython build configuration for compatibility with the Python version used for the project.

0.3 Design Goals

Development was incremental: start with a working holonomic point robot (not required, but useful for isolating planner bugs), then add each vehicle type in order of complexity. Shapely was used for collision detection from the start as a correct-if-slow baseline, with optimisation deferred until the planner was working. Performance work was largely driven by profiling with `cProfile` and `snakeviz`.

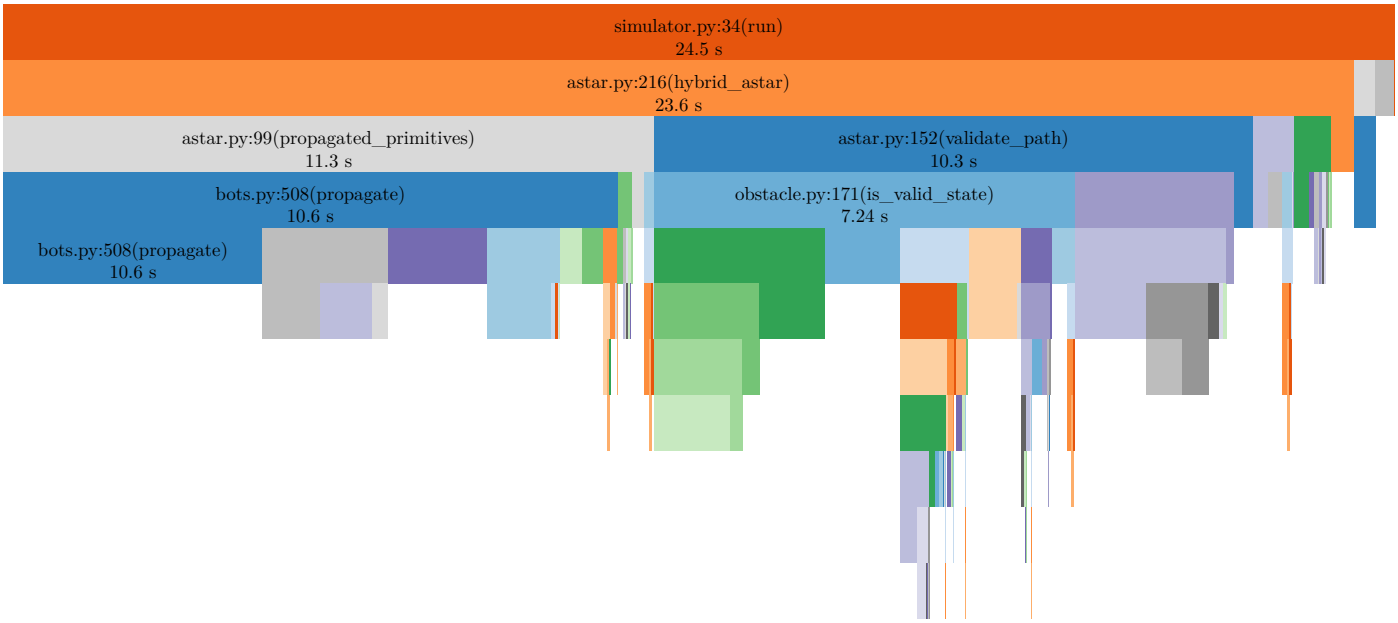

Figure 2: Icicle plot rendering of profiled trailer simulation. Profiling shows time is split almost equally between primitive generation (propagate, 10.6 s) and collision checking (validate_path, 10.3 s), with is_valid_state accounting for 7.2 s of that collision budget.

The code was also an experiment with modern Python's type system — `typing.Protocol` and generics were used to keep vehicle types interchangeable without inheritance, inspired by Rust traits. Interesting in practice, though probably not worth the overhead in the future.

[1] A spatial index structure for fast nearest-neighbour queries, in the same vein as an Octree

1 Approach

1.1 Obstacle Environment Implementation

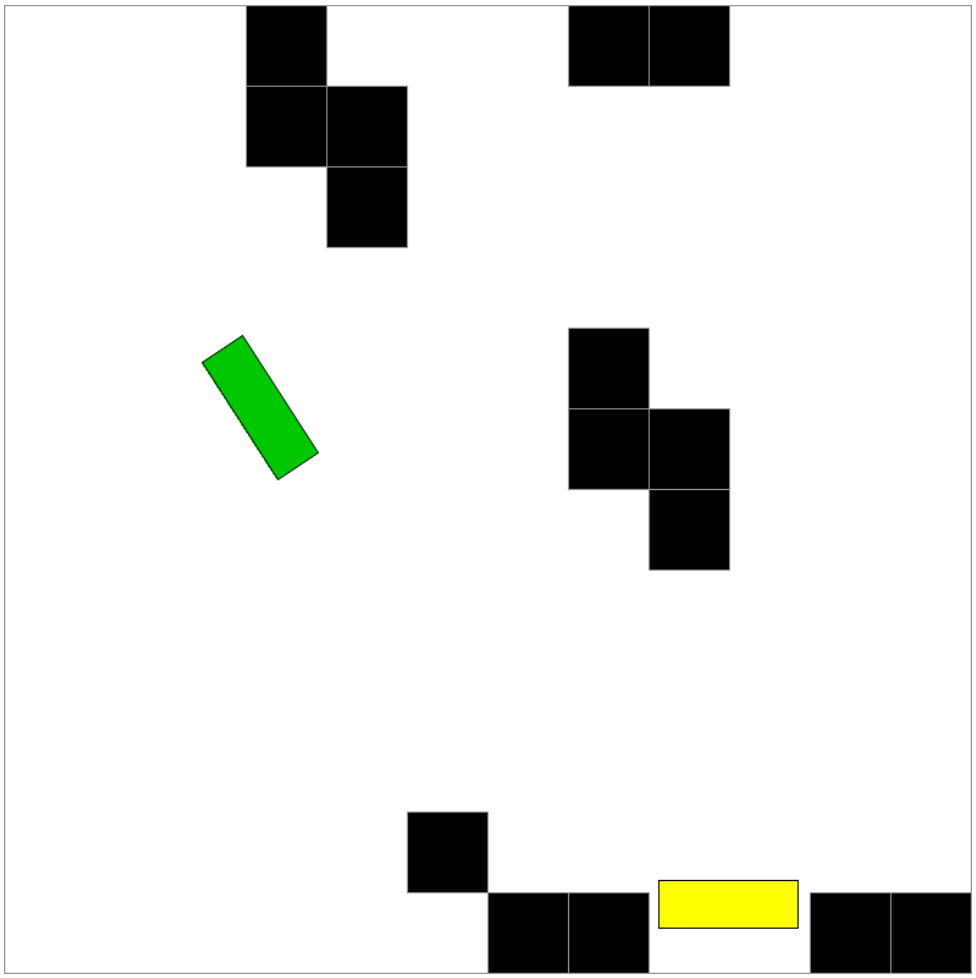

Figure 3: Obstacles generated given a 10% occupation requirement. Captured while driving the car in manual mode.

The environment consists of a fixed-size 2D grid with randomly placed axis-aligned obstacles, a fixed start and goal at opposite corners, and a printable RNG seed for reproducibility. Obstacles are stored as both a NumPy boolean grid for fast broad-phase rejection and a Shapely `STRtree` for exact intersection tests.

Obstacles are generated by randomly placing tetrominoes throughout the grid, until a set proportion of the grid is occupied by obstacles. The standard proportion is 10%. The vehicles (green rectangle) start in the top left, and attempts to navigate to the goal pose (yellow rectangle) in the bottom right. The starting and ending positions are always cleared.

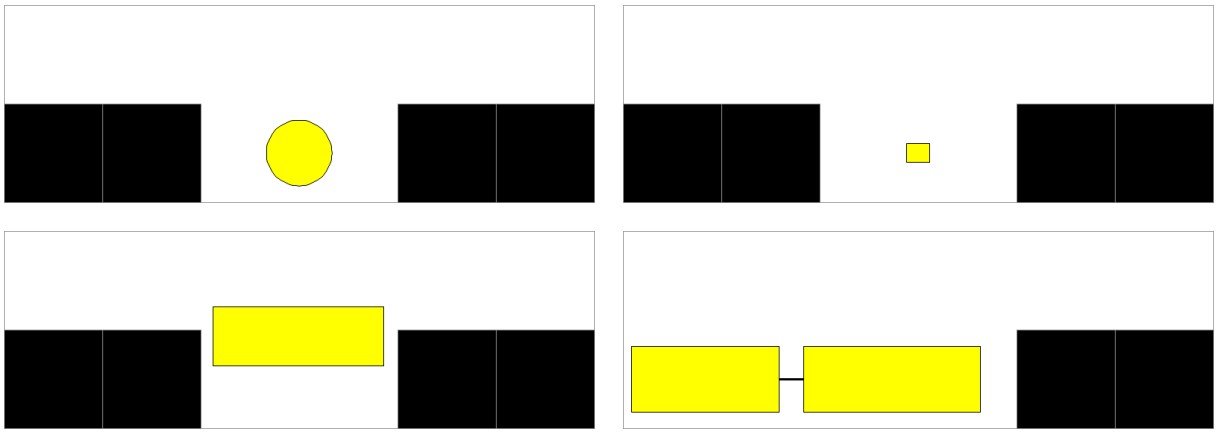

Figure 4: Goal poses for each vehicle type: point robot (top left), differential drive (top right), Ackermann car (bottom left), trailer (bottom right).

1.1.1 Car Goal Offset

Of specific note, the car's goal pose is offset further from the bottom edge than the other vehicles. Where the other vehicle goal poses are set 1 cell from the bottom edge, the car's is instead set 1.425 from that edge[2]. This decision was made to allow the goal position to be reachable: without it, the planner was unable to reach the goal at the standard offset.

Initial coarse tests suggested the feasibility cutoff was around 1.2 grid cells from the bottom edge, but this was an artifact of the heading cache rounding to 5° increments producing false positives. Systematic testing with fine collision checking found the true cutoff: an offset of 1.38 exhausts the search space without finding a path, while 1.39 succeeds. Above an offset of 1.44, the car no longer needs to make the parallel parking maneuver to reach the goal. The final offset of 1.425 was chosen to sit within this window, ensuring the goal is reliably reachable while preserving the parallel parking maneuver.

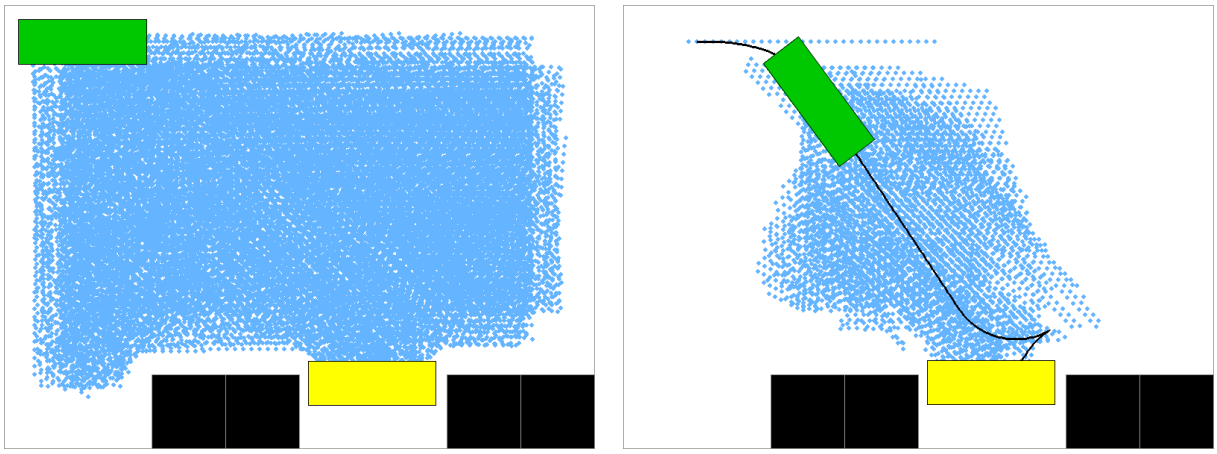

Figure 5: Offset 1.38 (left) exhausts the search after 30,004 expansions without finding a path. Offset 1.39 (right) succeeds in 5,719 expansions, confirming the feasibility cutoff.

[2]Goal positions are specified as an offset from the grid edge passed to grid_to_coords, which adds an additional 0.5 to center within the cell; the true geometric distance is therefore offset − 0.5 cells. Other vehicles use an offset of 1 (0.5 cells geometric distance); the car uses 1.425 (0.925 cells).

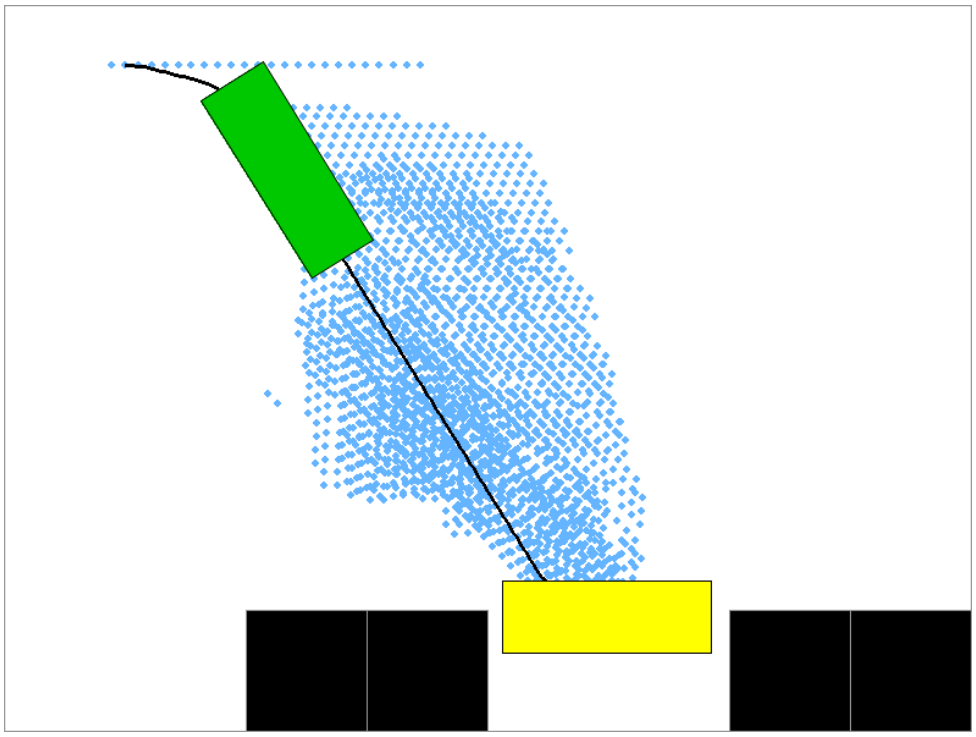

Figure 6: At offset 1.44 the car reaches the goal in 2,436 expansions without a parallel parking maneuver, establishing the upper bound of the target window.

1.2 Collision Detection

The collision checker was designed to handle two geometry types: rotated rectangular footprints (for the robot body, truck, and trailer) and a line segment (the hitch bar connecting truck to trailer). Obstacles are axis-aligned grid cells, stored as both a numpy boolean array for fast grid lookups and a shapely `STRtree` of `box` polygons for exact intersection tests.

Shapely is designed for general topological geometry analysis and is not inherently optimised for repeated per-frame queries against a fixed obstacle set. Naively calling `intersects` on individually translated geometries at every state check proved too slow for the planner's inner loop. Three optimisations were applied to address this, and their combined effect is quantified in Figure 9.

1.2.1 Heading cache

Rotating a Shapely geometry is expensive: internally, rotation calls `_affine_coords`, which walks every vertex of the polygon through a matrix multiply. To eliminate this per-query cost, each base shape is pre-rotated at 72 evenly-spaced headings (every 5°) at construction time. Per-state footprint queries look up the nearest pre-rotated shape and record only the (x, y) offset, deferring translation until an exact check is actually needed.

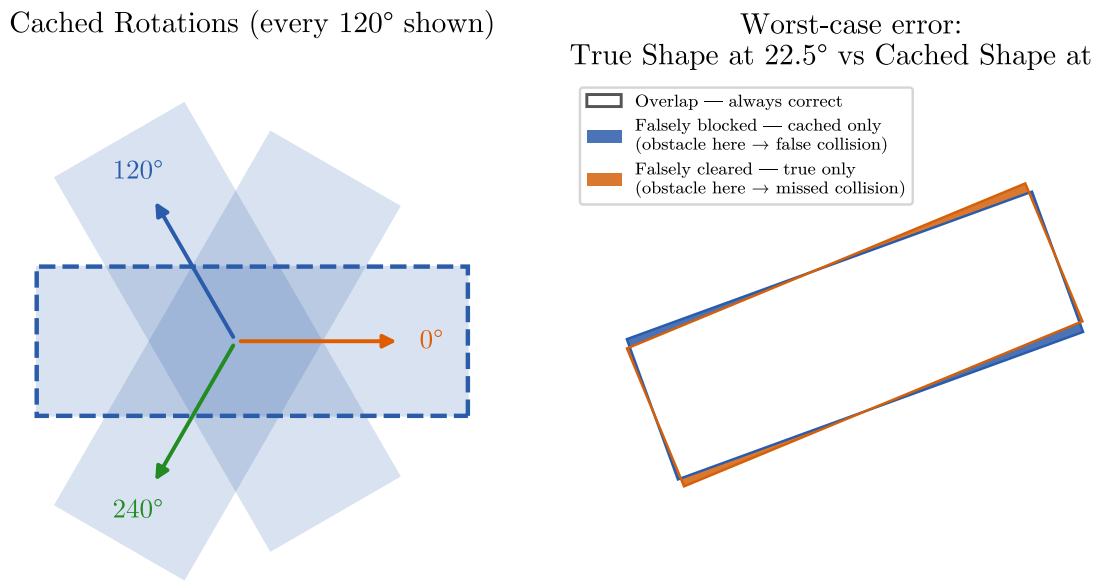

Figure 7: **Left:** three of the 72 pre-rotated car footprints stored in the heading cache (shown at 120° intervals for clarity; the full cache covers every 5°). Arrows indicate the forward direction of each cached shape. **Right:** worst-case approximation error — the true footprint at 22.5° versus the nearest cached shape at 20°, the maximum possible snap error being 2.5°. Errors only occur when an obstacle intersects exclusively one discrepancy region (~3.5% of vehicle area each); obstacles within the shared overlap are correctly detected by both shapes.

As seen in Figure 9, this nearly eliminates `rotate` entirely (974,294 calls → 3,622), and drives a 65× reduction in `_affine_coords` calls (1,948,445 → 29,917), representing the single largest source of time saved across both optimisations.

1.2.2 State validation

The state validator applies checks in order of increasing cost, exiting as soon as any check fails.

```
function IS_VALID_STATE(geometries):
    for each geometry in geometries:
        if geometry is a line segment:
            if any endpoint lies outside the environment boundary:
                return false
            if the segment intersects an obstacle:
                return false
        else: // Geometry is a rotated rectangle
            compute translated axis-aligned bounding box
            if bounding box lies outside the environment boundary:
                return false
            if no obstacle can possibly overlap the bounding box:
                continue  // broad-phase rejection
            compute translated geometry
            if geometry intersects any obstacle:
                return false
    return true
```

Listing 1: Pseudocode describing state validation process.

For rectangular footprints, the translated axis-aligned bounding box (AABB) is checked against the environment boundary first. If it lies outside, the state is immediately rejected without

touching any geometry. If it lies inside, the AABB is mapped to grid cell indices and the corresponding subgrid slice is checked for any occupied cell via `_has_possible_collision`. This eliminates the majority of states at negligible cost as the geometry is only translated and tested exactly against the `STRtree` (via `intersects_any`) if the broad phase reports a possible collision.

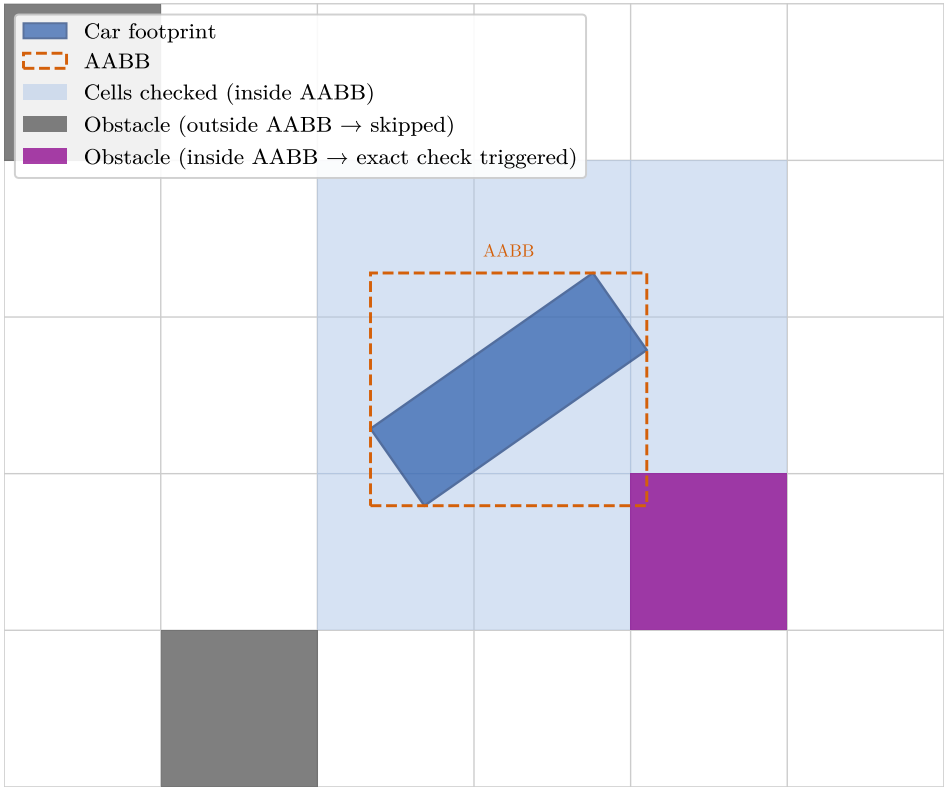

Figure 8: Broad-phase AABB collision check. The car footprint (blue) is rotated at 35°; its axis-aligned bounding box (orange, dashed) is mapped to grid cell indices and only the highlighted cells are queried for obstacles. The purple obstacle falls inside the AABB and triggers an exact Shapely intersection check, but does not intersect the footprint, so the state is accepted as free. The grey obstacles are outside the AABB and are therefore skipped entirely. This filter is conservative, but it catches the vast majority of states considered without requiring more expensive tests.

The effect is visible in Figure 9: `_has_possible_collision` appears only in the optimised profile (935,415 calls), while `intersects_any` — the expensive Shapely intersection — drops 41× from 938,273 to 23,006 calls. Crucially, `is_valid_state` is called roughly the same number of times in both profiles (487,071 vs 485,689), confirming that the AABB filter does not reduce collision accuracy, instead avoiding exact checks on states that are clearly free.

The combined effect of these two optimisations is a 3.98× end-to-end speedup (81.5 s → 20.5 s). For reference, `propagate` — which handles motion primitive generation and is unaffected by the collision optimisations — shows virtually identical call counts and cost in both profiles, confirming the speedup is attributable entirely to the collision checker.

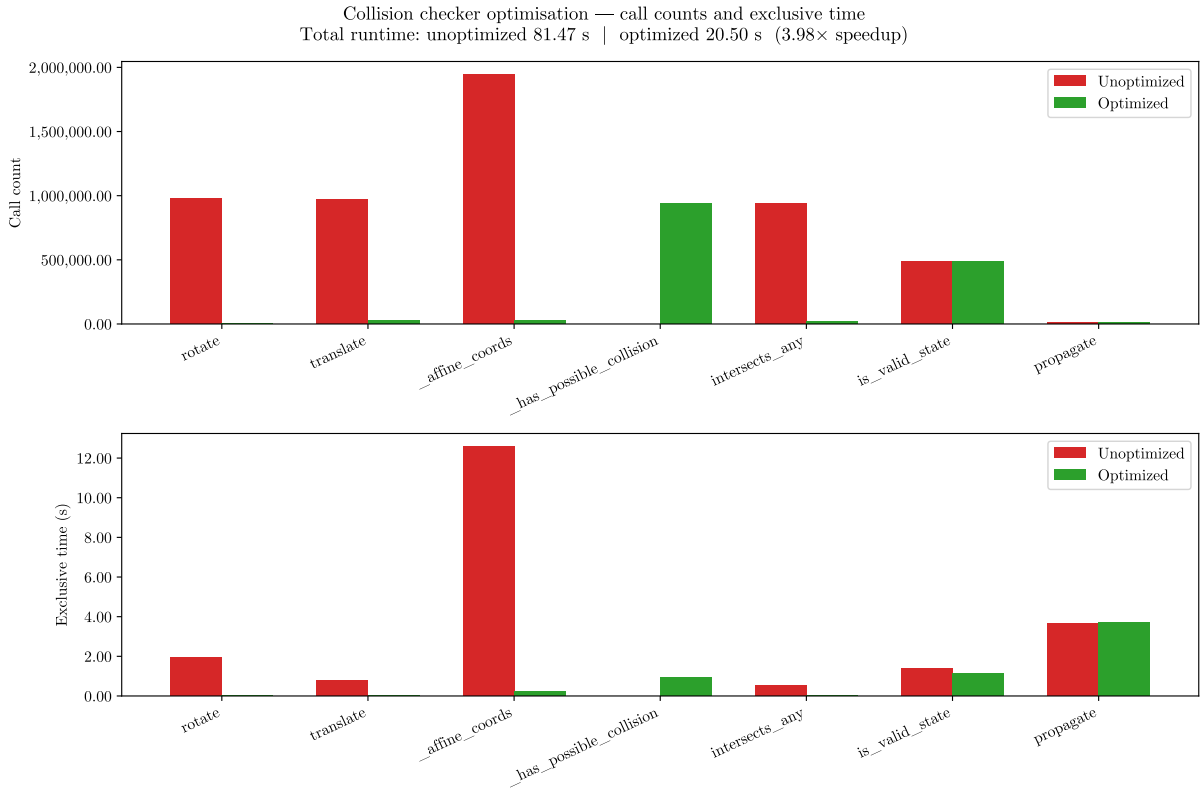

Figure 9: Call counts (**above**) and exclusive CPU time (**below**) for key functions, comparing the unoptimised and optimised collision checker. Total runtime: 81.5 s (unoptimised) vs 20.5 s (optimised), a 3.98× speedup. `propagate` is included as a control to show that primitive generation cost is largely unchanged between runs.

1.2.3 Line segment handling

The trailer's hitch bar connects the truck rear to the trailer's front axle. In this project, this is simplified to a line from the center of the rear axle of the truck to the trailers front axle. This cannot be represented as a box, so it is stored as raw endpoints rather than a Shapely geometry. Containment is checked by testing both endpoints against the boundary. Obstacle intersection uses Amanatides and Woo's fast voxel traversal algorithm [3] against an occupancy grid, stepping cell-by-cell along the segment and returning on the first occupied cell.

1.2.4 Path-level validation

During planning, entire primitive trajectories must be validated, not just individual states. A two-phase approach is used: first, the last state and every 4th intermediate state are checked using the approximate (cached, untranslated) footprints. Most invalid primitives are caught here. If that passes, and fine collision checks are enabled, the remaining states are checked with exact footprints.

1.3 Planner Implementation

1.3.1 Hybrid A*

The planner uses Hybrid A*, a variant of A* that operates over a continuous state space rather than a discrete grid. States are represented as continuous (x, y, θ) (and φ for the trailer), but duplicate detection uses a discretised grid: XY position is rounded to the nearest spacing multiple, and headings are floor-divided into fixed angular bins of `angular_spacing` size. A node

is only expanded once per discrete cell, preventing the search from cycling while still allowing the robot to reach any continuous pose within a cell.

Neighboring cells are generated from a node by sampling a subset of the possible control inputs, and then simulating the kinematics resulting from them.

The heuristic used is the Reeds-Shepp path length from the current state to the goal, ignoring obstacles. This is admissible and generally tighter than Euclidean distance for non-holonomic robots, as it accounts for turning radius constraints that Euclidean distance ignores [4, Section 8.1], [5, Page 2].

```
function HYBRID_ASTAR(start, goal):
    push start onto open heap
    while open heap is not empty:
        node ← pop lowest f = g + h node
        if node already visited: continue
        mark node visited
        if node is within terminal range of goal:
            path ← LAST_SHOT_CONNECTION(node, goal)
            if path is valid and collision-free:
                return POSTPROCESS(path)
        for each primitive in PROPAGATE(node.state):
            if primitive is collision-free:
                push endpoint onto heap with updated g cost
    return failure
```

Listing 2: Hybrid A* pseudocode

1.3.2 Primitive Generation

At each node, the bot generates a set of motion primitives by sampling control inputs. For car-like and trailer vehicles, this means sampling steering angles δ evenly between $-\delta_{\max}$ and $+\delta_{\max}$, combined with forward and reverse speed. For differential drives, different angular velocities ω are instead sampled. Each pair is integrated over n timesteps using an analytic arc trajectory formula[3] to avoid computational cost and error accumulation:

$$\begin{aligned} \theta(i) &= \theta_0 + \omega \cdot i \cdot dt \\ x(i) &= x_0 + R \cdot (\sin(\theta(i)) - \sin(\theta_0)) \\ y(i) &= y_0 - R \cdot (\cos(\theta(i)) - \cos(\theta_0)) \end{aligned} \tag{1}$$

where $R = \frac{\nu}{\omega}$, and i is the i-th timestep of dt during the turn. Although the Ackermann and differential drive models differ in how ω is produced, both can be reduced to the same unicycle model [6, Section 13.1.2.3]. Therefore, both kinds of vehicles may be modeled using the above formula.

The step count n is calculated to satisfy two constraints:

$$n_\nu \geq \frac{\text{spacing}}{|\nu| \cdot dt}, \quad n_\omega \geq \frac{\text{angular\_spacing}}{|\omega| \cdot dt} \tag{2}$$

The final value of n is simply $\max(n_\nu, n_\omega)$. Satisfying these constraints guarantees every primitive lands in different (x, y) cell and heading bin compared to the initial cell.

[3] For derivation, see Appendix A:

The cost of each primitive (and therefore edge in the A* graph) is the arc length of the primitive plus a weighted heading change, with an additive penalty for reversals to discourage the planner from exploring backwards.

Two vehicle-specific exceptions apply:

- The differential drive appends additional rotate-in-place primitives with $v = 0$. These primitives have zero arc length, their cost relying purely on heading change, allowing the planner to reason about in-place rotation as a distinct maneuver.
- The trailer's primitive generation is more involved. The truck follows the same analytic arc as any other Ackermann vehicle, but the trailer heading φ is coupled to the truck via a nonlinear ODE:

$$\dot{\varphi} = \frac{v}{M} \sin(\theta - \varphi) \tag{3}$$

  where M is the hitch-to-trailer-axle distance [6, Equation 13.19]. This cannot be integrated in closed form alongside the truck arc, so φ is propagated sequentially: the truck positions and headings are computed analytically in one pass, then φ is stepped forward through the resulting sequence using Euler's method.

  Paths that result in the trailer jackknifing[4] are treated as invalid: each step is checked against a jackknife limit via the condition $|\theta - \varphi| < 90°$. If the limit is exceeded, the primitive is discarded.

1.3.3 Last-Shot Connection

When a node falls within a euclidean distance of `terminal_radius` of the goal pose, a closed-form trajectory ignorant of obstacles is attempted directly to the goal rather than continuing to expand primitives. For car and trailer bots this is a Reeds-Shepp path [7]; for the differential drive bot it is a rotate-drive-rotate sequence.

This avoids the difficulty of landing exactly on the goal pose through discrete primitives, and is the mechanism by which the planner achieves precise heading alignment at the goal.

The connection is not attempted on every node within the terminal radius. Since the trajectory must be validated for collisions, attempting it from every qualifying node is expensive and unlikely to pay off far from the goal where obstacles are more likely to block a direct path. Following [4, Section 6.1.2], a distance-proportional probability gate is applied: the attempt probability scales linearly from 0 at the edge of the terminal radius to 1 at the goal position itself. The exact calculation is:

$$P = 1 - \frac{\|\boldsymbol{p} - \boldsymbol{p}_g\|}{R_t} \tag{4}$$

where $\boldsymbol{p} = (x, y)$ is the position component of the current state, $\boldsymbol{p}_g$ is the goal position, and R_t is the terminal radius.

The attempted path is validated for collisions before being accepted. For the trailer, the trailer heading is simultaneously integrated along the RS path and the attempt is rejected if jackknifing occurs. As the planner is unable to directly control the trailer heading, it rejects attempts if the trailer heading at the goal is outside a set tolerance.

[4] When a vehicle towing a trailer loses control, causing the trailer to swing out and form an acute angle with the towing vehicle, resembling a folding pocket knife.

1.3.4 Post-Processing

The raw path from the search is passed through two steps before playback:

- **Path smoothing** [8, Section 3.2] (`smooth_path`): two random indices are chosen, a direct connection is attempted via `generate_trajectory`, and if collision-free it replaces the span between those two indices. After repeating for 100 iterations, this step reduces total path length by replacing indirect routes imposed by the search order with more direct connections.

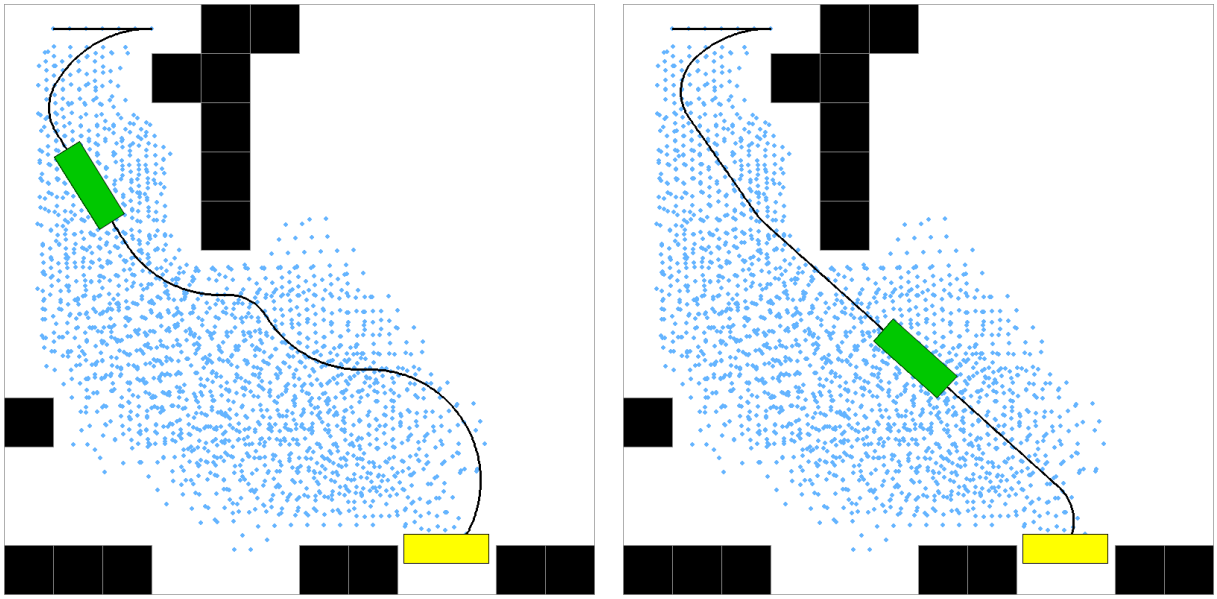

Figure 10: Car path before (**left**) and after (**right**) probabilistic shortcutting. The raw path takes a wide detour imposed by the order in which the search explored the space; smoothing finds direct Reeds-Shepp shortcuts that reduce total path length.

- **Resampling** (`resample_path`): the variable-density path is converted to uniform arc-length samples so the animation plays back at constant velocity. Pure-rotation segments (zero XY displacement) are detected separately and resampled at a distinct angular rate.

2 Results

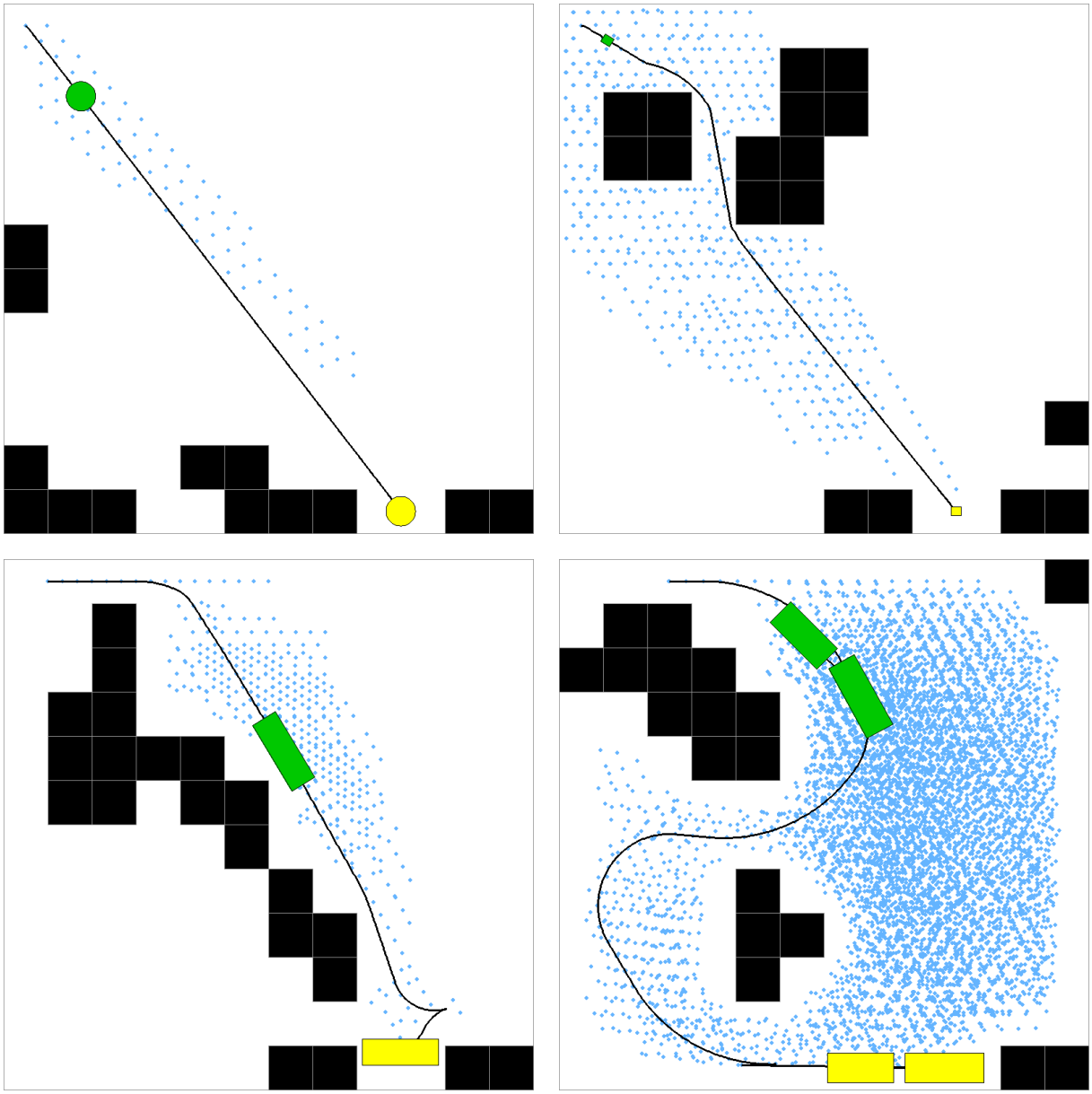

Figure 11: Navigation results for each vehicle type: point robot (**top left**), differential drive (**top right**), Ackermann car (**bottom left**), trailer (**bottom right**).

The planner works well across all four vehicle types. The trailer in particular produces some satisfying paths — watching it navigate tight spaces while keeping the trailer heading under control is a good demonstration that the coupled kinematics are being handled correctly. Performance ended up in a reasonable place after profiling, making iteration much faster than it was early on. The codebase also ended up in a good state structurally; having a clean interface between vehicle types made it easy to experiment with planner parameters and add features without things breaking unexpectedly.

2.1 Areas for Improvement

- **Obstacle-aware heuristic.** The Reeds-Shepp heuristic ignores obstacles, which can cause the planner to underestimate costs in cluttered environments. Augmenting it with a precomputed 2D Dijkstra cost-to-go map from the goal would give tighter estimates and likely reduce

node expansions. Other implementations [4], [5] have tried this approach, either individually or combined with obstacle-free distance metrics.

- **State representation boundary.** Throughout the planner, fully typed state objects are used rather than raw float arrays or tuples. This made the logic clean and easy to reason about, but there is a performance cost. NumPy operations on raw arrays would be substantially faster than constructing and passing around dataclass instances in the planner's inner loop. This was a conscious tradeoff in favour of correctness and clarity over speed.
- **Dataclass performance.** Switching from standard Python dataclasses to a library like `msgspec` could reduce the overhead of creating large numbers of short-lived state objects during search. Learning how to have these interact with NumPy would be interesting.
- **Full SAT-based collision.** Shapely is still used for the narrow-phase intersection checks. Replacing it entirely with raw NumPy checks using the Separating Axis Theorem would remove the last external geometry dependency and likely be faster. The current version is good enough, but it's a loose end. Other approaches to collision detection like circular bounding boxes could be explored as well.
- **Planner visualisation.** It would be nice to watch Hybrid A* progress in real time: drawing expanded nodes, edges, the open set, and the current best path as the search runs. The current sim only shows the final result. This wasn't a priority, but it would make debugging and tuning much more intuitive.
- **Full dynamics and controls simulation.** The current simulator only considers kinematics, with perfect awareness of its surroundings. A more realistic setup could add a simple tracking controller and model dynamics from actuators, or play with sensors and requiring the robot to explore before finding its way.

Bibliography

[1] G. Liemann, "pyReedsShepp." Feb. 2020.

[2] G.-H. Liu, "pyReedsShepp." Aug. 2016.

[3] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," in *EG 1987-Technical Papers*, , Ed., Eurographics Association, 1987. doi: 10.2312/egtp.19871000.

[4] K. Kurzer, *Path Planning in Unstructured Environments : A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle*. 2016. Accessed: Apr. 16, 2026. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-198534

[5] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Practical Search Techniques in Path Planning for Autonomous Driving," *AAAI Workshop - Technical Report*, p. , 2008, [Online]. Available: https://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf

[6] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006. [Online]. Available: http://lavalle.pl/planning/

[7] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards.," *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990.

[8] R. Geraerts and M. H. Overmars, "Creating High-quality Paths for Motion Planning," *The International Journal of Robotics Research*, vol. 26, no. 8, pp. 845–863, Aug. 2007, doi: 10.1177/0278364907079280.

Appendices

Appendix A: Derivation of Equation 1

Both the differential drive and Ackermann kinematics can be simplified to the unicycle kinematic model [6, Section 13.1.2.3]:

$$\begin{aligned} \dot{x} &= \nu \cdot \cos(\theta) \\ \dot{y} &= \nu \cdot \sin(\theta) \\ \dot{\theta} &= \omega \end{aligned} \tag{5}$$

The differential drive model is directly equivalent, while Ackermann steering has a dependency between the velocity ν and steering angle δ:

$$\omega = \nu \cdot \frac{\tan(\delta)}{L} \tag{6}$$

where L is the wheelbase (distance between front and rear axles).

When ν and ω are held constant, the heading evolves linearly as $\theta(t) = \theta_0 + \omega$, reducing the position integrals to standard trigonometric forms with known closed-form solutions.

Treating this as an initial value problem with state (x_0, y_0, θ_0) at $t = 0$, the heading integrates directly, as seen above:

$$\theta(t) = \theta_0 + \omega \cdot t \tag{7}$$

Substituting into the position equations and integrating:

$$\begin{aligned} x(t) &= x_0 + \int_0^t \nu \cdot \cos(\theta_0 + \omega \cdot \tau)\ d\tau = x_0 + \frac{v}{\omega} \cdot \sin(\theta_0 + \omega \cdot \tau)\Big|_{\tau=0}^{\tau=t} \\ y(t) &= y_0 + \int_0^t \nu \cdot \sin(\theta_0 + \omega \cdot \tau)\ d\tau = y_0 - \frac{v}{\omega} \cdot \cos(\theta_0 + \omega \cdot \tau)\Big|_{\tau=0}^{\tau=t} \end{aligned} \tag{8}$$

The radius of the circle traced by this arc can be found through the equation $R = \frac{L}{\tan(\delta)}$ [6, Section 13.1.2.1]. from Equation 6, $\omega = \nu \cdot \frac{\tan(\delta)}{L}$, so $R = \frac{L}{\tan(\delta)} = \frac{\nu}{\omega}$. Substituting $\frac{\nu}{\omega}$ for R:

$$\begin{bmatrix} x(t) = x_0 + R \cdot (\sin(\theta(t)) - \sin(\theta_0)) \\ y(t) = y_0 - R \cdot (\cos(\theta(t)) - \cos(\theta_0)) \end{bmatrix} \tag{9}$$

This can be extended to include the case of $\omega \approx 0 \Rightarrow R = \frac{v}{0}$, where the arc approaches a straight line as $R \to \infty$. As in this case the heading, and thus the components of the velocity remain constant, the kinematics of this case can be easily formulated from Equation 5:

$$\begin{bmatrix} x(t) = x_0 + \dot{x} \cdot t = x_0 + \nu \cdot t \cdot \cos(\theta_0) \\ y(t) = y_0 + \dot{y} \cdot t = y_0 + \nu \cdot t \cdot \sin(\theta_0) \\ \theta(t) = \theta_0 + \omega \cdot t = \theta_0 \end{bmatrix} \tag{10}$$

This result can be verified formally via L'Hôpital's rule applied to the arc equations as $\omega \to 0$:

$$\lim_{\omega \to 0} \frac{\nu \cdot (\sin(\theta_0 + \omega \cdot t) - \sin(\theta_0))}{\omega} = \frac{0}{0} = \lim_{\omega \to 0} \frac{\nu \cdot \cos(\theta_0 + 0 \cdot t) \cdot t - 0}{1} = v \cdot t \cdot \cos(\theta_0) \tag{11}$$

$$\lim_{\omega \to 0} \frac{\nu \cdot (\cos(\theta_0 + \omega \cdot t) - \cos(\theta_0))}{\omega} = \frac{0}{0} = \lim_{\omega \to 0} \frac{\nu \cdot -\sin(\theta_0 + 0 \cdot t) \cdot t - 0}{1} = -v \cdot t \cdot \sin(\theta_0) \tag{12}$$

which can be substituted back into the arc equations to get the straight line equations. Stationary ($\nu = 0, |\omega| > 0$) turns do not suffer from this instability, however this would require an ackermann car to have a steering angle of 90°, which would cause $\tan(\delta) \to \infty$ (Equation 6).