A* Global Planner – Developer Manual

Understanding the ROS 2 A* Path Planning Node

Astar Global Planner Project

February 2026

Contents

1 Introduction

This manual explains every component of the **A* Global Planner** – a standalone ROS 2 node that computes obstacle-free paths for a mobile robot. The planner is designed to work with the Nav2 navigation stack but can also run independently.

1.1 What Does This Node Do?

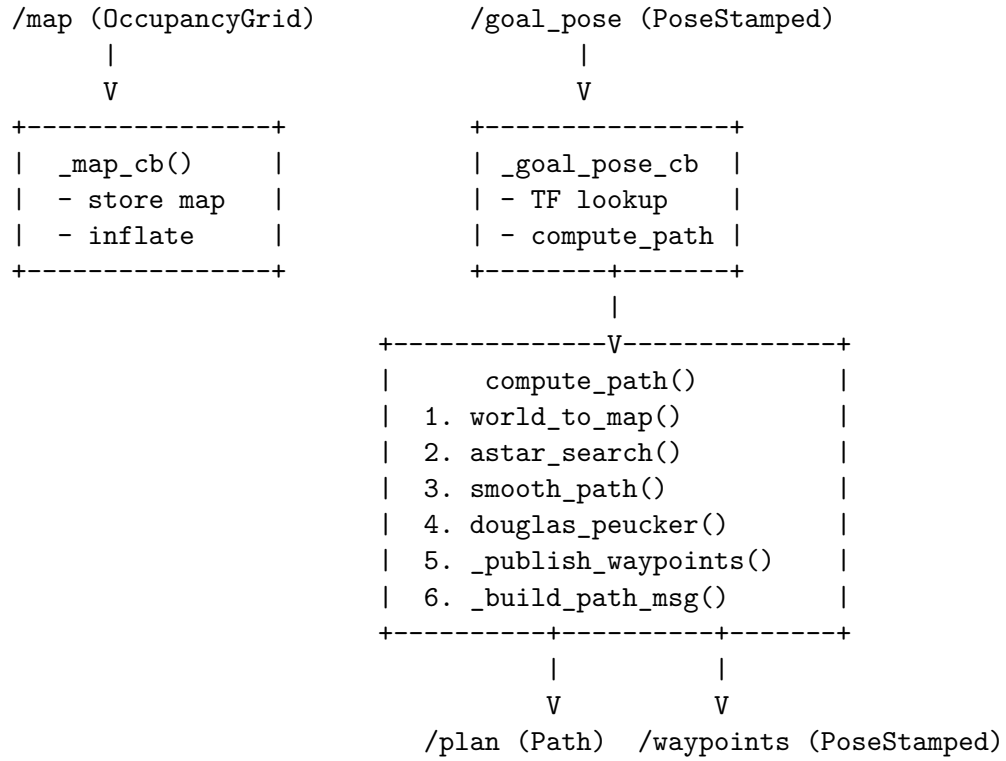
In simple terms, the node answers the question: **"Given where the robot is now and where it needs to go, what is the best route on the map?"**

It does this through a pipeline of four stages:

1. **Map ingestion** – reads the occupancy grid map and inflates obstacles
2. **A* search** – finds the shortest collision-free path on the grid
3. **Path smoothing** – removes the jagged grid-stepping artefacts
4. **Waypoint simplification** – reduces the path to a small set of key waypoints using Douglas-Peucker

The resulting path is published as a `nav_msgs/Path` message, and the simplified waypoints are published individually as `geometry_msgs/PoseStamped` messages on `/waypoints`.

1.2 High-Level Architecture



2 The Occupancy Grid & GridCostmap

2.1 What Is an Occupancy Grid?

An **OccupancyGrid** is a 2D grid where each cell holds a value from **0 to 100** (or **-1** for unknown):

Value	Meaning
0	Completely free space
1-49	Low-cost area (near obstacles but safe)
50-89	High-cost area (close to obstacles)
90-100	Lethal / occupied by an obstacle
-1	Unknown (not yet explored)

The map is published by SLAM or a map server on the `/map` topic.

2.2 The GridCostmap Class

The `GridCostmap` class wraps the raw `OccupancyGrid` message into a convenient interface for the planner.

2.2.1 Key Properties

Property	Type	Description
`width`, `height`	int	Grid dimensions in cells
`resolution`	float	Metres per cell (e.g. 0.05 = 5 cm)
`origin_x`, `origin_y`	float	World position of cell (0,0)
`data`	list[int]	Flat array of cost values, row-major
`lethal_threshold`	int	Cells >= this value are impassable (default: 90)
`inscribed_threshold`	int	Cells >= this are high-cost (default: 50)

2.2.2 Coordinate Conversion

The planner constantly converts between two coordinate systems:

- **World coordinates** (metres): `(wx, wy)` – where the robot actually is on the real map
- **Map coordinates** (cells): `(mx, my)` – integer grid indices

```
def world_to_map(self, wx, wy):
    mx = int((wx - self.origin_x) / self.resolution)
    my = int((wy - self.origin_y) / self.resolution)
    return mx, my

def map_to_world(self, mx, my):
    wx = (mx + 0.5) * self.resolution + self.origin_x
    wy = (my + 0.5) * self.resolution + self.origin_y
    return wx, wy
```

The `+0.5` in `map_to_world` centres the world coordinate in the middle of the cell rather than at its corner.

2.2.3 Cell Queries

- `in_bounds(mx, my)` – checks if the cell is within the grid
- `is_free(mx, my)` – returns `True` if the cell value is between 0 and `lethal_threshold`
- `get_cost(mx, my)` – returns the raw cost value (0–100)

2.3 Obstacle Inflation

2.3.1 Why Inflate?

The robot is not a point – it has a physical **radius**. If the planner only avoids cells marked as obstacles, the robot's body could still clip walls. Inflation expands every obstacle outward by the robot's radius so the planner treats the robot as a point but on a safely padded map.

2.3.2 How It Works

The `inflate_obstacles()` method uses **Breadth-First Search (BFS)**:

1. **Seed the BFS** with every lethal cell (cost >= 90) and unknown cell (cost = -1)
2. **Expand outward** cell by cell up to `inflation_radius_cells`
3. **Assign decaying costs**: cells nearest the obstacle get the highest cost, linearly decaying to `inscribed_threshold` at the edge

```
Obstacle:  XXXX
Inflated:  ..XXXX..

Cost profile (cross-section):
  90 --+          +-- 90  (lethal)
       | ######## |
  50 --+          +-- 50  (inscribed)
```

The inflation radius in cells is computed from the `robot_radius` parameter:

```
inflation_cells = ceil(robot_radius / resolution)
```

For example, with `robot_radius = 0.3 m` and `resolution = 0.05 m/cell`, inflation = 6 cells.

3 The A* Search Algorithm

3.1 What Is A*?

A* (pronounced "A-star") is a **graph search algorithm** that finds the shortest path between two points. It is the gold standard for grid-based path planning because it is:

- **Complete** – if a path exists, A* will find it
- **Optimal** – the path found is guaranteed to be the shortest (given an admissible heuristic)
- **Efficient** – it explores far fewer cells than brute-force search

3.2 How A* Works (Step by Step)

3.2.1 The Core Idea

A* maintains a priority queue (called the **open set**) of cells to explore. Each cell has a score:

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$ = **actual cost** from the start to cell n (accumulated as we walk)
- $h(n)$ = **heuristic estimate** of the cost from n to the goal
- $f(n)$ = **total estimated cost** of the cheapest path through n

A* always picks the cell with the **lowest f score** to expand next. This is what makes it "smart" – it explores toward the goal first.

3.2.2 The Heuristic

The heuristic used here is **Euclidean distance**:

$$h(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

This is **admissible** (never overestimates) for an 8-connected grid, which guarantees optimality.

3.2.3 8-Connected Movement

The robot can move in 8 directions from any cell:

```
NW N NE
W  .  E     Cardinal moves: cost = 1.0
SW S SE     Diagonal moves: cost = sqrt(2) = 1.414
```

3.2.4 The Algorithm

```
1.  Add START to open_set with g=0, f=h(start, goal)
2.  While open_set is not empty:
      a. Pop the cell with lowest f -> call it CURRENT
      b. If CURRENT == GOAL -> reconstruct path, done!
      c. Add CURRENT to closed_set (already explored)
```

```
      d. For each of the 8 neighbors:
           - Skip if obstacle or already in closed_set
           - Compute tentative_g = g(CURRENT) + move_cost x weight
           - If tentative_g < stored g of neighbor:
               -> Update neighbor's g and parent
               -> Push neighbor into open_set
3.  If open_set is empty -> no path exists
```

3.2.5 Costmap-Aware Weighting

The planner doesn't just find the shortest path – it prefers paths that stay away from obstacles. This is done through **cost weighting**:

```
if cost_val < inscribed_threshold:      # safe zone
    weight = 1.0 + (cost_val / inscribed_threshold) * 0.5
else:                                    # danger zone
    weight = 2.0
```

- In free space (cost near 0): `weight` near `1.0` – normal cost
- Near obstacles (cost near 49): `weight` near `1.5` – slightly penalised
- High-cost zone (cost >= 50): `weight = 2.0` – strongly penalised

This makes the path "prefer" to go through open space even if it's slightly longer.

3.2.6 Path Reconstruction

When the goal is reached, the path is reconstructed by following the `parent` pointers backward from the goal to the start, then reversing:

```
GOAL -> parent -> parent -> ... -> START
Reverse -> START -> ... -> GOAL
```

3.2.7 Safety Bail-Out

The search is capped at `max_iterations = 500,000` to prevent infinite loops on very large or unsolvable maps.

4 Path Smoothing (Gradient Descent)

4.1 Why Smooth?

The A* path follows grid cells, so it looks like a staircase:

```
Before smoothing:          After smoothing:
  .--.                       .
  |  .--.                     \
     |  .--.                    \
        |  .                      .
```

The `smooth_path()` function removes this jaggedness.

4.2 How It Works

The smoother uses an **iterative averaging** technique (a form of gradient descent). For each interior point (not start or goal), it pulls the point toward the midpoint of its two neighbours:

$$p_i^{\text{new}} = p_i + w_s \cdot \left(\frac{p_{i-1} + p_{i+1}}{2} - p_i \right)$$

Where:

- p_i is the current position of point i
- p_{i-1}, p_{i+1} are its neighbours
- w_s is the smoothing weight (default: 0.5)

This is applied to both the x and y dimensions independently.

4.2.1 Convergence

The process repeats until either:

- The maximum change in any point drops below `tolerance` (0.001 m) – **converged**
- `max_iterations` (100) is reached

4.2.2 Important Property

The **start and goal points are never moved** – only interior points are adjusted. This ensures the path still begins and ends exactly where intended.

5 Douglas-Peucker Path Simplification

5.1 Why Simplify?

Even after smoothing, the path may contain hundreds or thousands of closely-spaced points. A robot controller doesn't need (or want) that many – it needs a small number of **key waypoints** that capture the shape of the path.

5.2 The Douglas-Peucker Algorithm

The **Ramer-Douglas-Peucker** algorithm is a classic line simplification technique. It works by recursively removing points that don't contribute significantly to the path's shape.

5.2.1 How It Works (Visual Explanation)

```
Step 1: Draw a line from START to END
        Find the point FARTHEST from that line

        START .---------------------. END
                        ^
                  point P (distance d)

Step 2: If d > epsilon (threshold):
        -> KEEP point P
        -> Recurse on [START -> P] and [P -> END]

        If d <= epsilon:
        -> DISCARD all points between START and END
```

5.2.2 Step-by-Step Example

Given a path with 8 points and $\varepsilon = 0.15$ m:

```
Original:  A . . B . . . C . . D . . E
                  ^               ^
              far from line    far from line

Pass 1:    A ------- B -------- E        (B and E kept)
Pass 2:    A -- B -- C -- D -- E          (C, D checked)
Result:    A -- B -- D -- E               (C removed, too close to line B-D)
```

5.2.3 The Perpendicular Distance

The key measurement is the **perpendicular distance** from a point to the line segment between two reference points:

$$d = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

In the code, this is computed by the `_perpendicular_distance()` function, which projects the point onto the line segment and computes the distance to the projection.

5.2.4 Parameters

Parameter	Default	Effect
`epsilon`	0.15 m	Distance threshold. Smaller = more waypoints (more detail). Larger = fewer waypoints (more simplified).
`min_points`	3	Minimum waypoints to keep (start + at least 1 interior + goal)

5.2.5 Implementation Note

The code uses an **iterative stack-based** approach instead of recursion to avoid Python's recursion depth limit on very long paths.

5.2.6 Minimum Waypoints Guarantee

If Douglas-Peucker reduces the path below `min_points`, the algorithm falls back to **uniform sub-sampling** – picking evenly-spaced points along the original path.

6 The ROS 2 Node: AStarGlobalPlanner

6.1 Node Overview

The `AStarGlobalPlanner` class is a `rclpy.Node` that ties everything together. It handles ROS communication, parameters, and orchestrates the planning pipeline.

6.2 Parameters

All parameters can be set via a YAML file or command line:

Parameter	Default	Description
`map_topic`	`/map`	Topic to subscribe for the occupancy grid
`plan_topic`	`/plan`	Topic to publish the full path
`goal_topic`	`/goal_pose`	Topic to receive goal poses
`use_costmap_weights`	`True`	Enable cost-aware path planning
`do_smooth_path`	`True`	Enable gradient-descent smoothing
`do_simplify_path`	`True`	Enable Douglas-Peucker simplification
`simplify_epsilon`	`0.15`	Douglas-Peucker distance threshold (metres)
`simplify_min_points`	`3`	Minimum waypoints to retain
`lethal_threshold`	`90`	Cells >= this are impassable
`inscribed_threshold`	`50`	Cells >= this are high-cost
`robot_radius`	`0.3`	Obstacle inflation radius (metres)

6.3 Topics

6.3.1 Subscriptions

Topic	Type	QoS	Purpose
`/map`	`nav_msgs/OccupancyGrid`	Transient Local, Reliable	Receive the map
`/goal_pose`	`geometry_msgs/PoseStamped`	Default (depth 10)	Receive navigation goals

6.3.2 Publications

Topic	Type	Purpose
`/plan`	`nav_msgs/Path`	Full planned path (all waypoints)
`/waypoints`	`geometry_msgs/PoseStamped`	Individual waypoints (one message per waypoint)

6.4 TF2 Integration

The node uses **TF2** to determine the robot's current position:

```
transform = tf_buffer.lookup_transform('map', 'base_link', Time())
```

This looks up where `base_link` (the robot) is in the `map` frame. The transform tree typically looks like:

```
map -> odom -> base_link
```

6.5 The Planning Pipeline (compute_path)

When a goal is received on `/goal_pose`, the following sequence executes:

```
1. Get robot pose from TF (map -> base_link)
2. Convert start and goal from world -> map coordinates
3. Run A* search on the grid
4. Convert result back to world coordinates
5. [Optional] Smooth the path (gradient descent)
6. [Optional] Simplify with Douglas-Peucker
7. Publish waypoints individually to /waypoints
8. Build and publish Path message to /plan
```

6.6 Orientation Computation

Each waypoint includes an **orientation** (which direction the robot should face). This is computed from the heading between consecutive points:

$$\theta = \text{atan2}(\Delta y, \Delta x)$$

The yaw angle θ is then converted to a quaternion (rotation around Z-axis only):

$$q_z = \sin(\theta/2), \quad q_w = \cos(\theta/2)$$

7 Waypoint Publishing

7.1 How /waypoints Works

After Douglas-Peucker simplification, the node publishes each waypoint as a **separate `PoseStamped` message** on the `/waypoints` topic. This is useful for:

- Waypoint-following controllers that consume one waypoint at a time
- Visualisation in RViz (add a `PoseStamped` display on `/waypoints`)
- External nodes that need to know the key turning points

Each message contains:

- **Header**: timestamp and frame ID (`map`)
- **Position**: (x, y, 0.0) in world coordinates
- **Orientation**: quaternion facing toward the next waypoint

7.2 Listening to /waypoints

To echo waypoints from the command line:

```
ros2 topic echo /waypoints
```

To see how many waypoints were published:

```
ros2 topic hz /waypoints
```

8 Running the Node

8.1 Prerequisites

- ROS 2 (Humble or later)
- A map source (SLAM node or map_server publishing to `/map`)
- A TF tree providing `map -> base_link`

8.2 Launch

```
# Build the workspace
cd ~/ros2_ws
colcon build --packages-select astar_global_planner
source install/setup.bash

# Run the node
ros2 run astar_global_planner astar_global_planner
```

8.3 Sending a Goal

8.3.1 From RViz

Click **"2D Goal Pose"** in RViz – this publishes to `/goal_pose`.

8.3.2 From the Command Line

```
ros2 topic pub --once /goal_pose geometry_msgs/msg/PoseStamped \
  "{header: {frame_id: 'map'}, pose: {position: {x: 2.0, y: 3.0, z: 0.0}, \
  orientation: {w: 1.0}}}"
```

8.4 Tuning Tips

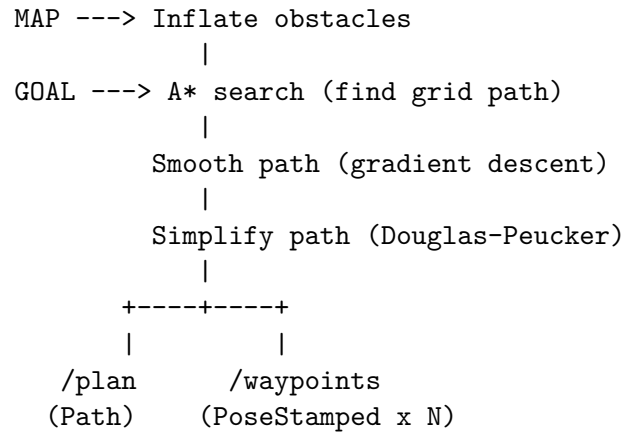
Situation	Adjustment
Path hugs obstacles too closely	Increase `robot_radius`
Path takes unnecessary detours	Decrease `robot_radius` or disable `use_costmap_weights`
Path has too many waypoints	Increase `simplify_epsilon` (e.g. 0.3)
Path cuts corners too aggressively	Decrease `simplify_epsilon` (e.g. 0.05)
Path is too jagged	Ensure `do_smooth_path` is `True`
Planning is too slow	The map might be very large; check grid dimensions

9 Glossary

Term	Definition
A*	A graph search algorithm that finds the shortest path using a heuristic
Admissible heuristic	A heuristic that never overestimates the true cost – guarantees optimal paths
BFS	Breadth-First Search – explores cells layer by layer from a source
Costmap	A grid where each cell has a traversal cost (0 = free, 100 = blocked)
Douglas-Peucker	A line simplification algorithm that removes unnecessary points
Gradient descent	An optimisation technique that iteratively moves toward a minimum
Inflation	Expanding obstacles outward by the robot's radius for safety
Lethal cell	A grid cell occupied by an obstacle (cost >= 90)
Occupancy Grid	A 2D map where each cell indicates if it is free, occupied, or unknown
PoseStamped	A ROS message containing position + orientation + timestamp
Quaternion	A 4-component representation of 3D rotation (x, y, z, w)
TF2	The ROS 2 transform library for tracking coordinate frames
Waypoint	A key point along the path where the robot should navigate through

10 Summary

The A* Global Planner pipeline:



Each stage has a clear purpose:

1. **Inflation** -> safety margin around obstacles
2. **A*** -> optimal, collision-free path on the grid
3. **Smoothing** -> removes grid-stepping artefacts
4. **Douglas-Peucker** -> reduces to essential waypoints
5. **Publishing** -> delivers the path and waypoints to downstream nodes