QEA Bridge of Doom Assignment

<http://nb.mit.edu/f/59440>

<https://canvas.instructure.com/courses/1774456/assignments/14606124>

# A brief Intro

To wrap up the robotics module of part 2 of spring 2020 QEA, we were assigned a project with simulated Neato robots and some basic open-loop robotics path planning challenges.

Prior to the project, we were assigned a lot of practice to develop our matlab plotting skills and grow our intuition and understanding of paths and 3d/ motion. We worked with some basic vectors and vector calculus to understand the basis of how motion works and how you can represent it and derive one type of motion-related data from another.

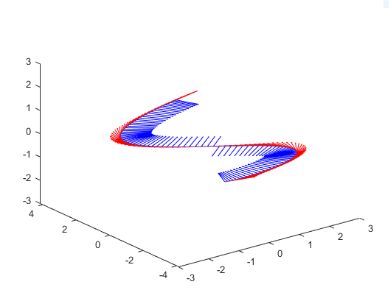
During the actual BOD assignment, we were tasked with enabling a simulated Neato robot to cross a bridge of defined curvature. The path can be represented mathematically in Matlab as:

R = 4\*[0.396\*cos(2.65\*(u+1.4));-0.99\*sin(u+1.4);0];

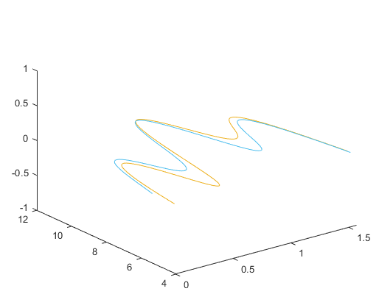
While elevation of the bridge was constant and factors like friction and air resistance (causing the robot to slip or act in more unpredictable ways) were manageable, the challenge lay in having a really solid understanding of simple mathematical motion theory and being able to translate that into Matlab code that the robot could understand efficiently and accurately. Another challenge also lay in grabbing data from the robot as it moved through the simulation and comparing the data to theoretical predictions.

# Methodology

Following the BOD assignment, the first set of steps I took were to gain a better visual understanding of the robot’s motion from a number of mathematical perspectives.

I began by plotting the bridge’s path along with some tangential and normal vectors, to see what the ideal motion path of the Neato should in theory look like.

The big picture is to figure out a way to drive the robot through this path in an open-loop; thus we logically must know the wheel velocities at every point in time of the Neato’s journey. In order to turn the bridge’s defined path into a set of wheel velocities that we could plug into the robot, you can begin by defining the value u as a constant B multiplied by a changing variable t. Now you have a core path function determined by a single variable t, representative of time, at all points of the motion.

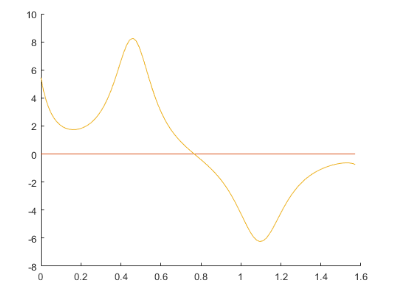
The next step is to understand the angular velocity as a function of time and the linear speed as a function of time, both of which are required to calculate the individual wheel speeds of the bot. To find the angular velocity we cross T with the derivative of T, and to find linear speed we take the magnitude of the velocity (which is just the derivative of the position function we started out with).

Left and right wheel velocities

Left Wheel Speed in Orange

Right Wheel Speed in Blue

We can use 2 simple equations that model the robot’s differential drive characteristics to map linear speed and angular velocity to left and right wheel speeds.

vL1 = speed - (d\*(omega(3)))/2

Linear speed & angular velocity

Angular Velocity in Orange

Linear Speed in Red

vR1 = speed + (d\*(omega(3)))/2

After plotting the two different wheel speeds, we find it quite easy to plot the value of angular velocity (omega in the above snippet) over time as well as the value of the linear velocity over time (speed in the above snippet).

## Actual Robot control

Now that we’re mathematically capable of generating the two robot wheel speeds from the initial plot of the BOD’s path, we’re essentially ready to write the code to control the robot. There are a couple main architectural differences in the code that was used to create the plots and the code that was actually used to run the robot.

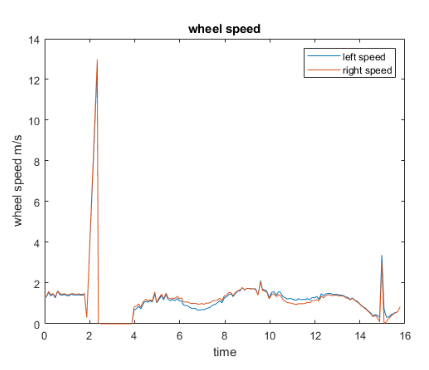
In short, these differences were

|  |  |
| --- | --- |
| Creating Plots & math code | Running Neato Code |
| Used symbolic variables for all variables to start with | Used symbolic variables for only the t variable |
| The value of B did not matter. | The value of B did matter. |
| Code wasn’t really centered around any specific mathematical function. | Code was somewhat centered around the wheel speed functions.  Ran a while loop and grabbed the time via rosstime(‘now’). Plugged the resulting difference between that ‘now’ and the ‘start’ time value into the functions we created for the left and right wheel speeds. |
| No ROS features used. | Used ROS publishing features to update the wheel speeds in nearly real time. |
| No ‘boilerplate code’. | Lots of ‘boilerplate code’ to setup the robot simulation (i.e. placing the robot in the right spot at the start of the simulation). |

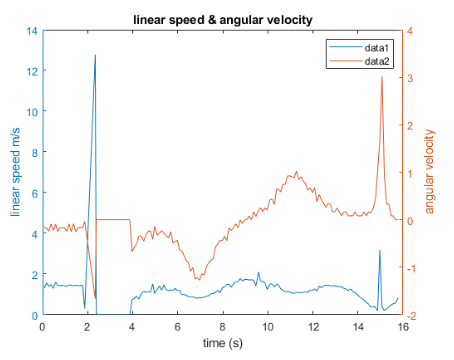
## Analyzing the encoder results

In order to analyze the encoder results we started by using the collectDataset\_sim.m function to collect the Neato’s encoder values as it crossed the Bridge of Doom. Of the data that the function returned, we used the linear travel of the two wheels, stored in columns 2 and 3, which aligned with the elapsed time, stored in column 1 of the data returned by the function.

Wheel Linear Travel

On the right you can see the plotted linear travel of the left and right wheels – not too impressive. However, once you derive those values to obtain the left and right Neato wheel speeds as a function of time, things begin to get more interesting. You can see the results in the graph on the right.

A couple things to note about this data – number 1, there’s a lot of noise. While in theory the graphs should appear smooth, there are little spikes and bumps along the entire time. Additionally, the wheel speeds in theory should never dip below 0 – yet both the left and right wheels spike upwards and then immediately below zero at around the 2 second mark, which should not happen.

The process of analyzing the encoder data was essentially the same as the initial mathematical and theoretical code, but with the entire workflow reversed, and working with real data from the start instead of Matlab’s symbolic values.

Thus the next logical step would be to derive linear speed and angular velocity of the Neato as it moved through the BOD. View our plotted results on the right. We did this with the differential drive motion equations provided to us, and substituting in the distance between the two wheels as .235 meters:

d=.235

V = (vL + vR)/2

omega = (vL-vR)/d

Now that we had the linear speed and angular velocity, we could walk through some for loops and recreate the entire motion path, using the time values initially provided in the encoder data. All we’d need to do is walk through every time value and calculate the position and angle at that time value using the previous step’s angular velocity and linear speed, which we already have data for, as shown above.

The code to reconstruct the position and angle of the Neato at every point in time is shown below:

path = zeros(length(t), 2);

theta = zeros(length(t), 1);

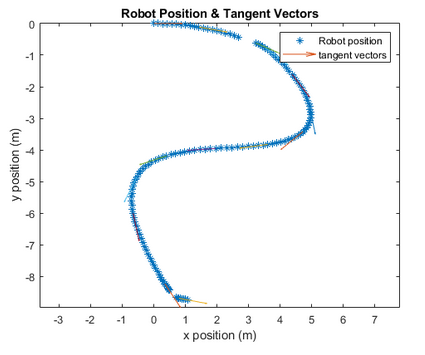
for n=1:length(dt)

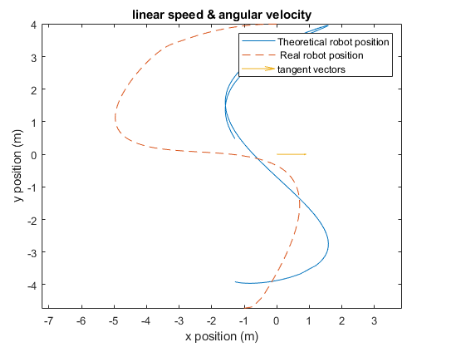
path(n+1,1) = path(n,1)+V(n)\*cos(theta(n))\*dt(n);

path(n+1, 2) = path(n,2)+V(n)\*sin(theta(n))\*dt(n);

theta(n+1) = theta(n)+omega(n)\*dt(n);

end

Shown on the right is the reconstructed path of the Neato with no changes made. While the Y axis is flipped in this plot, you can clearly see that the Neato’s path plotted at every time stamp with a blue star, and some of the tangent vectors at every 10 time steps.

And finally, shown below on the next page is the final plot of the recreated robot’s path, mapped against the theoretical path that was calculated all the way at the beginning of the report. The paths themselves don’t align/overlap very well, and took some quick mathematical adjustments to even get them to this level of visual overlap. The dashed curve representing the Neato’s actual reconstructed path has curves that aren’t perfect or smooth – there is evidence of slipping of the wheels during points in the path with high curvature.

# View my Neato crossing the BOD & all code!

<https://youtu.be/kyUhivl-2AA>

<https://github.com/aramachandran7/BOD_QEA>

## A couple quick final points to note about the Neato’s crossing of the BOD:

The Neato accelerates quickly when on straightaways but slows down significantly on tighter turns / curves. At the current value of B, which effectively determines how the Neato processes time (how fast it goes, basically), the Neato’s actual success ratio of BOD crossings is around 40%. If I were to lower its B value and resulting speed of crossing the BOD, I would increase its overall path completion consistency.