MAE 4180/5180, ECE 4772/5772, CS 3758

Autonomous Mobile Robots: Lab #1

Introduction to AMR's iRobot Create

Instructor:
Dr. Hadas Kress-Gazit

Objectives

In this lab, students will:

- Control the iRobot Create platform
- Change between coordinate frames
- Model sensor error
- Observe actuation error
- Observe the timing effects of obtaining different sensor information
- Perform feedback linearization for controlling a differential drive robot

This lab is designed to familiarize students with the iRobot Create platform that will be used for the remainder of the course. The Create is a differential drive (non-holonomic) robot equipped with several different sensors. For homework assignments and testing outside the lab, a simulator has been developed so you can debug your code without a physical robot. The simulator is designed to use the same functions as the MATLAB Toolbox for the iRobot Create (pages 10–11 of `Matlab_Toolbox_iRobot_create_doc.pdf` posted on Canvas).

Important note about saving files: Make sure to save your data frequently to a shared google drive, or better yet, have your working Matlab directory be on the USB drive.

Required Code (from Homework 1) and plots (to be shown to the TAs at the beginning of the lab)

- `turnInPlace.m`
- `readStoreSensorData.m`
- `OpenLoopControl.m` (From the Lab 1 assignment on Canvas)
- `backupBump.m`
- `limitCmds.m`
- `visitWaypoints.m`
- `feedbackLin.m`
- A plot of the robot trajectory when following waypoints [-1 0; 1 0] (last part of the question about Feedback Linearization)

1 Lab Manual

1.1 Set-up - Remote student

(a) Join the lab Zoom session (link on Canvas).

(b) Open the Twitch stream: https://www.twitch.tv/cu_mae_amr

(c) (One remote group member) create and share a Google drive or Box folder with the rest of the group. Add subfolders for each group member to put their code in.

(d) Put all the required files in the shared folder.

1.2 Station Set-up - In-Person student

(a) Join the lab Zoom session (link on Canvas) on the **lab computer** and share your screen.

(b) Open the shared folder created by one of the online group members. Add your files.

(c) Create a local folder on the lab computer and copy all the files there.

(d) Open Matlab and change the working directory of Matlab to be the folder that contains your files. **Make sure to periodically save data to your online folder.**

(e) Unplug your iRobot Create. Make sure to only pick up the robot from the *bottom*– not the sensor platform. **Be careful not to hit the markers on the top of the robot.** Put it on the floor next to your lab station.

(f) Take note of the name of your robot.

(g) Turn on the robot by pressing the power button. The Raspberry Pi takes about 20-30 seconds to boot.

(h) In your MATLAB Command Window, run `Robot = CreatePiInit('robotName')` where `robotName` is the robot name from step (f). The initialization process creates the variable `Robot`, which contains the port configurations for interfacing with the Create and the added sensors and the robot name; it has five elements:

- `Robot.Name` contains the robot name.
- `Robot.OL_Client` used for getting the robot pose ground truth from the Optitrack system.
- `Robot.CreatePort` used for sending commands and reading sensors from the robot.
- `Robot.DistPort` used for getting depth information from the realsense camera.
- `Robot.TagPort` used for getting tag information from the realsense camera.

(i) Check that you have connected to the robot properly by running `BeepRoomba(Robot.CreatePort)`. You should hear your robot beep.

(j) Put your robot on the field. Make sure you can access the robot's localization information by running `[x,y,theta] = OverheadLocalizationCreate(Robot)`.

(k) Put the robot in front of a tag, at least one foot away. Run `RealSenseTag(Robot.TagPort)` and `RealSenseDist(Robot.DistPort)`. Make sure you can get tag and depth information.

(l) If any of steps i-k fail, disconnect from the robot (run `CreatePiShutdown(Robot)`), shut it down, close Matlab, restart Matlab, turn the robot on and go back to step h.

Important:

- If the control function exits with an error, make sure to stop the robot by typing in the command window: `SetFwdVelAngVelCreate(Robot.CreatePort, 0, 0)`

- When you are done working with the robot, or you wish to restart Matlab or the connection to the robot, first run `CreatePiShutdown(Robot)` to disconnect properly from the robot.
- **Make sure to alternate the group member that is typing in the Matlab window.** Switch between every section of the lab manual. Remote students: this will be done by remote screen sharing.

1.3 Robot Motion

In this part of the lab you will collect data and observe how accurately the robot moves.

(a) The function `OpenLoopControl.m` controls the robot to perform several different motions - first straight line motions at different speeds, and then rotations at different angular speeds. Place the robot such that it is aligned with one of the global axes and it has **at least 10 feet free** in front of it. Run the function. **Make sure your robot has enough space to run the sequence of motions without running into the walls or other robots**.

(b) In the MATLAB Command Window, access the overhead localization data by typing: `global poseData;` (or just call the function with `poseData` as the output). You should have 12 rows. Save to a file.

(c) Edit the function `OpenLoopControl.m` to create different motions, for example, you can try

`SetFwdVelAngVelCreate` for a fixed time. Collect and save the pose data and the commanded actions.

(d) In the lab report you will be asked to compare the expected motion (based on the function) and the actual motion (based on the overhead localization). Make sure you have data for both. You will notice that sometime the overhead localization does not provide data; this will appear as NaN in the output. Make sure you have enough data to complete the lab report, and if you do not, run the functions again starting at a different initial pose.

1.4 RealSense Sensor Measurements

The realsense data: In this class, we process the RealSense information to create two types of measurements: depth and beacon (also referred to as tag) relative position. The depth information contains 9 points going from left (the angle is 27 deg in the sensor frame) to right (the angle is -27 deg in the sensor frame). The depth information is taken from the middle of the camera image.

Processing time of the RGBD information is not negligible. The Raspberry Pi continuously runs a loop that requests an image from the camera and then calculates depth and tag information. The information received by the depth and tag functions also contains the time elapsed since the image was taken.

In this part, you will find the sensor frame location in the robot frame and observe possible error sources for the depth and tag information.

(a) Choose a beacon (AprilTag).

(b) Position the robot such that the sensor is facing the beacon and the **robot center** is 2 ft away from the beacon. It is important that the center of the robot is aligned with the center of the beacon and the robot X-axis is perpendicular to the beacon plane.

(c) Read the beacon location using `RealSenseTag(Robot.TagPort)`. What are the different elements of the resulting measurement? What units are the measurements in? (Hint: you can type `help RealSenseTag`). You will use the measurements corresponding to the x and y position of the tag.

(d) Calculate the difference between the distance measured on the floor and the distance along the x-axis measured by `RealSenseTag(Robot.TagPort)`. This is the offset in the x-axis. The offset in the y-axis is the value measured by `RealSenseTag(Robot.TagPort)` in the y-axis.

(e) Read the distance information using `RealSenseDist(Robot.DistPort)`. What are the different elements of the resulting measurement? What units are the measurements in? (Hint: you can type `help RealSenseDist`). Save the middle measurement (angle 0). Repeat the measurement a few times and average the distance results. Write down the measurements and the average.

(f) Position the center of the robot 3 ft away from the beacon. Read the beacon location. Repeat the measurement and calculation. The x-axis offset will be the average of the two x-axis offsets, and the y-axis offset will be the average of the two y-axis offsets. Make sure to write down the offsets - you will use them in the lab report.

(g) Read the distance information. Save the middle measurement (angle 0). Repeat the measurement a few times and average the distance results. Write down the measurements and the average.

(h) Place the robot next to a wall, facing the opposite wall, along the width of the map, such that the robot X-axes is perpendicular to the wall. Read the distance information. Save the middle measurement (angle 0). Repeat the measurement a few times and average the distance results. Write down the measurements and the average.

1.5 Collecting Sensor Data

In this part of the lab you will collect sensor data and observe how delays and sensors data frequency depend on the functions you use.

(a) Edit `turnInPlace.m`, set the angular velocity `cmdW` to 0 and comment out the `Pause` function (the robot should remain stationary, and gather data). Place the robot such that it sees a corner and at least one beacon. Run `turnInPlace.m` for 30 seconds.

(b) In the MATLAB Command Window, access the sensor data by typing: `global dataStore;` (or just call the function with `dataStore` as the output). Make sure all the sensor data has been properly recorded (pose, odometry, bump, depth, beacon) – you may want to plot it – and save to a file.

(c) Run `turnInPlace.m` (with `cmdW` set to 0 and no pause) for 30 seconds after modifying the functions as follows. For each run, make sure all the relevant data has been recorded and save to file.

  (i) In the function `readStoreSensorData.m` comment out all sensor readings except for the overhead localization.

  (ii) In the function `readStoreSensorData.m` comment out all sensor readings except for the odometry (both distance and angle).

  (iii) In the function `readStoreSensorData.m` comment out all sensor readings except for the bump.

  (iv) In the function `readStoreSensorData.m` comment out all sensor readings except for the camera functions – depth and beacon.

  (v) In the function `readStoreSensorData.m` uncomment all sensor readings (i.e. collect all the data) but in `turnInPlace.m` comment out the command that sets the velocity for the robot.

(d) Uncomment the command that sets the velocity for the robot. In the function `readStoreSensorData.m` comment out all sensor readings except for the camera functions – depth and beacon. Run `turnInPlace.m` (with `cmdW` set to 0 and no pause) for 30 seconds for the following cases. For each run, make sure all the relevant data has been recorded and save to file. To figure out how many beacons the robot detects, run `RealSenseTag(Robot.TagPort)` before running `turnInPlace.m`.

  (i) The robot does not detect any beacons.

  (ii) The robot detects one beacon.

  (iii) The robot detects at least two beacons.

(e) In the function `readStoreSensorData.m` uncomment all sensor readings (i.e. to be able to collect all the data).

1.6 Running Your Control Program: Bump

In this part of the lab you will run your control function `backupBump`, collect sensor data for analysis and observe the behavior of the robot.

(a) Open `backupBump.m` (from any of the group members), and ensure that it calls the function `readStoreSensorData.m` to gather and store sensor data.

(b) Using `limitCmds.m`, set the robot's maximum velocity to 0.1m/s.

(c) Run `backupBump.m` and allow the robot to "bump" into the walls a few times before terminating the program. Make sure you run your program when the robot is in view of beacons.

(d) In the MATLAB Command Window, access the sensor data by typing: `global dataStore;` (or just call the function with `dataStore` as the output). Make sure all the sensor data has been properly recorded (pose, odometry, bump, depth, beacon) – you may want to plot it – and save to file.

(e) Place the robot close to a wall and run `backupBump.m`. Observe the behavior.

(f) In the function `readStoreSensorData.m` comment out all sensor readings except for the bump sensor. Place the robot close to a wall, run `backupBump.m` and observe the behavior. Part 2.2 will ask about the difference in behavior and responsiveness, if any was observed.

(g) Uncomment the other sensors in `readStoreSensorData.m`.

1.7 Visiting Waypoints

In this part of the lab you will control the robot so that it visits a set of waypoints in the map and observe the effects of ϵ on the resulting trajectories.

(a) Open your function `visitWaypoints.m`, and set your waypoints appropriately (coordinate the way-points with the TA so several groups can run at the same time).

(b) Using `limitCmds.m`, set the robot's maximum velocity to 0.1m/s.

(c) Run `visitWaypoints.m` to have the robot visit the series of waypoints.

(d) Once the robot has followed its path, access the sensor data in the MATLAB Command Window by typing: `global datastore;` (or just call the function with `dataStore` as the output). Make sure all the sensor data has been properly recorded – you may want to plot it – and save to file.

(e) Repeat for each member of your group. Use the same waypoints, but change the value of ϵ that is used by the feedback linearization. **Make sure to save (or write down) the waypoint coordinates and the value of ϵ for each run.**

1.8 Clean up

When you are done collecting all the data and after you make sure you have everything that you need for the lab report:

(a) Run `CreatePiShutdown(Robot)` to disconnect from the robot.

(b) Turn off the robot, return it to the lab station and plug it in.

(c) Make sure to leave your lab station clean, without any papers or other items. Follow COVID-19 protocols for sanitizing your station.

2 Post-Lab Assignment

The group should submit one post-lab report `Lab1report.pdf`, on Canvas, one week after the lab took place, by 11:59 PM.

2.1 Actuation and sensor errors (20 points)

(a) In 1.3, the function `OpenLoopControl.m` commands the robot to perform a series of motions. Create two figures, one for the distance traveled and one for the angle turned. On each of these figures, plot the actual data (from your saved PoseData) and the expected motions based on the function. Plot each motion segment (one command in `OpenLoopControl.m`, either a forward motion or angle turned) in a different color and use a different line style to distinguish between the expected and the actual motions.

(b) Based on the figures and analyzing the difference between the expected and actual motions, comment on the actuation errors you observe. Does the robot perform the desired motion perfectly? does the error, if any, depend on the speed? How can we mitigate such errors? How fast do you think we should drive the robot?

(c) For the depth data collected in 1.4, taking into account the location of the sensor with respect to the robot (estimated in 1.4), calculate the difference between the actual measurements and the expected measurements. Do they match? is the error constant? does it change with the distance? what could be the sources of the error? comment on the precision and accuracy of the depth sensor.

(d) If the distance measurement error is a constant bias (a constant error regardless of the distance), how would you take that into account when using the measurements? if the error is not constant, how would you take that into account?

2.2 Sensor frequencies and delays (25 points)

(a) For the data sets obtained in 1.5(a)–(c) observe and compare the timestamps of the data. What do you notice? how often can you get the different sensor information? how does this frequency depend on your code?

(b) What are the implications of these delays on your control code and the robot behavior? how might you mitigate any unwanted behaviors?

(c) For the dataset in 1.5(a) what do you observe about the timing of the depth and tag information with respect to the other sensors? When doing localization (as you will do in Lab 2), how should you deal with the delay in these measurements?

(d) For the datasets in 1.5(d), how does the number of tags affect the delays?

(e) In Part 1.6(e)–(f), did the robot react differently to the presence of a wall? How so?

(f) If so, what is the reason for the different behavior? If the behavior was identical, why would one expect commenting out parts of the code would lead to different behaviors?

2.3 Sensor Information (35 points)

(a) Plot the robot trajectory from Part 1.6(a)–(d).

(b) Indicate places where the robot's bump sensors were triggered (for example, with a *).

(c) How might you build an obstacle map if you only had information from the robot's bump sensors?

(d) Using the function `robot2global.m`, and the location of the camera in the robot-fixed frame you obtained during calibration, plot the locations of any detected beacons (indicate their locations with a different symbol than was used in part (b).) Keep in mind that the robot pose timestamps are not perfectly aligned with the beacon timestamps. How did you handle the timing mismatch? Did you pick the “closest” timestamp? Did you interpolate the data, and if so, which data? Justify your choice.

(e) Discuss the precision of the camera. Are the locations of the detected beacons consistent, or do they move around a lot?

(f) Write a function `depth2xy.m` to convert the raw depth measurements into 2D local robot coordinates (this function will be similar to `lidar_range2xy.m` from Homework 1.) Make sure to take into account the location of the camera you obtained during the lab. The minimum distance for the depth sensor is 17.5cm; if an obstacle is closer, the sensor will return 0. How are you taking into account the nonlinearity in the range error, if at all?

(g) Plot the lab map and the robot’s depth data from Part 1.6(a)–(d) in global coordinates (you may use the function `robot2global.m` from Homework 1 for the depth data). Keep in mind that the robot pose timestamps are not perfectly aligned with the depth timestamps. How did you handle the timing mismatch? Did you pick the “closest” timestamp? Did you interpolate the data? Justify your choice.

(h) How well did the depth sensor measure the truth map? For example, are the walls detected as straight lines? What may be some sources for errors?

(i) Load the depth returns from the stationary robot run in Part 1.5(a)–(c) (all the runs that have depth information). If the depth noise is modeled as a Gaussian, $w_{depth} \sim \mathcal{N}\left(0, \sigma^2\right)$, estimate σ.

2.4 Waypoint Follower (25 points)

(a) Plot all the robot trajectories from Part 1.7 on the same figure (you should have one for each member of the group) and indicate which ϵ was used by each robot for feedback linearization.

(b) Plot the waypoints on the same figure.

(c) Comment on any differences between the robots’ trajectories. Why wouldn’t the trajectories be exactly the same? Did members of the group handle this task differently?

(d) How did the value of ϵ for feedback linearization affect the robots’ trajectories? How would the trajectory have been different for a holonomic vehicle?

(e) For one of the ϵ values, run the same waypoints and the same code in the simulator. Plot the resulting (simulated) trajectory. On the same figure, plot the actual trajectory. Do you expect both trajectories to be the same? Are they? If they are not the same, explain what the possible sources of difference are.

2.5 Lab preparation (mandatory, not graded)

(a) Which robot did you use?

(b) For the code you wrote as part of Homework 1, were there any coding errors that were not discovered when using the simulator?

(c) How much time did you spend debugging your code in the lab?

(d) Did you observe any behaviors in the lab that you did not expect, or that did not match the simulated behaviors?

(e) What was the contribution of each group member to the lab and the report?