Cohort Session 2 & 3 (Wk 10)
Robot Controller

Objectives

1. We will program the robot to move along a wall to its side, maintaining a constant, desired distance from the wall. We will use a simple *proportional controller*. For this part, we will:
   - build a proportional controller for a robot and test it in simulation with different gains.
   - test it on a real robot.
2. We will also investigate one scheme to achieve better performance. In the delay-plus-proportional scheme, the rotational velocity depends on both the current value of the error $e[n] = d_i[n] - d_o[n]$ and on the previous value $e[n-1]$.

   For this scheme, we will
   - build a corresponding controller for the robot,
   - test it in simulation for the various gains, and
   - test it on a real robot.

Please work on this lab in a group of **two or three**. **Be sure to email your partner all the modified code, printouts and data**. You may have to use them during your exams.

1 Equipment & Software

Each partnership should have a laptop that reliably runs soar. Download the software distribution from Week 10 on **1D Project website**. The distribution contains:

- `smBrainPlotDistWork.py`: Template simple brain with a place for you to write the state machine for the controller.
- `propWallFollowBrainWork.py`: Template brain with a place for you to write the proportional controller.
- `delayPlusPropBrainWork.py`: Template brain for delay-plus-proportional scheme.

2 Proportional controller for the robot

Task

Implement a proportional controller to keep the robot a fixed distance from the wall in front of it.

In the previous week, you implemented a brain to control the robot to keep it a fixed distance from the wall in front of it. Most students implemented a *bang-bang* controller, which either went forward or backward at a fixed velocity. Many people had trouble making the robot stop oscillating. In this lab, we will solve that problem with a *proportional controller* in which the robot's velocity is proportional to the difference between its desired distance to the wall and its actual distance to the wall.

The template for your controller is in `smBrainPlotDistWork.py`, where the state machine `mySM` is defined to be a cascade of a `Sensor` state machine and a `Controller` state machine. Thus the output of the `Sensor` state machine (which represents the sensed distance to the wall, `dSensed`) is the input to the `Controller` state machine.

Your job is to complete the definition of the `Controller` state machine. Make the output of this machine be an instance of `io.Action` that specifies a rotational velocity of zero and a forward velocity that is proportional to the difference between the desired distance to the wall (`dDesired`) and the sensed distance `dSensed`. Let k represent the constant of proportionality between forward velocity and the difference `(dDesired-dSensed)` (take note of which term comes first in the subtraction).

Note: the distance to the wall decreases when the forward velocity is positive!

Implement the proportional controller by editing the `getNextValues` method, and run this brain in the `wallFinderWorld.py`. When you stop it, it will plot the sequence of distances to the wall during the run.

Checkoff 1

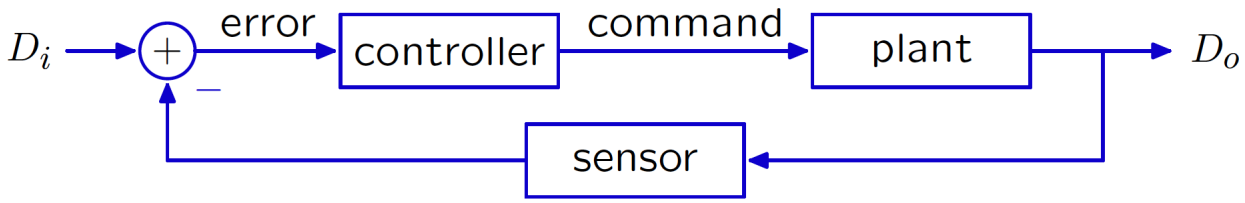
Find three different values of `k`, one for which the distance converges monotonically, one for which it oscillates and converges, and one for which it does not converge. Enter the gains (values of `k`) you found into Tutor. Make plots for each of these `k` values. Save screen shots for each of these plots.

Show your controller code and plots to a staff member, and discuss which gains generate which results, and speculate why. Email your code and plots to your partner. (Make sure the staff member enters your grade into Tutor.)

3 Staggering Proportions

Now, we generalize from the simple one-dimensional model, to a two-dimensional world, and seek to guide a robot such that it moves parallel to a wall, staying a specific distance away.

The basic structure of the system we build will remain the same:



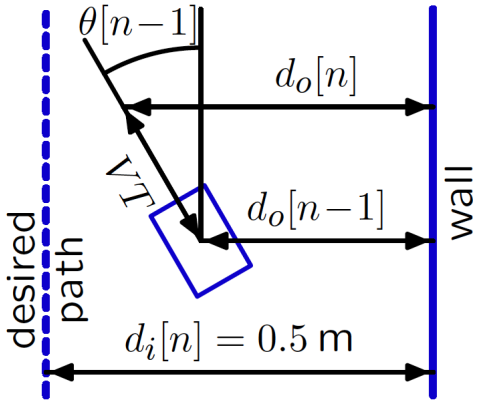
The symbols attached to the diagram are signals. Specifically, here are definitions of various symbols to be used in this lab:

- k : gain of the controller, a constant number
- V : forward velocity of the robot, a constant number
- T : the amount of time between discrete samples of the signals, a constant number
- D_i : desired distance of the robot to the wall, a signal whose samples are $d_i[n]$
- D_o : actual distance of the robot to the wall, a signal whose samples are $d_o[n]$
- E: error, equal to $D_i - D_o$, a signal whose samples are $e[n]$
- Θ : robot's angle with respect to the wall, a signal whose samples are $\theta[n]$
- Ω : robot's angular velocity, a signal whose samples are $\omega[n]$

4 Proportional Wall-Follower

> Task
> Build a proportional controller for a robot and test it in simulation with different gains .

The figure below illustrates a robot in a hallway, with its desired path a fixed distance from the right wall. We can build a controller with a fixed forward velocity of $V = 0.1\,\text{m/s}$, and adjust the rotational velocity $\omega[n]$ (not shown) to be proportional to the *error*, which is the difference between the desired distance $d_i[n] = 0.5\,\text{m}$ to the right wall and the actual distance $d_o[n]$. The constant of proportionality between the error and the rotational velocity is called the *gain*, and we will write it as k. Notice that when the rotational velocity $\omega[n]$ of the robot is positive the robot turns towards the left, thus increasing its angle $\theta[n]$.



Look in the file `propWallFollowBrainWork.py`. The brain has two state machines connected in cascade. The first component is an instance of the `Sensor` class, which implements a state machine whose input is of type `io.SensorInput` and whose output is the perpendicular distance to the wall on the right. The perpendicular distance is calculated by `getDistanceRight` in the `sonarDist` module. All of the code for the `Sensor` class is provided.

The second component of the brain is an instance of the `WallFollower` class. Your job is to provide code so that the `WallFollower` class implements a proportional controller.

Check Yourself 1
What should be the types of the input to and the output from a state machine of the `WallFollower` class? What should be the sign of k?

Detailed guidance:

Step 1. Implement the proportional controller by editing the brain. Then use soar to run your brain in the world `wallTestWorld.py`. The brain is set up to issue a command during the setup to rotate the robot, so it starts at a small angle with respect to the wall.

Step 2. Experiment with a few values of the gain k to determine how k affects the resulting behavior. Generate plots to illustrate the trends that you found. **Save** screen shots of your plots and email them to your partner.

Checkoff 2
Show your plots to a staff member.
Describe how the gain k affects behavior of the wall-follower. What value of k is best, and why?
Compare the effect of k in this section and in the previous section.
(Make sure the staff member enters your grade into Tutor.)

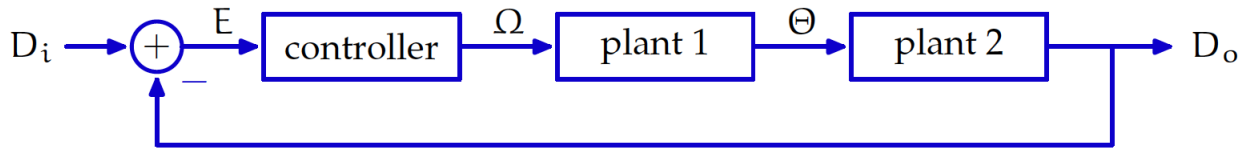
5 Remembrance of Things Past

We now move on to improving our wall-following controller.

We can make a better controller by processing the error signal E in a more sophisticated way than we did in the previous lab. For example, we could adjust the angular velocity using some combination of the present and previous values of the error,

$$\omega[n] = k_1 e[n] + k_2 e[n-1].$$

We refer to this controller as "delay plus proportional." The system still has the same form as the one from last week:



The subsystems that represent robot locomotion (plant 1 and plant 2) and the sensor (modeled as a wire) are the same. Only the controller has changed. (Remember that ω is lower-case Ω.)

5.1 Brain

Step 1. Implement the delay-plus-proportional controller by editing the `WallFollower` state machine class in `delayPlusPropBrainWork.py`. **Think very carefully about what you want to output in the first time step.**

As before, the brain has two parts connected in cascade. The first part is an instance of the `Sensor` class, which implements a state machine whose input is a sequence of instances of `SensorInputs` and whose output is the perpendicular distance to the wall. The perpendicular distance is calculated by `getDistanceRight` in the `sonarDist` module by using triangulation (assuming the wall is locally straight). If sensors 4 and 5 both hit the wall, then the value is fairly accurate. If only one of them hits it, then it's less accurate. If neither sensor hits the wall, then it returns 1.5. The code for the `Sensor` class is provided inside `delayPlusPropBrainWork.py`.

The second part of the brain is an instance of the `WallFollower` class. You should provide code so that the `WallFollower` class implements a state machine whose input is the perpendicular distance to the wall and whose output is an instance of the class `io.Action`. Think carefully about what you are going to store in the `state` of the machine, and how you will initialize `startState`.

Step 2. Run your brain in the world `wallTestWorld.py`. Determine how the behavior of the system is affected by the controller gains `k1,k2`. Try the gain values shown in Table 1 and determine which is the best values to use. Pay attention to the distance values being printed out by the brain. Save plots to illustrate the performance of each of your

k1	k2
10	-9.97
30	-29.77
100	-97.36
300	-271.74

Table 1: Gain Values

optimized gain pairs `k1, k2`. Choose names for these files so that you can remember the parameters that were used to generate each one.

The brain is set up so that whenever you click **Stop**, a plot will be displayed, showing how the perpendicular distance to the right wall changes as a function of time. To save this plot, take a screen shot. **This plot will disappear once you reload the brain.**

Keep the files for exams.

Don't change your controller to try to make it work better! Instead, try to understand the different kinds of failures that can happen and how they depend on the choice of gain or initial condition (you can change the initial position and angle of the simulated robot by dragging it with the right or left mouse buttons).

Check Yourself 2
Which of the four gain pairs work best in simulation?

`k1 =` `k2 =`

Which gains cause bad behavior?

Step 3. Run your code on a real robot as follows:

- Locate a free robot that is already set up with a wall (with at least two bubble-wrapped boards).
- Log into your laptop and attached the robot to it, and run your code.
- Get a measuring stick and be sure to start the robot at the same initial conditions (0.5 meters from the wall and rotated $\pi/8$ radians to the left) as in the simulator. Pay attention to the distance values being printed out by the brain.
- **Save a plot for each of your calculated gain pairs. Keep the files for exams.**

Check Yourself 3
Which of the four gain pairs work best on a robot?

`k1 =` `k2 =`

Are the best gains the same as in simulation?
Which gains cause bad behavior?

Checkoff 3
Show a staff member plots for the simulated and real robot runs, and discuss their relationship.

Explain how you chose the starting state of your controller.

Be sure both partners have the files, and the staff member enters your grade into Tutor.