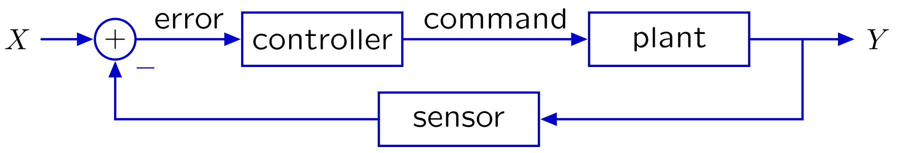
Design Lab 4: Staggering Proportions

We will be working in simulation, so you can use any machine capable of running "soar" and "idle -n". Both partners should log in to the Homework Tutor in the same browser, on two different tabs.

- **Athena machine:** Do `athrun 6.01 update` and `add -f 6.01`.
- **Lab laptop:** Do `athrun 6.01 update`.
- **Personal laptop:** Download design lab 4 zip file from course web page.

Code and data are in `~/Desktop/6.01/lab4/designLab/`.
Remember to email your code and plots to your partner.
You will need to bring these to your oral interview for discussion.

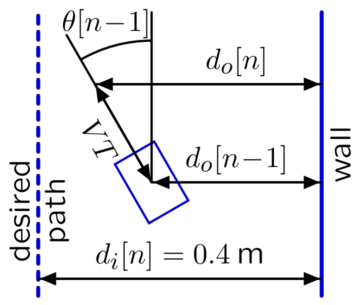
In this lab, we will program the robot to move along a wall, maintaining a constant, desired distance from the wall. We will use a simple **proportional controller**, with the same structure as the controller that we used for last week's **wallFinder** system.



We will observe the behavior of this week's **wallFollower** system, and we will analyze a signals and systems model to understand the behavior of this system.

1 Proportional wall-follower

The figure at the right illustrates the structure of the wallFollower system.



Fix the forward velocity $V = 0.1\,\text{m/s}$, and adjust the rotational velocity $\omega[n]$ (not shown) to be proportional to the *error*, which is the difference between the desired distance $d_i[n] = 0.4\,\text{m}$ to the right wall and the actual distance $d_o[n]$. Notice that when the rotational velocity $\omega[n]$ of the robot is positive the robot turns towards the left, thus increasing its angle $\theta[n]$. Let $T = 0.1$ seconds represent the time between steps.

Implement the proportional controller by editing the brain in `propWallFollowBrainSkeleton.py`. The brain has two parts connected in cascade. The first part is an instance of the `Sensor` class, which implements a state machine

Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

whose input is of type `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). All of 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 proportional controller.

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

Run your brain in the world `wallTestWorld.py`. Determine how the behavior of the system is affected by the gain `k` of the proportional controller. The brain in `propWallFollowBrainSkeleton.py` issues a command during the setup to rotate the robot, so it starts at a small angle with respect to the wall. Generate slime trails to illustrate how the system's performance depends on the value of the proportional gain `k`; for instance, does it affect the convergence of the output to the desired value?. **Be sure to take and save screen shots of some of your slime trails.**

Checkoff 1.
- Show your slime trails to a staff member, and explain the implications of the slime trails for choosing the "optimum" value of `k`.
- Compare the range of behaviors of the wallFollower system to the range of behaviors of the wallFinder system from last week.

2 Analytical model

The signals and systems approach allows us to predict analytically salient aspects of the behavior in these simulations for **any** value of the gain without having to run the simulation.

Develop an analytical model of the proportional wall follower as follows (see an analogous development for `wallFinder` in section 6.5 of the readings):

- Assume that the controller can instantly set the rotational velocity $\omega[n]$ to be proportional to the error signal $e[n] = d_i[n] - d_o[n]$. (This is an approximation, which simplifies the analysis. If you like, you could test the validity of this assumption after you have finished the lab.) Express this relation as a difference equation.

- Write difference equations describing the "plant," that is, an expression for $d_o[n]$, that depends on ω. It is useful to break this problem into two parts:

– an expression for $\theta[n]$, that depends on ω and

– an expression for $d_o[n]$, that depends on θ.

Linearize the relation between d_o and θ using the small angle approximation ($\sin\theta \approx \theta$). (This is an approximation, which simplifies the analysis. If you like, you could test the validity of this assumption after you have finished the lab.)

The subsystems represented by your difference equations connect together to form a system of the following form. Note that in this model there is no delay in the sensor. Label each wire in the block diagram with the name of the corresponding signal.

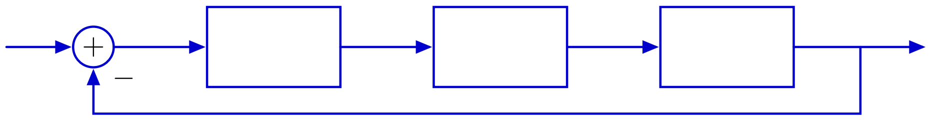


Figure 1 Structure of Control System

- Convert your difference equations into operator ($\mathcal{R}$) equations, and find the system function.

$$H = \frac{D_o}{D_i} =$$

Wk.4.2.2 Enter the system function into the tutor by entering the numerator and denominator polynomials.

- Find an algebraic expression for the poles of the system (ask a staff member to check it before you spend time computing periods).

- Assume that the time per step of the robot $T = 0.1s$, and that $V = 0.1m/s$. Determine whether the system's behavior is periodic for gains of 0.5, 1.0, 2.0, 4.0, and 10.0. If it is periodic, determine the period.

Checkoff 2. Show your system function to a staff member. Discuss what the poles reveal about the behavior of the system (oscillation? stability?). Explain what happens to the period as the gain increases.

3 Software model

Now, use the Python `SystemFunction` class to model and analyze the proportional wall follower, as follows. (We suggest that you open the file `dl4Work.py` (which imports `sf`) in idle and use it to do the work in this part of the lab.)

The `SystemFunction` class provides two primitive kinds of system functions, gain (`sf.Gain`) and delay (`sf.R`). They are implemented as functions, but named with uppercase variables by analogy with the `sm.Gain` and `sm.R` state machines.

```
def Gain(k):
    return sf.SystemFunction(poly.Polynomial([k]), poly.Polynomial([1]))
def R():
    return sf.SystemFunction(poly.Polynomial([1, 0]), poly.Polynomial([1]))
```

These can be combined with `sf.Cascade`, `sf.FeedbackSubtract`, and `sf.FeedbackAdd` to make any possible system function; and the structure of the combination will be the same as it would have been to build up the analogous state machine.

Implement Python functions called `controller`, `plant1`, and `plant2` that return instances of the `SystemFunction` class that represent the three blocks in **figure 1**. Pass the important parameters for each block (e.g., `k`) as inputs to the corresponding function.

Write a Python function `wallFollowerModel(k, T, V)` that calls the previous Python functions to make system functions for the components and composes them into a single `SystemFunction` that describes the system with `desiredRight` as input and `distanceRight` as output.

Checkoff 3. Discuss the structure of your code with a staff member.

Wk.4.2.5 Enter the definition for `wallFollowerModel`. Do not enter any `import` statements.

Use the `dominantPole` method of the `SystemFunction` class, and the function `sf.periodOfPole(p)` to determine whether your model predicts oscillatory behavior and, if so, with what period, for gains 0.5, 1.0, 2.0, 4.0, and 10.0. Compare to the values you derived by hand.

Visualization

You can construct an `LTISM` state machine that corresponds to your system function with (assuming `k` is set to some gain value):

```
mySF = wallFollowerModel(k)
sm = mySF.differenceEquation().stateMachine(previousInputs, previousOutputs)
```

where `previousInputs` and `previousOutputs` are lists of values used to initialize the state machine. The lengths of the lists depend on the order of the difference equation for the system, as they did in the `LTISM` class. The values in `previousInputs` would generally be `desiredRight`. The values in `prevousOutputs` would generally be some constant (that is different from `desiredRight`) indicating the initial distance from the wall. You could then simulate the wall following behavior with

```
result = ts.TransducedSignal(sig.ConstantSignal(desiredRight), sm)
```

To see what the resulting signal is like, just do `result.plot(end = 300)`. (Hint: remember the `-n` switch for `idle -n`.)