Lab3: Modeling for Simulation and Estimation

1 Introduction

When robot commands have the same meaning as the sensing (i.e both mean wheel speeds), three problems become essentially identical:

- Simulating the robot (simulation)
- Computing where the robot is from the encoders (estimation)
- Predicting where the robot will be from intended commands (prediction)

You use the first to help you write code without a robot, the second to provide basic motion feedback, and the third to make the robot go where you want.

2 Motivation

Simply put, having a robot simulator makes you about 10 times more efficient. While there is no substitute for testing code on the robot, there is also no substitute for (initially) debugging code on a simulator. Likewise, data loggers have an intermediate role (between simulation and hardware) in the development toolchain because they provide real data “on tap” without having to deal with a real robot.

3 Exercises

The most basic model of a mobile robot is a differential equation because you drive it with speed commands or measurements (velocity) but you care about where it is (its position). In general terms, the estimation model is of the form:

$$\dot{x} = f(x, z, t) \qquad \textbf{Eq 1}$$

Every letter but t corresponds to a vector and all of these vectors are functions of time. The word *model* can mean many things in many contexts. Here, it means a mapping between a measurement signal $z(t)$ and the motion $x(t)$ that is consistent with it.

3.1 Warm Up Exercise 1: The Robot ~~Whisperer~~ Listener

You can activate a piece of code precisely when new encoder data arrives by assigning a “listener” (also called as a callback function) with the following snippet:

```
robot.encoders.NewMessageFcn=@encoderEventListener;
```

Once this field is assigned, the function `encoderEventListener` will be called immediately whenever a new encoder data message arrives from the robot. You can turn off the listener/callback by setting `robot.encoders.NewMessageFcn=[];`

You also have to write `encoderEventListener` – the routine to do something when the data arrives - and it probably needs to exist on your path before you assign it to `robot.encoders.NewMessageFcn`. Three options for what it does are 1) to call your main program directly 2) to set a global variable that your main program loop is watching to see if it has changed yet or 3) to do the required computations right inside the

handler. In the third case, you should read up on “persistent” variables in MATLAB first. For case 2, you can watch in a (short, one line) infinite loop with a pause for a millisecond every time no change is detected. Then, exit that loop when a change is detected.

Here is the basic layout of a listener:

```
function encoderEventListener(handle,event)
%EncoderEventListener Invoked when new encoder data arrives.
% A MATLAB event listener for the Robot. Invoked when
% encoder data arrives.

 ... % Do some stuff
encoderDataTimestamp = double(event.Header.Stamp.Sec) + ...
      double(event.Header.Stamp.Nsec)/1000000000.0;

encoderFrame = encoderFrame + 1;

… % Do some more stuff
end
```

Any variable in here that is declared global can be read anywhere else you might want to read it. **Get this working**. You will need it for the rest of the course. Encoder messages are sent regularly by the robot whether the wheels are moving or not.

3.2 Warm Up Exercise 2: Measuring Velocity

While it is possible to do this lab by adding incremental displacements `ds` and avoiding time measurement, we want you to use `V*dt` to compute your displacements in order to make a point about timing. To get `V`, you differentiate the (left or right) encoder. In precise terms that means you measure `ds (which in reality = V*dt)` and you then divide it by `dt` where `dt` comes from the difference between two “time tags”[1] – “now” and the “last” time[2]. `ds` is the difference between two encoder readings. Think through the units of everything.

Drive the robot in a straight line for one or two seconds and plot its velocity versus time. If your time tags do not start at zero, subtract the initial value to make the graph more readable. Do a plot for each of three cases:

- Case 1: Read the encoder as fast as you can. Use your <u>laptop clock</u> to compute dt.
- Case 2: Respond quickly to Encoder Events. Use your <u>laptop clock</u> to compute dt.
- Case 3: Respond quickly to Encoder Events. Use the <u>event time tags</u> to compute dt.

[1] A time tag is the value returned when you read some clock. Every processor on a robot, and there may be many, has some kind of clock - and that is part of the issue.

[2] Memory of what happened the “last time I was here” in real-time code is a common thing. Use static variables for the last cycle memory if you exit the scope in which it would otherwise be defined.

3.3 Warm Up Exercise 3: Basic Simulation for Arbitrary Trajectories

Write the function or code snippet `modelDiffSteerRobot(vl, vr, t0, tf, dt)` that takes input wheel velocities **in mm/sec** in the `vl` and `vr` vectors, and returns the robot trajectory `x(t), y(t), th(t)` corresponding to those measurements. You can assume zero initial conditions in position and orientation in the plane. The last three arguments are initial time, final time, and time step for the integration.

The basic mechanism to do such a simulation is integration of the differential equation. Of course, time will be discrete so you will do discrete integration. Consult the course notes for the details. Be careful to update the angle `th` **before** you update the position variables `x` and `y` in each step of the integration. Better yet, look up the midpoint method online and use that.

3.3.1 Estimate a Cornu Spiral

Estimate the actual trajectory of this simulated robot for the following encoder measurements (in meters, based on the velocities computed as $v = 0.1$; $\omega = k_\omega t$; $k_\omega = 1/8$) for a time period up to $t_f = \sqrt{32\pi}$. Use the differential steer inverse kinematics formulas:

$$v_r = v + \frac{W}{2}\omega \,;\, v_l = v - \frac{W}{2}\omega \qquad \textbf{Eq 2}$$

Where `t` is time and it starts at zero. The right wheel velocity increases linearly with time while the left decreases linearly. Use a time step of 0.001 seconds to ensure good accuracy. You will have to simulate the clock as well with `t=t+dt` somewhere in the loop but **be careful to take this out later** when you use the real clock in later exercises.

This curve is called a *clothoid* or a *Cornu Spiral*. The robot should make exactly one revolution and end up at around $x, y = (0.25, 0.17)$. Plot the resulting trajectory. Plot x versus y with `axis equal`. Use a plot area (`xlim, ylim`) that extends from 0 to 0.4 units in both directions.

3.3.2 Estimate a Figure 8

For later, you will also need to simulate a figure 8 trajectory. Use the differential steer equations above based now on velocities computed as follows. Before the loop that measures time and decides when to stop (based on comparing the time to T_f), compute

- the parameters of the unscaled trajectory: $v(t) = 0.2$; $s_f = 1$; $t_f = s_f/v$; $k_\theta = 2\pi/s_f$; $k_k = 15.1084$;
- then, the scaling parameter and the scaled final time are: $k_s = 3$; $T_f = k_s t_f$;

Inside the loop, read the scaled time T using `toc()` and convert it to unscaled time with:

- $t = T/k_s$

Then, compute the rest of the trajectory as follows:

- $s(t) = v(t)t$; $\kappa = (k_k/k_s)\sin(k_\theta s)$; $\omega(t) = \kappa(t)v(t)$.

You may find that it saves a lot of run time to turn off the real time display and preallocate the plotting arrays. Note that the code `xArray(1,1:n)` means the matrix composed of the first n elements of the (longer) row vector `xArray`.

4 Warm Up Exercise 4: Real Time Graphics

Once that is working, retrofit real time graphics. Add the following before the loop (with your array names substituted):

```
myPlot = plot(myXArray, myYArray, 'b-');
xlim([0.0 0.5]);
ylim([0.0 0.5]);
```

And put this line in the loop:

```
set(myPlot, 'xdata', [get(myPlot,'xdata') x], …
        'ydata', [get(myPlot,'ydata') y]);
```

You may need a `pause(0.005)` (or so) in the loop to give the plot window time to update. Now you can see your solution in real time as it is computed !

5 Warm Up Exercise 5: Drive the Robot and Gather Data

Write a function or code snippet to drive the real robot on the above Cornu spiral and record at least the encoder data and the time. Send left and right wheel **commands** (in **m/sec**) that are equivalent to the above measurements.

You will find it super valuable to record the commanded and actual wheel speeds too, in order to help you debug this code while it is being developed. If you want to plot the actual velocity, you should process encoder readings immediately as they arrive and differentiate them. The "actual" velocities derived from the encoders should look pretty close to what you are commanding but there will be a delay. "Immediately" means it is OK to check every millisecond or so to see if there is new data. It is critical that the robot move at about the right speed to ensure that sending commands based on the elapsed time will also be sending them at the right position. You can use `tic` and `toc` to find precise elapsed time for the purpose of generating commands.

Don't expect the path to be perfect but it can be close. Errors as large as 1% to 5% of distance travelled are typical.

Or this task, **and from now on,** you can choose to use the change in wheel encoder readings directly to compute the change in robot position and orientation. This approach makes your code more immune to timing issues but it also makes it harder for you to detect timing issues if they occur.

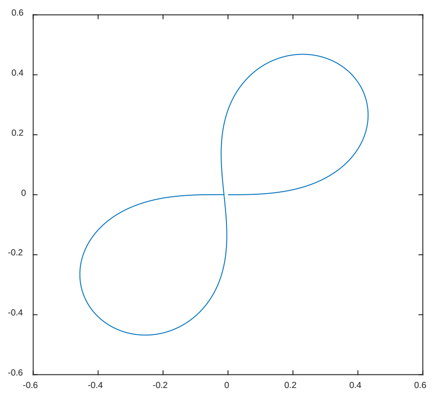

Figure 1: Challenge Task Path. A figure 8 of one revolution.

6 Challenge Task: Drive the Robot and Estimate its Pose

Now, implement a real time state estimation system for your robot with interactive graphics. Drive the robot on the figure 8 (in Figure 1 ☺) and **compute the position of the robot from the encoder feedback** rather than from the commands.

7 Graded Demonstration and Report

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

Show your robot executing the challenge task trajectory with a real time state estimate appearing in a window[3] as the robot moves.

[20] Actual path is correct shape[4].

- 20 points if 0<err<5cm,
- 15 points if 10<err<7cm,
- 10 points if 20<err<9 cm,
- 5 points if 0<err< 11cm,
- 0 points otherwise.

"err" is defined as the average error in the terminal position of the two wheels of the robot.

[5] Graph updates in real time.

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

~~7.1 Report~~

~~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 (and in other grey boxes above, if any):~~

~~[2] For the challenge task, does the robot end up where it is supposed to end up? To what accuracy? Suggest reasons for your answer.~~

~~[2] For the challenge task, does the robot (actually your laptop) "know" a) if and b) how well it is or is not following the path?~~

[3] All graphs produced for labs in this course should have (1) a title and (2) axis labels that include (3) a description of the units. Plot different signals on the same graph using (4) different colors with (5) no point markers (triangles, squares, etc.). Use a (6) legend to describe which is which. Failure to comply with these rules will result in a 20% reduction in your grade. Whenever possible, produce graphs in real time as the robot moves, but its OK to plot results at the end of the test if real-time graphics would otherwise affect performance.

[4] In 2016, Al's solution is accurate to 2 cm.

8 Notes and Hints

1. Most of the hints from last week still apply.
2. When planning your attack, always think about a) what can be done without the robot, b) what can be done with just some test data that is real, c) what must be done on the robot, and d) what must be done with the robot actually moving on the floor.
3. The value returned by `tic` can only be used as an argument to `toc`. Much badness happens, and much time is wasted, if you try to use it as a time. Read the documentation before you use these functions.
4. Be careful not to mix up real and simulated time. Your measurements of what time it is and how much time has passed should be based on readings from the same clock. There are at least three clocks that you may use 1) the clock on the robot (time tags in events) 2) The clock on your laptop (`tic` and `toc`) or the clock you are simulating when you write `t = t + dt`. Don't mix em up. The basic rules are 1) used encoder time tags to compute dt for purposes of error differentiation and integration 2) use laptop clock for prediction calculations and 3) synchronize the laptop clock and simulation clocks. It is not necessary and it is very complicated to try to synchronize the laptop and robot clocks. Delay calculations are a form of predictive synchronization.
5. A final subtlety about time is you may define command signals based on the assumption that time starts at zero. For such signals, you need to read the clock right before they are first used (call that result the clock bias) and subtract the bias before passing time into the control signal to compute the signal value.
6. In a few labs, writing a big loop will start to become impossible to debug. If you want to get ahead of things and make this lab a little simpler, do the following.
    a. Implement a MATLAB class that encapsulates the robot state (pose and velocities, and time) as well as simulated encoders. Make it able to integrate the state based on time or based on an update to encoders. Give it a `sendVelocity` method and give it a public struct for the encoders(i.e.
    ```
    encoders=
    struct('LatestMessage',struct('Vector',struct('X',0,'Y',0))
    );).
    ```
    b. Make it possible to initialize the pose, the time, and the encoders.
    c. Allocate some large arrays in the instance you create and store everything you would ever want to plot in them. Every time you take one step in the integration, put the new data in these arrays.

When you flesh all that out, you can create one instance for estimating state from the encoders, and another instance to act as your simulator[5] for testing purposes. Now you can simulate anything and plot anything and you never have to write that code again. Once this is working, you can redo the lab using the new infrastructure and

[5] It is true that we have provided a simulator already, but one point of this lab is to demonstrate that your estimation and control systems will STILL need to have embedded simulators in them that integrate motion commands or sensing. For control, your code needs a simulator to predict the consequences of sending out a particular command. For estimation, your code needs a simulator to figure out where the robot is.

make sure it still works. We suggest you don't start out with the ambition to write an object oriented solution until you try the more straightforward approach first.

7. One impact of delays is that if you stop your control loop right when the command signals are over, you will not process the feedback that they generate perhaps 0.1 seconds later. A simple approach is to command zero velocity until the delay expires and then exit the loop.
8. A `robot.stop` command is possibly more robust than sending zero velocity once.
9. Error checking can save a lot of time. Situations like forgetting to start the encoder listener should occur once to teach you how painful it is to detect it. Once you learn that lesson, make whatever piece of code is alive most of the time check that a) listener/callback function exists before you start b) it is being repeatedly activated by the arrival of encoder messages from the robot (check the data or build a frame counter into your handler). You can treat the robot object similarly. Unfortunately, it is necessary to use `clear all` fairly often and that deletes the robot object so you can check if it exists and if not, create it automatically in your code.