Design Lab 6 *6.01 – Fall 2011*

Sizable Following

Goals: We have seen that the simple *proportional controller* does not provide satisfactory performance in guiding a robot to follow along a wall. In this lab, we investigate two improved controllers for the `WallFollower`:

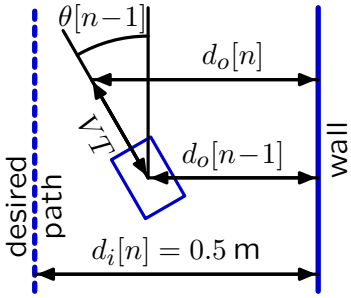
- A `delayPlusProp` controller which depends on the previous distance to the wall as well as the current one
- An `anglePlusProp` controller which depends on the current angle to the wall, as well as the current distance

1 Introduction

Resources: This lab should be done with a partner. Each partnership should have a **robot** and lab **laptop** or a personal laptop that reliably runs soar. Do `athrun 6.01 getFiles` to get the files for this lab, which will be in `Desktop/6.01/designLab06`, or get the software distribution from the course web page. The relevant files in the distribution are:

- `delayPlusPropBrainSkeleton.py`: template of brain for delay-plus-proportional controller.
- `anglePlusPropBrainSkeleton.py`: template of brain for angle-plus-proportional controller.
- `designLab06Work.py`: file with imports for making system functions and for optimization.
- See also **Chapter 5** of the course notes.

Be sure to mail all of your code and plots to your partner. Each of you will need to bring copies with you to your interview.



Last week, we used 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 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; the first controller depends on the previous distance to the wall as well as the current one, and the second controller depends on the current angle to the wall, as well as the current distance.

Some of the software and design labs contain the command athrun 6.01 getFiles. Please disregard this instruction; the same files are available on the 6.01 OCW Scholar site as a .zip file, labeled Code for [Design or Software Lab number].

2 Remembrance of Things Past

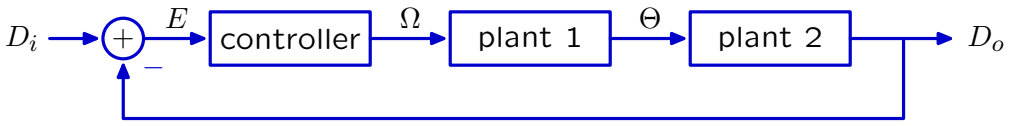
Objective: Make a controller that depends on the previous distance to the wall as well as the current one, using the following steps:

- Build a model of the delay-plus-proportional controller-plant-sensor system, using the `SystemFunction` class.
- Use the model to find the best gains.
- Build a corresponding controller for the robot.
- Test it in simulation for the various gains.
- Test it on a real robot.

A better wall-following controller can be made 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.

Detailed guidance:

Check Yourself 1. Draw a block diagram of the controller.

Remember that ω is lower-case Ω.

2.1 Model

Step 1. 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 `designLab06Work.py` in `idle` and use it for 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.

  You can use `sf.FeedforwardAdd`, and `sf.FeedforwardSubtract`, two combinators for simple addition of systems, as well as any of the other `sf` combinators. They are all documented in the **`sf` module** page of the **Software Documentation** section of the **Reference Material** from the course web page.

  Use your answer to last week's problem **Wk.5.3.4** to help with this.

Wk.6.1.1. Part 1 Enter your `delayPlusPropModel` code into the tutor.

2.2 Picking gains

We want to pick the values of `k1` and `k2` to get the best stable behavior. As we saw in lecture, the speed of convergence is improved by **reducing the magnitude of the dominant pole**. Let's consider the case of a single gain first, in the system you modeled last week. You could construct a function $f(k)$ that computes the magnitude of the system's dominant pole. Now, you want to find the value of k that produces the minimum value of this function.

Finding minima of a function is an optimization process, which can be accomplished using routines from the **lib601 optimize module**, as documented in the **Homework 2 assignment**. This documentation is reproduced below for your convenience.

Given a function $f(x)$, how can we find a value x^* such that $f(x^*) \leqslant f(x)$ for all x? If f is differentiable, then we could do this relatively easily by taking the derivative, setting it to 0 and solving for x. This gets tricky when the function f is complicated, when there may be multiple minima, and/or when we wish to extend to functions with multiple arguments. For functions that aren't differentiable (such as those involving max or abs), there is no straightforward mathematical approach at all. In one dimension, if we know a range of values of x that is likely to contain the minimum, we can plausibly sample different values of x in that range, evaluate f at each of them, and return the sampled x for which $f(x)$ is minimized.

The procedure **`optimize.optOverLine`** does this. It is part of the **lib601 optimize module**, and is called as follows:

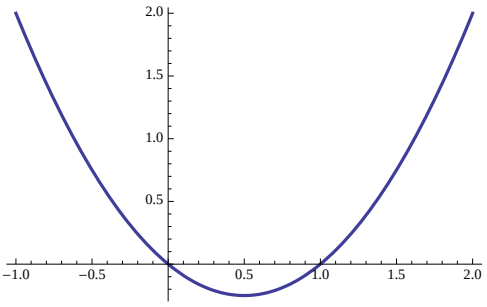
```
optimize.optOverLine(objective, xmin, xmax, numXsteps, compare)
```

- `objective`: a procedure that takes a single numeric argument (x in our example) and returns a numeric value ($f(x)$ in our example),
- `xmin, xmax`: a range of values for the argument,

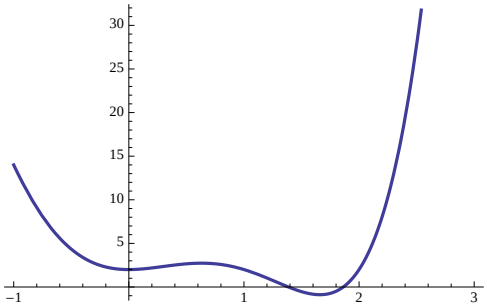
- `numXsteps`: how many points to test within the range,
- `compare`: an optional comparison function that defaults to be `operator.lt`, which is the less than, `<` , operator in Python. This means that if you don't specify the `compare` argument, the procedure will return the value that **minimizes** the objective.

This procedure returns a tuple `(bestObjValue, bestX)` consisting of the best value of the objective ($f(x^*)$ in our example) and the x value that corresponds to it (x^* in our example).

Check Yourself 2. Here are graphs of two functions. Use `optimize.optOverLine` to find the minimum of one of them. The function f has a minimum at $x = 0.5$ of value -0.25; the function h has a minimum at $x = 1.66$ with value -0.88.



$f(x) = x^2 - x$



$h(x) = x^5 - 7x^3 + 6x^2 + 2$

Step 2. Consider four different values of `k1`: 10, 30, 100, and 300. For each value of `k1`, use `optimize.optOverLine` to determine the value of `k2` that minimizes the magnitude of the least stable pole.

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 with 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`.

We encourage you to define and use the `bestk2` procedure stub that is included in `designLab06Work.py` for this purpose.

`k1`	`k2`	magnitude of dominant pole
10	-9.8	0.99
30	-29.76	0.98
100	-97.34	0.946
300	-271.68	0.772

Wk.6.1.1. 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.

2.3 Brain

Step 3. Implement the delay plus proportional controller by editing the `WallFollower` state machine class in `delayPlusPropBrainSkeleton.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 `io.SensorInput` 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.**

Step 4. Run your brain in the world `wallTestWorld.py`. Determine how the behavior of the system is affected by the controller gains `k1,k2`. Use only the gains that you determined in the previous step. Pay attention to the distance values being printed out by the brain. Save plots to illustrate the performance of each of your optimized gain pairs `k1, k2`. Choose names for 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 3. Which of the four gain pairs work best in simulation?

`k1 =` `k2 =`

Which gains cause bad behavior?

Step 5. Connect your lab laptop to a robot. You may need a long ethernet cable (or ask a staff member for help switching to the wireless network).

Go to the edge of the room or out to the hallway to find a long (at least two bubble-wrapped boards) wall to work with.

Run your brain on a real robot. 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 your oral interview.

Check Yourself 4. 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 1. **Wk.6.1.2**: 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?

Explain how you chose the starting state of your controller.

Be sure both partners have the files.

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

Objective: Make a controller that depends on the current angle to the wall, as well as the current distance, using the following steps:

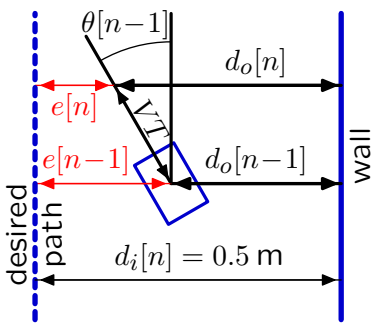
- Build a model of the angle-plus-proportional controller-plant-sensor system, using the `SystemFunction` class.
- Use the model to find the best gains.
- Build a corresponding controller for the robot.
- Test it in simulation for the various gains.
- Test it on a real robot.

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? A related issue is why the best values of $-k_2$ were just a bit smaller than those for k_1.

A better way to think about the delay-plus-proportional controller is as proportional-plus-derivative, where the derivative here is the first difference. As we will see below, if we

re-parameterize the gains in terms of differences instead of delays, then the relation between the gains ($k_2 = -k_1 + \epsilon$) is no longer mysterious.

For this particular problem, the difference $e[n] - e[n-1]$ can be interpreted graphically in terms of the angle $\theta[n-1]$ (see right figure). 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 $n-1$.



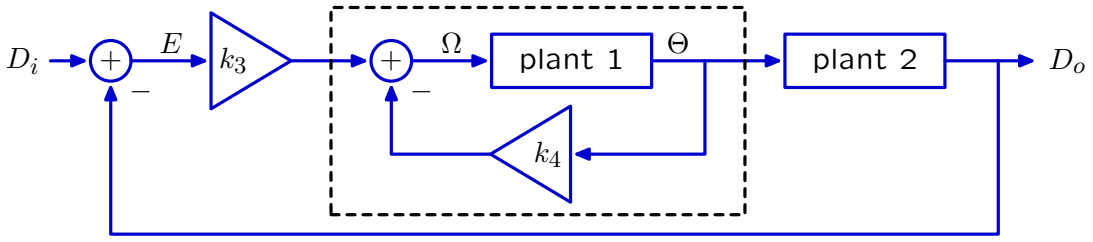
Our robot has more than just one sensor – it actually has eight sonars arranged at slightly different angles – so we can measure the angle directly. How well could a control system work if it had up-to-date information about both position and angle? We can answer this question by analyzing an angle-plus-proportional controller:

$$\omega[n] = k_3(d_i[n] - d_o[n]) + k_4(\theta_d - \theta[n])$$

where $\theta_d = 0$ represents the desired angle (as measured relative to the wall), so that

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

as shown in the following block diagram.



Notice that this controller has a feedback loop to control the angle Θ that is *inside* a second feedback loop to control the distance D_0.

Detailed guidance:

3.1 Model

Step 6. In `designLab06Work.py`, 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.

> **Wk.6.1.3 Part 1** Enter your `anglePlusPropModel` code into the tutor.

Step 7. 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	0.632	0.968
3	1.091	0.945
10	2.0	0.899
30	3.46	0.827

Wk.6.1.3 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.

3.2 Brain

Step 8. 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`.

Step 9. 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 5. Which of the four gain pairs work best in simulation?

`k3 =` `k4 =`

Which gains cause bad behavior?

Step 10. Go to the edge of the room or out to the hallway to find a long (at least two bubble-wrapped boards) wall to work with.

Run your brain on a real robot. 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 your oral interview.

Check Yourself 6. Which of the four gain pairs work best on a robot?

`k3 =` `k4 =`

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

Checkoff 2. **Wk.6.1.4**: 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?

Which controller (delay-plus-proportional or angle-plus-proportional) performs better? Explain why.

Be sure both partners have the files.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011