Project 1 Report - EE4308

Designing Nav2 Controller and Planner Plugins

A0315270N
A0315021B
A0314731M

March 3, 2025

Contents

1 Introduction

Goal of this project was to let a turtlebot navigate along multiple waypoints in an environment with obstacles. To achieve this, we designed Nav2 controller and planner plugins. We aimed to improve the pure pursuit controller from lab 1 using various heuristics. Using this controller, a robot should follow a path found by an A* planner and smoothened by a Savitsky-Golay algorithm in both a virtual and a physical environment.

2 Background and Related Work

In this section we discuss the relevant existing algorithms and their limitations.

2.1 Regulated Pure Pursuit Controller

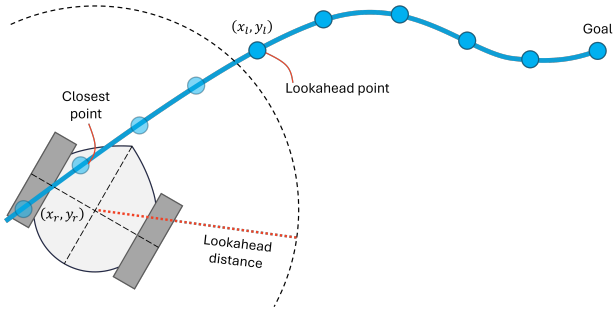


Figure 1: Pure Pursuit

Pure pursuit is a commonly used path tracking algorithm that calculates the needed angular velocity ω for a robot to reach a certain lookahead point on the desired path from the robots current position. In its most basic form the linear velocity v is kept a constant.

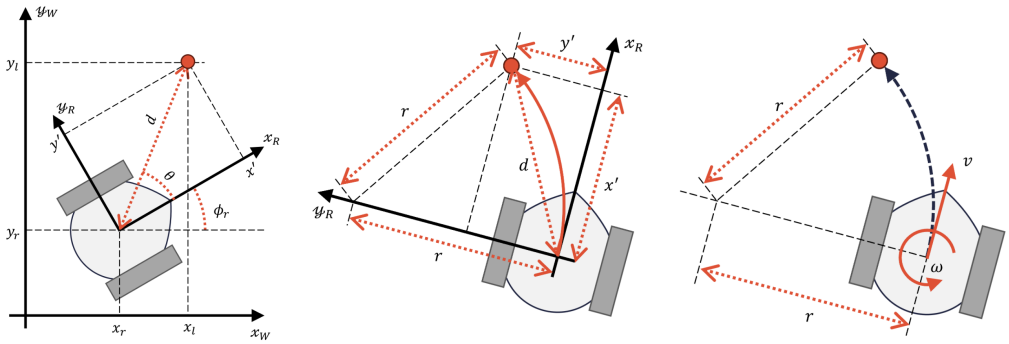


Figure 2: Pure Pursuit; Geometry

At first we need to find the lookahead point in the robot frame. Let (x_l, y_l) be the coordinates of the lookahead point in the world frame and (x_r, y_r) be the coordinates of the robot in the world frame. The coordinates of the lookahead point in the robot frame (x', y') can be found as follows:

$$\Delta x = x_l - x_r \qquad x' = \Delta x \cos(\phi_r) + \Delta y \sin(\phi_r)$$

$$\Delta y = y_l - y_r \qquad y' = \Delta y \cos(\phi_r) - \Delta x \sin(\phi_r)$$

From the following equation the robot's needed curvature can be calculated:

$$r^2 = (x')^2 + (r - y')^2 \implies c = \frac{1}{r} = \frac{2y'}{(x')^2 + (y')^2} = \frac{2y'}{d^2}$$

The required angular velocity is then given by $\omega = vc$.

Keeping the linear velocity and the lookahead distance a constant is generally not a good idea due to multiple reasons:

- The constant lookahead distance can lead to cutting of corners, as the whole corner might be within the lookahead distance and the next point that is being approached, is already after the corner.
- Also, the constant speed poses a risk when operating close to obstacles as a reduced speed would be reasonable to increase the path tracking accuracy and reduce the risk of crashing.
- The constant speed combined with a maximal rotational speed, defines a minimal turning radius, making it impossible to follow paths with high curvature closely.

To address those weaknesses, regulated pure pursuit (RPP) introduces the following heuristics on top of pure pursuit:

Curvature heuristic **Obstacle heuristic** **Varying lookahead**

$$v_c = \begin{cases} v' \frac{c_h}{c} & \text{if } c_h < c \\ v' & \text{otherwise} \end{cases} \qquad v = \begin{cases} v_c \frac{d_o}{d_{prox}} & \text{if } d_o < d_{prox} \\ v_c & \text{otherwise} \end{cases} \qquad L_h = v g_l$$

The **curvature heuristic** considers the desired linear velocity v' and the curvature threshold c_h. If the curvature needed to reach the next lookahead point is bigger than c_h, the linear velocity will be reduced.
The **obstacle heuristic** depends on the proximity threshold d_{prox} and the distance to the next obstacle d_0. If an obstacle is closer to the robot than the threshold, the linear velocity will be reduced.
The **varying lookahead** considers the lookahead gain g_l to compute the adjusted lookahead distance L_h, which linearly depends on v. This leads to closer tracking of the path for low velocity and bigger lookahead for high velocity.
The interplay of the curvature heuristic and the obstacle heuristic with the varying lookahead lead to fast movement of the robot when the space is clear and the path has no to little curvature, but reduces the speed and increases path tracking for passages close to obstacles or with high curvature. That reduces the problem of corner cutting, leads to careful movement in close spaces, and allows following high curvature paths.

Problems and Limitations of Regulated Pure Pursuit

As with normal pure pursuit, regulated pure pursuit also causes short-cutting parts of the path with high curvature turns even so in a reduced manner compared to regular pure pursuit [1]. The reason for this behavior lies in the fact that pure pursuit algorithms determine the rotation steering by finding a point at a lookahead distance along the path and computing a curvature to reach it. For sharp corners on the path, as long as there is a lookahead distance, the actual turning point lies before the corner. This smooths out the actual path of the robot compared to the desired path, but also introduces tracking errors.

Another problem of (regulated) pure pursuit controllers is their fundamental lack of considering robot dynamics when calculating the required velocities to reach the lookahead point. [1] This problem stems from the fact that (regulated) pure pursuit is a purely geometrical tracking approach. This

lack of considering dynamics lead to tracking errors of the path.

The biggest problem of the regulated pure pursuit is the instability of the curvature heuristic, combined with a varying lookahead, as they form a positive feedback loop. When having a big curvature, the curvature heuristic will reduce the velocity. Because of this, the lookahead will be reduced, leading the robot to aim for a closer point on the path. If the robot is not exactly on the path, this results in the robot rotating towards the path. This correction leads to a high curvature and will then reduce the velocity even more and so on. This can result in the robot not facing in the path direction any more, but orthogonal to it, and lead to oscillatory behaviour, once this positive feedback loop is triggered.

Another severe issue that remains unsolved by both the curvature heuristic and the other improvements from regulated pure pursuit pathfollowing is the case, when a new path is generated, needing the robot to go opposite to the direction it is facing right now. The y' in the curvature calculation goes to zero if θ is around π as $y' = \sin(\theta)$. This results in the robot going in the wrong direction as the curvature is small and the speed is therefore not regulated.

2.2 A* Path Planner

A* is a widely used path planning algorithm that works very similar to dijkstra with the key difference that it introduces a distance heuristic. I.e. instead of searching a way to the target point in any direction (like dijkstra), A* prioritizes points that lie in the direction of the goal. This is done by using a costmap that contains a heuristical distance (resp. cost) of the points on the map that is higher for points further away from the goal and lower for points closer to the goal. The pseudocode is given by the following:

Algorithm 1 A* Path Planning

```
1: function ASTAR
2:     Initialize empty OpenList
3:     Initialize all nodes with g = ∞ and no parent
4:     Set g(start) ← 0
5:     Queue start node into OpenList
6:     while OpenList is not empty do
7:         n ← node with lowest f-cost in OpenList
8:         if n was previously expanded then
9:             continue
10:        else if n is at goal then
11:            Reconstruct path from n to start
12:            Reverse path to start → goal order
13:            Convert path from map coordinates to world coordinates
14:            Apply Savitzky-Golay smoothing
15:            return path
16:        end if
17:        Mark n as expanded
18:        for each accessible neighbour m of n do
19:            g̃ ← g(n) + (distance(n, m) × (map cost(m) + 1))
20:            if g̃ < g(m) then
21:                g(m) ← g̃
22:                Parent of m ← n
23:                Add m to OpenList with updated f-cost
24:            end if
25:        end for
26:    end while
27:    return No path found
28: end function
```

Problems and Limitations of A*

1. High Memory Usage: The algorithm uses a lot of memory as many nodes are stored. In environments that are complex or have fine discretization the memory used could grow exponentially. This is especially an issue when the computational power on the robot hardware is very limited.

2. Suboptimal Paths Due to Discretization: Due to discretization, the algorithm only finds straight and diagonal paths following the grid structure, which may be suboptimal. This problem could be solved with the use of Theta*, which allows direct line-of-sight movements. [3]

3. Suboptimality in Weighted Cost Maps: When cost maps are used as terrain maps, A* avoids high-cost areas near obstacles and tends to make longer detours. This could lead to the robot avoiding passable paths that are near obstacles. With this in mind, it is crucial to tune the cost map so that the planner can find paths through tight gaps that the robot can fit through while also avoiding impassable areas.

2.3 Savitsky-Golay Smoothing

To smoothen the path found by A* we implemented a Savitsky-Golay (SG) algorithm which locally fits a polynomial curve over a set of points on the path. It applies a kernel $\mathbf{A}_{1,:}$ to a moving window of regularly spaced points, and it can be calculated as described in the following steps:

We first need to calculate the Vandermonde matrix $\mathbf{J}$ whose elements are given by $j_{r,c} = (-m + r - 1)^{c-1}$, where the subscript r denotes the row and c the column of the corresponding element.

$$\mathbf{J} = \begin{bmatrix} 1 & -m & \cdots & (-m)^{p-1} & (-m)^p \\ 1 & -m+1 & \cdots & (-m+1)^{p-1} & (-m+1)^p \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & m-1 & \cdots & (m-1)^{p-1} & (m-1)^p \\ 1 & m & \cdots & m^{p-1} & m^p \end{bmatrix} \in \mathbb{R}^{(2m+1)\times(p+1)}$$

The kernel matrix $\mathbf{A}$ can be calculated as follows:

$$\mathbf{A} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,2m+1} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,2m+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{p+1,1} & a_{p+1,2} & \cdots & a_{p+1,2m+1} \end{bmatrix} \in \mathbb{R}^{(p+1)\times(2m+1)}$$

The convolution kernel $\mathbf{A}_{1,:}$ is acquired by extracting the first row of $\mathbf{A}$:

$$\mathbf{A}_{1,:} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,2m+1} \end{bmatrix} := \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_{2m+1} \end{bmatrix}$$

The smoothened path can be calculated as

$$\hat{p}_i = \sum_{j=-m}^{m} p_k \alpha_{j+m+1}, \qquad k = \begin{cases} 1, & \text{if } i+j < 1 \\ n, & \text{if } i+j > n \\ i+j, & \text{otherwise} \end{cases},$$

where p can be substituted either by x or y and subscript i denotes the i^{th} point on the path.

Problems and Limitations of Savitsky-Golay Smoothing

Savitsky-Golay smoothing algorithms generally exhibit two key weaknesses:

1. Near the boundaries (i.e., the start and end of the data set), there are insufficient neighboring data points to form a complete window. This lack of data causes the polynomial fitting process to become less reliable, leading to inaccuracies in the smoothened output. [2]
2. Points at the boundaries of the window have a big influence on the polynomial fit and additionally, when faced with high frequency noise, SG filters showcase poor high frequency noise suppression. [2]

Let's consider problem 1 first. As described in the mathematical description, our implementation repeats the start- and endpoint whenever the kernel calculation would require a point with index $k < 1$ (before the starting point) or $k > n$ (after the endpoint). This implementation could cause some artifacts at the boundaries of the path if there are big gradients at the start or end of the path. However, we did not notice this during testing since the data generated by A* seems to be smooth enough which is why we can move on to problem 2.

In our context, high frequency noise refers to rapid, small-scale fluctuations in the path. These kind of zigzag patterns are rather uncommon in A* outputs due to its distance heuristics which would punish the additional distance covered. Nevertheless, such a path could still occur due to a bad quality costmap, which is why we should still consider this weakness of SG smoothing algorithms. The problem stems from the fact that the SG kernel is

discontinuous at positions m and $-m$ which introduces high Fourier components at high frequencies. [2] Therefore, the high frequency noise suppression of SG smoothing is undesirably weak. On top of that, due to the notable discontinuities at the borders of the kernel, points relatively far away from the current point that is being calculated still influence the polynomial fit by quite a significant amount. One straight forward solution for this problem is to apply weights w on the kernel to make the kernel continuous at the borders of the respective interval and to reduce the influence of the data points close to the boundaries of the window on the fit. We will discuss this improvement using a Hann-square window function further in section 3.

3 Proposed Improvements

In this section we describe our simple solutions to improve the existing algorithms.

3.1 Pure Pursuit Controller Improvements

The two problems with RPP introduced last arise from using the curvature for regulating the robot's velocity. To address these, a replacement for the curvature heuristic was used:

$$v_c = v' \left(1 - \frac{\theta}{\pi}\right)$$

This heuristic regulates the linear velocity depending on the angle between the direction the robot is facing and the position of the lookahead point relative to the robot. This heuristic does not suffer from the issue that y' depends on $\sin(\theta)$ and therefore decreases for $\theta > \pi/2$. And the curvature is approaching infinity for $\theta \to \pi/2$.

A minimal lookahead distance was also added to encourage the robot to always face in the direction of the path. A maximal lookahead distance is implicitly given by the robots maximal velocity v_{max} and the lookahead gain g_l.

3.2 Savitsky-Golay Smoothing Improvements

Applying a Window Function

As described in section 2.3. we want to weigh points on the path that are closer to the point of interest more during the fit using a window function. Additionaly, we would like to make the kernel continuous at $k = -m$ and $k = m$ to achieve a much more robust frequency behaviour, i.e. better high frequency noise suppression, as a nice byproduct. We chose a Hann-square window which can be described by:

$$w(n) = \left[0.5 \cdot \left(1 - \cos\left(\frac{2\pi n}{L-1}\right)\right)\right]^2, \qquad n \in \{0, 1, \ldots, L-1\}$$

The length of the window is given by $L = 2m + 1$ which corresponds to the length of the SG kernel. After calculating the weights $w(n)$ they are multiplied elementwise to $\mathbf{A}_{1,:}$ to obtain the weighted Savitsky-Golay (WSG) kernel. In the following two plots, the difference between the kernel with SG and with WSG are shown using $m = 15$ and $p = 6$.

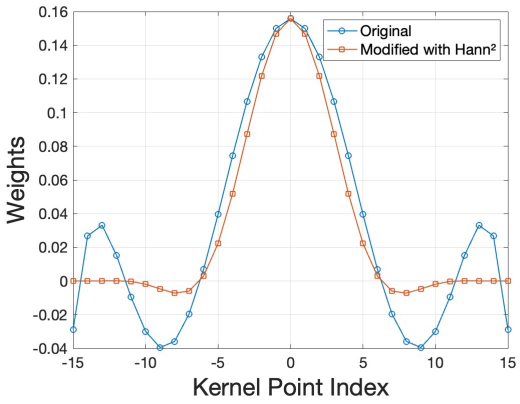


Figure 3: SG vs WSG with Hann^2

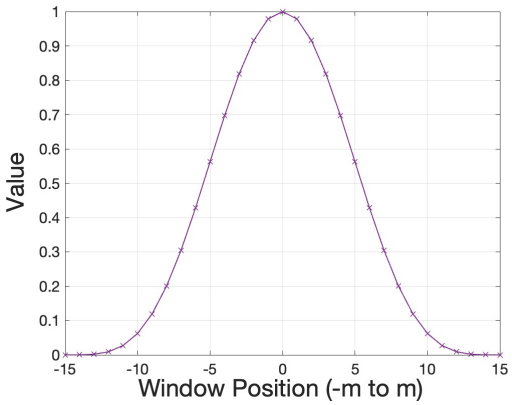


Figure 4: Hann^2 Weights

These weights are calculated in the same function as the kernel, which, as described later, is only a one-time calculation. Thus, this improvement does not cause any substantial additional computational costs.

Poorly Conditioned, Close to Singular Matrices

While experimenting with different orders and window sizes of the SG Kernel in Matlab, we often got warnings that $(\mathbf{J}^T\mathbf{J})$ is poorly conditioned and close to singular which means that calculating the direct inverse is numerically

unstable. To mitigate this issue when inverting the nearly singular matrix we calculate it via its pseudo-inverse. I.e. instead of calculating

```
Eigen::MatrixXd J_kernel = (J.transpose() * J).inverse() *
    ↪J.transpose();
```

we used

```
Eigen::MatrixXd J_kernel = (J.transpose() *
    ↪J).completeOrthogonalDecomposition().pseudoInverse()
    ↪* J.transpose();
```

Computational Costs of Matrix Operations

Another thing to consider is the computational cost of the matrix-matrix multiplications and the matrix inversion performed in the kernel calculation. At first, we implemented one function `Planner::applySavitzkyGolaySmoothing` that performed both the kernel calculation and the actual path smoothing. In this case, the kernel (including the window function) gets calculated every time the function gets called. But since the SG kernel stays constant the entire time, we split the smoothing function into two functions:

- `Planner::buildVandermondeMatrix()` which only calculates the kernel.
- `Planner::applySavitzkyGolaySmoothing(...)` that only performs the actual path smoothing.

This way, we can move the kernel calculation into `Planner::configure` which means that the `Planner::buildVandermondeMatrix()` is only a one time calculation (including calculating and applying the weights w), which greatly reduces the amount of computationally expensive matrix inversions that would have had to be performed multiple times otherwise.

4 Methodology

In this section we explain the experimental setup and parameter tuning methodologies.

4.1 Experimental Setup

Our experiments were conducted in two stages: simulation-based testing and real-world validation.

Simulation-Based Testing

In the first phase, we used the maps from Lab 1 that we created utilizing Gazebo and RViz to simulate SLAM and navigation in a controlled virtual environment. Briefly summarized, we mapped a simulated world using Cartographer and teleoperated the virtual TurtleBot to generate an occupancy grid in Lab 1. This map was then used in Project 1 to evaluate and fine-tune our navigation algorithms, including our regulated pure-pursuit controller, A* path planner and our smoothing algorithm. By iterating within the simulation, we adjusted parameters and verified the controller's ability to follow planned paths efficiently.

Real-World Testing

Once the algorithm was optimized in simulation, we deployed it on a **physical TurtleBot** in an actual lab environment, which featured a **different map** from the virtual one. This phase allowed us to evaluate the robustness of our implementation in real-world conditions, including sensor noise, wheel slip, and environmental variations. By comparing results between simulation and reality, we assessed the effectiveness and real-world performance of our algorithms.

4.2 Define Performance Metrics

To tune our parameters efficiently during the simulation-based testing, we had to define the metrics that determine the performance of our algorithms. These include:

- Avoid collisions with obstacles: This is the number one criterion for our algorithms.
- Path following accuracy: Keep lateral error from the path minimal.
- Smoothness: Avoid excessive oscillations.
- Responsiveness: Avoid cutting sharp corners too much.
- Convergence behavior: The robot does not take too much time to stabilize after disturbances.
- Speed regulation: Maintaining smooth velocity transitions and trying to increase speed when all the other conditions are met.

These goals were evaluated by looking at runs and comparing them.

4.3 Regulated Pure Pursuit Parameter Tuning

As a first step, we wanted to achieve a robust RPP controller, which means that we had to tune the following parameters: The lookahead gain, the min lookahead distance, the desired linear velocity and the proximity threshold. The curvature threshold was not tuned, as the improved curvature heuristic does not rely on a tuned parameter. The maximal allowed linear and angular velocities and the maximal allowed xy-threshold were fixed and not allowed for tuning by the task description due to limitations of the robot hardware. Our tuning methodology was as follows:

1. **Initial Values**

   We start with a moderate lookahead gain of ≈ 1 and a minimum lookahead distance of just above zero meters. For the proximity threshold, a value of 0.5m was used as a starting point.

2. **Parameter Tuning Process**

   Those parameters were tuned evaluating the runs and seeking improvements following these rules:

   (a) Lookahead Distance Parameters (g_l, l_{min})

   Increase g_l if the vehicle oscillates or reacts too aggressively to small deviations and decrease g_l if the vehicle shortcuts turns or struggles in tight curves.

   Adjust the min lookahead distance to handle the trade-off between responsiveness and path tracking vs. smoothness and stability. The smaller the lookahead distance, the closer the robot follows the path. If it is too small, the robot starts to oscillate.

   (b) Desired Linear Velocity (v')

   Start low (approx. 50% of max speed) and gradually increase, ensuring robustness.

   (c) Proximity Threshold (d_{prox})

   If d_{prox} is too high, the robot slows down too much even when far away from obstacles. If d_{prox} is too low, the risk of collision due to bad path tracking close to obstacles is too high.

A set of parameters was evaluated for various cases, such as straight lines, curved paths around obstacles, and edge cases such as placing the goal behind the robot or close beside the robot.

The final parameters ended up being $l_g = 2.5$, $l_{min} = 0.1$, $v' = 0.2$ and $d_{prox} = 0.3$.

4.4 A* Planner Parameter Tuning

As the cost map of our environment was already given, there wasn't much to tune for the A* planner. But using smaller costs near obstacles could have been beneficial, as it would have allowed the robot to navigate through narrower spaces and move more closely along obstacles. The choice of the Euclidean heuristic over the Manhattan heuristic was made because the robot moves in a continuous space and is capable of moving diagonally.

4.5 Savitsky-Golay Smoothing Parameter Tuning

Our smoothing algorithm only has two parameters that require tuning: The half-window size m and the polynomial order p. The goal is to reduce noise or sharp discontinuities in the path while ensuring the vehicle follows a feasible and accurate trajectory. We used the following steps for tuning:

1. **Understand Parameter Effects**

   Half-window size m: A larger m leads to a smoother path, but it can cause shortcuts or substantial deviations from the intended route. A smaller m means the robot follows the path more precisely, but unnecessary sharp turns or noise may be retained.

   Polynomial order p: A higher p preserves natural curves in the path better, reducing distortions at corners. A lower p causes stronger smoothing but may oversimplify complex paths.

   Constraint: $p \leq 2m$ to ensure the fit is well-defined because we need at least $p + 1$ points to fit a polynomial of order p. Otherwise the fit is not uniquely determineable.

2. **Start With an Initial Safe Baseline**

   We started with $p = 2$ (quadratic) for general smoothing and with $m = 5$ (a pretty small window).

3. **The Actual Parameter Tuning Process**

   At first we fixed p and tuned m. We start with a small m and then gradually increased it. To check if m is large enough, we observed if sharp turns are sufficiently smoothed. Then we chose the smallest m for which we could observe enough smoothing. For us, $m = 20$ seemed to give nice and smooth results.

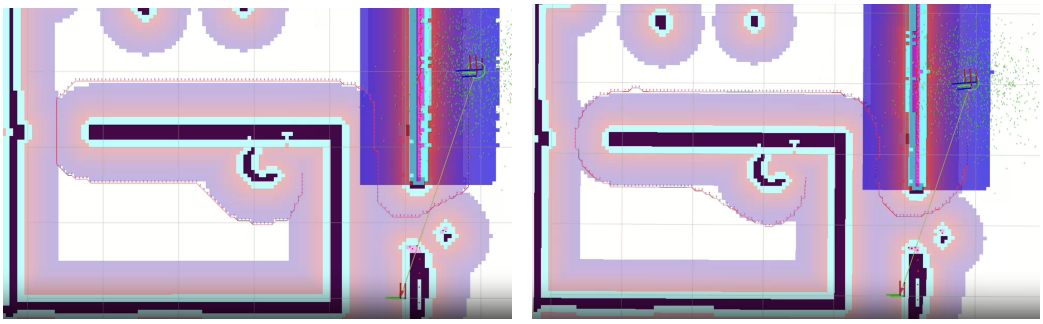
   As a second step we tune p for the shape preservation of the path. If oscillations appear in the path, we should decrease p. $p = 2$ is a safe starting point but we increased it to 3 to handle smooth curves better.

5 Results and Discussion

Our implementation of the A* planner and the regulated pure pursuit controller was successful both in simulation and in physical testing. The system was able to plan and execute paths to target locations with a high reliability.

5.1 Simulation-Testing

In simulation, the A* planner and Pure Pursuit controller enabled the robot to navigate effectively through a predefined environment. To evaluate the impact of the path smoother, we conducted five runs from a fixed starting position to a fixed goal, once using smoother and once without it. The paths for both scenarios can be seen in Figure 5.



(a) Path without smoothing (b) Path with smoothing

Figure 5: Comparison of paths with and without smoothing

As shown in Table 1, the mean execution time with smoothing was 72.72s, compared to 75.01s without smoothing. This is an improvement of approximately 2.3 seconds. Also, the standard deviation was lower (0.27 vs. 0.44), showing a more consistent performance when using the smoother. The improvement is due to reduced sharp turns and fewer abrupt slowdowns from the controller.

With this a more continuous and efficient motion is achieved. Without the smoother, the robot followed a more rigid A* path with abrupt changes in direction. As the controller also slows down the robot in sharp corners, some of the time loss can be attributed to these tighter turns.

5.2 Physical Testing

The system was successfully set up and tested on a real robot in the physical world. The robot was able to independently travel to three different locations, as planned by the project supervisors. This proved that the planner works well in real-life situations and is reliable outside of simulations.

	With Smoothing	Without Smoothing
Trial 1	72.56	75.38
Trial 2	72.73	74.36
Trial 3	72.34	74.62
Trial 4	73.13	75.50
Trial 5	72.86	75.20
Mean	72.72	75.01
Standard Deviation	0.27	0.44

Table 1: Comparison of execution times (in seconds) with and without smoothing

6 Conclusion and Recommendations

In this project, we successfully designed and implemented a Nav2 controller and planner plugins to enable TurtleBot to navigate through an obstacle-filled environment. To do so, we used regulated pure pursuit with a small enhancement to follow a path generated by A* that is refined by an improved version of Savitzky-Golay smoothing.

The RPP algorithm introduces significant improvements over the pure pursuit but introduces instability due to the interplay of curvature heuristic and adjusted lookahead distance. It also struggles to track paths that head into the opposite direction as the robot is facing. This was mitigated by modifying the curvature heuristic to depend on the robot's heading instead of the curvature.

The Savitsky-Golay Smoothing improves the path's smoothness but faced challenges for computational stability and had issues with calculations at the boundaries of the path. By using a more stable way to compute the pseudoinverse of the Vandermonde matrix and introducing a weighted window function, those issues were handled.

A fine-tuned cost map would have been beneficial, allowing the planner to find shorter paths by moving closer to obstacles, as the robot still had plenty of space.

The algorithm worked well in simulation where it was developed. It was transfered first try onto real hardware in a different environment. This demonstrates stability against sensor noise, against wheel slip and other non-idealities.

Overall, the project was successful while leaving room for further exploration and more experiments to find the planner's limitations.

References

[1] Steve Macenski*, Shrijit Singh, Francisco Martín, Jonatan Ginés, "Regulated Pure Pursuit for Robot Path Tracking," arXiv, 2023.

[2] Michael Schmid*, David Rath, Ulrike Diebold, "Why and How Savitzky–Golay Filters Should Be Replaced," ACS Publications, 2022.

[3] Daniel S. Koenig, Lihong Shao, Maxim Likhachev, Sven Koenig, "Theta*: Any-Angle Path Planning on Grids," *Journal of Artificial Intelligence Research*, vol. 33, pp. 869–877, 2008. https://doi.org/10.1613/jair.2994.