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

EECS/ME/BIOE C106A/206A Fall 2023

Goals

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

- Build a perception pipeline to detect the position of a brightly colored 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

---

*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 Realsense Camera

As you can see, the turtlebots now have Realsense cameras on them! As a refresher, Realsense cameras are RGBD cameras, meaning that they provide us with an RGB image and a depth image where each pixel on the depth image is how far away it is from the camera. More specifically, Realsense D435i cameras have both an RGB camera to get color and then two more cameras to create a separate stereo depth camera setup to get depth. Just like humans need two eyes to perceive depth, the Realsense needs two stereo cameras to perceive depth (don't believe me; close one of your eyes)! The Realsense then aligns the image feed from the RGB camera and depth image from the stereo depth cameras to have the full RGBD image

2.2   Starting up the Realsense Camera

The realsense camera by Intel connects to lower level camera drivers in something called the RealSense SDK that actually run the camera. Intel developed a ROS bridge that connects this SDK to ROS to spin up ROS nodes. Since we are running the camera on the Turtlebot's Raspberry Pi, we should ssh into the Turtlebot and launch the camera there. Open another terminal to ssh into the Turtlebot with

```
ssh fruitname@fruitname
```

Login with the password `fruitname2022`. In ssh run

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

Now **in SSH**, Using the Realsense ROS bridge, we can bring up several camera nodes at once with a launch file.

```
roslaunch realsense2_camera rs_camera.launch mode:=Manual color_width:=424 \
color_height:=240 depth_width:=424 depth_height:=240 align_depth:=true \
depth_fps:=6 color_fps:=6
```

Note: If you receive "No Realsense Devices Found" error, unplug the realsense for 10 seconds, kill the realsense launch file and then replug the camera and restart the launch file. Other errors such as "Resource temporarily unavailable, number X" and "Asic Temperature value is not valid" are fine and can be ignored. Here we are setting the resolution of the image, `color_width` and `color_height`, along with the resolution of the depth image, `depth_width` and `depth_height`, to the lowest resolution settings the Realsense allows so the Raspberry Pi on the Turtlebot does not crash. The `align_depth` parameter aligns the depth image to the RGB image. This is crucial when overlaying

depth data on RGB data, ensuring that when we map RGB pixels to depth points, the mapping is correct. Lastly, `depth_fps` and `color_fps` are set to 6 frames per second to save compute on the Turtlebot's Raspberry Pi. You can view your new image by running rviz= 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 Realsense camera is working correctly, you should be able to add a new **Image** object under `/camera/color/image_raw`. You should now be able to see your Realsense image feed!

2.3   Finding a Goal Position

Now lets convert the camera frame coordinates into world frame coordinates so our path planner can use it later. Recall in ROS coordinate frames are linked together by transformations in TF trees. In this case, we have a TF tree between `odom` and `base_footprint` and another separate TF tree containing `camera_link`, which is the camera frame. We need to be able to define the goal positions that the camera detects in the world frame, so we need to indirectly establish a link between the two. To connect the trees we can run:

```
rosrun tf static_transform_publisher 0 0 0 0 0 0 base_footprint camera_link 100
```

to define a transformation between the `base_footprint` and the `camera_link`. In this case, we are assuming the frames are coincident with a 0 transform. This command publishes a static 0 transform at 100Hz (100 times a second) which makes the `base_footprint` frame a parent of the `camera_link` frame in the TF tree.

2.4   The Camera Instrinsics

The Realsense camera is constantly publishing it's camera intrinsics (the K matrix) to the topic `/camera/color/camera_info` which we have define as a class variable for you. Fill in the `camera_info_callback()` function to properly index into this K matrix. To see how to index it view the ROS Docs for the CameraInfo.msg.

2.5   Finding the Cup: Color Thresholding

Color thresholding is one of the simplest yet most effective methods for object detection in images, especially when the object of interest has a distinct color. We will use color thresholding and locate a cup based on its color.

2.5.1   Basics of Color Thresholding

Color thresholding works by segmenting an image based on pixel values. For instance, if we know our cup is blue, we can define a range of blue shades and filter out all other colors from the image. The result is a binary image where the pixels within our defined range are set to white (255) and all other pixels are set to black (0). Some limitations:

1. Sensitivity to Lighting: Changes in lighting can affect the perceived color of objects. While HSV alleviates this to some extent, drastic lighting changes can still pose challenges.

2. If there are other objects with a color similar to the cup, they might also be detected.

2.5.2   Convert to the Right Color Space:

While our images are usually in RGB (Red, Green, Blue) color space, it's often beneficial to represent our image in HSV (Hue, Saturation, Value) color space when doing color thresholding. HSV is more robust to lighting changes and allows us to focus on the 'Hue', which is a more accurate representation of 'color'.

2.5.3   Thresholding

Define a range (lower and upper bound) for the Hue, Saturation, and Value channels that capture the color of the cup. You can use the code we provide to find the HSV value of the cup. Move the cup to take up the entirety of the realsense image. Then in code, we are printing of the HSV values of the average of the center row of the image in the function `process_images()`. Now using the mean printed to the terminal, hardcode this mean value as the color of the cup add some margin (around -50) to define your bound below this value (we already gave you the upper bound), as there is noise and variation in the real world! You may have to expand this range more if you can't get the color filter to work. Using the defined range, create a binary mask where pixels within the range are set to white and everything else is set to black.

2.5.4   Post-Processing

After thresholding, it's common to have some noise in the image and random points that make it past the color filter. While there are better ways to filter out this noise, we can simply take the mean position of all points left in the mask and call that the center of our cup, which works reasonably well.
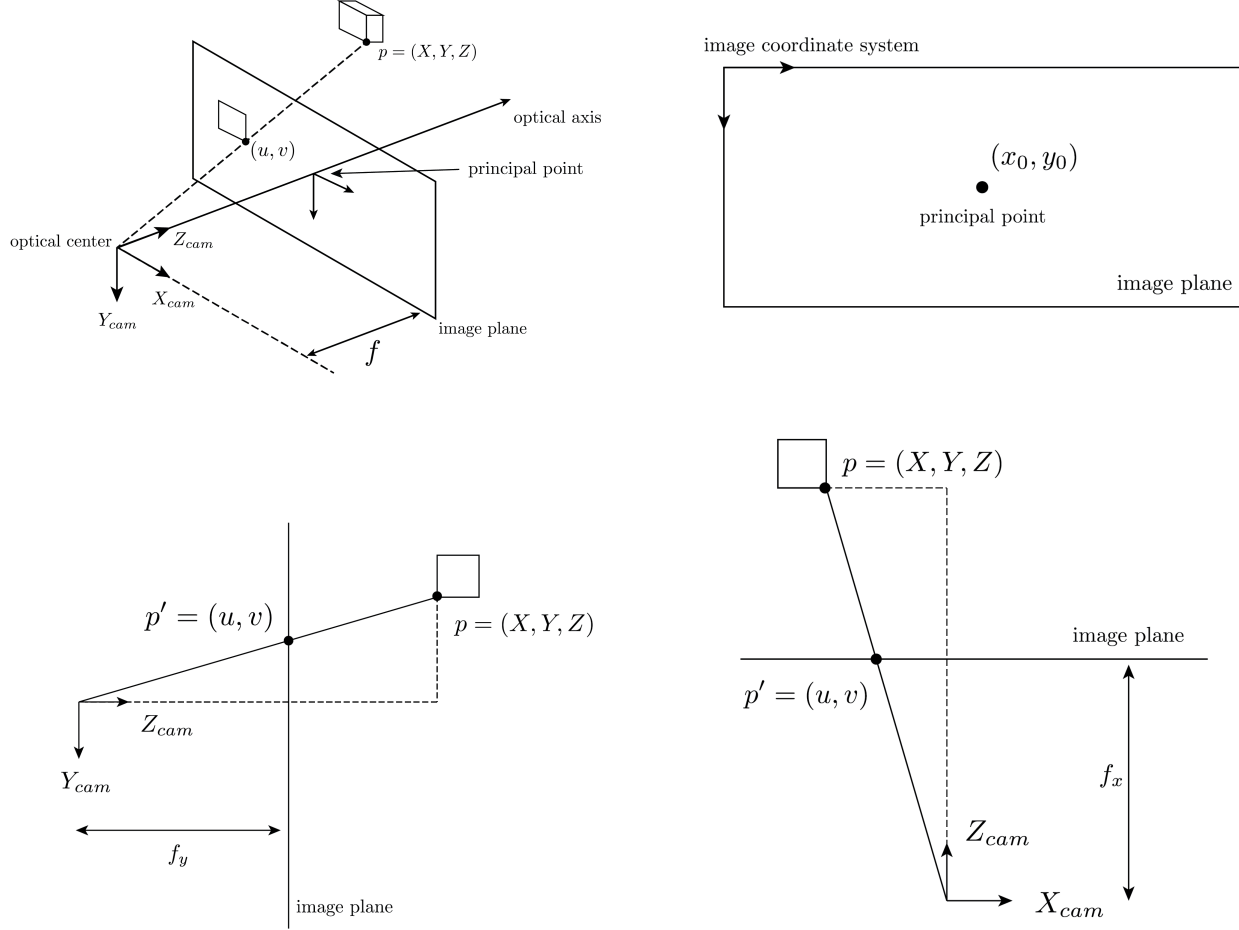
2.6   Camera Projection

Figure 1: Geometry behind a Pinhole Camera

Given $X_{cam}, Y_{cam}, Z_{cam}$ and the camera intrinsics, we can convert pixels on the image u, v into X, Y where

$$X = \frac{(u - x_0) \times Z}{f_x} \tag{1}$$

$$Y = \frac{(v - y_0) \times Z}{f_y} \tag{2}$$

2.7   How did we Derive the Projection Equations

Place the optical center of the camera at the origin of the 3D space and let $P_{\text{world}} = (X_{\text{cam}}, Y_{\text{cam}}, Z_{\text{cam}})$. The image plane is positioned at $Z = f$, where f is the focal length. Using similar triangles, the relationship between world coordinates and image coordinates can be derived. For the x-coordinate:

$$\frac{X_{\text{cam}}}{Z_{\text{cam}}} = \frac{u - x_0}{f_x} \tag{3}$$

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

Rearranging, you get:

$$X_{\text{cam}} = \frac{(u - x_0) \times Z_{\text{cam}}}{f_x} \tag{4}$$

Similarly, for the y-coordinate:

$$\frac{Y_{\text{cam}}}{Z_{\text{cam}}} = \frac{v - y_0}{f_y} \tag{5}$$

Which rearranges to:

$$Y_{\text{cam}} = \frac{(v - y_0) \times Z_{\text{cam}}}{f_y} \tag{6}$$

2.8 The Intuition

The term $\frac{Z_{\text{cam}}}{f_x}$ provides a scaling factor. When the 3D point is closer to the camera (small Z_{cam}), its image appears larger on the image plane and vice versa. The subtraction of x_0 and y_0 (or 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.

The realsense camera publishes its camera intrinsics (focal length etc.) to a rostopic. Find this topic and use it's data to convert pixels into points using the camera intrinsics. You can now run

```
rosrun perception object_detector.py
```

Just like how we added an **Image** object in RVIZ when we started the Realsense camera, if you open the `/detected_cup` ROS topic in RVIZ (`rviz` in the terminal to launch) as an **Image** object you will be able to see your thresholding algorithm running live! If you are getting an error about no `odom` frame existing, make sure your Turtlebot bringup and static transform are running.

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 that your color filter is working properly by displaying it in RVIZ.

- Demonstrate that your object detection in RVIZ is working (you should see a green dot on the $/detected_cup =$ $topicExplainwhatanRGBDcameradoes.$

---

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 with 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.
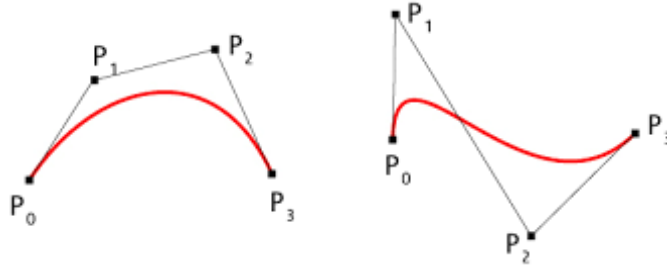
Figure 2: Cubic Bézier curves

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:

$$B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t)t^2 P_2 + t^3 P_3 \tag{7}$$

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:

$$P_1 = \begin{bmatrix} x_1 + cos(\theta_1) * \text{offset} \\ y_1 + sin(\theta_1) * \text{offset} \end{bmatrix} \tag{8}$$

$$P_2 = \begin{bmatrix} x_2 - cos(\theta_2) * \text{offset} \\ y_2 - sin(\theta_2) * \text{offset} \end{bmatrix} \tag{9}$$

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:

$$\theta_i = \arctan\left(\frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) \tag{10}$$

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 `rosrun plannedcntrl trajectory.py` for now)! You won't be able to move the turtlebot yet, though (that's the next part).

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.

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:
   $$P_{\text{out}} = K_p \times \text{error} \tag{11}$$

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.

   - Formula:
   $$I_{\text{out}} = K_i \times \int \text{error} \, dt \tag{12}$$
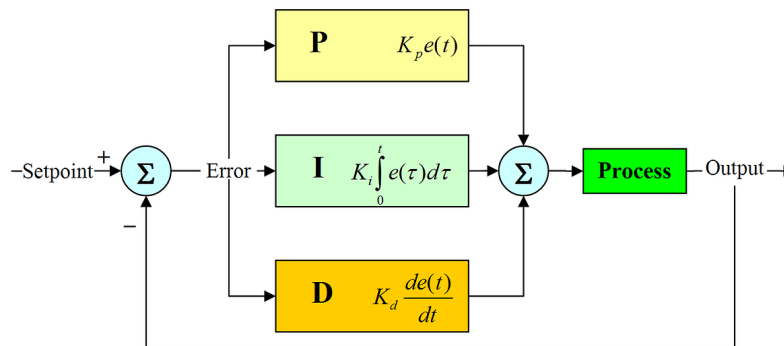
3. **Derivative (D)**:

Figure 3: Hmmm, where have I seen this before

- 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:

$$D_{\text{out}} = K_d \times \frac{d(\text{error})}{dt} \tag{13}$$

The combined output from all three terms is computed as:

$$\text{Output} = P_{\text{out}} + I_{\text{out}} + D_{\text{out}} \tag{14}$$

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. Start from the given starter code for *turtlebot_control.py*.

4.5.1 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 not be in terms 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. Open a new terminal and run `roscore`

2. Open another terminal to ssh into the Turtlebot with

```
ssh fruitname@fruitname
```

Login with the password `fruitname2022`

3. In ssh run

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch --screen
```

4. Launch the realsense node

```
roslaunch realsense2_camera rs_camera.launch mode:=Manual color_width:=424 \
color_height:=240 depth_width:=424 depth_height:=240 align_depth:=true \
depth_fps:=6 color_fps:=6
```

5. Create the static transform in the TF tree with

```
rosrun tf static_transform_publisher 0 0 0 0 0 0 base_footprint camera_link 100
```

6. Now everytime you run `rosrun perception object_detector.py` your Turtlebot will find the cup and navigate towards it!

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!

- Sign up for 106/206B!