AA 274A: Principles of Robot Autonomy I
Problem Set 3: Perception
Due date: Tuesday, October 29

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/PrinciplesofRobotAutonomy/Course1_Fall19_HW3.git` in a terminal window.

For your final submission, you will submit a single pdf with your answers for written questions (denoted by the ✎ symbol) and the python files that contains your code for code questions (denoted by the 🖥 symbol).

NOTE: In this problem set, we'll be using a very popular open-source computer vision package named OpenCV. To install it, please execute `pip install opencv-python`

Problem 1: Camera Calibration

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 will 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 (there are a couple of versions online; use the version here: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.220.534`). 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 from Lecture 5):

- (X, Y, Z) A point in a world coordinate system (attached to each checkerboard)
- (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 editing methods in the `CameraCalibrator` class which `cal_workspace.py` calls. Please take a look at the code and see how `cam_calibrator.py` and `cal_workspace.py` interact before you begin.

NOTE: Any code for Problem 1 should be run within your VM or Docker (there's a dependency on a ROS component).

To take a look at the chessboard images that you will be processing, run `./cal_workspace.py`. The corner grid is 7×9 and the side length of each square is $d_{\text{square}} = 20.5$ mm. The corner locations $(u_{\text{meas}}, v_{\text{meas}})$ for each chessboard are extracted for you using OpenCV. You should see something like in Figure 1 (click on the image to go to the next image).

Let's begin!

Note: You do not need to include these images in your write-up (unless you want to for your own future reference). While grading, we will run your code and these images should be generated.

(i) Modify `genCornerCoordinates` 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 $(u_{\text{meas}}, v_{\text{meas}})$!

(ii) 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.

(iii) 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.

(iv) Next modify `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.)

(v) You 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.

(a) Now you can check to see how well you 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. Refer to Figure 2 to see what the expected results should be. (Click on the image to move to the next image.)

(b) As a second check, pass your extrinsic parameters to `plotBoardLocations` to see the estimated locations and orientations of the chessboards relative to the camera. Refer to Figure 3 to see what the expected results should be. (Hit enter in the terminal to move to the next image.)

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.

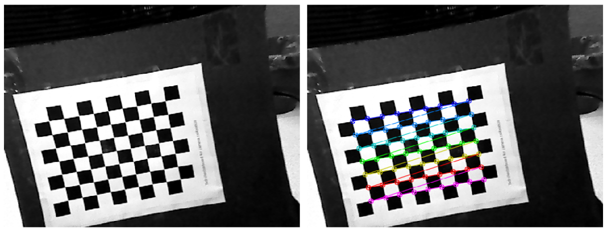

Figure 1: Corner extraction of chessboards.

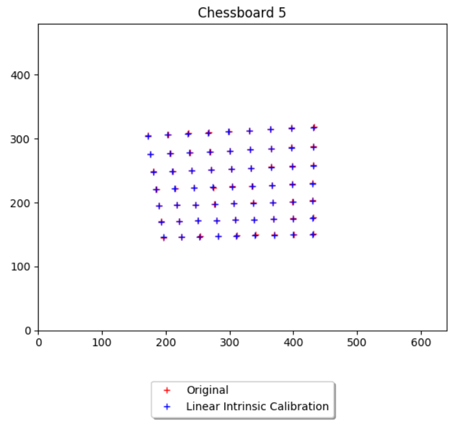

Figure 2: Example plots for (v)(a).

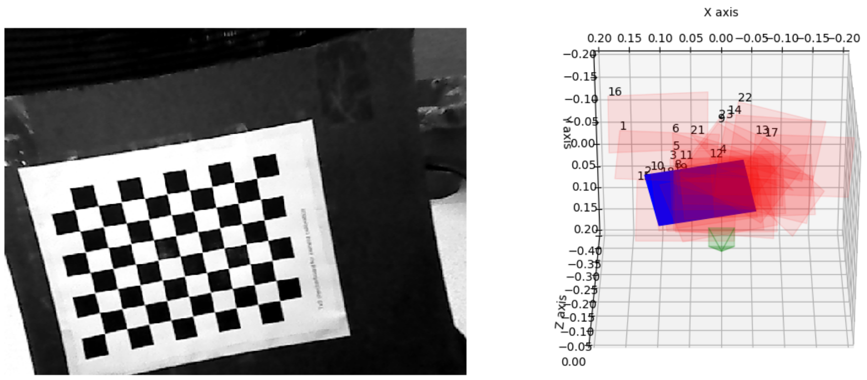

Figure 3: Example plots for (v)(b).

Problem 2: Line Extraction

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:

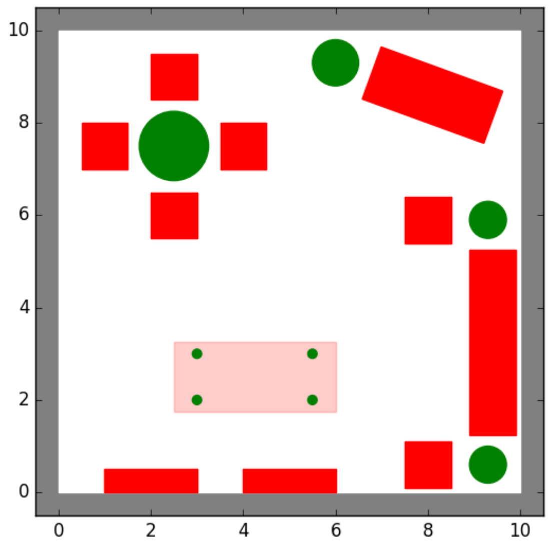

Figure 4: 2D Schematic of a typical 10 m x 10 m indoor environment.

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 connection's length is $r \geq 0$ (see Figure 5). The goal of line fitting in polar coordinates is to minimize

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

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

$$\alpha = \frac{1}{2}\text{arctan2}\left(\frac{\sum_i^n \rho_i^2 \sin 2\theta_i - \frac{2}{n}\sum_i^n \sum_j^n \rho_i \rho_j \cos\theta_i \sin\theta_j}{\sum_i^n \rho_i^2 \cos 2\theta_i - \frac{1}{n}\sum_i^n \sum_j^n \rho_i \rho_j \cos(\theta_i + \theta_j)}\right) + \frac{\pi}{2}, \qquad r = \frac{1}{n}\sum_i^n \rho_i \cos(\theta_i - \alpha) \tag{3}$$

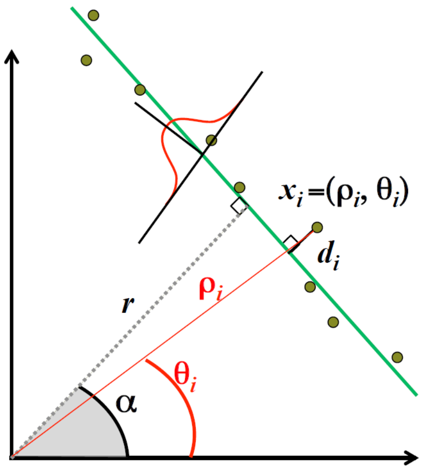

Figure 5: In polar coordinates, a line fitted to data (θ_i, ρ_i) can be uniquely defined by (α, r). We make the assumption that there is Gaussian noise on the range measurement (ρ_i) but none in the angle (θ_i).

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: Euclid's algorithm for finding the greatest common divisor of two nonnegative integers

```
1  function SplitLinesRecursive ($\theta, \rho, a, b$);
   Input  : $\theta \in \mathbb{R}^N$, $\rho \in \mathbb{R}^N$, start index $a \in \mathbb{N}$, end index $b \in \mathbb{N}$
   Output: $\alpha \in \mathbb{R}^M$, $r \in \mathbb{R}^M$, line indices $i \in \mathbb{N}^{M\times 2}$
2  $\alpha, r \leftarrow$ FitLine($\theta_{a:b}$, $\rho_{a:b}$);
3  if $b - a \leq$ MIN_POINTS_PER_SEGMENT then
4      return $\alpha, r, (a, b)$;
5  end
6  $s \leftarrow$ FindSplit($\theta_{a:b}$, $\rho_{a:b}$, $\alpha$, $r$);
7  if $s$ is not found then
8      return $\alpha, r, (a, b)$;
9  end
10 $\alpha_1, r_1, i_1 \leftarrow$ SplitLinesRecursive($\theta$, $\rho$, $a$, $a + s$);
11 $\alpha_2, r_2, i_2 \leftarrow$ SplitLinesRecursive($\theta$, $\rho$, $a + s$, $b$);
12 return $(\alpha_1, \alpha_2), (r_1, r_2), (i_1, i_2)$;
```

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`>_<`$n_{\texttt{pts}}$`>.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 6 illustrates these three data sets.

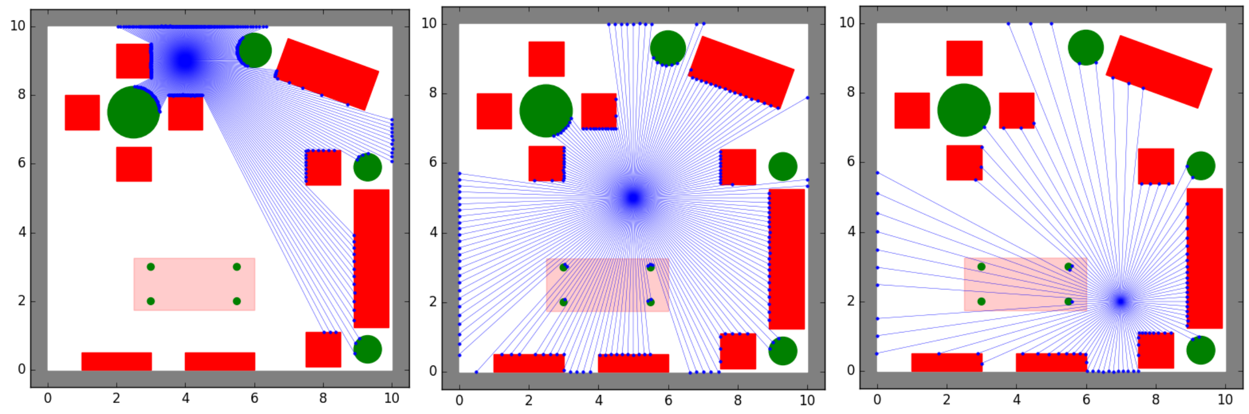

Figure 6: Lidar range data for three different locations in the room and three different resolutions, corresponding to `rangeData_4_9_360.csv`, `rangeData_5_5_180.csv`, and `rangeData_7_2_90.csv`, respectively

(i) 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 the `SplitLinesRecursive`, `FindSplit`, `FitLine`, and `MergeColinearNeigbors` functions. More details can be found in the script comments.

There are four 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`: The minimum length of a line segment
- `MAX_P2P_DIST`: The maximum distance between two adjacent points in 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.

(ii) Submit three plots showing the extracted lines for each of the data sets; include your segmentation parameter choices with each plot.

Problem 3: Linear Filtering

The field of computer vision includes processing complex and high dimensional data (up to millions of pixels per image) and extracting relevant features that can be used by other components in a robotic autonomy stack. Nowadays, many computer vision techniques rely on deep learning and machine learning algorithms for classification and depend heavily on computational tools that can efficiently process, learn, and do inference on the data. However, these methods can be quite computationally expensive to use (often requiring GPU acceleration). What else can we do if we have a limited computational budget? How did robots perceive and detect objects in their environments before the advent of machine learning?

In this problem you will familiarize yourself with Linear Filtering, an important precursor to Template Matching.

(i) As a warm up, let us analyze how grayscale filters affect grayscale images when correlated with them. Recall from lecture that the **correlation** operator $G = F \otimes I$ is

$$G(i,j) = \sum_{u=0}^{k} \sum_{v=0}^{\ell} F(u,v) \cdot I(i+u, j+v)$$

The correlation operator takes as input an image $I \in \mathbb{R}^{m \times n}$ and filter $F \in \mathbb{R}^{k \times \ell}$ and produces an image $G \in \mathbb{R}^{m \times n}$. In words, G is an image whose pixels are a locally-weighted sum of the original image I's pixels (weighted by the entries in F). Knowing this, if I is

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

manually compute G as the result of $G = F \otimes I$ when F is:

(a) $F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

(b) $F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

(c) $F = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$. For this example, also add a one sentence explanation as to what this filter is doing to the image, and why this might be useful in computer vision. If you cannot answer this right away, come back to this after you've implemented correlation and run your code with this filter!

(d) $F = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$. For this example, also add a one sentence explanation as to what this filter is doing to the image, and why this might be useful in computer vision. If you cannot answer this right away, come back to this after you've implemented correlation and run your code with this filter!

Note: Along edges, we implicitly zero-pad the input image I so that the output G has the same height and width as I (this is commonly known as "same" padding).

(ii) We'll now switch gears to correlation with color images. Prove that correlation can be written as a dot product of two specially-formed vectors. Specifically, find the vectors $\mathbf{f}$ and $\mathbf{t}_{ij}$ such that

$$G(i,j) = \sum_{u=0}^{k} \sum_{v=0}^{\ell} \sum_{w=0}^{c} F(u,v,w) \cdot I(i+k, j+\ell, w) \text{ can be written as } G(i,j) = \mathbf{f}^T \mathbf{t}_{ij}$$

(iii) Implement the correlation operation within the `corr` function in `linear_filter.py` which takes in a color filter F and color image I and outputs G, where $G = F \otimes I$. Specifically, implement it using the dot product result you showed above (otherwise it will be too slow). Remember to implement zero-padding to maintain correct sizes! Run your code with the filters in `linear_filter.py` on the provided image `test_card.png` and include the resulting images and an explanation as to what each filter is doing in your submission.

(iv) Typical implementations of correlation on these sizes of images and filters run in less than 20 ms. Report the runtime of your correlation function, how does it compare? Suggest two ways it could be sped up (no need to actually implement these, also, only reason about the grayscale case for these suggestions).

HINT: Do we need to compute each of the pixels in G sequentially?

HINT: What is the runtime of the correlation operator as-is (how many multiplication and addition operations need to be performed)? What happens if the filter you use can be written as an outer product, i.e. $F = \mathbf{f}\mathbf{f}^T$, where $\mathbf{f} \in \mathbb{R}^{k\times 1}$ and $F \in \mathbb{R}^{k\times k}$? As it turns out, many filters have this property!

(v) How could you obtain the vector $\mathbf{f}$ in the second hinted speed-up method above (i.e. $F = \mathbf{f}\mathbf{f}^T$)? Give a mathematical condition on F that enables this speed-up method to be used. This problem does not require solving the previous one and, again, only reason about the grayscale case.

(vi) We've seen that correlation can be written as a dot product, and, from linear algebra/geometry we know that the dot product can be viewed as a measure of similarity between two vectors (think about why this makes sense). Thus, the value of $G(i,j)$ obtained from correlation can be viewed as a measure of similarity between the filter and the image at that location. As a result, by correlating F across an input image I we can find regions in I which are similar to F. This is "Template Matching", we are literally trying to "match" a "template" F somewhere inside an input image I. After correlating the filter across the image, we simply choose the highest values of $G(i,j)$ and center a filter-sized bounding box there, yielding our detection.

While correlation alone will work, unfortunately the value it gives is hard to threshold (i.e. to identify when a detection is made), as the output value $G(i,j)$ has a scale that is dependent on the content of the image (and thus changes per image). One way of fixing this is to normalize the correlation value (making a perfect match yield a value of 1). This is called **normalized cross-correlation**. Formally,

$$G(i,j) = \frac{\mathbf{f}^T \mathbf{t}_{ij}}{\|\mathbf{f}\| \cdot \|\mathbf{t}_{ij}\|}$$

Implement this within the `norm_cross_corr` function in `linear_filter.py`

(vii) You might have heard the term "convolution" being thrown around recently. Convolution is similar to correlation; it is also linear operator that, from input matrices F and I, creates an image $G = F * I$. Specifically,

$$G(i,j) = \sum_{u=0}^{k} \sum_{v=0}^{\ell} F(u,v) \cdot I(i-u, j-v)$$

Explain how convolution can be implemented with correlation.

Problem 4: Template Matching

Building on the previous problem, in this problem you will familiarize yourself with Template Matching, a classic computer vision technique for detecting specific objects within an image.

(i) Fill in `template_match` wintin `template_matching.py` and call it with the template `valdo.png` to find Valdo, Waldo's buff cousin, within the provided `clutter.png` image. Include the detection image with your submission.

HINT: OpenCV has functions for this! `cv.matchTemplate(img, template, method=cv2.TM_CCORR_NORMED)` will perform template matching with `template` on `img` with normalized cross-correlation. Note that it will only perform the cross-correlation for you, you will still have to threshold the output to find matches!

(ii) Looks like template matching works pretty well when the template is taken straight out of the image. How does it generalize? What if we wanted to take a template of an object from one image, to detect that object in other images? Run your template matching implementation with the stop sign template `stop_signs/stop_template.jpg` on the other images available in the `stop_signs/` directory. Use these detection images to help answer the following question.

(iii) How does template matching perform on other images? Suggest two ways (excluding scaling) to improve template matching.

Problem 5: Stop Sign Detection and FSM in ROS

This problem will be done in groups. Once assigned a group number, you can TurboVNC into your group account on `genbu.stanford.edu`. DO NOT try to log into another group's account.

IMPORTANT: You will need to pull new files to your `asl_turtlebot` package. Go to the package directory and make sure that no files show up when you type

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

If there are any files that are listed as having been changed, you should check them out so that they do not conflict with the new files.

```
$ git checkout thefilethatyouchanged.py
```

Now you can pull the new changes.

```
$ git pull
```

If the above fails, delete the `asl_turtlebot` directory and clone the `asl_turtlebot` repository again (https://github.com/StanfordASL/asl_turtlebot).

There is probably no need to rebuild `asl_turtlebot` since there are no new custom message types that need to be compiled, but you may want to rebuild just in case.

```
$ cd ~/catkin_ws
$ catkin_make
```

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 the problem, you will be asked to implement a part of your stack that will allow the Turtlebot to move towards a goal and *stop* if it sees a stop sign on the way there. This is similar to what you would expect an autonomous car to do on the road.

This is an example of how your perception algorithms (computer vision/object detection) would be used to inform your path planning and control modules.

Here is a description of the new code you have just pulled and also a bird eye's view of what this problem is about.

- `pose_controller.py` The controller node. This is a ROS node wrapper for the `PoseController` class you wrote in homework 1.
- `detector.py` The image detector node. We have pretrained a Convolutional Neural Network (CNN) to classify Gazebo stop signs and calculate bounding boxes for them [1]. However the CNN detector can be slow, so it is disabled by default and the node instead uses a simple color thresholding to detect stop signs (IMPORTANT: this second detection method will NOT work on the real robot or in any more complicated scenario, but we opted to also implement this in order to reduce the computational burden of running the simulation - your real Turtlebots will be running the CNN though so poke at it!). This node takes in camera images, camera information and laser scan information in order to estimate the location of the stop sign in the Turtlebot's reference frame.
- `supervisor.py` The supervisor node. This will encode your Finite State Machine (FSM) for your robot. Your robot will be in different *states* and your state machine will encode how your robot may transition between these states.

[1] In general, when you are deploying robots in the real world, you will need to train a network on images relevant to your robot. Fortunately, there exists pretrained CNN classifiers (Inception, MobileNet) where you need to only train the last few layers for your specific dataset. In this case, we have collected a dataset of Gazebo stop signs and trained the network using those.

- `turtlebot3_signs.launch` The launch file that will launch Gazebo (with the Turtlebot3 and a stop sign), `detector.py` and `pose_controller.py`. By default, this launch file does not start Gazebo's graphical interface (which can be enabled by passing `gui:=true` to the launch file) and instead starts a simplified 2D visualizer `gazebo_plot.py`. The detector should also start another window, which corresponds to the camera images being sent by the robot (it should look mostly gray). Feel free to have a poke around this launch file to get an idea of how launch files work. It is likely you may write your own as part of your final project.

The Turtlebot is receiving raw camera images[2] from the (sophistcated white box) camera we have placed on it and from our detector node, it can identify and detect stop signs. It is also receiving camera information reminiscent of Problem 1. Using this information, we can estimate the direction of the stop sign relative to the Turtlebot and using laser scan information, estimate how far away the stop sign is. We can then act intelligently with the knowledge that there is a stop sign at some known relative position from the robot.

In this problem, you will be drawing upon your knowledge from Problem 1 and also design your own state machine for the Turtlebot to obey traffic rules (stop at a stop sign before moving past it.) The goals of the problem are:

- Incorporate perception into your decision-making.
- Learn about and use Finite State Machines (FSM) to command your robot to execute non-trivial tasks.
- Gain more familiarity with ROS by working with multiple ROS nodes that publishes/subscribes to multiple topics.
- Learn about different ROS messages and how to use them.

(i) Take a look at `supervisor.py`. What information does it publish? What is the message type?

(ii) Copy your `P2_pose_stabilization.py` file from HW1 to `~/catkin_ws/src/asl_turtlebot/HW1`. Then modify `pose_controller.py` so that it subscribes to a topic of desired goal poses and feeds these poses to your `PoseController` to compute control commands. The required information for this is in the comments.

(iii) Fortunately, the simulated camera provides its own calibration so you do not have to implement Problem 1 again (we promised you would have to go through the pain only once!). But you want to extract the necessary information from this calibration in order to transform objects from pixel coordinates to a heading angle. This information is encoded in the `CameraInfo` ROS message. Read the documentation `http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html` to help you extract the focal lengths and principal points in the `camera_info_callback` function in `detector.py`. More information is given in the code. Pay close attention to the dimension of the field you may be accessing in the message.

(iv) Given the focal lengths and principal points, edit `project_pixel_to_ray` in `detector.py` to compute the ray/direction (as a unit vector) of the stop sign in the camera's frame of reference. You may find Figure 7 helpful (note that in our case u in the code would be $\tilde{x}$ in the diagram, cx in the code would be $\tilde{x}_0$ in the diagram and that we have different focal lengths for each axes of this camera).

You will now implement a FSM for your Turtlebot. A FSM (`https://en.wikipedia.org/wiki/Finite-state_machine`) is defined by a finite list of states, and conditions that describe how you can transition in and out of a state. You must also specify your initial state. This can be viewed as a mathematical model for your logic.

[2]We have simulated a camera in Gazebo.

Initially, the Turtlebot should just be navigating to its desired pose (in `POSE` mode). If it sees a stop sign though, it must then stop for *3 seconds* (see `STOP_TIME` variable) when it is sufficiently close to it (approximately 0.5 meters - see `STOP_MIN_DIST` variable). Then it shall "cross the intersection" (i.e. move while ignoring any stop sign it might be seeing) and then eventually transition back into its regular pose seeking state (the mode denoted as `POSE`).

(vi) Draw out the state diagram for a FSM respecting the description above and that you think you could implement. Think carefully about what the transitions should be and think whether or not the Turtlebot will get "stuck" in an undesirable loop.

(vii) Code up your FSM. Note that if you run the code before editing `supervisor.py`, the Turtlebot will just move towards $(x_g, y_g, \theta_g) = (1.5, -4, 0)$ without stopping at the stop sign (traffic violation!). To help you get started, we have given you a template of what a FSM could look like and a list of states that we think may be useful (see the `Mode` class). From this template, you would need to implement two new states (look at the `loop` method) and one callback (look at the `stop_sign_detected_callback` method) for it to accomplish the task. However, you are free to modify the code beyond this.

To test your FSM and see how your turtlebot performs, type:

```
$ roslaunch asl_turtlebot turtlebot3_signs_sim.launch gui:=true
```

This one command will run Gazebo, `pose_controller.py`, `detector.py` and `gazebo_plot.py`. The `gui:=true` flag enables Gazebo rendering. You should also see a window that simulates what the camera is seeing, which should be nothing initially. This might take a few seconds to load.

To test `supervisor.py`, in a new terminal window, run `supervisor.py` (remember how to run a node from homework 1?). If done correctly, your Turtlebot should start moving. The terminal window in which you are running the supervisor should also be printing mode changes.

(viii) Take a screenshot of the path and velocity profile that should get drawn in one of the windows. We should be able to see the turtlebot navigate to its goal location, and most importantly see the 3 seconds of stopping in the velocity profile (corresponding to stopping at the stop sign).

$$\tilde{x} = f\,\frac{X_C}{Z_C} + \tilde{x}_0, \qquad \tilde{y} = f\,\frac{Y_C}{Z_C} + \tilde{y}_0,$$

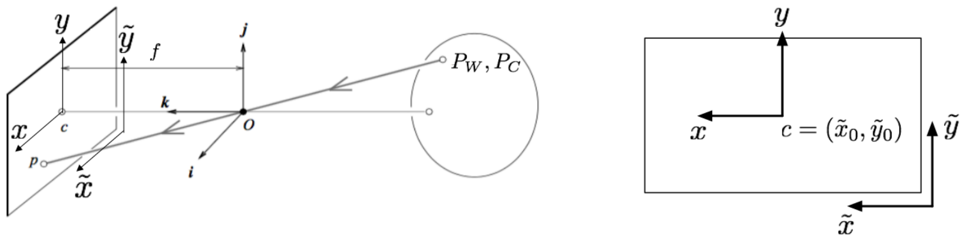

Figure 7: Projecting a point from pixels to camera frame

Running on your local machine (optional): If you want to try running this code on your local non-Linux machine, you'll need to run the launch file without `gui:=true` since Gazebo is too slow inside virtual machines. The procedure for running with Docker is as follows. The procedure for running with the Windows VM is the same, just without a need to prepend commands with `./run.sh` or `./run.sh --display 1`

1. Update the Docker image.

```
$ cd aa274-docker
$ git pull
$ ./build_docker.sh
```

2. **Terminal 1:** Run roscore.

```
$ ./run.sh roscore
```

3. **Terminal 2:** In a new terminal window (leave `roscore` running), run the launch file with the plot rendered to display 1. You will be able to view this display through TurboVNC. You can use any display number you want (doesn't have to be 1); this just changes the TurboVNC hostname you connect to in the next step.

```
$ ./run.sh --display 1 roslaunch asl_turtlebot turtlebot3_signs_sim.launch
```

   Enter in a password for your TurboVNC session.

4. **TurboVNC:** Connect to display 1 with the server hostname `localhost:1` and the password you set above.

5. **Terminal 3:** Run the supervisor to start the Turtlebot.

```
$ ./run.sh rosrun asl_turtlebot supervisor.py
```

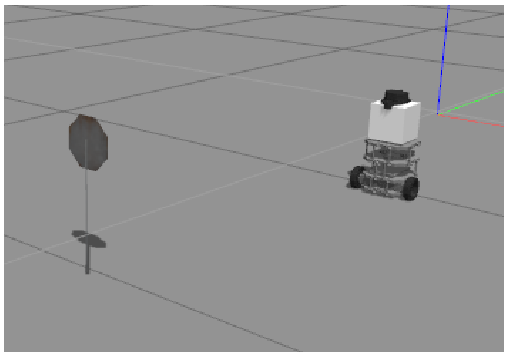

Figure 8: Turtlebot and stop sign in Gazebo.

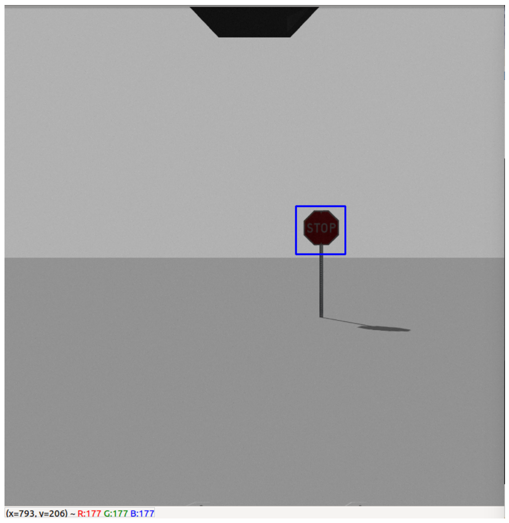

Figure 9: Camera's view of the stop sign.

Extra Problem: Image Pyramids

We now have a working template matching implementation! However, there is a big catch that we have not addressed yet. What if the object in the image is much larger or much smaller than our template? What if it is not exactly in our plane of view? Template matching alone cannot address these. We'll now dive into addressing these challenges.

The main topic we'll be focusing on in this question is image pyramids. The core idea of image pyramids is scaling an image up (zooming in) and down (zooming out). Thus, if there is an object that matches our template in the image, even if it has a different scale than our template, we'll detect it.

(i) As a start, let us implement scaling down as selecting every even-indexed row and column to create a 1/2-sized image. Fill in `half_downscale` within the `image_pyramids.py` file. Test your code by passing in ndarrays of varying shape, checking that their downscaled shapes match your expectations.

(ii) Call `half_downscale` three times on `test_card.png` to get a 1/8-sized image. Include this image in your submission. Comment on its quality, why did we lose so much detail compared to the original?

(iii) A method to fix this is to first blur the image before downscaling it. First, explain why this will help fix the issue we saw above. Then, implement this in `blur_half_downscale` within `image_pyramids.py`.

Note: You do not need to use your code from Problem 3. Actually, we encourage you not to as it is better to get exposure to OpenCV's methods. To blur an image before downsampling, use `cv2.GaussianBlur(image, ksize=(5, 5), sigmaX=0.7)`

(iv) Call `blur_half_downscale` three times on `test_card.png` to get a 1/8-sized image. Include this image in your submission. Compare this image to the original test card and the one downsampled with `half_downscale`.

(v) Great! Now let's focus on the other end: upscaling. As a start, let us implement scaling up as repeating every row and column twice to create a 2x-sized image. Fill in `two_upscale` within the `image_pyramids.py` file. Test your code by passing in ndarrays of varying shape, checking that their upscaled shapes match your expectations.

HINT: `numpy.repeat` is a helpful function here.

(vi) Call `two_upscale` three times on `favicon-16x16.png` to get an 8x-sized image. Include this image in your submission. Comment on its quality, why does it look so blocky?

(vii) To fix this, we can use a more sophisticated method to upscale images: bilinear interpolation. As the name implies, it linearly interpolates in both the x and y dimensions to produce an upscaled image.

On the web, there are many long and complicated formulas for computing bilinear interpolation. However, we are going to ignore them and perform bilinear interpolation simply via correlation! To do this,

1. Create an image which is the desired output size.
2. Copy the original image's pixels to every n pixels in the output image, the rest being zeros.
3. Return G where $G = F \otimes I_{\text{scaled}}$ and the filter F is a provided 2D tent function (search this online to see why it is called this).

For example, if we were upscaling an image I by a factor of 3, the process would look like this

$$I = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow I_{\text{scaled}} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \end{bmatrix}$$

$$G = F \otimes I_{\text{scaled}} = F_{\text{provided}} \otimes \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 1.33 & 1.67 & 2 \\ 1.67 & 2 & 2.33 & 2.67 \\ 2.33 & 2.67 & 3 & 3.33 \\ 3 & 3.33 & 3.67 & 4 \end{bmatrix}$$

Implement bilinear interpolation this way in `bilinterp_upscale`. Call it with `scale=8` on `favicon-16x16.png` to get an 8x-sized image. Include this image in your submission and compare it to the one upscaled with `two_upscale`. Finally, explain intuitively why this method works, i.e., why does this yield a bilinearly-interpolated image? If you need a hint, print out the filter in the code.

Note: You do not need to use your correlation function from Problem 3 to perform the last correlation step. Instead, use OpenCV's `filter2D` method like so: `cv2.filter2D(image_scaled, -1, filt)`

(viii) Finally, we can implement full template matching with Gaussian image pyramids. Specifically, you'll be combining what you made in the previous problems into one algorithm for object detection and classification:

1. Create a Gaussian image pyramid of the originally given image. A "Gaussian image pyramid" is simply a series of up and downscaled versions of the image (called so because it looks like a pyramid when visualized in 3D).
2. Perform template matching on each image in the Gaussian pyramid.
3. Return a detection if the normalized cross-correlation score is greater than a given threshold.

Implement this within `scaled_template_matching.py` and run it on the image and template specified within. Include your detection image with your submission.

HINT: OpenCV has many functions for this! `cv2.pyrUp(img)` and `cv2.pyrDown(img)` can be used to scale `img` up or down by a factor of 2 (applying Gaussian blur along the way). `cv.matchTemplate(img, template, method=cv2.TM_CCORR_NORMED)` will perform template matching with `template` on `img` with normalized cross-correlation. Note that it will only perform the cross-correlation for you, you will still have to threshold the output to find matches! You can also use your solution from Problem 4 to do the template matching part.

(ix) Run your template matching with Gaussian image pyramids algorithm with the stop sign template `stop_signs/stop_template.jpg` on the other images available in the `stop_signs/` directory. How does it perform on these images?

The behavior you're seeing at the end of this problem is one of the main reasons why modern robotic perception algorithms don't use such algorithms anymore, including your Turtlebots!

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.