Lab 8: Fullstack Robotics - Perception, Planning, and Control [*]

EECS/ME/BIOE C106A/206A Fall 2025

Goals

By the end of this lab you should be able to:

- Build a perception pipeline using YOLO to detect the position of a goal object.
- Use the goal position to create a trajectory to the object using a path planner
- Have the Turtlebot execute the trajectory with a PID controller

Contents

[*]Remodeled by Daniel Municio (Fall 2025). Developed by Eric Berndt, Karim El-Refai, Charles Xu, Kirthi Kumar, and Martin Zeng (Fall 2023). The PID control was wrong - as stolen then corrected in Lab 7, developed by Mingyang Wang and Shrey Aeron (Fall 2023)

1 Introduction

In this lab, we delve into the core components of full-stack robotics: perception, path planning, and control, using the TurtleBot as our robotic platform (Bootleg Tesla edition).

1. Perception: The TurtleBot will utilize sensors and image processing to detect a cup within its environment.
2. Path Planning: Upon detection, the robot determines a trajectory using a path planner to the cup.
3. Control: Equipped with a planned path, control algorithms guide the TurtleBot smoothly to its destination.

1.1 Starter Code

Before every lab, you should pull the updated starter code. To pull the lab 8 starter code, navigate to the `ros_workspaces/` directory and run

```
git pull starter main
```

2 Perception

2.1 The Glorious Logitech C920

In this lab, we will be using the Logitech C920, instead of the Intel RealSense. Unfortunately, the Logitech C920 does not have stereovision, so we are going to have to come up with another method of extracting depth from the image.

2.2 Starting up the Camera

To launch our camera, we must first ssh into the turtlebot

```
ssh fruitname@fruitname
```

Login with the password: `fruitname2025`. **In ssh** run

```
ros2 launch turtlebot3_bringup robot.launch.py
```

Now, launch the `usb_cam` launch file, which will publish the camera, and its intrinsic parameters, to ROS.

```
ros2 launch usb_cam usb_cam.launch.py
```

You can view your new image by running `rviz2` in the terminal to launch Rviz. Then click `Add` and then `By Topic` to see all the topics that Rviz can currently see. If your camera is working correctly, you should be able to add a new **Image** object under `/image_raw`. You should now be able to see your turtlebots camera feed!

2.3 The Camera Instrinsics

The launch file also includes a publisher which is constantly publishing the camera intrinsics (the K matrix) to the topic `/camera_info` which we have subscribed too for you. Inside of the `perception` package, you should find a file called `cone_publisher.py`. Fill in the `camera_info_callback()` function to properly index into this K matrix. To see the contents of `CameraInfo.msg`, you can view the ROS2 Docs for the CameraInfo.msg.

For the perception portion of this lab, we will need to use a virtual environment. To activate it, run this, INSIDE your distrobox

```
source ~ee106a/venvs/lab8/bin/activate
```

You will only need to run this in terminals where you run perception code

Task 1: Write code to extract the relevant intrinsic data from the `CameraInfo.msg` in `camera_info_callback()`. For this lab, we're going to need the intrinsic parameters f_x, f_y, c_x, c_y from the camera matrix K.

2.4 Finding the Cup: YOLO-based Object Detection

Modern object detection has evolved beyond simple color-based or pointcloud filtering methods to deep learning approaches that can robustly detect objects regardless of lighting conditions or color variations. In this lab, we will use a YOLO (You Only Look Once) image model to detect and segment the cup in our camera feed.

2.4.1 YOLO Segmentation Model

YOLO is a state-of-the-art, real-time object detection and segmentation system. Unlike traditional methods that rely on color thresholding, YOLO uses a convolutional neural network that has been trained on thousands of images to recognize object shapes, textures, and contexts. The segmentation variant of YOLO provides us with:

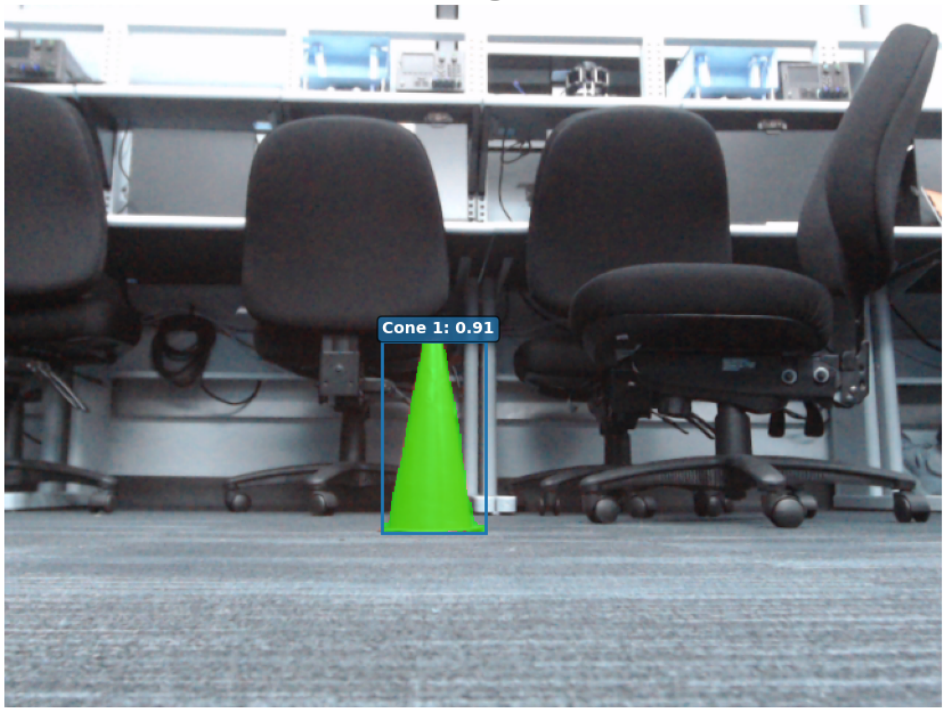
1. **Bounding boxes**: Rectangular regions defining where objects are located in the image
2. **Class labels**: What type of object is detected (e.g., cup, bottle, person)
3. **Segmentation masks**: Pixel-level masks showing exactly which pixels belong to the detected object
4. **Confidence scores**: How confident the model is about each detection

Daniel M's FSAE Autonomous team has a nice YOLOv8 model trained to recognize blue and yellow cones used in the Autonomous competition. We will use this model to recognize cones for this lab. If you're interested in how it was trained, you can look into this Google Collab Notebook. It was trained using the FSOCO dataset. The dataset it was trained on, is mostly outdoor track images, so the model will not perform incredibly amazingly in the classroom, but Daniel swears that it's very good for their use case.

2.4.2 Segmentation Output

When YOLO processes an image, it outputs a binary segmentation mask where pixels belonging to the detected cup are set to 1 (or 255 in 8-bit format) and all other pixels are set to 0. This mask can be directly used to identify the location and extent of the cup in the image. An image of what this looks like is shown below:

YOLO Cone Segmentation



2.5 Camera Projection

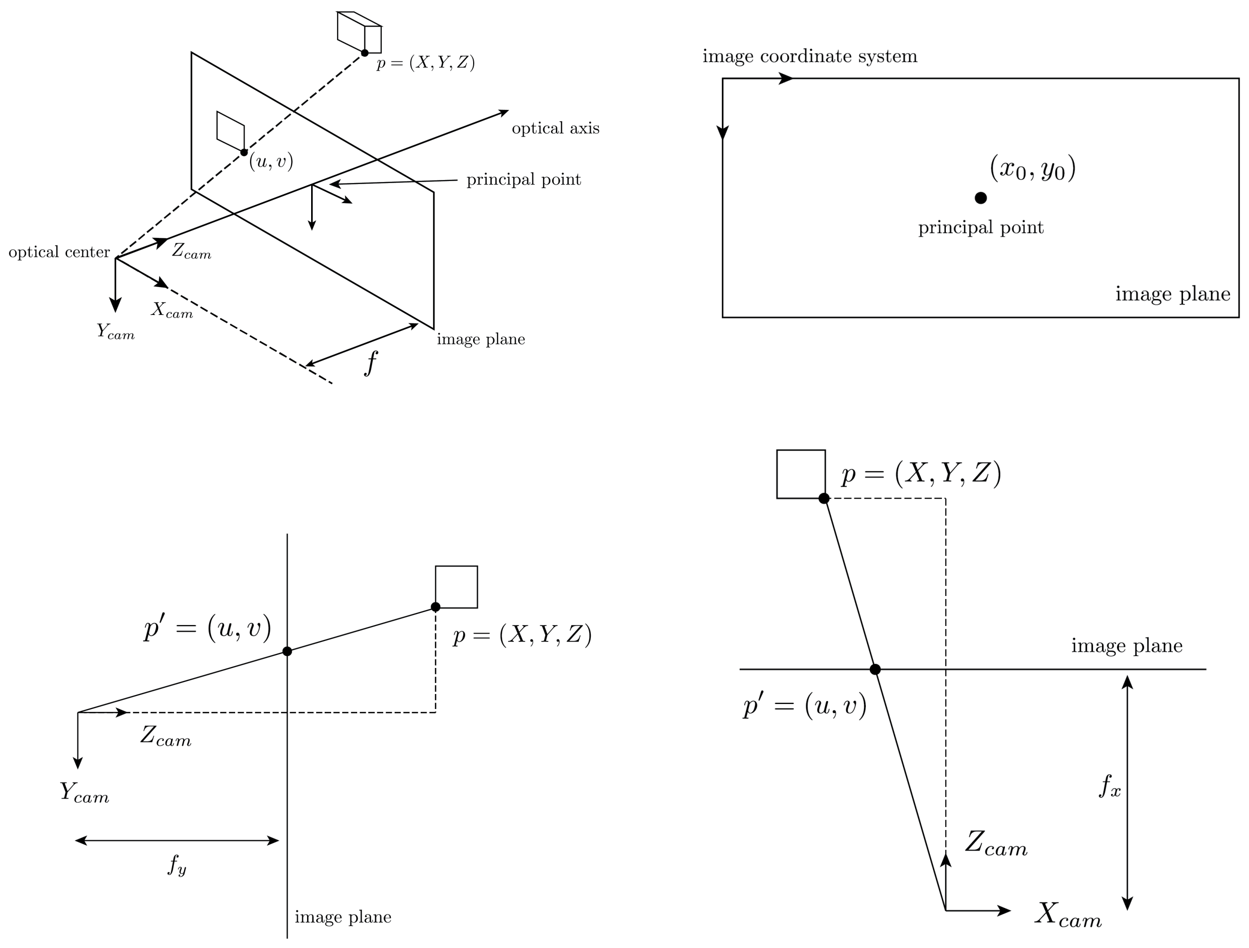


Figure 1: Geometry behind a Pinhole Camera

Given a point's 2D pixel coordinates (u, v) and its depth Z (in the camera's Z_{cam} direction), we can find its 3D coordinates (X, Y) in the camera's frame using the inverse projection equations:

$$X = \frac{(u - c_x) \times Z}{f_x} \tag{1}$$

$$Y = \frac{(v - c_y) \times Z}{f_y} \tag{2}$$

2.6 Deriving the Projection Equations

Place the optical center of the camera at the origin of the 3D space (i.e. $G_{cam->world} = I$) and let a 3D point be $P = (X, Y, Z)$. The image plane is positioned at $Z = f$, where f is the focal length. Using similar triangles, the relationship between 3D coordinates and image coordinates can be derived. For the x-coordinate:

$$\frac{X}{Z} = \frac{u - c_x}{f_x} \tag{3}$$

Here, u is the x-coordinate of the image point P_{image} on the image plane, and c_x (sometimes written as x_0) is the x-coordinate of the principal point.

Rearranging, you get:

$$X = \frac{(u - c_x) \times Z}{f_x} \tag{4}$$

Similarly, for the y-coordinate:

$$\frac{Y}{Z} = \frac{v - c_y}{f_y} \tag{5}$$

Which rearranges to:

$$Y = \frac{(v - c_y) \times Z}{f_y} \tag{6}$$

If we organize the camera intrinsic parameters into a matrix (commonly labeled as K) then the equation below is equivalent to what we just proved above:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix} \tag{7}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

2.7 The Intuition

The projection equations (e.g., $u - c_x = f_x \frac{X}{Z}$) show an inverse relationship between depth Z and the projected image size. As an object gets closer to the camera (a smaller Z), its projected pixel coordinates (u, v) move further from the principal point, making it appear larger. The subtraction of c_x and c_y is a translation, accounting for the fact that the principal point might not be at the exact center of the image.

2.8 Estimating Distance from Pixel Count

One of the challenges when working with a monocular (single) camera is estimating the distance to an object. However, if we know the approximate real-world size of the object we're detecting, we can estimate its distance based on how large it appears in the image.

For a 1D case using width or height:

$$Z = \frac{f_x \cdot W_{\text{real}}}{w_{\text{pixels}}} \tag{9}$$

where:

- Z is the distance from the camera to the object
- f_x, f_y are the focal lengths of the camera in the x and y directions (from camera intrinsics)
- W_{real} is the known real-world width of the object (e.g., diameter of a cup in meters)
- w_{pixels} is the width of the object in pixels

For a 2D case using area, we have:

$$Z = \sqrt{\frac{f_x \cdot f_y \cdot A_{\text{real}}}{A_{\text{pixels}}}} \tag{10}$$

where:

- A_{real} is the known real-world area of the object (e.g., cross-sectional area of a cup in square meters)
- A_{pixels} is the area of the object in pixels (the number of pixels N in the segmentation mask)

The 2D area-based approach is often more stable because it considers both dimensions of the object simultaneously, making it less sensitive to the object's orientation relative to the camera. This estimated depth Z, combined with the object's 2D pixel-center coordinates (u, v) and the back-projection equations (Equations 1 & 2), allows us to compute the full 3D position (X, Y, Z) of the cup in the camera's coordinate frame.

2.8.1 Calculating Object Center from Pixel Count

Once we have the segmentation mask, we need to determine where the cup is located. We can do this by:

1. **Count Pixels:** Sum all the non-zero pixels in the segmentation mask to get the total area, A_{pixels}.
2. **Compute Distance (Z):** Use A_{pixels} and Equation 10 to calculate the depth Z of the cone.
3. **Find Pixel Center (u, v):** Calculate the mean of the segmentation mask's non-zero pixel coordinates to find the object's center (u, v) in the image.
4. **Convert to Camera Coordinates (X, Y, Z):** Use the depth Z (from Step 2), the pixel center (u, v) (from Step 3), and the back-projection **Equations 1 & 2** to find the cone's 3D position (X, Y, Z) in the *camera's* coordinate frame.
5. **Transform to World Frame:** Apply the camera's static transform (from the robot's URDF) to convert the (X, Y, Z) camera-frame coordinates into the robot's world frame (e.g., `/odom`). This final 3D position is the goal for the path planner.

Task 2: Implement the remainder of `image_callback()` in `cone_publisher.py` You can test your code by running:

```
ros2 run perception cone_publisher
```

Checkpoint 1

Submit a checkoff request for a staff member to come and check off your work. At this point you should be able to:

- Demonstrate your code successfully identifying the position of the cone.
- Explain how you extracted depth without having a stereo vision setup
- Comment on any issues this method of extracting depth may have

---

3 Path Planning

3.1 What is a Trajectory

Imagine that you were given a starting pose and a target pose for a robotic system, say the end-effector pose of a robot arm, or the position of a mobile robot), there can be infinitely many ways to get from the current pose to the target pose. How to effectively generate an "optimized" trajectory from A to B is the problem that trajectory planning solves. The word optimized can mean vastly different things depending on the context, ranging from a humanoid robot simply not falling over, to bin-picking manipulators executing multiple picks and place actions in one second.

3.2 Bézier Curve

In this lab, we are simply concerned with how our TurtleBots can smoothly reach the target pose. One way to do this is by fitting a Bézier curve. A Bézier curve is a parameterized curve commonly used in robotics and computer graphics to fit an arbitrary desired shape awith few discrete "control points". We can use these control points to specify properties of the trajectory such as the starting and ending position, as well as the orientation.

Specifically, given the current TurtleBot pose and a target pose in the world frame, we can generate a trajectory with the following cubic Bézier curve equation:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \tag{11}$$

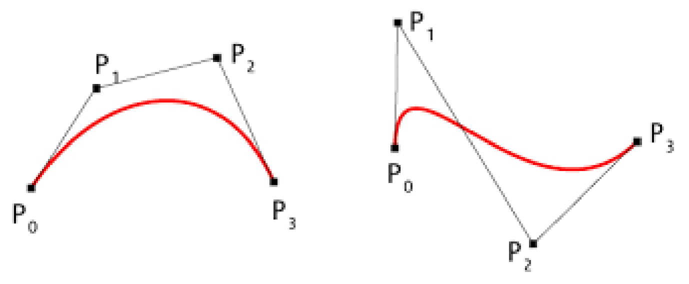


Figure 2: Cubic Bézier curves

where P_0 and P_3 are set to the starting point (x_1, y_1) and ending point (x_2, y_2) of the desired trajectory respectively. P_1 and P_2 are intermediate control points set to a fixed distance away from the starting and ending point at the desired start and end orientation:

$$P_1 = \begin{bmatrix} x_1 + cos(\theta_1) * \text{offset} \\ y_1 + sin(\theta_1) * \text{offset} \end{bmatrix} \tag{12}$$

$$P_2 = \begin{bmatrix} x_2 - cos(\theta_2) * \text{offset} \\ y_2 - sin(\theta_2) * \text{offset} \end{bmatrix} \tag{13}$$

After fitting the desired curve to the start and end pose, we sample a set of waypoints (x_i, y_i) along the curve at a fixed interval for our controller to track. Along with x_i and y_i values, we also need a desired rotation θ_i for the controller to track, which we can compute using the arctan of neighboring waypoints:

$$\theta_i = \arctan\left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) \tag{14}$$

3.3 Coding

Open up *trajectory.py*. There are four functions in this file, not including our main function.

- *plot_trajectory*: takes in waypoints, a list of points that we want to get to on the trajectory, and outputs a graph with projected orientation arrows. You don't have to fill out anything in this function, but you should spend some time to understand the output plots for debugging purposes.
- *bezier_curve*: uses p0, p1, p2, p3, and t, which are values used in the cubic Bézier curve equation. You will have to fill in the formula (but it is given above).
- *generate_bezier_waypoints*: see the previous section for the explanation. You don't have to fill out anything in this function, but you should spend some time understanding its purpose and functionality.
- *plan_curved_trajectory*: takes in a target position as an (x, y) tuple. Here you'll have to initialize some tf_2 variables (when did we last see that? hint: a previous lab), do a transform to orient us to the right frame, and get our target pose in a format that we want. You can play around with the `offset` and `num_points` parameters here, too.

Once you're done, your code should be able to generate some trajectories and nice plots (you can test by running `python3 src/plannedcntrl/plannedcntrl/trajectory.py` for now)! You won't be able to move the Turtlebot yet, though (that's the next part).

Checkpoint 2

Submit a checkoff request for a staff member to come and check off your work. At this point you should be able to:

- Show your path planner plot.
- Explain what a Bézier Curve is and how we can use it for trajectories.
- Show your submitted Feedback Form!

4 Control

4.1 What is PID Control

PID (Proportional-Integral-Derivative control) is a type of feedback control system widely used in industrial control systems and various other applications requiring continuously modulated control. A PID controller continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms. Watch the PID video from MathWorks if you want to learn more.

4.2 Components of PID Control

1. **Proportional (P)**:
   - The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant known as K_p, the proportional gain constant.
   - Formula:
   $$P_{\text{out}} = K_p \times \text{error} \tag{15}$$
2. **Integral (I)**:
   - The integral term is concerned with the accumulation of past errors. If the error has been present for an extended period of time, it will accumulate (integral of the error), and the controller will respond by changing the control output in relation to a constant K_i known as the integral gain.
   - In this Turtlebot lab we recommend setting K_i to 0 to make the PID controller a PD controller, as the **I** term here is more likely to cause issues than help.
   - Formula:
   $$I_{\text{out}} = K_i \times \int \text{error}\, dt \tag{16}$$
3. **Derivative (D)**:
   - The derivative term is a prediction of future error, based on its rate of change. It provides a control output to counteract the rate of error change. The contribution of the derivative term to the overall control action is termed the derivative gain, K_d.
   - Formula:
   $$D_{\text{out}} = K_d \times \frac{d(\text{error})}{dt} \tag{17}$$

The combined output from all three terms is computed as:

$$\text{Output} = P_{\text{out}} + I_{\text{out}} + D_{\text{out}} \tag{18}$$

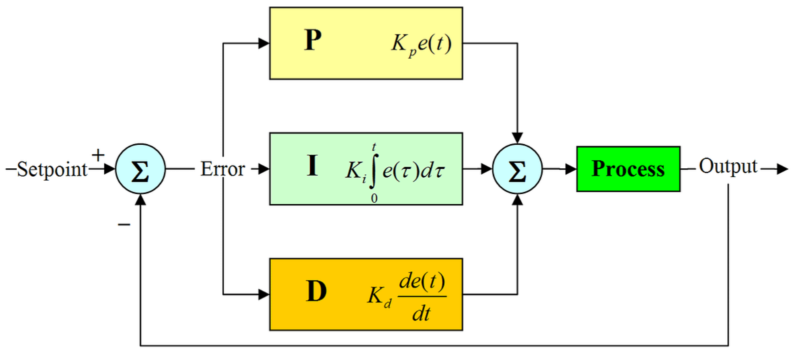


Figure 3: Hmmm, where have I seen this before

4.3 Advantages of PID Control

- Versatility: PID controllers can be used for a wide range of applications.
- Stability: Properly tuned PID controllers can provide stable control for many processes.
- Improved transient response: The controller can reduce the overshoot and settling time of a system.

4.4 Challenges

- Requires tuning: The K_p, K_i, and K_d values need to be properly set for optimal performance, which can sometimes be a complex task.
- Not suitable for all processes: Some systems might not benefit from one or more of the PID terms.

4.5 Coding

What we're doing here isn't perfectly conventional PID control, but it builds off of what we've done in Lab 4 previously. But now, we have to make sure we can dynamically track `each` waypoint from our list of waypoints, and move on whenever we successfully come close enough to a waypoint.

4.5.1 Turtlebot Dynamics

A Turtlebot uses a unicycle model which only turns in terms of x direction (forward) and yaw (rotation around the z axis), so your error should be in terms of yaw instead of y as we have no way of directly controlling it on the Turtlebot.

5 Fullstack Robotics

Now we can have Perception, Path Planning and Control! Time to put it to the test. Open up a new terminal and run the following. **Note:** Everytime you run the move the Turtlebot with the full stack commands, please rerun the Turtlebot bringup sequence to reset the position of the `odom` (world) frame.

1. **Important: ctrl+c out of all running processes and close their terminals**
2. Open another terminal to ssh into the Turtlebot with

   ```
   ssh fruitname@fruitname
   ```

   Login with the password `fruitname2022`
3. In ssh run

   ```
   ros2 launch turtlebot3_bringup robot.launch.py
   ```

4. Launch the realsense node

   ```
   ros2 launch usb_cam usb_cam.launch.py
   ```

5. Run your custom Turtlebot PID (realistically PD) controller

   ```
   ros2 run plannedcntrl turtlebot_control
   ```

6. Now everytime you run

   ```
   ros2 run perception cone_publisher
   ```

   your Turtlebot will find the cup and navigate towards it!

7. Everytime you navigate to the cup or move the Turtlebot, restart the Turtlebot bringup sequence. This makes sure the `/odom` frame is reset to always be coincident (essentially the same as) the `/turtlebot` frame. This makes sure our trajectory is not wildly off when starting a run.

```
ros2 launch turtlebot3_bringup robot.launch.py
```

Checkpoint 3

Submit a checkoff request for a staff member to come and check off your work. At this point you should be able to:

- Show Fullstack robotics in action!
- Submit the feedback form.
- Sign up for 106/206B!
