10xConstruction Take Home Assignment

Robotics Software Role

Name: Pranav P

Gmail: pranv0723@gmail.com

1.0 Objective

Implement a path smoothing algorithm and a trajectory tracking controller for a differential drive robot navigating through a series of 2D waypoints. The goal is to generate a smooth trajectory from discrete waypoints and ensure the robot follows this trajectory accurately.

Index

2. Task1

Path Smoothing Given a list of 2D waypoints: waypoints = [(x0, y0), (x1, y1), ..., (xn, yn)]

Implement a smoothing algorithm to generate a smooth trajectory

Deliverables: A function that takes discrete waypoints and returns a smooth, continuous path.

2.1 Introduction

In many real-world applications of robotics and autonomous systems, a robot's motion is guided by a set of discrete waypoints that define its desired path. However, directly following these waypoints often leads to abrupt changes in direction or velocity, which are undesirable for smooth and stable motion. To address this, the task of **path smoothing** aims to transform a sequence of discrete 2D waypoints into a continuous and differentiable trajectory that a robot can follow seamlessly.

The objective of this task is to design and implement a **smoothing algorithm** that takes a list of 2D waypoints—each representing a position in space—and outputs a smooth, continuous path passing through or near them. The resulting trajectory should preserve the overall shape of the original path while eliminating sharp turns and discontinuities.

This process not only enhances the physical feasibility of motion (by ensuring smooth velocity and acceleration profiles) but also improves control stability and efficiency during trajectory tracking. Through this task, I aim to demonstrate an understanding of interpolation techniques, such as cubic splines or other parametric representations, which are commonly used to achieve path continuity and smoothness in robotic navigation and motion planning.

2.2 Interpolation Techniques for Path Smoothing

To generate a smooth trajectory from discrete waypoints, interpolation methods are employed to create a continuous mathematical representation of the path. Among the many available techniques, three common ones are **Catmull–Rom splines**, **Natural Cubic Splines**, and **B-splines**. Each method has distinct characteristics that influence how a robot or autonomous vehicle behaves while following the resulting path.

---

1. Catmull–Rom Splines

Catmull–Rom splines are **interpolating curves** that pass directly through all given waypoints. They are defined locally, meaning each segment of the curve depends only on a few nearby points. This makes them computationally efficient and easy to implement in real-time robotic systems.

Advantages:

- The curve naturally passes through all the waypoints (no need for manual fitting).
- Local control — modifying one waypoint affects only the nearby curve segments.
- Smooth C^1continuity (continuous first derivative), which ensures no sudden direction changes.

Disadvantages:

- Not C^2continuous — acceleration is not smooth, which can cause jerky motion or sudden velocity shifts for a robot.
- At sharp turns, overshooting can occur, leading to unrealistic or unsafe paths.

If a robot follows a Catmull–Rom spline path, it would move smoothly between points but may experience noticeable jerks in acceleration at junctions, especially during fast movements or when navigating tight turns.

---

2. Natural Cubic Splines

Natural cubic splines are piecewise cubic polynomials that ensure both **first and second derivatives are continuous** across segments, giving them C^2continuity. This means the path, velocity, and acceleration all change smoothly, which is ideal for stable robotic motion.

A physical analogy can be drawn from the **bending of a metal scale**:
If you fix a flexible metal strip at several points (representing waypoints), it bends naturally to minimize the **bending energy**, forming a smooth curve with no abrupt kinks. The natural cubic spline mathematically replicates this behavior by minimizing the integral of the square of the second derivative — essentially reducing curvature and sudden changes in direction.

Advantages:

- Guarantees smooth position, velocity, and acceleration (C^2 continuity).
- Produces natural-looking paths with minimal curvature.
- Excellent for dynamic systems where acceleration smoothness directly affects control stability.

Disadvantages:

- Slight computational overhead compared to Catmull–Rom splines.

Global influence(can be pro and con) — Adjusting one waypoint slightly modifies the entire curve, which can be both an advantage and a disadvantage. It's advantageous when a globally smooth trajectory is desired, as it ensures continuity across all segments and prevents abrupt transitions between local portions of the path. However, it can be a limitation when fine-tuning a specific section of the trajectory without affecting others.

For our bot, which needs to move steadily and maintain smooth turning without jerks, the natural cubic spline offers the best trade-off. The minimized bending energy results in gentle transitions, which reduces wheel slippage and ensures better controller performance.

---

3. B-Splines

B-splines (Basis Splines) are **non-interpolating** curves — they approximate the path rather than passing through each waypoint. They use a set of control points to define the curve, providing a high degree of flexibility.

Advantages:

- Strong local control — changing one control point affects only a local portion of the curve.
- High stability and smoothness (can achieve C^2or even higher continuity).
- Robust for complex shapes and widely used in computer graphics and CAD.

Disadvantages:

- The curve does not necessarily pass through the waypoints, requiring additional logic to ensure the robot reaches target locations.
- Slightly harder to tune and visualize for path-following applications.

For a mobile robot, B-splines can generate beautifully smooth trajectories, but since the path does not exactly go through waypoints, the bot might miss certain target locations unless additional tracking corrections are implemented.

Why Natural Cubic Polynomial Works Best in Our Case

In our task, the primary goal is to generate a **smooth trajectory that passes through all waypoints** while maintaining continuity in both velocity and acceleration. Since our bot's motion system requires consistent turning without sudden torque changes, **Natural Cubic Splines** provide the most suitable solution.
They balance smoothness and precision — ensuring the bot follows the intended waypoints while maintaining stable dynamics. The physical analogy of minimizing bending energy mirrors how we desire our trajectory to "bend" naturally, avoiding sharp corners or abrupt motions.

Interpolation Method	*Passes Through Waypoints*	*(C^1) Continuity*	*(C^2) Continuity*	*Local Movement*
Catmull–Rom Spline	Yes	Yes	No	Yes
Natural Cubic Spline	Yes	Yes	Yes	No (Global)
B-Spline	(Approximate)	Yes	Yes	Yes

Key Parameters for Trajectory Smoothness in Mobile Robots

C1 Continuity:
Ensures the first derivative of the path is continuous, meaning the robot's direction changes smoothly without sharp corners. This prevents sudden heading changes and allows the robot to maintain a steady velocity through waypoints.

C2 Continuity:
Ensures the second derivative of the path is continuous, providing smooth changes in acceleration and curvature. This helps the robot move without jerks, improving stability and control precision.

Local Movement:
Refers to how a change in one waypoint affects the curve. With strong local movement, only nearby sections of the path adjust, enabling quick, localized corrections without altering the entire trajectory.

2.3 Methodology

We want to find a curve $s(x)$that passes through all the given points but bends as little as possible. To do this, we minimize the total "bending energy," which is the integral of the square of the second derivative $(s''(x))^2$. When we apply calculus to find the smoothest possible shape, it turns out that the curve between any two points must be a **cubic polynomial**. That's why natural cubic splines are made up of smooth cubic pieces joined together. The condition $s''(x_0) = s''(x_n) = 0$simply means the curve's ends are free to relax — just like the ends of a metal scale that aren't being bent. This process naturally leads to the equations we solve to find the spline's shape.

Algorithmic steps (implementation recipe)

1. Inputs: knot vectors $\{x_i\}$and data $\{y_i\}$. Compute $h_i = x_{i+1} - x_i$.
2. Build the right-hand side vector $r_i = 6(\frac{y_{i+1}-y_i}{h_i} - \frac{y_i-y_{i-1}}{h_{i-1}})$for $i = 1, \dots, n-1$.
3. Build tridiagonal matrix $A$with diagonal entries $2(h_{i-1} + h_i)$and off-diagonals $h_{i-1}$and h_i.
4. Solve $A\mathbf{M} = \mathbf{r}$for $\mathbf{M} = [M_1, \dots, M_{n-1}]^\top$(use Thomas algorithm — $O(n)$for tridiagonal systems).
5. Set $M_0 = M_n = 0$.
6. Compute coefficients $a_i, b_i, c_i, d_i$for each segment using the formulas above.
7. Evaluate $S(x)$as needed for plotting or to output parametric points for your robot controller.

2.4 Derivation

Date :

Natural Cubic Spline

Here, in general polynomial eq^n we use $(x - x_i)$ instead of just x so that each segment of the curve starts counting from its own starting point. This makes the math simpler & keeps the equations easier to handle for each small section of the path.

$S_i(x_i)$ → refers to the polynomial in i^{th} segment. It ranges from (x_i, x_{i+1})

Given : $[(x_1, y_1), (x_2, y_2) \cdots, (x_n, y_n)]$ data points

Derivation :

$$y_i = S_i(x_i) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad - (1)$$

$$S_i'(x_i) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \quad - (2)$$

$$S_i''(x_i) = 2c_i + 6d_i(x - x_i) \quad - (3)$$

In natural cubic spline, each segment has its own unique polynomial with unique set of coefficient values of (a, b, c, d).

But they maintain smoothness in the curve, by ensuring both first and second derivatives are continuous across segments, giving them C^2 continuity.

Date :

An analogy to intuitively understand this algo :

bending of a metal scale -

if we fix a flexible metal strip at several points, it bends naturally to minimize bending energy. By using this example we can say

$$S''_0(x_0) = 0 \quad \& \quad S''_{n-1}(x_n) = 0$$

because before the first point & after the last point there is no bending.

Now, let $M_i = S''_i(x_i)$, $M_{i+1} = S''_i(x_{i+1})$

by sub $x = x_i$ in ③

$$M_i = S''_i(x_i) = 2C_i$$

$$\therefore \quad C_i = \frac{M_i}{2} \quad — ④$$

by sub $x = x_{i+1}$ in ③

$$M_{i+1} = S_i(x_{i+1}) = 2C_i + 6d_i(x_{i+1} - x_i)$$

let $h_i = x_{i+1} - x_i$ //

$$M_{i+1} = M_i + 6d_i \cdot h_i$$

$$d_i = \frac{M_{i+1} - M_i}{6h_i} \quad — ⑤$$

Date :

Sub $x = x_i$ in ①

$y_i = S_i(x_i) = a_i$

$\therefore a_i = y_i$ — ⑥

Sub $x = x_{i+1}$ in ①

$y_{i+1} = S_i(x_{i+1}) = y_i + b_i \cdot h_i + c_i h_i^2 + d_i h_i^3$

$b_i = \frac{y_{i+1} - y_i}{h_i} - c_i h_i - d_i h_i^2$

Sub $a_i = \frac{M_i}{2}$ & $d_i = \frac{M_{i+1} - M_i}{6h_i}$

$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{M_i h_i}{2} - \frac{(M_{i+1} - M_i) \cdot h_i}{6}$

$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6}(2M_i + M_{i+1})$ — ⑦

Now we have the values of

Sub a_i, b_i, c_i, d_i in ①

$S_i(x) = M_i \frac{(x_{i+1} - x)^3}{6h_i} + M_{i+1} \frac{(x - x_i)^3}{6h_i} + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6}\right)(x_{i+1} - x)$

$+ \left(\frac{y_{i+1}}{h_i} - \frac{M_{i+1} \cdot h_i}{6}\right)(x - x_i)$ — ⑧

$h_i = x_{i+1} - x_i$

$M_i = S_i''(x_i)$, $M_{i+1} = S_i''(x_{i+1})$

Date :

To compute M_i

differentiating ④ :

$$S_i'(x) = -\frac{M_i(x_{i+1}-x)^2}{2h_i} + \frac{M_{i+1}(x-x_i)^2}{2h_i} - \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6}\right) + \left(\frac{y_{i+1}}{h_i} - \frac{M_{i+1}\cdot h_i}{6}\right) \quad -⑤$$

Evaluate $S_i'(x)$ at the left & right ends

at the left side of x_i, (interval of segment $\in [x_{i-1}, x_i]$

$$S_{i-1}'(x_i) = \frac{M_i \cdot h_{i-1}}{6} + \frac{M_{i-1} h_{i-1}}{3} + \frac{y_i - y_{i-1}}{h_{i-1}} \quad -⑥$$

at the right side of x_i (interval of segment $\in [x_i, x_{i+1}]$)

$$S_i'(x_i) = -\frac{M_i h_i}{3} - \frac{M_{i+1}\cdot h_i}{6} + \frac{y_{i+1} - y_i}{h_i} \quad -⑦$$

Now,

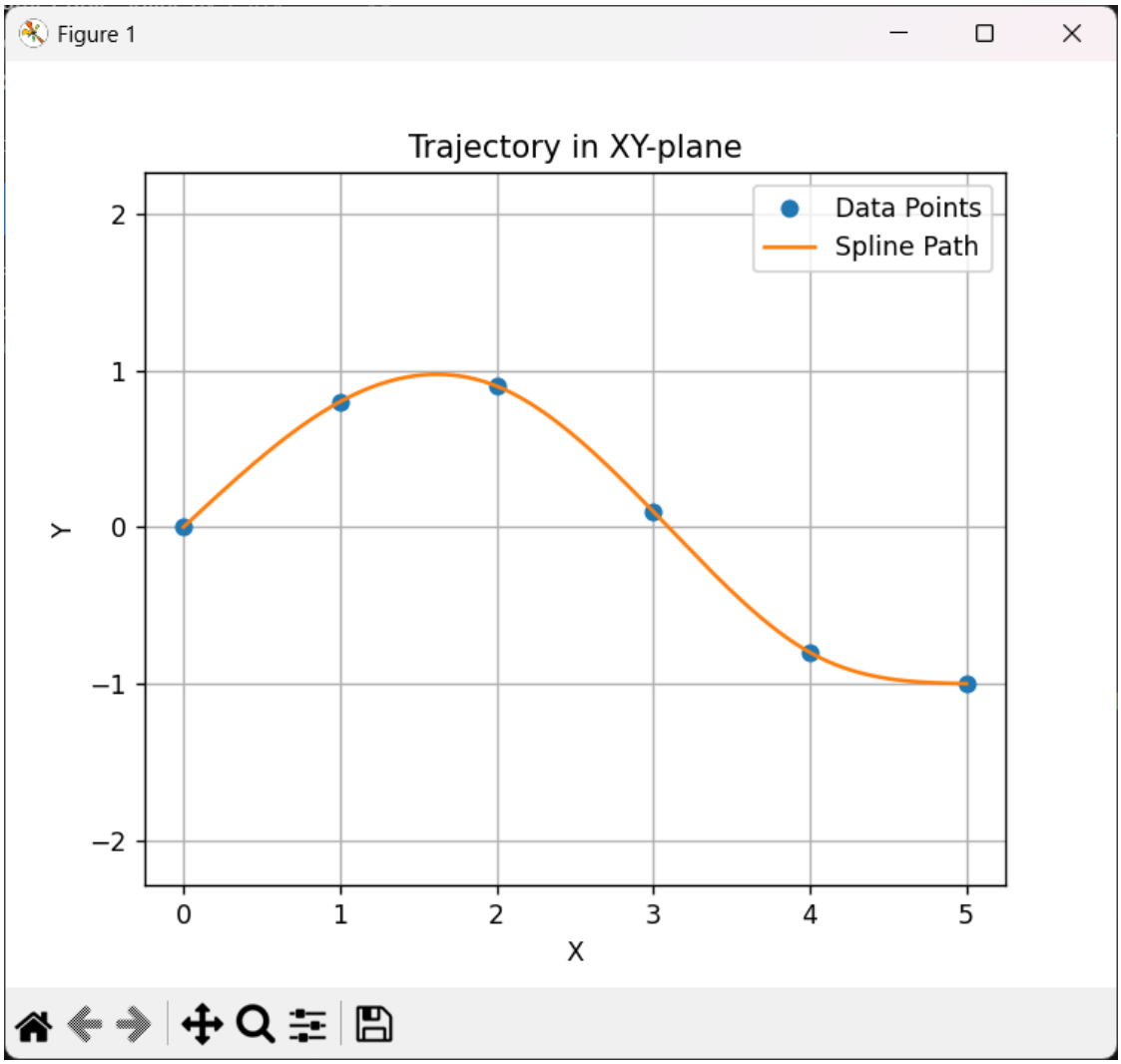
for slope continuity first derivative must be equal

$$S_{i-1}'(x_i) = S_i'(x_i)$$
$$⑥ \quad = \quad ⑦$$

$$h_{i-1}\cdot M_{i-1} + 2(h_{i-1} + h_i) M_i + h_i M_{i+1} = 6\left[\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right]$$

Also property of natural spline: $M_0 = 0$, $M_n = 0$

2.5 Result



The smoothing algorithm using the **Natural Cubic Spline** was successfully implemented to generate a continuous and smooth trajectory from a set of discrete waypoints. The resulting path maintained both $C^1$and C^2continuity, ensuring smooth transitions in direction and acceleration.

When plotted, the trajectory passed through all given waypoints with minimal curvature, resembling the natural bending of a flexible strip. The spline effectively eliminated sharp turns and discontinuities, making it well-suited for robotic motion planning. Compared to Catmull–Rom and B-splines, the natural cubic spline produced a globally smooth path that balanced accuracy (passing through points) and motion stability.

2.6 Conclusion

This task demonstrated how **Natural Cubic Splines** can be used to create physically realistic and dynamically stable paths for a mobile robot. By minimizing bending energy, the spline ensures gentle curvature changes, leading to smoother velocity and acceleration profiles — essential for precise and safe trajectory tracking.

Among the techniques studied, the natural cubic spline provided the most reliable performance for robotic applications where both path accuracy and smooth motion are critical. The approach not only improves the robot's control response but also reduces mechanical stress, making it an effective and elegant solution for path smoothing in real-world navigation.

3 Task2

Trajectory Generation From the smoothed path, generate a time-parameterized trajectory:

Sample the path at regular intervals Optionally assign velocity profiles (e.g., trapezoidal or constant velocity)

Deliverables: Time-stamped trajectory in the form: trajectory = [(x0, y0, t0), (x1, y1, t1), ..., (xn, yn, tn)]

3.1 Introduction

After obtaining discrete waypoints, the next step was to generate a smooth and continuous trajectory that the robot could follow. Instead of directly connecting the waypoints with straight lines, a **Natural Cubic Spline** interpolation was used. This method ensures smooth transitions between points by maintaining continuity in both the first and second derivatives of the path, resulting in a realistic and physically feasible motion for the robot.

The spline function takes the list of waypoints and computes intermediate points between them, forming a smooth curve that minimizes sudden direction changes. The path is then sampled at regular intervals to create a dense sequence of (x, y)points. These sampled coordinates represent the trajectory that the controller uses during tracking.

Although time parameterization or velocity profiles (like constant or trapezoidal velocity) were not implemented, the controller in Task 3 handles timing implicitly. It processes the trajectory points in real time based on the robot's motion feedback, effectively linking spatial progression with actual simulation time.

The generated trajectory was saved as a CSV file and later visualized in RViz and matplotlib to confirm smoothness and correctness. This ensured that the robot had a continuous and well-defined path to follow for the trajectory tracking stage.

3.2 Time-Stamped Trajectory Data

Time(s)	X	Y
0.00	0.000	0.000
0.20	0.147	0.136
0.40	0.295	0.270
0.60	0.446	0.402
0.80	0.600	0.529
1.00	0.760	0.649
1.20	0.928	0.759
1.40	1.105	0.853
1.60	1.292	0.924
1.80	1.487	0.967
2.00	1.687	0.973
2.20	1.883	0.939
2.40	2.070	0.869
2.60	2.244	0.771
2.80	2.406	0.655
3.00	2.560	0.528
3.20	2.707	0.393
3.40	2.850	0.253
3.60	2.990	0.110
3.80	3.129	-0.034
4.00	3.268	-0.178
4.20	3.408	-0.320
4.40	3.553	-0.459
4.60	3.703	-0.591
4.80	3.862	-0.713
5.00	4.033	-0.818
5.20	4.216	-0.898
5.40	4.409	-0.951
5.60	4.607	-0.981
5.80	4.806	-0.995
6.00	5.000	-1.000

4.0 Task 3 - Trajectory Tracking Controller

Implement a controller that makes a simulated robot follow the trajectory

Deliverables: A controller function that outputs velocity commands based on current state and trajectory and simulation showing robot tracking performance

---

4.1 Introduction

Once a smooth and continuous path has been generated, the next major challenge is ensuring that the robot can accurately follow that path in real time. This process is known as **trajectory tracking**. In this task, the objective is to design and implement a **controller** that takes the robot's current position and orientation as feedback and continuously computes the velocity commands required to keep it on the desired trajectory.

In simple terms, a **trajectory tracking controller** acts like the robot's brain while it moves. The path generated from Task 2 only provides where the robot *should* go — that is, a list of *(x,y)(x, y)*(x,y) coordinates (and optionally, time *tt*t). However, the robot doesn't naturally know how to move along that curve smoothly. The controller bridges this gap by calculating the correct **linear** and **angular velocities** so that the robot's actual movement stays as close as possible to the reference trajectory.

In this project, a **PID (Proportional–Integral–Derivative) controller** is used for trajectory tracking. The PID controller continuously measures the **difference (error)** between the robot's current pose (from the odometry sensor) and the reference trajectory point.

- The **Proportional (P)** term reacts to the current error, giving immediate correction.
- The **Integral (I)** term accumulates past errors, reducing steady-state drift.
- The **Derivative (D)** term predicts future error trends, adding stability and smoothing the response.

By tuning the PID gains, the robot can be made to follow the path with minimal oscillation and overshoot.

In this implementation, the controller runs as a **ROS2 node**, subscribing to the /odom topic to get real-time robot pose and publishing control commands (/cmd_vel) at a fixed rate. The code continuously reads the next target point from the smoothed trajectory and adjusts the velocities based on how far and in which direction the robot is from that point. Once the robot gets sufficiently close, it automatically moves on to the next target.

Overall, the trajectory tracking controller ensures that the robot does not just *know* the path but can also **move along it smoothly, accurately, and autonomously**. This step effectively connects perception (path planning) with action (motion execution), forming the core of autonomous navigation.

4.2 Methodology

1. Overview of the Control Process

The main goal of the trajectory tracking controller is to make the robot move along the smoothed trajectory generated from Task 2. To achieve this, the controller operates in a **feedback loop**. At each instant, it compares the robot's current pose — obtained from odometry data — with the corresponding reference point on the trajectory. Based on the difference (called the **error**), it computes the linear and angular velocity commands that should be sent to the robot's wheels.

The entire process repeats at fixed time intervals (in this case, every 0.05 seconds), allowing the robot to continuously correct its motion in real time.

---

2. Input and Output of the Controller

Inputs:

- Current robot pose from /odom: *(x,y,θ)*
- Reference point from trajectory: *(xr,yr)*

Output:

- Linear velocity command: v
- Angular velocity command: ω

These commands are published to the /cmd_vel topic, which drives the robot in the simulator.

---

3. Error Computation

The first step is to calculate the **position and orientation errors** between the robot and the reference trajectory point:

$$\mathrm{dx} = x_r - x, \quad \mathrm{dy} = y_r - y$$

$$\text{Distance Error: } e_d = \sqrt{(\mathrm{dx})^2 + (\mathrm{dy})^2}$$

This distance error tells how far the robot is from the desired point.

The desired heading (direction from the robot to the target) is:

$$\theta_r = \tan^{-1}\left(\frac{\mathrm{dy}}{\mathrm{dx}}\right)$$

The **angular error** is the difference between the desired and current headings:

$$e_\theta = \theta_r - \theta$$

To ensure that the robot always takes the shortest turn, the angular error is normalized using:

$$e_\theta = \mathrm{atan2}(\sin(e_\theta), \cos(e_\theta))$$

4. PID Control Law

The controller uses **two independent PID loops** — one for linear velocity and one for angular velocity.

(a) Linear Velocity Control

The linear velocity v is adjusted based on how far the robot is from the target:

$$v = K_p^{lin} \cdot e_d + K_i^{lin} \int e_d \, dt + K_d^{lin} \frac{de_d}{dt}$$

- The proportional term moves the robot forward toward the goal.
- The integral term compensates for steady-state errors (e.g., when the robot moves slightly slower than expected).
- The derivative term smooths sudden changes in error.

(b) Angular Velocity Control

The angular velocity $\omega | omega$ω aligns the robot toward the target point:

$$\omega = K_p^{ang} \cdot e_\theta + K_i^{ang} \int e_\theta \, dt + K_d^{ang} \frac{de_\theta}{dt}$$

By combining these two commands, the robot can move and rotate simultaneously, smoothly following the curve.

---

5. Gain Tuning

The PID gains *(Kp,Ki,Kd)* were tuned experimentally:

- Higher ***Kp*** increases responsiveness but may cause oscillations.
- The ***Kd*** term dampens overshoot and stabilizes turning.
- The ***Ki*** term corrects accumulated errors over time, though it is kept small to avoid slow oscillations.

For this project, typical tuned values were:

Linear: Kp = 0.5, Ki = 0.01, Kd = 0.1
Angular: Kp = 1.0, Ki = 0.0, Kd = 0.3

6. Path Advancement and Termination

The controller updates the trajectory index based on time or distance.
Once the robot comes within a small radius (e.g., 0.1 m) of the target point, it moves to the next trajectory point.
When the robot reaches the final waypoint, the velocities are set to zero, and a completion message is displayed.

---

7. ROS2 Node Operation

The entire algorithm runs inside a ROS2 node (trajectory_pid) with the following workflow:

1. **Subscription:** Receives /odom data (current pose).
2. **Computation:** Calculates ed, eθ , and the corresponding v, ω using the PID control laws.
3. **Publication:** Sends the computed velocities to /cmd_vel.
4. **Loop:** Repeats every 0.05 seconds until the path is completed.

---

8. Summary

The PID trajectory tracking controller ensures smooth and stable motion of the robot along the precomputed spline path. It effectively combines real-time feedback with continuous control, enabling the robot to follow complex curved trajectories with precision. This method is reliable, easy to tune, and well-suited for differential drive robots like the TurtleBot3.

---

9. Future Work: Obstacle Avoidance

The current system accurately follows a predefined trajectory but does not yet handle obstacles that may appear in the robot's path. As future work, a simple **LiDAR-based obstacle avoidance** mechanism can be integrated to make the robot more autonomous and reactive.

The idea is to divide the LiDAR scan into **three regions** — *front*, *left*, and *right*. The minimum distance in each region is monitored continuously. If any obstacle is detected within a safe distance (e.g., 0.5 m), the robot adjusts its motion accordingly:

- **Front obstacle:** Stop and turn toward the side with more clearance.
- **Left obstacle:** Turn slightly right.
- **Right obstacle:** Turn slightly left.
- **No obstacle:** Continue PID-based trajectory tracking.

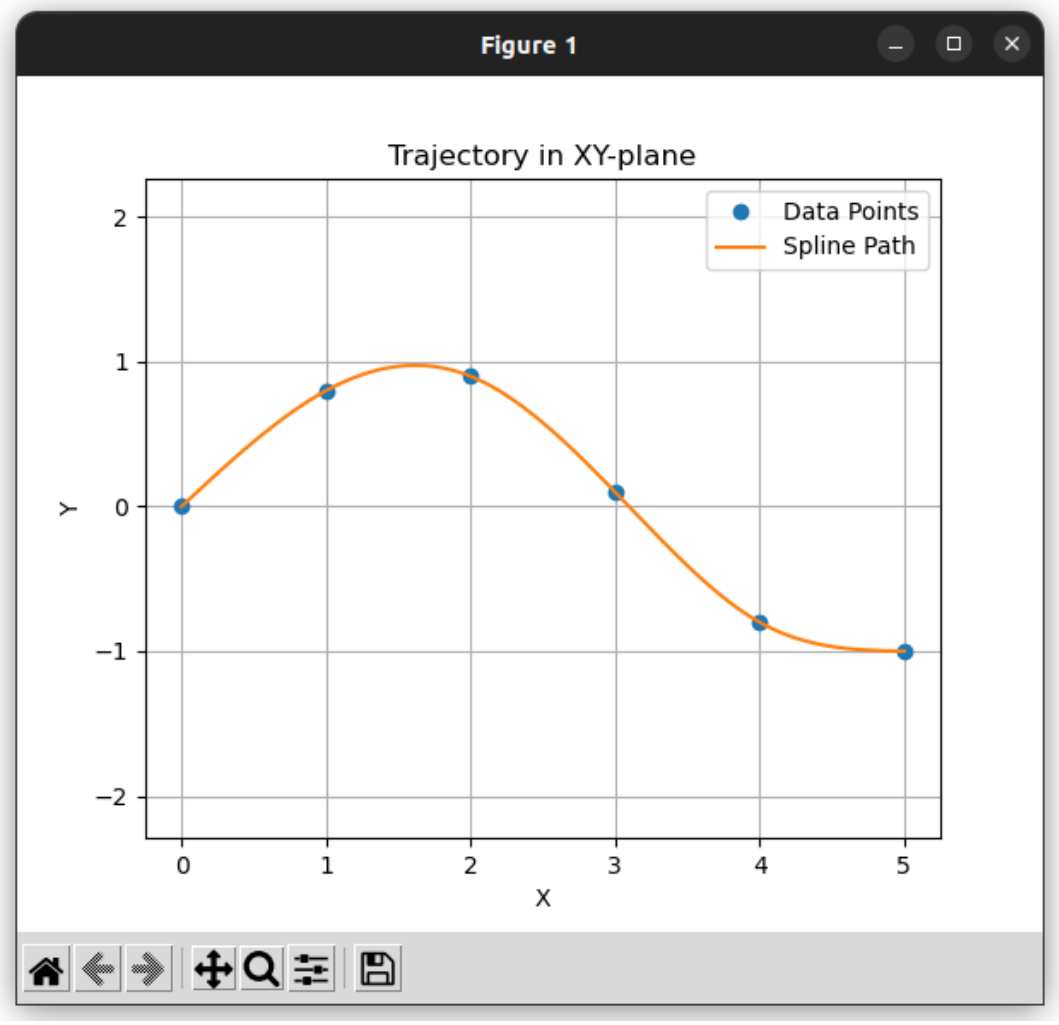
This logic allows the robot to **temporarily override** the trajectory controller when obstacles are present and resume tracking once the path is clear. It provides a simple, reactive layer of safety that can later be expanded into more advanced methods such as local re-planning or dynamic window approaches.

4.3 Results and Discussion

The trajectory tracking controller was successfully implemented and tested in the ROS2 simulation environment. The smoothed trajectory generated from the **Natural Cubic Spline (Task 2)** was used as the reference input for the PID controller. The robot was able to follow the given path with stable motion and minimal deviation.

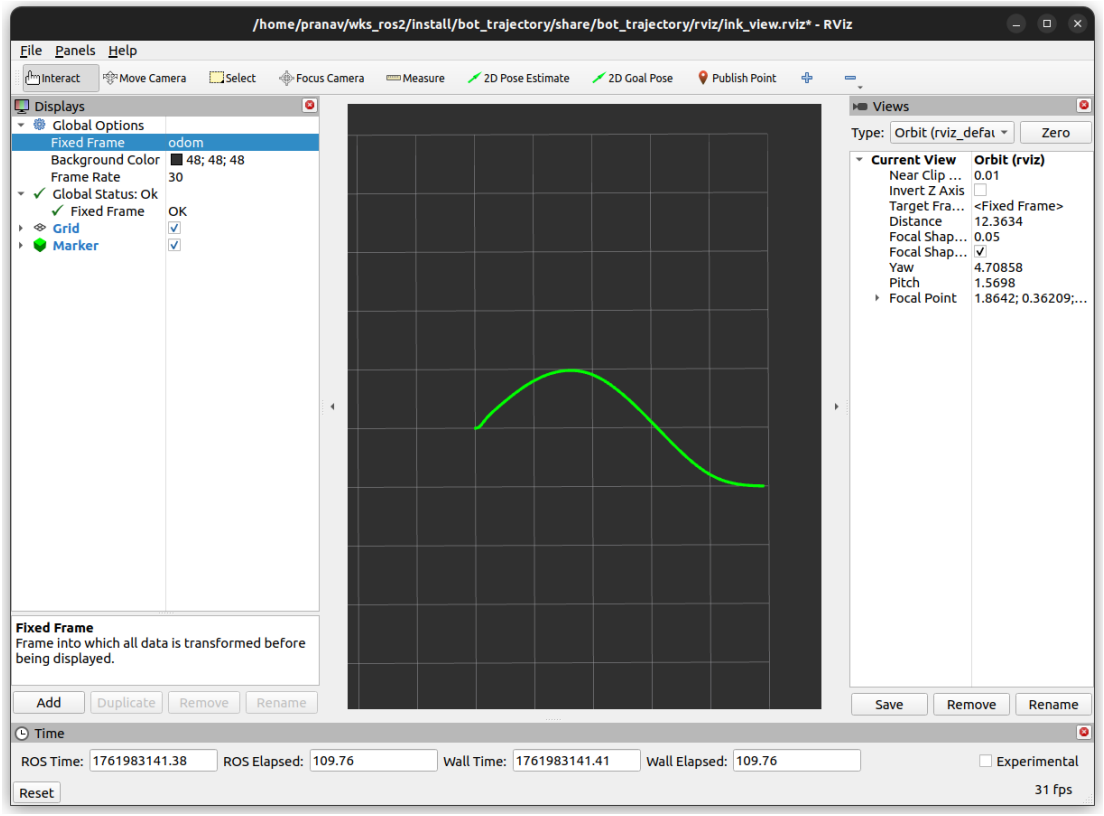
In simulation, the robot started from the initial waypoint and progressed along the curve while continuously correcting its orientation and position using feedback from odometry. The PID control parameters ensured smooth turning and accurate path convergence without significant overshoot or oscillation.

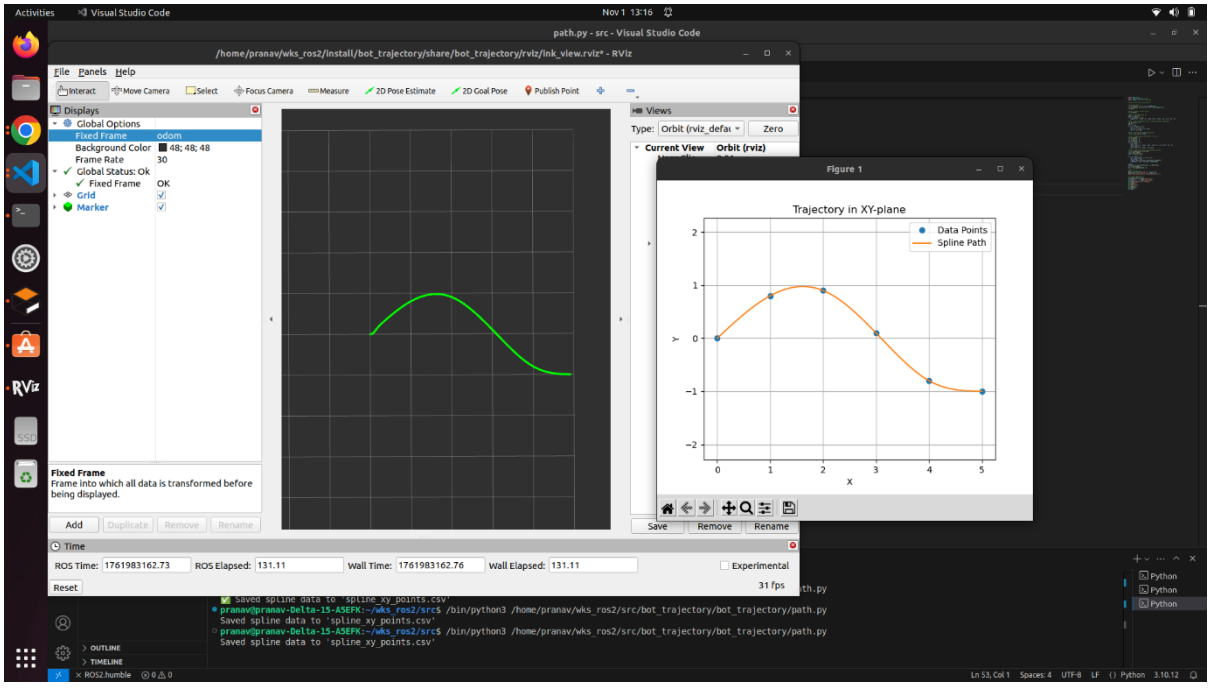
Original Trajectory
(Plot showing the continuous spline trajectory generated from discrete waypoints.)



Path Tracked in RViz

(Visualization of the actual robot path following the spline in simulation.)





By comparing the two paths, it can be seen that the **tracked trajectory closely follows the reference path**, confirming the effectiveness of the controller.

Overall, the results demonstrate that the controller achieves reliable trajectory tracking in a differential drive robot setup. The system is modular and can easily be extended to include **reactive obstacle avoidance** using LiDAR, as described in the future work section.

4.4 Conclusion

This project successfully demonstrated the complete workflow of autonomous navigation for a differential-drive robot — from generating a smooth trajectory to accurately tracking it using a feedback controller. Starting with discrete waypoints, the **Natural Cubic Spline algorithm** was used to produce a smooth, continuous path that minimized sudden changes in curvature, ensuring feasible motion for the robot.

The **trajectory tracking controller**, implemented using the **PID control method**, effectively translated this smooth path into real-time motion commands. Through continuous feedback from the robot's odometry data, the controller adjusted linear and angular velocities to minimize positional and angular errors, enabling precise path following in simulation. The results confirmed that the tracked path closely matched the reference trajectory, validating the control strategy's performance and stability.

This work establishes a strong foundation for further enhancements such as **reactive obstacle avoidance** using LiDAR, **adaptive PID tuning**, or **trajectory re-planning** for dynamic environments. Overall, the project demonstrates a clear understanding of path smoothing, control system design, and ROS2-based implementation — key components of modern mobile robot navigation.

5.0 Extension to a Real Robot

To extend this work from simulation to a real robot, several hardware and software adaptations would be required while keeping the core logic unchanged.

The PID-based trajectory tracking and path generation algorithms can remain the same, but the robot's **sensor inputs and actuator commands** would come from physical hardware rather than simulated topics. The /odom topic in simulation would be replaced by real odometry data obtained from **wheel encoders**, **IMU**, or **visual odometry** sensors mounted on the robot. Similarly, the velocity commands published to /cmd_vel would be interpreted by the robot's **motor drivers** through its onboard controller.

A **LiDAR or depth camera** can be integrated for real-time obstacle detection. This would allow the same LiDAR-based obstacle avoidance logic to be used on the physical platform to ensure safe navigation. The robot's onboard computer (like a Raspberry Pi or Jetson Nano) would run the ROS2 nodes, communicating with sensors and actuators over standard protocols such as I^2C, UART, or CAN.

Calibration would play a major role in real-world deployment. PID gains may need to be re-tuned to account for real-world factors such as wheel slip, uneven surfaces, and sensor noise. Additional filtering methods, like a **Kalman Filter** or **sensor fusion**, can be introduced to improve position accuracy and stability.

Overall, by connecting the same ROS2 framework to hardware interfaces and refining sensor feedback, the developed simulation can be directly extended into a fully functional **autonomous mobile robot** capable of following paths and avoiding obstacles in real environments.

6.0 AI Tools Used

AI support (such as **ChatGPT**) was used to:

- Understand mathematical concepts behind **cubic spline interpolation** and **PID control** in simpler terms.
- Get suggestions on how to structure the **ROS2 nodes** for publishers and subscribers and generating code for launch files and path_marker.py
- Improve the **readability** of Python scripts used for trajectory generation and tracking.