AA 274: Principles of Robotic Autonomy
Problem Set 2: Perception
Due February 13

Problem 1

In this problem, the objective is to estimate the intrinsic parameters of a camera, which will allow you to accurately project any point in the real world onto the pixel image output by the camera.

To accomplish this, we be using a popular method proposed in Z. Zhang, "A Flexible New Technique for Camera Calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2000. This method uses images of a known pattern on a 2D plane, such as a chessboard, captured at different positions and orientations. By processing at least 3 (and preferably many more) images, the camera parameters can be often be accurately estimated using a direct linear transformation as described in lecture.

In performing this calibration, it will be important to keep the relevant coordinate frames in mind (the paper by Zhang will be the main reference, so note any differences in notation):

- (X, Y, Z) A point in a world coordinate system
- (x, y) Ideal, distortion-free, normalized image coordinates
- (u, v) Ideal, distortion-free, pixel image coordinates
- $(\breve{x}, \breve{y})$ Real, distorted, normalized image coordinates
- $(\breve{u}, \breve{v})$ Real, distorted, pixel image coordinates

The observed points we extract in the $(\breve{u}, \breve{v})$ frame for calibration can be denoted by $(u_{\text{meas}}, v_{\text{meas}})$.

The scripts `cam_calibrator.py` and `cal_workspace.py` are given to provide a framework for the calibration. You will be adding methods to the `CameraCalibrator` class and testing them by modifying and running `cal_workspace.py`.

Task: Run `cal_workspace.py` to view the chessboard images we will be processing. The corner grid is 7×9 and the side length of each square is $d_{\text{square}} = 22.5$ mm. The corner locations $(u_{\text{meas}}, v_{\text{meas}})$ for each chessboard are extracted for you using OpenCV.

1. Modify `genCornerWorldCoordinates` to generate the world coordinates (X, Y) for each corner in each chessboard. It is important that the ordering corresponds exactly to the points in $(u_{\text{meas}}, v_{\text{meas}})$!
2. Next modify `estimateHomography`, using the singular value decomposition (SVD) method outlined in Appendix A of [1] to estimate the homography matrix H for each chessboard.
3. Use SVD again in `getCameraIntrinsics` to estimate the linear intrinsic parameters of the camera, using the homographies H. These parameters should be packed into a single matrix A. As a sanity check, the skewness parameter γ should be small ($|\gamma| \ll \alpha$) and the principal point (u_0, v_0) should be near the center of the image pixel dimensions.

4. Next modifying getExtrinsics, use your estimated A and the H for each chessboard to estimate the rotation R and translation t of each chessboard when the images were captured. (Note that your initial R estimates will likely not be genuine rotation matrices! Once again, SVD comes to the rescue — see Appendix C in [1] for details.)

5. We are now in a position to create some important coordinate transformations. Implement transformWorld2NormImageUndist and transformWorld2PixImageUndist in order to switch from (X, Y, Z) to (x, y) or (u, v) in the undistorted image frames. It will be helpful to make use of homogeneous and inhomogeneous coordinates.

    - Now we can check to see how well we are doing! Pass your estimated camera matrix A and chessboard extrinsic parameters R and t into the plotBoardPixImages function (leave the k argument unspecified) to see where your calibration is mapping the corners, compared to the original measurements.
    - As a second check, pass your extrinsic parameters to plotBoardLocations to see the estimated locations and orientations of the chessboards relative to the camera.

6. We finish our camera calibration by estimating radial distortion parameters k. This can be done using least-squares, as outlined in Section 3.3 of [1]. (The previous transformations may prove useful here!) Modify estimateLensDistortion for this final part of the calibration, and fill in the final transforms transformWorld2NormImageDist and transformWorld2PixImageDist to make use of k. Note that here we are only estimating two parameters, but the D matrix can easily be augmented to estimate higher orders of radial distortion as well as the location of the center of radial distortion, which is not in general identical to the principal point (u_0, v_0).

    - Finally we reach the true payoff! First, check the new movement of the corners, once again using plotBoardPixImages, but this time including k as an argument.
    - Now pass your estimated camera parameters A and k into the undistortImages function to apply your calibration to the original chessboard images. You should be able to compare the original (left) to the image after applying A (center) and finally after adjusting for radial distortion (right). Feel free to experiment with removing images from webcam_12 folder to see how the calibration deteriorates. (however, your submission should of course be based on the full set of images)

In ROS, the camera calibration parameters you have just calculated are often sent to a set_camera_info service broadcast by the package running a given camera. They are then packed into a .yaml file in a standard location from which they can be automatically loaded whenever the camera starts. Pass your calibration parameters into the writeCalibrationYaml function to generate this configuration file.

Submission: You will be submitting your code in cam_calibrator.py and cal_workspace.py as well as your generated configuration file webcam_calibration.yaml.

Problem 2

In this problem, you will implement a line extraction algorithm to fit lines to (simulated) Lidar range data. Consider the overhead view of a typical indoor environment:

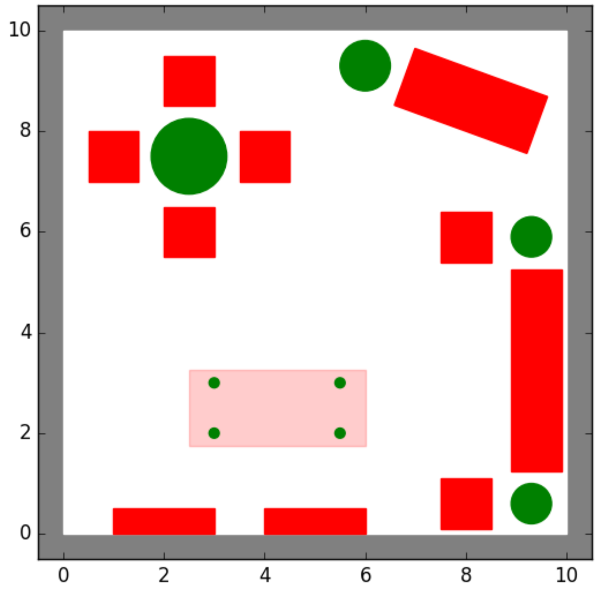


Figure 1: 2D Schematic of a typical 10 m x 10 m indoor environment.

A mobile robot needs to explore and map this room using only a (2D) LIDAR sensor, which casts equally spaced laser beams from the center of the robot to form a 360° view. The first step in mapping is to extract meaningful information from the range measurements. *Line Extraction* is a common technique used to fit a series of straight line segments to range data in an attempt to define the border of objects in the environment.

Line Fitting

A range scan describes a 2D slice of the environment. Points in a range scan are specified in a polar coordinate system with the origin at the location of the sensor. It is common to assume that the noise on measurements follows a Gaussian distribution with zero mean, some range variance and negligible angular uncertainty. We choose to express a line using polar parameters (r, α) as defined by the line equation (1) for the Cartesian coordinates (x, y) of the points lying on the line

$$x \cos\alpha + y \sin\alpha = r, \tag{1}$$

where $-\pi < \alpha \leq \pi$ is the angle between the x-axis and the shortest connection between the origin and the line. This connections length is $r \geq 0$ (see Figure 2). The goal of line fitting in polar coordinates is to minimize

$$S = \sum_i^n d_i^2 = \sum_i^n (\rho_i \cos(\theta_i - \alpha) - r)^2 \tag{2}$$

for the n data points in the set. The solution of this least squares problem gives the line parameters:

$$\alpha = \frac{1}{2}\arctan\left(\frac{\sum_i^n \rho_i^2 \sin 2\theta_i - \frac{2}{n}\sum_i^n\sum_j^n \rho_i\rho_j \cos\theta_i \sin\theta_j}{\sum_i^n \rho_i^2 \cos 2\theta_i - \frac{1}{n}\sum_i^n\sum_j^n \rho_i\rho_j \cos(\theta_i + \theta_j)}\right), \qquad r = \frac{1}{n}\sum_i^n \rho_i \cos(\theta_i - \alpha) \tag{3}$$

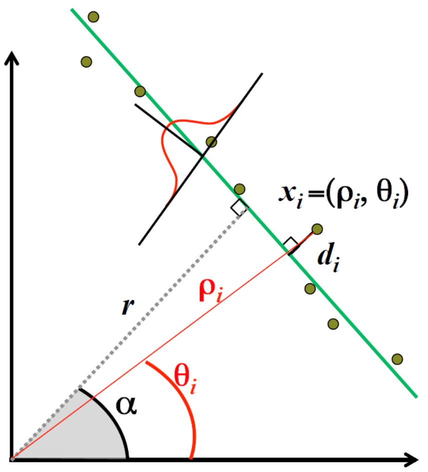


Figure 2: In polar coordinates, a line fitted to data (θ_i, ρ_i) can be uniquely defined by (α, r). We make the assumption that there is Gaussian noise on the range measurement (ρ_i) but none in the angle (θ_i).

Line Extraction

There are many algorithms that have been successfully used to perform line extraction (e.g. Split-and-Merge, Line-Regression, RANSAC, Hough-Transform, etc.). Here, we will focus on the "Split-and-Merge" algorithm, which is arguably the fastest, albeit not as robust to outliers as other algorithms. See Algorithm 1 below and Section 4.7.2.1 in the textbook [2] for more details.

Algorithm 1: Split-and-Merge

```
Data: Set $S$ consisting of all $N$ points, a distance threshold $d > 0$
Result: $L$, a list of sets of points each resembling a line
$L \leftarrow (S), i \leftarrow 1$;
while $i \leq len(L)$ do
    fit a line $(r, \alpha)$ to the set $L_i$;
    detect the point $P \in L_i$ with the maximum distance $D$ to the line $(r, \alpha)$;
    if $D < d$ then
        $i \leftarrow i + 1$
    else
        split $L_i$ at $P$ into $S_1$ and $S_2$;
        $L_i \leftarrow S_1$; $L_{i+1} \leftarrow S_2$;
    end
end
Merge collinear sets in $L$;
```

The scripts `ExtractLines.py` and `PlotFunctions.py` are provided to structure and visualize the line extraction algorithm. You will be modifying/adding functions in `ExtractLines.py` to perform the Split-and-Merge line extraction.

There are three data files provided, `rangeData_<`x_r`>_<`y_r`>_<`n_{pts}`>.csv`, each containing range data from different locations in the room and of different angular resolutions, where `<`x_r`>` is the x-position of the robot (in meters), <y_r> is the y-position, and <$n_{\texttt{pts}}$> is the number of measurements in the 360° scan. The provided function `ImportRangeData(filename)` extracts `x_r`, `y_r`, `theta`, and `rho` from the csv file. Figure 3 illustrates these three data sets.

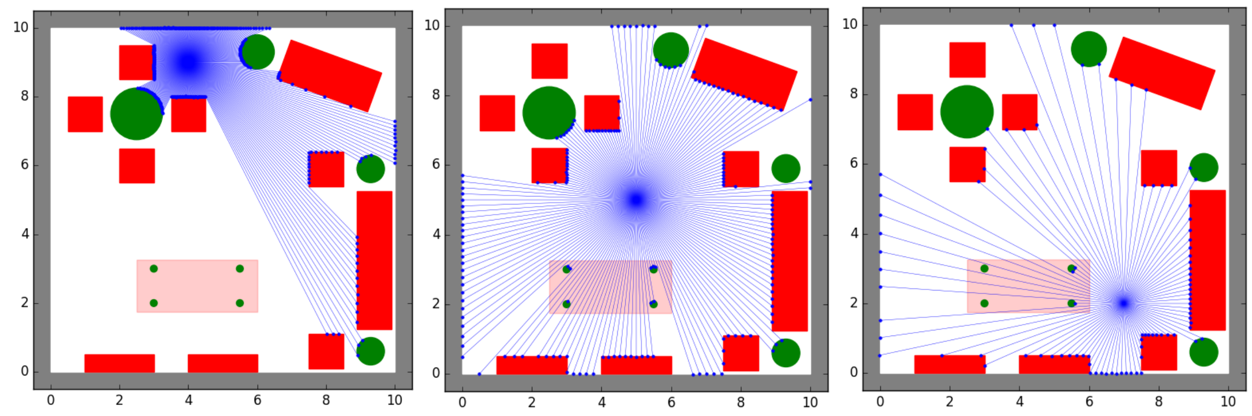


Figure 3: Lidar range data for three different locations in the room and three different resolutions, corresponding to `rangeData_4_9_360.csv`, `rangeData_5_5_180.csv`, and `rangeData_7_2_90.csv`, respectively

Task: For each of the three data sets, run `ExtractLines.py` to extract line segments from the data and plot them on the map. The main `ExtractLines` function has been provided for you. Your job is to populate `SplitLinesRecursive`, `FindSplit`, `FitLine`, and `MergeColinearNeigbors` functions. More details can be found in the script comments.

There are three suggested parameters to control segmentation:

- `LINE_POINT_DIST_THRESHOLD`: The maximum distance a point can be from a line before the line is split
- `MIN_POINTS_PER_SEGMENT`: The minimum number of points per line segment
- `MIN_SEG_LENGTH`: minimum length of a line segment

These parameters act as knobs you can tune to better fit lines for each set of range data. You are welcome to add other parameters/constraints as you see fit.

NOTE: There is not one correct answer to this problem. Slightly different implementations of the algorithm may produce different lines. However, *better* results will, of course, smoothly fit the actual contours of the objects in the room and minimize the number of false lines (e.g. that jump between objects). Also feel free to edit the `ExtractLines` function or any of the plotting in `PlotFunctions.py` if you'd like.

Submission: You will submit the three plots showing the extracted lines, the number of lines extracted for each, and the modified `ExtractLines.py`.

Problem 3

To be released:
This problem will give you some experience with visual recognition via neural networks. The text will be provided on Thursday (February 2nd), after the associated lecture.

Problem 4

Let's now get back to building our stack for the turtlebot. In the context of ROS, a stack usually means a collection of nodes and other software that is used on a robot. In this problem, you will be asked to implement the part of your stack that will allow the turtlebot to reach specific targets, like a charging station, using a monocular camera.

Before we start writing more features for our robotic stack, we need to start breaking it down into modules by making good use of the ROS pub/sub architecture.

Task: Modify your `controller.py` from problem set 1 so that instead of having a hard-coded goal state, it subscribes to a topic called `/turtlebot_control/position_goal` using the `Float32MultiArray` message (from the `std_msgs.msg` package). Look into the ROS documentation for how to use that message.

Now in order to enable your robot to accomplish complex tasks, we need to implement a simple finite state machine (FSM) that we will also use to send goals to our pose controller.

Task: Place the provided `supervisor.py` script inside your scripts directory (don't forget to make it executable). Modify the supervisor node to publish simple position goals to your controller node over the topic `/turtlebot_control/position_goal`.

We now have two nodes in our stack: one of them sending pose setpoints; the other one receiving them, running a pose controller, and sending commands to the robot. We could simply run both nodes at the same time, but in order to make our stack easier to use, let's write a launch file that will allow us to start both nodes using a simple command.

Task: Create a `launch` folder in your package (at the same level than your scripts directory). Inside that folder, create a file called `turtlebot.launch`. Include the content below in it. Then fill in the missing fields so that the launch file starts both your controller and your supervisor. The field 'type' refers to the name of your script.

```
<launch>
  <node pkg= FILL  type= FILL  name="turtlebot_controller"/>
  <node pkg= FILL  type= FILL  name="turtlebot_supervisor"/>
</launch>
```

Fiducials are markers that are used to make computer vision tasks easier. Roboticists have used them to simplify difficult computer vision tasks in many environments. Notably, Amazon Robotics (formerly Kiva Systems) has used them on the floors of their warehouses in order to guide autonomous forklifts.

We will now use a package that leverages a very similar algorithm to the one you implemented in problem 2 to detect the pose of a fiducial and use it to guide our robot to a goal location.

First let's update some of our software.

Task: There is a bug in Gazebo that could affect our task. It was luckily fixed last fall by the community. We will need that fix in order to properly simulate our camera. Upgrade your gazebo by running:

```
$ curl -ssL http://get.gazebosim.org | sh
```

Task: We also added necessary code to the `asl_turtlebot` package. Update the `asl_turtlebot` package using git:

```
$ cd ~/catkin_ws/src/asl_turtlebot
$ git pull
```

Task: Install the `apriltags_ros` package by cloning it in your `catkin_ws/src` folder and building it using catkin.

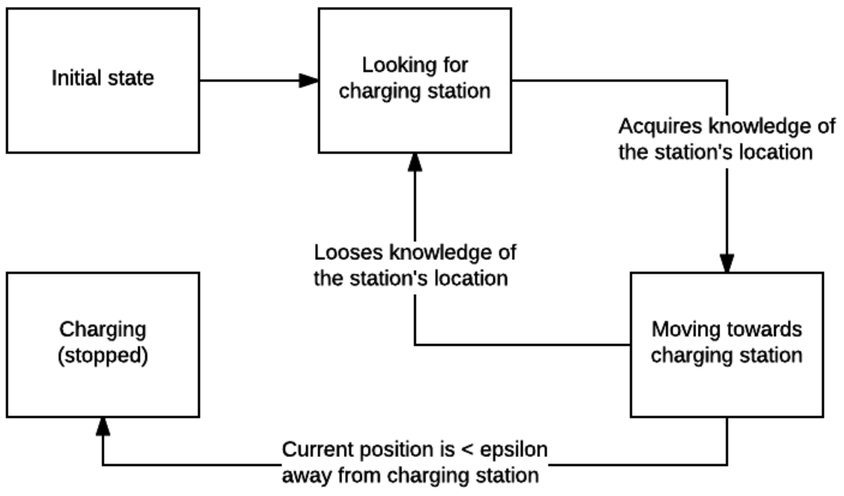
```
$ cd ~/catkin_ws/src
$ git clone https://github.com/RIVeR-Lab/apriltags_ros
$ cd ~/catkin_ws
$ catkin_make
```

You should now be able to launch a simulation containing the turtlebot and a large fiducial as well as your software stack.

```
$ roslaunch asl_turtlebot turtlebot_april_sim.launch
$ roslaunch turtlebot_control turtlebot.launch
```

Task: Finally, implement a state machine in your supervisor. The state machine should make your robot look for the charging station (the box with the large fiducial on it), go to it once it finds it, and stop moving once it is close enough (0.3 meters).

You are given significant flexibility in this implementation task. We encourage you to discuss architectures with your classmates. There are many solutions to this problem. The only requirement is that your state machine implements the following specification:


Initial state
Looking for charging station
Acquires knowledge of the station's location
Looses knowledge of the station's location
Charging (stopped)
Moving towards charging station
Current position is < epsilon away from charging station

Keep in mind that you will likely be reusing your supervisor and controller nodes as part of the final project. There are also many ways for you to decide what the robot will do while in the state "Looking for charging station", the most obvious one being to spin on itself until the robot sees the fiducial.

Notes on the FSM: The action of looking for the charging station might require you to send open-loop angular velocities to the robot. This can conflict with your `controller.py`. One way around this is to add two subscribers to your controller. One of them subscribing to a `/turtlebot_control/velocity_goal` topic (`Float32MultiArray`), and the other subscribing to a `/turtlebot_control/control_mode` topic (for example a string topic like in the "Hello World" from problem set 1). The `control_mode` topic can then tell your controller whether it should listen to the setpoints published on `/turtlebot_control/velocity_goal` or on `/turtlebot_control/position_goal` at any particular moment. This is, however, only a suggestion.

Notes on the AprilTags: The specific package we decided to use to detect the position of fiducials for this homework is based on AprilTags `https://april.eecs.umich.edu/wiki/AprilTags`. You can read about the algorithm behind AprilTags in [3]. It combines techniques similar to the one you implemented earlier in order to detect lines in images as well as the one you implemented to calibrate your camera to narrow down the pose of the tag. AprilTags are very similar to the popular ARTags.

Notes on the tf library: You will notice that the supervisor uses a library called `tf` in order to get the position of the fiducial in the world. There is too much to this package for us to cover it in this homework, but feel free to look into its documentation `http://wiki.ros.org/tf`. What you need to know is that we have already set up the assignment so that there is a chain of transforms linking the global inertial frame to the frame of the tag by going through the camera frame. This chain gets updated every time the fiducial is detected in the camera frame. However those transforms have an expiration time. Therefore the `tf` library will stop being able to report the position of the fiducial once it has disappeared from the camera's field of view for some amount of time (not immediately!). Keep this in mind when implementing your finite state machine.

Submission: Create a rosbag recording with the position setpoints (`/turtlebot_control/position_goal`) and the transform tree (i.e. the topic called `/tf`) for a simulation of the robot reaching its charging station. You will also submit your `supervisor.py` and `controller.py`.

References

[1] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[2] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed. MIT Press, 2011.

[3] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proc. IEEE Conf. on Robotics and Automation*, 2011.