E190Q – Autonomous Robot Navigation

Lab 3

Point Tracking

INTRODUCTION

Robots often need to move to a point with a desired orientation. This can be difficult when the robot has nonholonomic constraints. The robot cannot simply move laterally, making it difficult to come up with a controller using intuition.

The goal of this lab is to get students to implement a closed loop controller that will drive the robot to any desired state (position and orientation). Odometry will be the sole method used for estimating the robot's state (i.e. localization).

BACKGROUND

As shown in class, a controller has been developed based on the following coordinate transformation:

$$\rho = (\Delta x^2 + \Delta y^2)^{0.5}$$
$$\alpha = -\theta + atan2(\Delta y , \Delta x)$$
$$\beta = -\theta - \alpha$$

Once the new variables have been calculated, desired forward velocity v and desired rotational velocity w can be calculated. Note, control gains are defined at the top of `Navigation.cs`, but they may not be optimal values.

$$v = k_\rho \rho \qquad w = k_\alpha \alpha + k_\beta \beta$$

Using v and w, we can determine the desired wheel velocities $\dot{\varphi}_1$ and $\dot{\varphi}_2$. The following equations were derived and are used.

$$\omega_1 = \frac{r\dot{\varphi}_1}{2L}$$

$$\omega_2 = \frac{-r\dot{\varphi}_2}{2L}$$

$$w(t) = (\omega_1 + \omega_2)$$
$$v(t) = L(\omega_1 - \omega_2)$$

Recall that this controller works well if the goal point is in front of the robot, that is if α lies between $-\pi/2$ and $+\pi/2$.

However if the goal is behind the robot, then modifications to the controller are required to give shorter more direct paths involving the robot moving in reverse. That is, we first redefine the transformation as:

$$\rho = (\Delta x^2 + \Delta y^2)^{0.5}$$
$$\alpha = -\theta + atan2(-\Delta y , -\Delta x)$$
$$\beta = -\theta - \alpha$$

We also redefine the control law to have the robot work in reverse.

$$v = -k_\rho \rho \qquad w = k_\alpha \alpha + k_\beta \beta$$

When implementing this controller, make sure your robot never exceeds the maximum allowable velocity of $0.25\ m/s$, and that controller gains must satisfy the necessary conditions for stability.

EXPERIMENTS

Use the most recent version of the base code from the course website. All coding for steps 1 through 4 will occur within the function `FlyToSetPoint()`. Recall that this function will be called for each iteration of the control loop. You will need your odometry localization code from lab 2. You will not need any wall positioning or odometry error characterization code.

1. Transform to new coordinate system

Using the robot state estimate [`x_est y_est t_est`], and the desired state [`desiredX desiredY desiredT`], calculate the position of the robot Δx, Δy relative to the goal position.

Now use the equations above to calculate the state [`pho alpha beta`] in the new coordinate system. Remember to check if the goal is behind the robot and recalculate the state if necessary.

It is often a good idea to make sure that ALL angles lie within $-\pi$ and π.

2. Calculate Desired Wheel Velocities

Now that the transformation is complete, implement the control law to determine the desired velocities v and w, respectively represented by `desiredV` and `desiredW` in your code. You must experiment with different gain values `Kpho, Kalpha,` and `Kbeta`. Remember to reverse direction if the goal is behind the robot.

From desired robot velocities v and w, you can calculate the desired wheel velocities `desiredRotRateL` and `desiredRotRateR`.

Remember to convert these desired wheel velocites from *m/s* to *encoder pulses / second* before sending them to the robot.

You can check if the robot is close enough to the desired state and zero the desired wheel velocities.

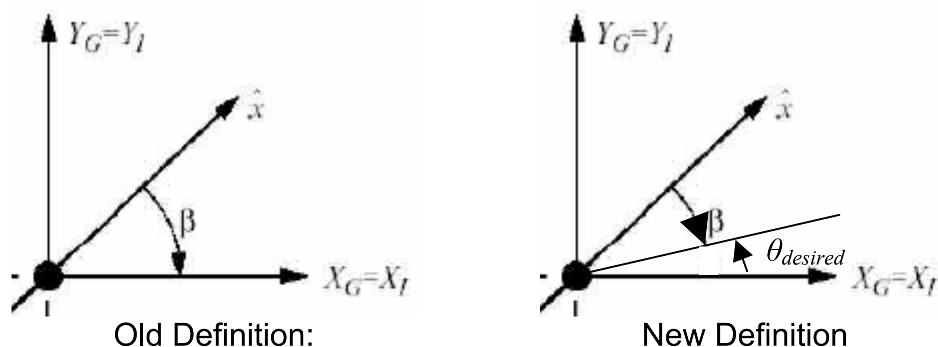
3. Track Desired Positions

After building and running the application, select *Simulation* mode. Try entering *[1 0 0]* in `DesiredX`, `DesiredY`, and `DesiredT` fields located next to the *FlyToSetPoint* button. Click on the *FlyToSetPoint* button. The robot should move forward *1 m*. Now try tracking the point *[0 0 0]*. The robot should return to the origin.

Keep trying new points to track, making sure the robot always moves to the desired locations. At this point the robot will always end with orientation *0* degrees.

4. Track desired positions and orientations

To make the robot track desired orientations, the state variable β must be modified to include `desiredT`. Simply adding this to β will force the robot to track the desired orientation, (See figure below).



Old Definition: New Definition

Now test the controller for many desired position/orientation combinations in *Simulation* mode.

5. Jaguar Wheel Velocity Control

The Jaguar's DC motor velocity control is not functional. It is up to you to add such a controller. In the function named `CalcMotorSignals()`, which is called from the main control loop, you need to add a Proportional-Integral-Derivative (PID) controller. This Controller will determine the duty-cycle of the pulse width modulation controller as a function of the desired wheel velocities.

First you need to get estimates of the wheel velocities in pulses per second. There are global variables `diffEncoderPulseR` and `diffEncoderPulseL` that should

be set in your odometry function. These can be used, along with `deltaT`, to determine the wheel velocity.

For this PI controller, the velocity error can be defined as the difference between the desired wheel velocity and the estimated:

$$e_l = \dot{\varphi}_1 - \dot{\varphi}_{1_est}$$
$$e_r = \dot{\varphi}_2 - \dot{\varphi}_{2_est}$$

The controller for each track becomes

$$u_{pwm}(t) = K_P\, e(t) + K_I \int e(t)\, dt + K_D\, de(t)/dt$$

Feel free to read up on PID control – it will be useful. Tuning these control gains can take some time. The instructor can help give guidance if required. Make sure the control signals u_{pwm} sent to the jaguar are between 0 and 32767.

First, do a check to make sure the robot is not moving faster than $0.30\ m/s$. Next, consider the controller's performance as the jaguar gets very close to the goal.

6. Tracking trajectories

Using the point tracker you just developed, implement a trajectory tracker that enables the Jaguar to autonomously follow a hard coded trajectory, (this may be useful when your motion planner autonomously constructs a collision-free trajectory).

Hard code a trajectory that includes both straight line path segments and circular arc path segments.

Use the function `TrackTrajectory()` located in Navigation.cs. You will have to make sure the control thread calls `TrackTrajectory()` instead of `FlyToSetPoint()`.

DELIVERABLES

1. Demonstration

Before the end of the final day of this lab, you must demonstrate to the Instructor that your point tracker/trajectory tracker is working properly. In both simulation and hardware mode, the 2D graphics window should show the robot and estimate moving towards desired goal states.

Part of your grade will be based on performance: How stable is the controller on the real robot, how close does it come to desired states, etc. Demos are due by the end of lab on Friday March 1st.

2. Submissions

In a 5-10 page report, present your methods and results for both point tracking and trajectory tracking. Be sure to include the following sections: abstract, introduction, background, problem definition, control design, results, conclusion.

Note, all lab documents in this class will follow the template found at:
http://www.ieee.org/conferences_events/conferences/publishing/templates.html

The report is due 9:00am, Friday, February 27th.