Lab1: Robot Familiarity & Data Logger

1 Introduction

This lab will introduce you to the robot and related infrastructure. It is deliberately very easy because we want you to work out the bugs in your basic workflow early.

Before you start you should familiarize yourself with the startup documents on the web site, especially the robot API documentation.

Always read the entire writeup (especially the notes and hints) before attempting anything. Otherwise you may waste a lot of time.

2 Motivation

In this lab, you will connect to the robot, make it move, read some data, and plot the result. As you will quickly discover, working on a real piece of hardware can be slow and frustrating, so you will also develop one of the most rudimentary tools in the roboticist's toolkit – the *data logger*.

Data loggers have immense value in a real time control system because things happen too fast for you to see, you cannot quantify them, and even if you ask the robot to quantify what is happening, you cannot assimilate a stream of numbers on a screen very well. Real time systems generate too much data too fast for you to deal with. The key value of a data logger, however it is constructed, is that it is fast enough to record all data when "talking" to the robot yet you can also read back its data and present it slow enough, and in useful graphical form, so as to not become overwhelmed.

You may think that you should get the robot to do the right thing first and then figure out how to plot a graph for this lab. That is the wrong approach. You need the graphs to debug, even for this simplest of possible labs. Hopefully, you will also learn that working on the hardware is hopelessly inefficient when you are debugging the code for the first time. The solution to that problem is to use the simulator that we provide.

You will probably make dozens of silly mistakes and introduce lots of bugs and wonder how anyone could write a million lines of code for a robot that does anything other than destroy the robot - when you can barely move 10 cm without issues! As they say, Rome was not built in a day, and today is day 1.

3 Exercises

You will find that the robot commands are very low level – almost at the level of sending voltages to the motors. We did that deliberately. Everything more sophisticated than the basic API will be your job, and you will be amazed by what you have done after 12 weeks.

3.1 Warm Up Exercise 1: Basic Open Loop Velocity Control

For this exercise, we will start with the MATLAB command window. If you are a veteran and you know how to do code blocks in the editor, go for it. To prepare to tackle the task, take some small steps first. First, it makes sense to learn how to make the mobile robot not move. Of course, zero velocity for both wheels means stop. That is your way to stop the robot.

It's a good idea, right before making the robot move, to execute `sendVelocity(robot,0,0)` in the MATLAB command window. Then, to stop the robot for any reason, you can ctrl-C to stop the program that is running, and get the stop command back with one or two up arrows and then hit return to stop the robot. Use a really slow speed like `sendVelocity(robot,0.01,0.01);` and practice stopping the robot before doing anything else. In a pinch, you can always pick the robot up.

Now, read about code sections. Read this:

http://www.mathworks.com/help/matlab/matlab_prog/run-sections-of-programs.html

Now go into the editor and start a new block with the line

%% Lab 1 Task 1 Move the Robot

We will use the notation v_l for the left wheel and v_r for the right wheel. The next step is to issue the commands in the following table for $v = 5\frac{cm}{sec}$. Send the indicated commands for the indicated time and then stop the robot. You will have to do this by writing a loop that measures the time elapsed and sends the command over and over until it is time to stop. You are likely to find it necessary to put a call to `pause(0.005)` in the loop to give other parts of the system (like the robot interface) time to do their thing. You can use `tic` and `toc` to measure the time elapsed but be careful to not use the value returned by `tic` for any purpose other than as an argument to `toc`. Use `help tic` (better yet `doc tic`) to find out about this or any other MATAB function.

Caution: When you set the robot velocity, the robot continues to execute that velocity until you send a different command, or about 1 second has elapsed. Even if you do not want to change the velocity, you still have to send the same command again before the deadline. If you do not send the same or (a different) command before the deadline, the robot will automatically come to a stop. This *watchdog timer* is a basic aspect of such systems. Because of it, if your laptop dies or the link goes down on either end, the robot will not drive off by itself. Just think how much time you would have wasted looking for the bug in your code if you did not read that caution statement above. Always read the writeup carefully.

You can also "waste time" in MATLAB with the `pause()` function. This pauses MATLAB, not the robot, and it can be useful. There is no penalty for sending the same command as fast as possible but if you cause MATAB to pause for, say, 50 milliseconds, after each velocity command is sent, your CPU will be able to update your graphs more quickly etc.

Table 1: Robot Commands

Case	v_l	v_r	Time (secs)	Comment
1	v	v	4	
2	$-v$	$-v$	4	

Congrats. You just made your first robot move.

3.2 Warm Up Exercise 2: Basic Simulation

Instead of measuring time, the robot is equipped with an encoder on each wheel that reports the distance it has travelled **in mm**. Write a small MATLAB program that assumes the robot moves at $v = 5\frac{cm}{sec}$ for Case 1 above and generates a simulated encoder reading (**in mm**) called `leftEncoder` accordingly. Start the encoder at the value `leftStart` (this some arbitrary nonzero number like 6934) and then increment the encoder inside a while loop based on a) the commanded velocity and b) the elapsed time (computed with the MATLAB `tic` and `toc` functions). Inside the MATLAB loop, compute a distance called `signedDistance` by reading the generated `leftEncoder` reading, subtracting the initial encoder value `leftStart`, and converting units to cm. Finally, have the while loop terminate after 20 cm of motion and introduce a delay of 1 millisecond (to advance real time a little) in each iteration of your loop with `pause(0.001)`.

Congrats. You just wrote a robot simulator. To do that you integrated the differential equation $\frac{ds}{dt} = v$.

3.3 Warm Up Exercise 3: Basic Plotting

Create two arrays called `timeArray` (e.g. `timeArray = zeros(1,1)` etc.) and `distArray` and fill them up with the time and distance values during each iteration of the loop in the above simulator. MATLAB arrays will add columns automatically if you write beyond their present length (e.g. `timeArray(2) = 0.01`). This is inefficient, but OK for this lab. Use `plot(timeArray,distArray)` to produce a graph on screen after the simulation terminates. It should look like a straight line ending at around (2,10).

Don't go any further until you have done these three warm up exercises.

3.4 Challenge Task

Now, make the robot execute the "challenge" task (Figure 1) for real and produce a graph of the real left encoder output versus time. The robot does not need to turn at all. It can drive backwards in the second motion. You should terminate the motions as soon as the encoder reaches or exceeds the intended distance.

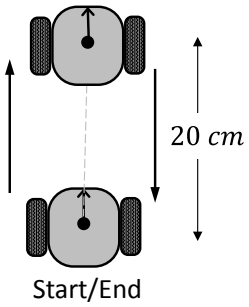

Figure 1: Challenge Task Path. The robot should move forward 20cm, stop and wait 2 seconds, and then move backward 20 cm and stop.

4 Graded Demonstration and Report

Only this part is graded. Points are included in square braces [thus].

Now you have to remove the bits of code that are for simulation, send real commands to the robot for each step, and read the real encoders. Execute the motions at $v = 5\frac{cm}{sec}$ in sequence so that the robot moves forward and back as specified in Figure 1. Terminate based on distance travelled rather than elapsed time as you did in your simulator. You do not have to do anything fancy. Just send zero velocity when the indicated distance is reached or exceeded. Plot the encoder readings (y) versus time (x) for the left wheel on your computer display in real time and show it to us. Subtract the initial encoder reading and convert to centimeters for a nicer display.

[10] Robot moves under computer control at all

[10] Robot moves correct distance, more or less. 3 cm error is too much. 5/4 cm is about as good as can be expected.

[10] Graph is correct (meaning it is what the robot really did).

~~4.1 Report~~

No reports in this lab.

~~Bring your report to the lab period and hand it in **before** the lab period starts. In it, include the content and answers to the questions provided below:~~

~~[2] Include your graph of the left encoder output for the challenge task.~~

~~[6] Comment, in terms of at least 3 nonidealities, on how the graph differs from what you would ideally expect. Suggest what may be the causes for each of nonidealities.~~

~~[2] Suppose you have a variable called `signedDistance` and it is used to terminate the backward motion toward the start point with a snippet something like `while(abs(signedDistance)>0.001){keep moving}`. What is wrong with~~

~~this test and why?~~

~~[2] How could you compensate for the delay in stopping the robot using prediction?~~

~~[10 **bonus**] Why was it important/useful to **not** reset the memory of the initial encoder value before moving backwards assuming that you wanted to get exactly back to where you started? That is, why drive back to zero rather than try to drive back 10 cm.~~

5 Notes and Hints

1. We will never look at your code, so feel free to slowly convert the same code snippet to the final result in this lab.
2. In a pinch, there is always "printf". In MATLAB, leave the semicolon off the end of a carefully chosen line or two in the loop to figure out what is going on. Or use `disp()`. There is an actual C-like `fprintf` command for formatted output.
3. Warm up exercises 2 and 3 can be done before you ever connect to the robot. One person can be doing that while another is figuring out the connection stuff.
4. This robot won't hurt you but it can hurt itself. Please do not go any faster than 15 cm/sec (per wheel) in this lab. Always be kind to your robots. Then they will love you unconditionally.
5. If you are alone, put a `pause(6)` or so as the first line of code, and then run to the robot in < 6 seconds so you can grab it if it freaks out.
6. It is very convenient to use the command line or MATLAB text editor in cell mode to do the initial motion exercise. That is, there is no need actually write an m file with three lines in it.
7. Many communication links will fail badly if you try to push too much data over them. For now, we recommend a `pause(0.001)` be placed in every loop that reads the robot data structure. It is possible to read this memory so fast that the wireless never writes any data to it.
8. While you are just testing, it is a very good idea to turn the robot over and just watch the wheels until you are sure your loops terminate properly and the speeds and durations are about right. The encoders measure wheel rotation, not true distance, so neither the robot nor your code will know the robot is on its back spinning its wheels.
9. MATLAB does not clear the values in variables automatically before you execute a new function or set of statements. One symptom is code that works only the first time you try it. Be careful to clear every variable necessary at the start of any procedure. You can use `clear all` when unsure about initial conditions but that will also delete the robot object so you will need to recreate it and open the wireless link again in that case.
10. When this lab is done, all you need is to read the position of a mouse or touchpad to implement a wireless joystick for your robot. Look up `[x y] = ginput(n)` for example (you need a plot on screen to use it).