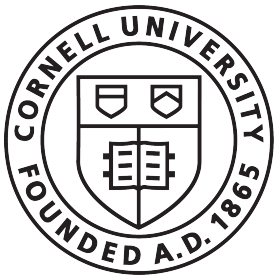


MAE 4291: Supervised Senior Design Experience

Path Tracking Control of a Small Mobile Robot

Jin Hyun (Addy) Park
Supervisor: Prof. Mark Campbell

May 17, 2024

Preface

This report is structured into two sections. The first section consists of an executive summary, which contains a high-level overview of the project's objectives, design decisions, results, and conclusions that are required for the senior design report. In the second section, a detailed implementation report delves into the different software components involved and the details of the implementing the path-tracking controller for the robot. In the Appendix section, I listed the steps required to run all the different software components. Additionally, I provided a list of ROS topics for interfacing with the Thymio's sensor and actuators. Although not required for the report, the goal of the second section and the appendix is to serve as a reference for anyone interested in working with the Thymio and building on it in the future.

Contents

Part I

Executive Summary

1 Overview

1.1 Objective

The primary objective of this project was to design a compact, autonomous robot capable of accurately tracking a predefined trajectory in order to mimic pedestrian movements in an urban setting. This robot is intended to serve as an important component within a testbed environment for the Jackal project and for future autonomous driving research in general. By mimicking realistic pedestrian behavior, the robot will help researchers test and refine the decision-making algorithms of autonomous vehicles.

1.2 Design Approach

To design such a robot, the approach that was taken at a high level is outlined below:

1. **Select a Suitable Robot**: Identify and purchase a small robot that meets the performance requirements for the project.
2. **ROS Integration**: Integrate the robot with ROS (Robot Operating System) to enable use of ROS functionalities and ensure compatibility with the OptiTrack motion capture system which will be used for the robot's localization.
3. **Controller Design**: Design and implement a path-tracking controller for the robot within the ROS environment.

Given the limited time frame of the project, I opted to purchase a robot rather than building one from scratch. This approach allowed me to focus on developing software for the robot's path-tracking controller.

2 Design Requirements and Constraints

2.1 Design Requirements

In order for the robot to mimic a pedestrian in an urban setting, it was important that the design satisfied the following performance requirements:

1. The robot should be able to accurately follow reference trajectories that mimic human movement patterns, such as moving back and forth between two points (similar to crossing a street) and navigating rectangular paths (like crossing multiple crosswalks at an intersection). Also, for accurate tracking, the robot should not deviate more than 10 cm from the reference path at any given time.
2. Assuming that the Jackal robot will be moving no faster than 1 m/s, the small robot should be able to move at a speed of around 10 cm/s (10% of the speed of the Jackal). This is a realistic representation of how fast pedestrians move compared to moving vehicles in an urban setting.
3. The robot must be capable of being integrated with the ROS environment, enabling us to utilize ROS tools. This integration allows us to develop all the necessary software within the ROS framework and build on existing tools (don't have to write everything from scratch!).

4. The robot should maintain consistent performance for approximately 30 minutes to last through extended testing periods.

5. The robot should be able to accommodate motion capture sensors.

6. The robot should be less than one fifth of the size of Jackal robot for realistic scaling between a vehicle and a pedestrian.

7. The robot should have extended projected lifetime of support.

8. The robot must be capable of wireless communication with a PC running ROS.

2.2 Trade Study

2.2.1 Robot Model

I identified five robot models that potentially fit the requirements of this project. These models were selected based on specific criteria: they are smaller than a fifth of the size of the Jackal robot, capable of rotating in place so that it can make sharp turns, able to communicate wirelessly, and available within a reasonable price range.

Then, a trade study was performed in order to decide on the robot model to purchase. The trade study was based on six most important factors which were guided by the design requirements discussed in Section 2.1. The following bulletpoints explain the importance of each factor:

- **ROS driver availability**: Developing your own driver for a robot can be time-consuming. To complete the project within the length of a semester, it would be desirable if there's a ROS driver for the robot already available.

- **Maximum speed**: The robot should be able to travel at about 10 cm/s (10% of the speed of the Jackal).

- **Battery life**: Assuming that a single test run will last around 10 minutes, the robot should have long enough of a battery life to be able to run for at least three iterations of the test.

- **Ability to accommodate motion capture sensors**: Because a motion capture system will be used for localization of the robot, the robot needs to be able to accommodate motion capture sensors.

- **Wireless communication type**: Because ROS will be running on a desktop PC and will be receiving sensor data and sending commands to the robot, the robot should be able to communicate with the PC wirelessly.

- **Projected lifetime of support**: It is desirable to have extended support for the robot in case hardware/software problems arise in the future.

Based on these six factors, a trade study table was created to evaluate the candidacy of each robot model:

Robot model	ROS driver available?	Maximum speed	Battery life	Can rotate in place?	Can accommodate MoCap sensors?	Wireless communication type	Projected support lifetime
Anki Vector	Yes	Unavailable	30 mins	Yes	Yes	WiFi	Available but unreliable
Anki Cozmo	Yes	Unavailable	45 mins	Yes	Yes	WiFi	Available but unreliable
Sphero Mini	Yes	1 m/s	45 mins	Yes	No	Bluetooth	Indefinite
Thymio II	Yes	15 cm/s	3 hrs	Yes	Yes	2.4 GHz, protocol 802.15.4	Indefinite
Makeblock mBot	No	50 cm/s	1 hr	Yes	Yes	Bluetooth	Indefinite

Table 1: Trade Study for robot design

Factors were weighted based on their relative importance. The relative importance, in descending order, are: 1. Motion capture sensor accommodation, 2. ROS driver availability, 3. Battery life, 4. Maximum speed, 5. Projected

lifetime of support, 6. Wireless communication type. After careful evaluation, the Thymio II (which we'll referred to as just "Thymio" from now) emerged as the top choice due to its long battery life and comprehensive documentation for its ROS driver (which supports various ROS distributions, including ROS2 in case the lab switches to ROS2 in the future). The Anki Vector and Cozmo were also considered viable options. However, their ROS package documentation is less thorough than that of the Thymio.

2.2.2 Controller Design

To satisfy the tracking performance requirement stated in Section 2.1, we require a controller that ensures robust path-tracking capabilities for the robot. Open-loop controllers, which operate by issuing a predefined sequence of control inputs to the robot's motors, do not consider the robot's actual state during its maneuver. This lack of feedback makes them unsuitable for the purpose of this project. For instance, different patches of the carpet on the field may have different coefficients of friction. These varying external conditions, called "disturbances," influence the robot's state through the system dynamics, which may then be corrected if feedback is present. This leads to the idea of a closed-loop controller. This type of controller continuously monitors the robot's state and adjusts the control inputs based on feedback. This approach allows for real-time corrections of any deviations from the desired trajectory, correcting for any discrepancies between what the controller "thought" the system would do in response to a control input and what the system actually ends up doing. Thus, closed-loop controllers are a better choice for achieving accurate path-tracking required for this project.

Another way to classify controllers involves considering whether they incorporate a model or not. By "model," we are referring to a mathematical model that describes how the system evolves over time. For instance, the Thymio, a differential drive robot, is governed by the following dynamic model:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} vcos(\theta) \\ vsin(\theta) \\ \omega \end{bmatrix}$$

This is a nonlinear system of the form $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$, where the state vector is $\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$, the input vector is $\mathbf{u} = \begin{bmatrix} v & \omega \end{bmatrix}^T$ and f is a nonlinear function in $\mathbf{x}$ and/or $\mathbf{u}$. Although model-based control design allows the use of optimal control techniques (because the controller has *apriori* knowledge of how the system will respond to certain control inputs prior to applying them, such as which directions are more controllable than others), the inherent nonlinearities of this system complicate the application of model-based control approaches. Moreover, we don't need to bother with the control policy being optimal, as long as it meets the design requirements. Therefore, I opted to develop my controller without incorporating the system's model. There are three control strategies that I considered:

- **Position control with fixed reference point**: This strategy involves navigating the robot through a predefined set of waypoints. The robot sequentially targets each waypoint as a *fixed* reference point to steer towards. For each waypoint, the robot calculates the necessary linear velocity command v and angular velocity command ω to point itself towards the target and minimize the distance to the target. The linear velocity is determined based on the distance to the goal, scaled by a constant k_v, and the angular velocity is computed using the arctangent of the slope formed by the line connecting the robot's current position to the goal. This method is simple and effective for reference trajectories that consist of straight line segments. The control law is given by:

$$v = k_v \sqrt{(x - x_g)^2 + (y - y_g)}$$

$$\omega = tan^{-1}(\frac{y_g - y}{x_g - x})$$

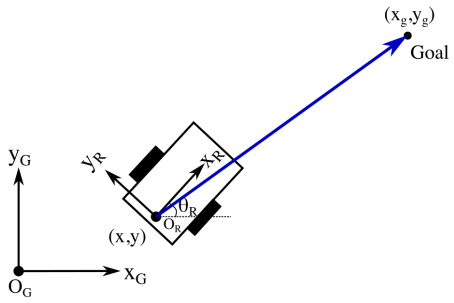


Figure 1: Position control with fixed reference point. Frame $\{O_G, x_G, y_G\}$ is the global frame. Frame $\{O_R, x_R, y_R\}$ is the body frame attached to the robot.

- **Pure pursuit controller (PD method)**: In this strategy, instead of a static reference point, the reference point is a function of time and moves along the desired path to be tracked:

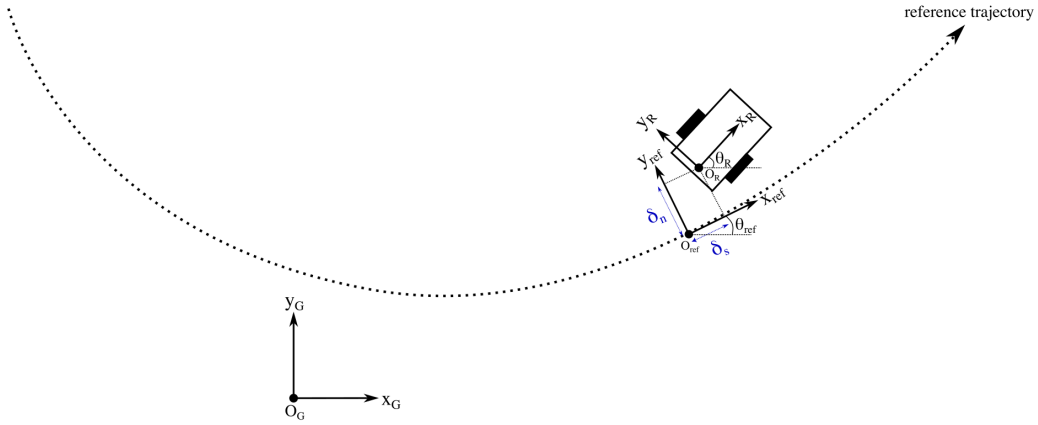


Figure 2: Pure pursuit control (PD method). Frame $\{O_G, x_G, y_G\}$ is the global frame. Frame $\{O_{ref}, x_{ref}, y_{ref}\}$ is a frame that represents the current reference (goal) pose. Frame $\{O_R, x_R, y_R\}$ is the body frame attached to the robot.

  This method is designed to compute and reduce three key errors:

  - **Along-track error** $\delta_{\mathbf{s}}$: Measures the longitudinal deviation along the path, indicating how far ahead or behind the robot is relative to the target point on the trajectory.
  - **Cross-track error** $\delta_{\mathbf{n}}$: Measures the lateral deviation from the path, indicating how far the robot is to the left or right of the target point.
  - **Heading error** $\delta_\theta (= \theta_{ref} - \theta_R)$: Measures the angle difference between the robot's current heading and the reference heading (which is the angle of the line tangent to the trajectory at the target point).

  The controller calculates the linear and angular velocities required to correct these errors, effectively steering the robot back towards the reference trajectory and aligning its heading with the path. Because the heading error is proportional to the rate of change of the cross-track error, this method is equivalent to PD control. Details of this method are discussed in Section 5.

- **Pure pursuit controller (geometric method)**: The geometric method, depicted in Figure 3,, utilizes a receding horizon approach.

  In this method, instead of steering directly towards the current target point (labeled O_{ref}) as done in the PD method, the robot "looks ahead" by a certain lookahead distance from its current position to identify a forward

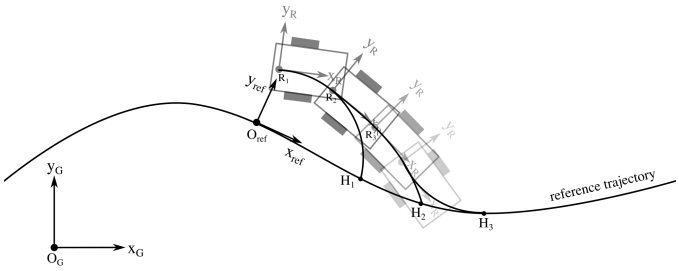


Figure 3: Pure pursuit control (geometric method). Frame $\{O_G, x_G, y_G\}$ is the global frame. Frame $\{O_{ref}, x_{ref}, y_{ref}\}$ is a frame that represents the current reference (goal) pose. Frame $\{O_R, x_R, y_R\}$ is the body frame attached to the robot.

target point, known as the "horizon point." The horizon points are labeled H_i, where i denotes the time step. At each time step, it calculates a constant-curvature path from the robot's position (labeled R_i) that will lead it to intersect this horizon point H_i. It follows this trajectory until the next time step, where it recalculates a constant curvature trajectory that intersects the next horizon point H_{i+1}. By doing this iteratively, the robot reduces its deviation from the planned trajectory. Because the robot is always "looking ahead" in the reference trajectory, this method is likely better suited than the PD method for trajectories with sharp turns since it's able to anticipate upcoming turns and start adjusting in advance.

There were three factors that were considered when I decided which controller to implement:

- **Implementation difficulty and code complexity**: It's important that the code for implementing the controller is not overly complex. If too complex, it will be difficult to implement in the first place and difficult to debug if problems arise.
- **Types of paths it can track**: The controller should be able to handle a wide variety of paths. This flexibility will make the controller suitable for a wide range of scenarios.
- **Ability to track desired velocity**: The controller should also be capable of tracking a desired velocity. This will make the controller suitable for if we want to test the Jackal's object tracking capabilities for objects traveling at different speeds.

"Accuracy" was not included a factor because it was difficult to judge which controller would be more accurate without implementing all three controllers.

Just as was done for choosing the robot model, a trade study was conducted to evaluate which of the three controllers was most suitable for this project based on the three criteria mentioned above. Below is a table that summarizes the findings:

Controller	Implementation difficulty & Code complexity	Types of trajectories it can track	Can it also track desired velocity?
Fixed reference	Easy/Simple	Paths with straight edges	No
Pure Pursuit (PD)	Moderate	Any kind of path	Yes
Pure Pursuit (receding horizon)	Difficult/Complex	Any kind of path	Yes

Table 2: Trade study for controller design

Although the design requirements listed in Section 2.1 only require the robot to track trajectories composed of straight segments, the fixed reference controller is rather restrictive for future adaptability. Considering future projects where the robot might need to follow curved trajectories, I opted to implement the pure pursuit controller using the PD method. This choice offers a good balance, as it's not significantly more complex the fixed reference controller yet it provides the capability to track any kind of path. It would also allow us to track reference velocities which may be useful. Implementing the geometric method would require code that is much more complicated as it would require implementing a function to compute a constant-radius path from the robot's current position to the horizon point. Therefore, I chose to implement the pure pursuit controller using the PD method.

2.3 Testing

To test the robot's tracking performance, I tested the robot's ability to track two types of trajectories. The first trajectory is a straight path, aimed to mimic a pedestrian crossing the street back and forth. The second trajectory is a rectangular path, intended to mimic a pedestrian navigating multiple crosswalks consecutively at an intersection.

2.3.1 Tracking a Straight Path

To test that the robot can track a straight path to 10 cm accuracy, I gave it a reference trajectory that moves back and forth for a distance of 1 meter at a speed of 0.1 m/s. I collected data of the robot's position over time for multiple cycles and plotted the trajectory. The plot is given below:

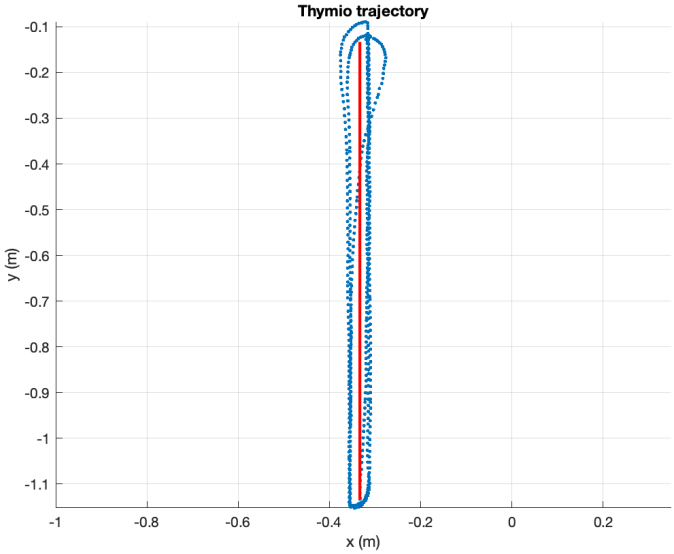


Figure 4: Plot of the robot's trajectory following a straight line reference trajectory. Blue points represent robot's position and the red line represents the reference trajectory.

Here is a video of the Thymio tracking a straight path: https://youtu.be/bu4Qtofysp8
Also, here is a video of the Thymio tracking a straight path but viewed in RViz: https://youtu.be/GOCAFdszizE

From the plot, we can see that the maximum deviation from the reference trajectory is less than 10 cm (i.e. 0.1 m). Therefore, the design successfully met the tracking performance requirement for tracking a straight path.

As for tuning the gains of the controller, I found that the gains $k_s = 1, k_n = 20, k_\theta = 5$ work the best. The cross-track gain k_n is a lot higher than the heading gain k_θ because of the difference in the scales of the errors. The cross-track error δ_t tends to be on the 0.01 meter scale whereas the heading error tends to be on the 0.1 radian scale (and on 1 radian scale during sharp turns). The along-path gain k_s was set to 1 as increasing k_s beyond 1 led to jerky movements in the along-track direction (robot switches back and forth between going full speed and stopping).

2.3.2 Tracking a Rectangular Path

As same as above, to test that the robot can track a rectangular path to 10 cm accuracy, I gave the controller a rectangular reference trajectory with 1 m sides. I collected motion capture data of the robot's position over time and plotted it. The plot is given in Figure 5. Note that the robot is able to track the rectangular trajectory better than a straight line trajectory because the turns required at the corners are less sharp.

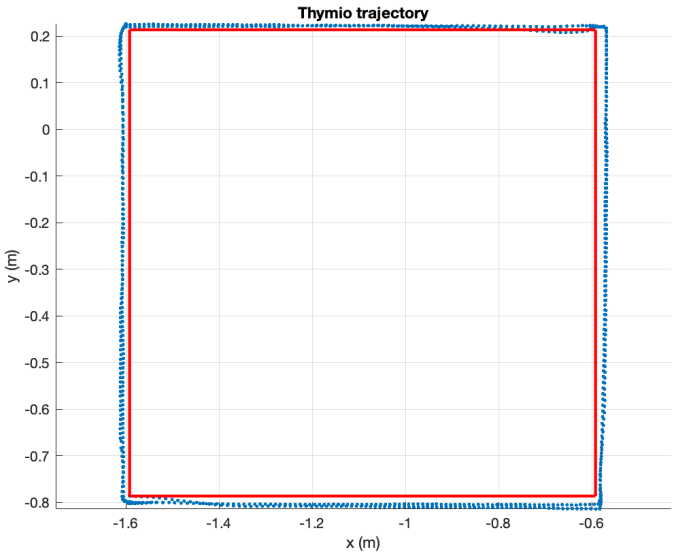


Figure 5: Plot of the robot's trajectory following a rectangular reference trajectory. Blue points represent robot's position and the red lines represents the reference trajectory.

Here is a video of the Thymio tracking a square path: https://youtu.be/Kpww3YYDXUM
Also, here is a video of the Thymio tracking a square path but viewed in RViz: https://youtu.be/-a7T4RWL7_w

As can be seen in the plot, the maximum deviation from the reference trajectory is less than 10 cm. Therefore, the design successfully met the tracking performance requirement for tracking a rectangular path.

2.3.3 Bonus: Tracking a Star!

To test that the robot can track more complex trajectories, I gave it a reference trajectory that traces out a star. Again, I collected motion capture data of the robot's position over time and plotted it. The plot is shown on the next page. Here is a video of of the robot tracking a star: https://youtu.be/JqWFIwH4wMk

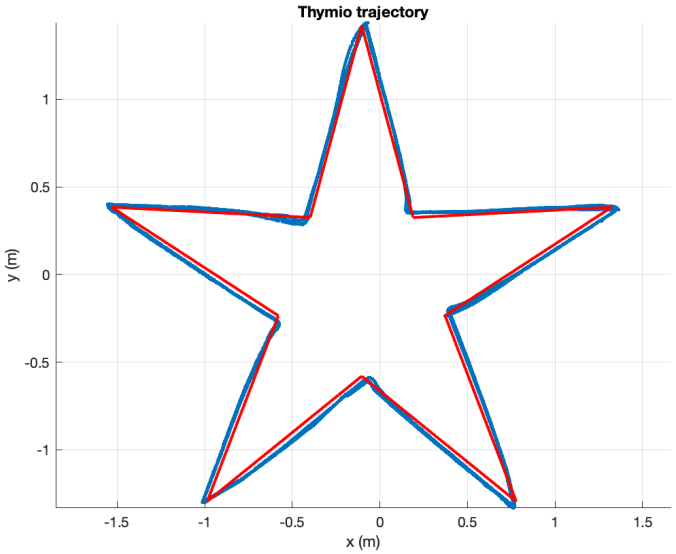


Figure 6: Plot of the robot's trajectory following a star reference trajectory. Blue points represent robot's position and the red lines represents the reference trajectory.

2.4 Tracking a Curved Trajectory?

Unfortunately, I wasn't able to implement and test curved trajectory tracking due to a lack of time. However, in principle, tracking a curved trajectory could possibly be easier than tracking trajectories made up of straight lines for two reasons:

1. As will be discussed in Section 6.1.1, when dealing with reference trajectories that are made up of straight lines, there is a need to segment the trajectory into multiple pieces. The logic required to do this was probably the most challenging part of writing the code. However, if there is a functional expression for the curved trajectory, you could use a library like `numpy` to generate reference pose at every time step (by parameterizing the path by time and plugging in the time at each time step) without the need to segment the path into multiple pieces.

2. Reference trajectories that are made up of straight lines are characterized by sharp turns at the vertices. As seen in Figure 4, the part of the trajectory where error is greatest is at the vertices where the robot must make a sharp turn. Because smooth curved trajectories have smooth turns, it should be more error-prone.

2.5 Design Evaluation

In the previous section, I showed that the robot satisfied the tracking performance requirement. Seven other design requirements were listed in Section 2.1. Let us go through the design requirements one by one and check that all design requirements were met:

- The robot was able to track the straight, rectangular, and star reference trajectories at a speed of 10 cm/s. Therefore, this requirement was met.

- The controller was implemented within the ROS environment. Therefore, this requirement was met.

- At full battery level, robot is able to operate for around three hours. Therefore, this requirement was met.

- The robot was able to accommodate motion capture sensors, allowing motion capture data to be used for localization. Therefore, this requirement was met.

- The robot is less than one fifth of the size of the Jackal robot. Therefore, this requirement was met.

- There is indefinite support available for this robot from the manufacturer (Mobsya). So this requirement was met.
- Finally, the robot is able to communicate with the PC wirelessly. Therefore, this requirement was met.

Since all eight requirements outlined at the beginning of the project were met, this design can be considered a successful design.

2.6 Conclusion

Overall, this project was a very valuable learning experience. The were two important skills that I learned throughout this project.

1. **Reading documentation**: I started off not having any idea how ROS could be integrated with the physical robot. Because of this, I spent a lot of time reading through the documentation for the ROS drivers developed for the Thymio. In hindsight, going through every page of the documentation wasn't entirely necessary as all that I needed to know in order to implement the path-tracking controller were the topics to publish and subscribe to in order to read sensor data from the robot and send actuator commands to the robot. However, by reading through the whole documentation, I began to understand the structure of robotics software in general, which is typically organized in "layers." The lower layers are usually abstracted away, so users only need to understand the topmost layer on top of which they are building. Additionally, becoming proficient in reading software documentation is a valuable skill that will be useful in the future.
2. **Familiarity with ROS**: Although I was introduced to ROS in robotics courses, most of the setup was pre-configured, and we only had to fill in function definitions. However, writing my own ROS module from scratch has significantly increased my familiarity with ROS.

3 Executive Summary Questions

1. **What are the desired function(s) of your design?**
   Refer to Section 2.1.

2. **What constraints related to the main function(s) must your design satisfy?**
   One important constraint was that robot had to be with within a reasonable price range. This placed a limit on which commercial robots were viable for this project. Budget will be a constraint no matter where you go (academia, industry, etc.). Another constraint was time. Because this project had to be completed within the span of a semester, it was preferable if the robot had a ROS driver that had already written been by someone.

3. **What are the performance objectives of your design? (Give quantitative metrics as much as possible)**.
   Refer to Section 2.1.

4. **What alternative design concepts were considered?**
   Refer to Sections 2.2.1 and 2.2.2.

5. **What analyses were used to select among these alternative design concepts?**
   Trade studies were used to select among the alternative designs. Refer to Sections 2.2.1 and 2.2.2.

6. **What industry or society standards were used to inform or evaluate your design?**
   My project didn't really consider any industrial or societal standards in the design process. However, some societal standards in robotics might be related to safety and ethical standards for what the design could be used for.

7. **Which concepts or skills learned in your coursework were applied to the design? Projects are expected to make substantial use of MAE and related ENGRD classes. Please provide a list with each entry providing the department and number of the course, plus a brief description of the particular concept or skill used.**
   MAE 2030/4730: Kinematics and reference frame concepts were used.
   MAE 3260/4780/6780: Controller design concepts from these courses were used in designing the path-tracking controller.
   MAE 4760: ROS was introduced in this course. Having a foundation in ROS was certainly helpful for this project. Robot kinematics and control were also covered in this course which were helpful.
   MAE 3780/4190: These courses helped me get familiar with different hardware components of a robot. I also learned to solder in this course, which was useful when I had to solder the wireless module onto the robot's motherboard.

8. **Evaluate your design, relative to its function(s) and constraints. How well did your design meet each of the performance objectives? How well does your design compare to other, existing solutions to the problem?**
   Refer to Sections 2.3 and 2.5 in the report. Existing solutions include some of the other small robots that were available in the lab, such as the Anki Vector. Although I haven't worked with the Vector robots, Professor Campbell mentioned that these robots had poor maneuverability on carpet. The robot purchased for this project (Thymio) didn't have any issues maneuvering on the carpet.

9. **What impact do you see your design, if implemented, having upon public health, safety, and human welfare, as well as upon current global, cultural, social, environmental, and economic concerns?**

The designed robot is intended to be used as part of a testbed environment for autonomous driving research. By contributing to this research, the goal is to enhance autonomous vehicle technology, making it safer and more reliable than it currently is.

10. **What format did your design take? For example, is it a complete set of CAD drawings, a working prototype, a full finished product, a system configuration, a process map, or something else?**
    My design consists of a physical robot with an implemented path-tracking controller. Because the robot itself was purchased, my main contribution was the software implementing path-tracking control.

11. **Describe each student's role in the design project if it was a group project.**
    This was an individual project.

Part II

Detailed Design and Implementation

In the second part of the report, I will explain how all of the different software pieces used in this project work in tandem and the details of implementing the pure pursuit controller.

4 Software Architecture

There are two important software components for the Thymio: `asebaros` and `ros-thymio`. These were written by Dr. Jerome Guzzi and the documentation for the two drivers is available at https://jeguzzi.github.io/ros-aseba/index.html. The goal of this section is to summarize the documentation and share what I've learned about how all of the software pieces fit together.

Note: Upon completing the project, I realized that you actually don't need to know all of the details of how the ROS packages work – you can get away with just knowing which topics to subscribe/publish to in order to access different sensor data and be able to send actuator commands. The different ROS topics for the Thymio are listed in Section A. Nonetheless, it was still interesting to understand how the different pieces work together and the information in this section might be helpful for debugging purposes.

4.1 Overview of `asebaros` and `ros-thymio`

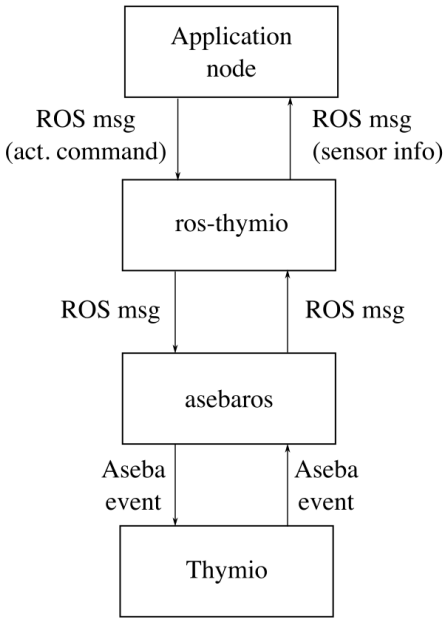


Figure 7: Schematic of the software stack.

The Thymio robot cannot run ROS natively. Natively, it runs software called Aseba, which is an event-based architecture (i.e. mainly works by detecting certain events and executing a specified block of code upon detection of that event). The inability of the Thymio to run ROS natively necessitates a ROS package that can serve as the bridge

between Aseba and ROS. `asebaros` is a ROS node that acts as this bridge. It effectively acts as a translator that translates ROS messages into a form that Asbea, running on the robot, can understand (called Aseba events) and vice versa. This has the effect of allowing you to access Aseba functionalities from the ROS environment, including retrieving sensor data from sensors on the Thymio and sending commands to the actuators on the Thymio. `asebaros` can be used for any robot running Aseba and is not specific to the Thymio robot. `asebaros` interacts directly with Aseba running on the robot and so it sits at the bottom layer of the software stack shown in Figure 7.

A second component to the software stack is `ros-thymio`, which implements Thymio-specific functionalities. Its main function is to set up ROS topics that let us interface with sensors and actuators that are specific to the Thymio robot. It also converts raw sensor/actuator data that `asebaros` retrieves from Aseba running on the robot to high-level data in SI units. `ros-thymio` interacts with `asebaros` (and doesn't interact with Aseba on the robot directly) and sits in the layer above `asebaros`.

Thirdly, the application layer sits at the top of the stack. This layer represents application nodes that users can write to implement high-level functionalities for the Thymio such as path-tracking. These nodes can subscribe to the topics configured by `ros-thymio` to receive sensor data from the Thymio robot. Similarly, they can publish to the topics configured by `ros-thymio` to send commands to the robot's actuators.

4.1.1 What does `asebaros` do?

First, I think it's useful to know a little bit about how Aseba works.

Nodes: An Aseba node is different from a ROS node. In Aseba, a "node" refers to a robot or a microcontroller (or anything that can run an *Aseba script*, which is explained later). "Aseba network" just refers to a collection of Aseba nodes.

Events: Aseba events are used to trigger a certain behavior from a node (i.e. robot). That is, once a node detects a certain event, it will execute the block of code defined for that specific event. An Aseba event can be thought of as being consisted of two things:

1. **Event name**: Each event is identified by a unique name, which is used to trigger event handlers (i.e. specified code that's executed in response to a certain event) corresponding to that event on the Aseba nodes. This is like a label that lets an Aseba node know what type of event has been broadcasted and whether the node should respond to it or not.

2. **Payload**: In addition to the name, an event carries a "payload," which is like the packet of data associated with that event. The even payload contains data/information that provides data to the event handlers about how to respond to the event. For example, if a `setspeed` event is to tell a node (robot) to start moving, the payload might contain the speed at which the robot should move at.

Scripts: A script in Aseba is a set of programmed instructions written in the Aseba programming language. An Aseba script can be thought of as consisting three things: (1) variable declaration & initialization, (2) event handlers, and (3) functions/subroutines.

- *Variable declaration and initialization*: Like any other programming languages, Aseba allows you to create variables, which you must declare and initialize. Also, there are a bunch of pre-defined variables that are automatically declared for you. These variables are specific to the hardware of the robot you are programming. Mainly, these pre-defined variables provide direct access to data from the robot's sensors and actuators:

1. Sensor Variables: Sensors such as proximity sensors, touch sensors, and accelerometers have corresponding variables in Aseba that hold the current sensor readings. For instance, the variable `prox.horizontal` is used for holding proximity sensor readings on the Thymio.
2. Actuator Variables: Variables that control actuators such as motors. For example, for the Thymio, the speeds of the left and right motors are controlled by writing to the variables `motor.left.target` and `motor.right.target`.

- *Event handlers*: Each event handler is associated with a specific event and contains the code that executes when that event is triggered. After the initial setup of variables, the script essentially enters a loop where it waits/listens for events. The microcontroller continuously monitors for any events that have corresponding handlers in the script. When an event occurs, the script executes the block of code defined in the corresponding event handler. This code block can modify variables, control the robot's actuators, or even trigger other events.

Here is an example script that causes the top LED of the Thymio to blink every 1 second:

```
# Variable declaration & initialization
var counter = 0
timer.period[0] = 100

# Event handler for timer0
onevent timer0
    counter += 1
    if counter > 10 then
        callsub blinkLED
        counter = 0
    end

# Function/subroutine to blink LED
sub blinkLED
    call leds.top(32, 0, 0)
    call leds.top(0, 0, 0)
```

The first section of the script creates a variable `counter` and sets it to a value of 0. Aseba provides two built-in timers (a pre-defined variable so it doesn't need declaration) that counts down for a specified time period. `time.period[0] = 100` initializes the period of the first timer to 100 milliseconds. If you needed a second timer that counts down from, say 1000 milliseconds, you would initialize `time.period[1] = 1000`.

The second section of the script creates an event handler for the timer initialized in the first section. `timer0` is a built-in event that is triggered every time the first timer has finished counting down from the specified time period. So every 100 milliseconds, it increments the variable `counter` by 1. And if `counter` is larger than 10 (i.e. 1 second has passed), it calls a subroutine called `blinkLED`.

The third section of the script defines a subroutine that blinks the top LED of the Thymio robot. It does this by calling a built-in function `leds.top()`.

For more details about Aseba programming, refer to the documentation at https://mobsya.github.io/aseba/index.html.

With these concepts, you can gain a better understanding of how `asebaros` works. `asebaros` does the following things:

- **Provide Read/Write access to Aseba node variables from ROS**: `asebaros` makes information about Aseba nodes (like sensor and actuator variables) accessible from within the ROS environment. Sensor/actuator variables are updated in real-time in Aseba, and through `asebaros`, these updates are made accessible to ROS. In addition to allowing you to remotely read from Aseba nodes' variables from within the ROS environment, it also allows you to write to the Aseba node variables from ROS (e.g. allows you to set the motor speed by writing to Aseba node variables `motor.left.target` and `motor.right.target`).
- **Load Aseba scripts from ROS**: `asbearos` enables you to deploy Aseba scripts to Aseba nodes directly from the ROS environment. This means you can program and reprogram Aseba nodes without leaving the ROS environment.
- **Send/Receive Aseba global events from ROS**: `asebaros` publishes information about Aseba events (event name, payload) to ROS topics. This integration ensures that events generated by Aseba can be received and acted upon from within the ROS environment. Conversely, ROS can also generate messages that are broadcast as events to Aseba nodes. For example, if the Thymio generates an `'ObstacleDetected'` event, this event is translated into a ROS message and published to a ROS topic. This message can then be received by a ROS node which can determine a new maneuver to avoid the obstacle and send the information about the new maneuver (like a velocity command) back to the Aseba node as an Aseba event.

In summary, `aseba` has two main functionalities. It allows you to read from and write to Aseba nodes' variables from the ROS environment. It also has a two-way translation capability. It translates ROS messages (which has been published to a topic) into Aseba events, which Aseba running on the robot can understand. Conversely, it also translates Aseba events into ROS messages, which are then published to ROS topics, which ROS can understand. This bidirectional translation is necessary because Aseba can only understand and respond to Aseba events and ROS can only understand ROS messages.

Above, the processes of getting/setting Aseba node variables and the translation of Aseba events to ROS messages and vice versa were described as being distinct processes (for the sake of simplicity). However, getting/setting Aseba node variables also happens through events. Now that we are familiar with how Aseba events work, here is an example from the documentation that shows how the process of getting a sensor variable occurs through events:

```
# Example event handler from script running on Thymio
onevent prox
    emit proximity prox.horizontal
```

Above is an event handler for a built-in event with name `prox`, where, every time sensor data for the proximity sensor is updated, this event is triggered. When triggered, it broadcasts another event called `proximity` with event payload `prox.horizontal` which contains the proximity sensor readings (a list with 7 elements, one for each of the seven proximity sensors on the robot). The keyword `emit` is used to trigger events, followed by the event name and the payload. Then, the `proximity` event is detected by `asebaros` and translated into a ROS message of message type `Event`, and published to a topic created for this specific event type, `aseba/events/proximity`. The ROS message might look something like this:

```
asebaros_msgs/Event:
    data: [928, 3212, 2039, 1292, 1029, 0, 0]
```

Note that the ROS message holds the event payload `prox.horizontal` in the `data` field. Now, the proximity sensor reading can be accessed from the ROS side.

4.1.2 What does `ros-thymio` do?

The ROS package `ros-thymio` does mainly two things. It first converts raw data from the Thymio's sensors and actuators to high-level data in SI units. Then it creates a set of topics for these high-level data to be published to or retrieved from, which your application node can interface with (instead of interfacing with the topics that contain raw data like `aseba/events/proximity` in the previous example).

To demonstrate this, the documentation continues on from the previous example. Recall that the following message has been published to the topic `aseba/events/proximity`:

```
asebaros_msgs/Event:
    data: [928, 3212, 2039, 1292, 1029, 0, 0]
```

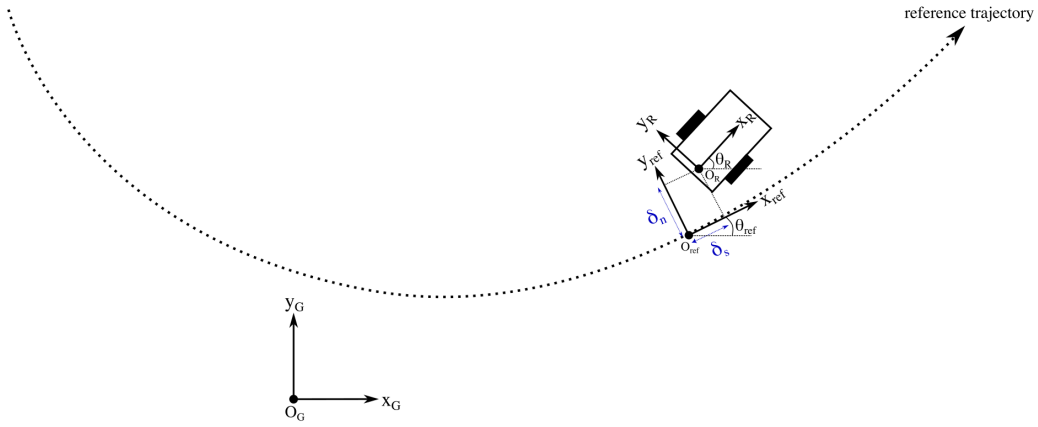
Let's say we want to get the proximity reading from just the center proximity sensor (instead of all seven). A problem with the data above is that the units are not in physical SI units and therefore are not intuitive. The `thymio-driver` node contained inside the `ros-thymio` package converts this raw data into SI units and also provides other information that are specific to the central proximity sensor on the Thymio. Then it publishes the converted data to the topic `proximity/center`, which your application node can subscribe to and receive sensor readings from.

While your application node could interface with (i.e. subscribe/publish to) the topics used by `asebaros` (such as `aseba/events/proximity`) directly, this is inconvenient because the raw data that is being communicated through these topics have non-intuitive units. Therefore, it's better to interface with the topics provided by `ros-thymio` (such as `proximity/center`) instead as the data being communicated through these topics have SI units, which are intuitive. If you do not need to access the raw data of the sensors and actuators, you don't need to understand what `asebaros` does and only need to understand how `ros-thymio` works. A list of topics provided by `ros-thymio` is summarized in Section A.

5 Path-tracking Implementation

5.1 Pure Pursuit (PD method)

This section discusses the details of the pure pursuit controller. Recall Figure 2:



- **Along-track error** $\delta_{\mathbf{s}}$: Measures the longitudinal deviation along the path, indicating how far ahead or behind the robot is relative to the target point on the trajectory.
- **Cross-track error** $\delta_{\mathbf{n}}$: Measures the lateral deviation from the path, indicating how far the robot is to the left or right of the intended trajectory.
- **Heading error** $\delta_\theta (= \theta_{ref} - \theta_R)$: Measures the angle difference between the robot's current heading and the reference pose heading (which is the angle of the line tangent to the trajectory at the reference point).

The open-loop control is given by:

$$u_{OL}(t) = \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{x}_{ref}(t)^2 + \dot{y}_{ref}(t)^2} \\ \dot{\theta}_{ref}(t) \end{bmatrix}$$

This control input would track the reference trajectory exactly if the robot perfectly followed the specified linear and angular velocity commands. However, the linear and angular velocity controllers that are embedded somewhere within Thymio's firmware cannot track the reference linear & angular velocities perfectly. Therefore, an additional term that corrects the errors must be added to the open-loop control:

$$u_{CL}(t) = u_{OL}(t) + K \begin{bmatrix} \delta_s \\ \delta_t \\ \delta_\theta \end{bmatrix}$$

where $K = \begin{bmatrix} k_s & 0 & 0 \\ 0 & k_n & k_\theta \end{bmatrix}$. This is the closed-loop control.

Here is a block diagram that summarizes the path-tracking controller:

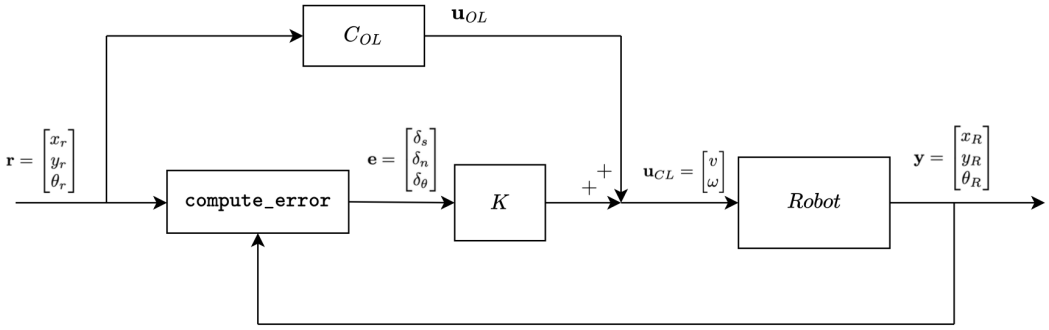


Figure 8: Block diagram of the path-tracking controller. The "Robot" block has been left as a black box since no model was used. C_{OL} is the open-loop controller.

6 path_tracking Module

The ROS package `path_tracking` was written to implement the pure pursuit controller. This package contains two nodes: (1) `reference` and `controller`. In the software stack described in Figure 7, this package sits at the application layer, meaning that it makes use of the `ros-thymio` package to indirectly interface with the Thymio in order to retrieve sensor data from and send actuator commands to the Thymio. It does not directly interact with `asebaros`. The complete code has been uploaded at https://github.com/apark2459/thymio-path-tracking.

To give a brief overview, `reference` generates a reference pose at every time step (based on the reference trajectory that the user inputs) and passes it onto the `controller` node. The `controller` node receives the Thymio's current pose, compares it to the reference pose received from the `reference` node, and computes the necessary control inputs to correct the errors. It then publishes this control input as a `Twist` message to a topic that `ros-thymio` has created (as described in Section 4.1.2) and is subscribed to. Then, as described in Sections 4.1.1 and Section 4.1.2, `ros-thymio` converts this velocity command to a raw data format that Aseba understands and publishes it to a topic that `asebaros` has created and is subscribed to. Then `asebaros` converts this ROS message containing raw data into an Aseba event, which Aseba running on the Thymio can detect and command the motors on the Thymio. Figure 9 is a diagram that schematically describes this process.

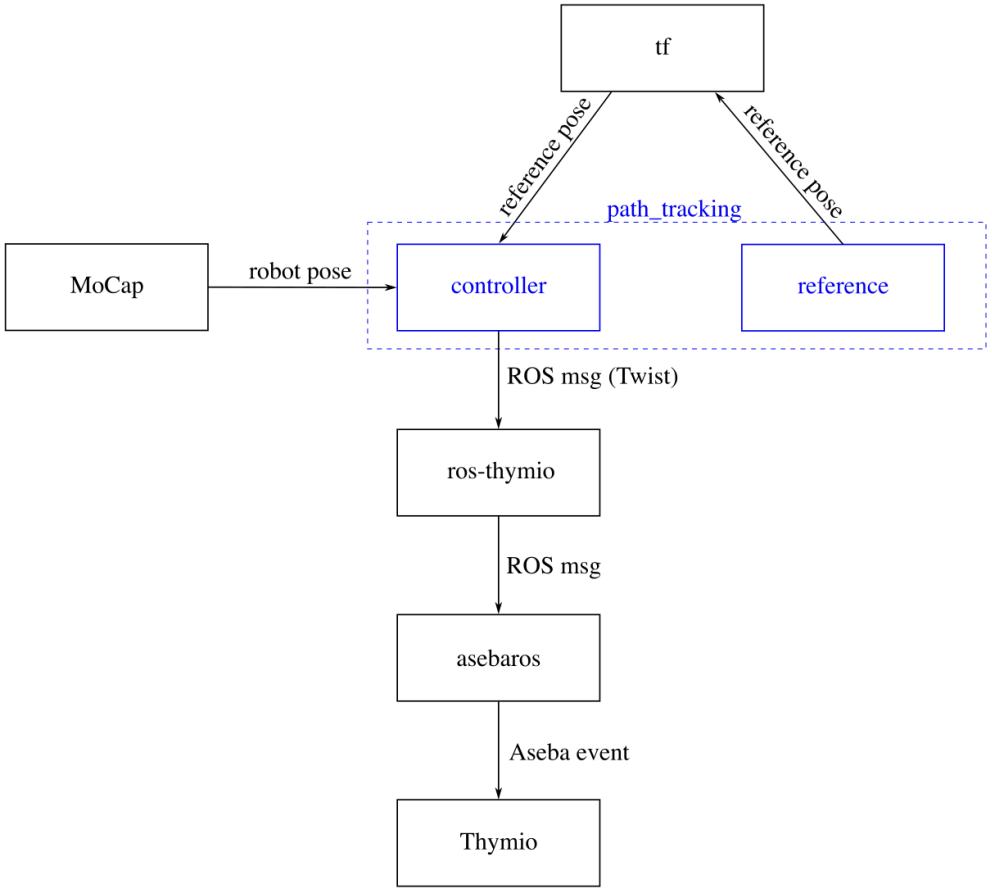


Figure 9: Diagram of how `path_tracking` package interacts with other software components.

6.1 `reference` Node

The main function of this node is to generate the reference pose that the Thymio should track at every time step. Then it turns it into a `tf` frame and broadcasts it such that the node `controller` can access it.

`reference` takes in two ROS parameters: (1) `waypoints` and (2) `velocity`. `waypoints` is a list that contains the waypoints that defines the reference trajectory. `velocity` is the velocity associated with the reference trajectory (i.e. the reference velocity). For example, if you want to define a square reference trajectory with sides equal to 1 meter that travels at 0.1 m/s, `waypoints` should be `[[0,0],[1,0],[1,1],[0,1]]` and `velocity` should be 0.1.

Note: The code is written such that the waypoints should be defined relative to Thymio's location at the start of running the node and not relative to the global coordinate system of the motion capture system. For example, let's say that the Thymio is initially located at `[3.14,3.14]` in the global coordinate system. If you want the Thymio to travel through the waypoints `[[3.14,3.14],[4.14,3.14],[4.14,4.14],[3.14,4.14]]` in the global coordinate system, you should pass in `waypoints=[[0,0],[1,0],[1,1],[0,1]]` as the parameter value.

Given a finite number of waypoints, `reference` creates a *continuous* reference trajectory. It does so by interpolating between the waypoints at constant time intervals. For example, for the waypoints `[[0,0],[1,0],[1,1],[0,1]]`, the interpolated points would look like what's shown like Figure 10.

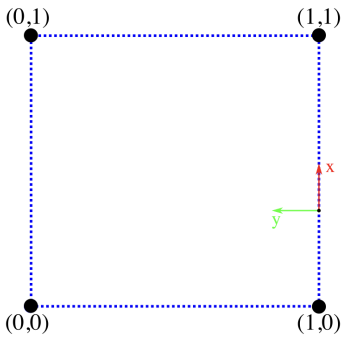


Figure 10: Interpolation between waypoints.

The black points represent the waypoints and the blue dots in between the waypoints represent the interpolated points. A single reference *pose* has also been shown in the diagram (a reference pose also has orientation, defined by the x and y axes in red and green).

6.1.1 How the code works

Given a list of waypoints, `reference` generates reference poses in real time by using a "divide and conquer" approach:

1. Given the list of waypoints, the code divides the trajectory into multiple line segments and numbers them. For example, for the square path in Figure 10, the numbering would be as follows:

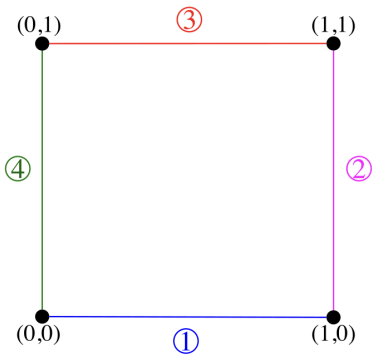


Figure 11: Diagram showing how reference trajectory is segmented into multiple line segments.

2. Then it loops through each segment and calls the function `pose_for_segment`, which is a function that returns reference poses on a single segment as a function of time. This approach is efficient as you can call the same function iteratively for every segment rather than writing separate code for each segment. For example, for segment 2 in Figure 10, `pose_for_segment` would return reference poses along the segment at as time evolves from t_0 to t_4:

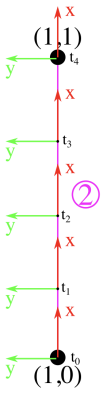


Figure 12: Diagram showing how reference trajectory is segmented into multiple line segments.

3. Finally, when it has completed the last segment in the reference path, it starts over from the beginning.

6.2 `controller` Node

The `controller` node receives the robot's pose from the motion capture system and computes the pose error. Using this error, it computes the necessary control inputs required to correct the error using the equations given in Section 5.1.

6.2.1 How the code works

The `controller` node computes the control input $u = \begin{bmatrix} v & \omega \end{bmatrix}^T$ by following these steps:

1. **Compute the error**: As seen in Figure 2, the three error terms can be obtained by expressing the robot's pose in the reference pose frame. That is, $\delta_s = -x_R^{ref}$, $\delta_t = -y_R^{ref}$, and $\delta_\theta = \theta_t - \theta_R$, where the superscripts denote the reference frame that the coordinates are being expressed in. To do this, it uses the `transform` method in the `tf2_ros` library.

2. **Compute the control inputs**: Given the error, the function `compute_control` uses the equations in Section 5.1 to compute the necessary control inputs to correct the error.

3. **Publish control inputs**: Publish the control input as a `Twist` message to the `cmd_vel` topic.

6.3 Next Steps

If I were to continue this project, the next step would be to extend the `path_tracking` module to multiple Thymio robots, where you would be able to give unique reference trajectories to each Thymio robot and have them track those trajectories simultaneously. There already exists a driver avaialble for interfacing with multiple Thymios in the `ros-thymio` package, called `multi_thymio_driver`.

A Appendix

In the appendix, some of the topics provided by `ros-thymio` (the ones I think are useful) are listed. The procedure for running all of the difference pieces of the software is also outlined.

A.1 Installing `asebaros` and `ros-thymio`

Install `asebaros` and `ros-thymio` by following the instructions at https://jeguzzi.github.io/ros-aseba/installation.html. Make sure to install both `asebaros` and `ros-thymio`. You should install these drivers into a new workspace (I named my workspace `thymio_ws`, which I will be using as the workspace name in this section).

A.2 Connecting the Thymio to the PC

First, run `roscore`. Then, to establish a connection between the Thymio and the PC, turn on the Thymio and insert the USB dongle to the PC. Then, run the following commands in the terminal window:

```
cd ~/thymio_ws
source ~/thymio_ws/install/setup.bash
source ~/thymio_ws/devel/setup.bash
export DEVICE="ser:device=/dev/ttyACM0"
export SIMULATION=False
sudo chmod 666 /dev/ttyACM0
roslaunch thymio_driver main.launch device:=$DEVICE simulation:=$SIMULATION
```

The line `sudo chmod 666 /dev/ttyACM0` basically gives ROS permission to read and write data to the USB dongle to communicate wirelessly with the Thymio. Once this permission is given, launching `ros-thymio` will connect ROS to the Thymio. If connection is successful, a message "`thymio-II is ready at namespace`" should appear in the terminal window.

Note: You do not have to separately run `asebaros` on another terminal. The `thymio_driver` launch file automatically runs `asebaros` for you. Trying to run both `thymio_driver` and `asebaros` will give you an error.

A.3 Running the `path_tracking` Module

After successful connection, you can run application nodes on the Thymio. But before launching the `path_tracking` module, you first need to launch the motion capture node because the `path_tracking` module relies on data from the motion capture system. Run the motion capture node by running the following command on another the terminal window:

```
roslaunch mocap\_optitrack mocap.launch
```

Then finally, run the launch file for the `path_tracking` module on a separate terminal window:

```
cd ~/thymio_ws
source ~/thymio_ws/install/setup.bash
source ~/thymio_ws/devel/setup.bash
roslaunch path_tracking main.launch
```

To specify the waypoints and reference velocity, open the `main.launch` launch file inside `/thymio_ws/src/path_tracking/launch` and edit the parameter values.

A.4 ROS Topics

In this section, I listed the ROS topics that I think are relevant/useful for developing ROS modules for the Thymio. A more comprehensive list of topics is available at https://jeguzzi.github.io/ros-aseba/thymio_driver.html.

A.4.1 Sensor readings

Subscribe to these topics to retrieve different sensor readings from the Thymio.

- **Ground sensors**: Messages from this topic contain the range measurement measured by the proximity sensors on the bottom of the Thymio. There are two of these sensors.
  - **Topic**: `ground/{left,right}`
  - **ROS message type**: `sensor_msgs/msg/Range`
- **IMU**: Messages from this topic contain the IMU sensor readings on the Thymio.
  - **Topic**: `sensor_msgs/msg/Imu`
  - **ROS message type**: `sensor_msgs/msg/Imu`
- **Joint States**: Messages from this topic contain "joint states" of the Thymio. By joint state, it's referring to the angular position and angular velocity of each wheel.
  - **Topic**: `joint_states`
  - **ROS message type**: `sensor_msgs/msg/JointState`
- **Odometry**: Messages from this topic contain the odometry data of the Thymio which includes position, orientation, linear velocity, and angular velocity. This is based on the encoder information.
  - **Topic**: `odom`
  - **ROS message type**: `nav_msgs/msg/Odometry`
- **Proximity sensors**: Messages from this topic contain range measurements measured by the proximity sensors on the front and rear sides of the Thymio.
  - **Topic**: `proximity/{left,center_left,center,center_right,right,rear_left,rear_right}`
  - **ROS message type**: `sensor_msgs/msg/Range`

A.4.2 Actuator commands

The only relevant topic to publish to (I think) is the velocity command topic:

- **Velocity command**: Publish to this topic to send a velocity command to the Thymio. The velocity command consists of a linear velocity field and an angular velocity field.
  - **Topic**: `cmd_vel`
  - **ROS message type**: `geometry_msgs/msg/Twist`

Note: Even though `aseba/events` topics are listed on the documentation, you probably won't ever need to bother with the these topics unless you want to access the raw sensor/actuator data.

References

[1] J. Guzzi, "ROS for Aseba and Thymio", available at:https://jeguzzi.github.io/ros-aseba/.

[2] Mobsya, "Aseba Documentation", available at: https://mobsya.github.io/aseba/.

[3] T. Bhattacharjee, "Path-following Control," CS 4750: Foundations of Robotics, Cornell University, 2023, unpublished.