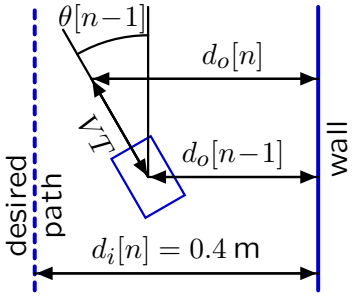
Design Lab 5: Sizable Following

You will need a lab laptop. Do `athrun 6.01 update`.
Code is in `~/Desktop/6.01/lab5/designLab/`.
Remember to email your code and plots to your partner.
You will need to bring these to your oral interview for discussion.

Last week, you wrote a **proportional** controller to move the robot parallel to a wall while trying to maintain a constant, desired distance from the wall. The forward velocity V was set to a constant (0.1 m/s) and the angular velocity $\omega[n]$ was proportional to the error signal $e[n]$, which was the difference between the desired distance $d_i[n]$ and current distance $d_o[n]$.



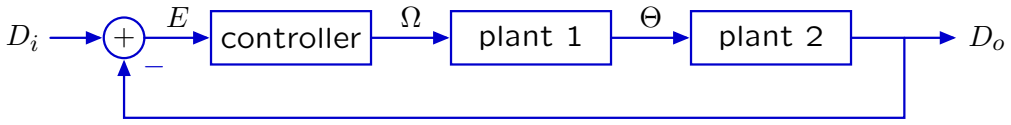
Unfortunately, no value of the proportionality constant k gave good performance. Large values of k gave fast oscillations and small values of k gave large errors, especially when the initial angle of the robot was not parallel to the wall. In this lab, we will develop two new types of controllers to achieve better performance.

1 Remembrance of Things Past

We can make a better wall-following controller by processing the error signal E in a more sophisticated way. 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." Notice that the system has the same form as the one from last week.



The subsystems that represent robot locomotion (e.g. plant 1 and plant 2) and the sensor are the same. Only the controller has changed.

1.1 Model

Use the Python `SystemFunction` class to model and analyze the behavior of whole system when the robot has a delay-plus-proportional controller, as follows. Open the file

`dl5Work.py` (which imports `sf`) in idle and use it to do the work in this part of the lab. Write a Python procedure `delayPlusPropModel`, that takes gains `k1` and `k2` as input and returns a `SystemFunction` that describes the behavior of the system when the robot has a delay-plus-proportional controller. You can assume that $T = 0.1$ seconds and $V = 0.1$ m/s. Use `sf.Cascade`, `sf.FeedbackAdd`, `sf.FeedbackSubtract`, `sf.FeedforwardAdd`, and `sf.FeedforwardSubtract` to compose primitive `sf.Gain` and `sf.R` components. There is documentation available for `sf.FeedforwardAdd` and `sf.FeedforwardSubtract` in the **Software Documentation** linked off of the **Reference Material** page of the course web page.

Wk.5.1.1. Part 1 Enter your `delayPlusPropModel` code into the tutor. You might find your answer to last week's problem Wk.4.2.5 helpful.

Consider four different values of `k1`: 10, 30, 100, and 300. For each value of `k1`, use `optOverLine` to determine the value of `k2` that minimizes the magnitude of the least stable pole.[1] You implemented a version of `optOverLine` in Week 2; our implementation is available in the `optimize` module, which is imported by `dl5Work.py`.

k1	k2	magnitude of dominant pole
10		
30		
100		
300		

Wk.5.1.1 Part 2 Enter the values of `k2` and the magnitude of the associated dominant pole that you found for each of the values of `k1` above.

[1] Be sure to test both positive and negative values of `k2`, be sure that you have tested a large enough range, and be sure that, ultimately, you have sampled at a granularity of at least `0.1`. Rather than setting up one long minimization run with a wide range and a small granularity, it's better to start a coarse granularity to find the right rough value, and then search more finely around that. Hint: the magnitude of `k2` should not be bigger than that of `k1`.

1.2 Brain

Implement the delay plus proportional controller by editing the `WallFollower` state machine class in `delayPlusPropBrainSkeleton.py`.

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 6 and 7 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.

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`.

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, as described on our web page. **This plot will disappear once you reload the brain.**

Run your brain in the world `wallTestWorld.py` (in the `worlds` directory). Determine how the behavior of the system is affected by the controller gains `k1,k2`. Save plots to illustrate the performance of each of your optimized gain pairs `k1, k2`. Name these files so that you can remember the parameters that were used to generate each one. Keep the files for your oral interview. **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.**

Check Yourself 1. Which gains work best in simulation? `k1 =` `k2 =`

Which gains cause bad behavior?

Run your brain on a robot. Try to start it at the same initial conditions (distance and angle to wall when starting) as in the simulator. Save a plot for each of your calculated gain pairs. Keep the files for your oral interview.

Check Yourself 2. Which gains work best on a robot? `k1 =` `k2 =`

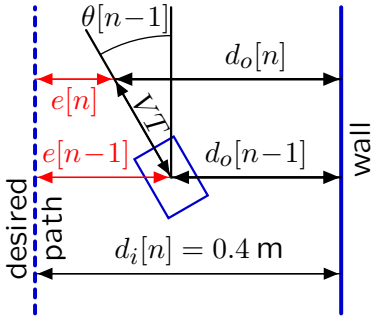
Are the best gains the same as in simulation?
Which gains cause bad behavior?

Checkoff 1. Show a staff member plots for the simulated and real robot runs, and discuss their relationship. How is the robot's behavior related to the magnitude of the dominant pole, for each of the gain pairs?

2 If we are facing in the right direction, all we have to do is keep on walking

Why does the delay-plus-proportional controller from the previous section behave better than the proportional controller from last week? The only difference is access to $e[n-1]$. So why should **old** information be helpful?

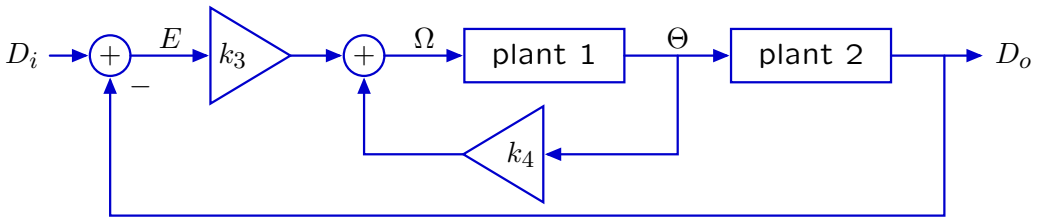
As we can see in the figure (right), if we know $e[n]$ and $e[n-1]$, then we can calculate $\theta[n-1]$. Thus, the delay-plus-proportional controller can base the next angular velocity (its output) on both the position AND angle of the robot. Notice however that the information about position is for time n while the information about angle is for time $n-1$.



How well could a control system work if it had up-to-date information about both position and angle? We can answer this question by analyzing a angle-plus-proportional controller, where

$$\omega[n] = k_3 e[n] + k_4 \theta[n]$$

as shown in the following block diagram.



2.1 Model

Write a Python procedure `anglePlusPropModel` that takes gains `k3` and `k4` as input and returns a `SystemFunction` that describes the system with angle-plus-proportional control. Assume that $T = 0.1$ seconds and $V = 0.1$ m/s. Use `sf.Cascade`, `sf.FeedbackAdd`, `sf.FeedbackSubtract`, `sf.FeedforwardAdd`, and `sf.FeedforwardSubtract` to compose primitive `sf.Gain` and `sf.R` components.

Wk.5.1.3 Part 1 Enter your `anglePlusPropModel` code into the tutor.

For `k3` equal to 1, 3, 10, and 30, determine values of `k4` that minimize the magnitude of the least stable pole of the angle-plus-proportional system.

`k3`	`k4`	magnitude of dominant pole
1		
3		
10		
30		

Wk.5.1.3 Part 2 Enter the values of `k4` and the magnitude of the associated dominant pole that you found for each of the values of `k3` above.

2.2 Brain

Implement the proportional plus angle controller by editing the `WallFollower` state machine class in `anglePlusPropBrainSkeleton.py`.

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 a sequence of **pairs of the perpendicular distance to the wall on the right and the angle to the wall**.

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 a **pair of the perpendicular distance to the wall on the right and the angle to the wall** and whose output is an instance of the class `io.Action`.

Notice that if sonar 6 or 7 is out of range, then we cannot calculate the angle, and the second component of the output of the `Sensor` machine will be `None`. When that happens, your brain should set the angular velocity to `0`.

Test that your brain works in the `soar` simulator with the `wallTestWorld.py` world. Save a plot for each of your calculated gain pairs. Keep the files for your oral interview.

Check Yourself 3. Which gains work best in simulation? `k3 =` `k4 =`

Which gains cause bad behavior?

Run your brain on a robot. Try to start it at the same initial conditions (distance and angle to wall when starting) as in the simulator. Save a plot for each of your calculated gain pairs. Keep the files for your oral interview.

Check Yourself 4. Which gains work best on a robot? `k3 =` `k4 =`

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

Checkoff 2. Show a staff member plots for the simulated and real robot runs, and discuss their relationship. How is the robot's behavior related to the magnitude of the dominant pole, for each of the gain pairs? How does the behavior of the angle-plus-proportional controller compare to that of the delay-plus-proportional controller?

6.01 Introduction to Electrical Engineering and Computer Science I
Fall 2009