MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 5, Issued: Tuesday, Mar. 4: Includes revised design lab

Overview of this week's work

In software lab

- Work through the software lab.
- Paste your solution to Question 7 into the tutor.

Before the start of your design lab on Mar 4 or 5

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section PS.6.2.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Take the nanoquiz in the first 15 minutes; don't be late.
- Work through the design lab with a partner, and take good notes on the results of your work.

At the beginning of your next software lab on Mar 11 or 12

- Submit online tutor problems in section PS.6.3.
- Submit written solutions to questions 7–9, 11, 14–19, and 20. All written work must conform to the homework guidelines on the web page.

Get the lab5 files via `athrun 6.01 update` or from the home page.

Software Lab: System Functions for Robot Controller

We saw in lecture this week that the behavior of a linear time-independent system can be completely characterized using a ratio of two polynomials in R, the delay operator. We call this characterization the *system function*. Further, we saw that systems composed of multiple LTI components can be described using system functions that are relatively simple compositions of the system functions of the components. It is possible to solve such problems by hand, with a paper and pencil, but why do that when we can get a computer to do the work for us?!

In this software lab and the next we will build and use a set of tools for analyzing, simulating, and combining system functions. The file `SystemFunctionSkeleton.py` contains the definition of the class `SystemFunction`, with some methods filled in, and others just documented, but with `pass` instead of the method body. We'll fill in its methods this week and next. We'll also give you `SystemFunction.pyc`, which is a complete compiled version of that module.

Making System Functions

The module `SystemFunction` contains a function that constructs a new instance of the `SystemFunction` class, given a characterization of that system function as a difference equation. Assume that the difference equation has the form

$$a_0 y[n] + a_1 y[n-1] + ... + a_k y[n-k] = b_0 x[n] + b_1 x[n-1] + ... + b_j x[n-j] \ ,$$

where X is the input signal and Y is the output signal. It is crucial that the first coefficient of both polynomials be from the same time step; but they can go back different depths in history.

To make a new system function, say corresponding to the difference equation

$$9y[n] - 4y[n-1] + 5y[n-3] = 2x[n-1] \ ,$$

we'd make the call

```
> sf = systemFunctionFromDifferenceEquation([9, -4, 5], [0, 2])
```

You can see from printing out the system function, that we store it as a ratio of polynomials in R:

```
> sf
SF(2.000R/5.000 R**2 + -4.000R + 9.000)
```

Question 1: Describe in English what this system does:

```
systemFunctionFromDifferenceEquation([1], [0, 0, 1])
```

Question 2: What call would you make to generate the system function for a system with this difference equation:

$$y[n] = 2y[n-2] + 3x[n-3] \ ?$$

Understanding a system

A crucial thing to understand about a system is whether, even in the absence of input, its output will go to zero, or whether it will grow unboundedly. We can find this out by determining the *poles* of the system. So, for example, if you ask for the poles of a system function, you'll get a report on the poles of the system and their implications for the stability of the system:

```
> sf = systemFunctionFromDifferenceEquation([9, -4, 5], [0, 2])
> sf.poles()
Magnitude of poles:  [0.7453559924999299, 0.7453559924999299]
[(0.22222222222222227+0.71145824860364981j),
 (0.22222222222222218-0.71145824860364981j)]

> sf2 = systemFunctionFromDifferenceEquation([3, -4, 5], [0, 2])
> sf.poles()
Magnitude of poles:  [1.2909944487358056, 1.2909944487358056]
Danger.  Warning.  Unstable system.
[(0.66666666666666663+1.1055415967851332j),
 (0.66666666666666663-1.1055415967851332j)]
```

Another way to get an understanding of how a system is working is to start it at some initial conditions, and simulate it by running the associated difference equation forward. We have built facilities for doing this into the `SystemFunction` class.

Let k be the order of the polynomial on the Y terms (also called the order of the system) and j be the order of the polynomial on the X terms. We need j initial input values and k initial output values in order to be able to specify all future values of the system. In addition, we need to provide an input sequence; for generality, instead of providing a list, we use a function from n to $x[n]$.

So, given initial values (the lists $[x[0], \ldots, x[j-1]]$ and $[y[0], \ldots, y[k-1]]$) and an input source, the `SystemFunction.plotSequence` method plots the sequence of Y values.

So, for example, you can do

```
> sf2 = systemFunctionFromDifferenceEquation([3, -4, 5], [0, 2])
> sf2.plotSequence([1], [0, 1], lambda x: 1, n = 30, idle = True)
```

and get the plot shown in figure 1. The last two arguments are optional. `n` controls the number of points that are plotted; `idle` controls whether the plotting will work nicely with Idle. *If you call this with* `idle` *set to* `True`*, then you will have to close the plotting window before your program that called this method (or the Python shell) can continue to run.*

Question 3: Experiment with plots of `sf2` above with different initial conditions and inputs.

Robot meets wall

Now, let's return to the problem of the robot driving up to the wall. We would like to determine whether this system is stable. Doing so will require several steps, laid out in the questions below.

Question 4: Imagine a robot driving straight toward a wall. We would like to use a difference equation to model this system, where the output, D, is the distance from the robot to the wall in meters, and the input, V is the robot's current forward velocity in meters per second. Assume that each state transition corresponds to the passage of 0.2 seconds, and that D at time t depends on V at time $t-1$. Write a difference equation to model this system. (Do this on paper).

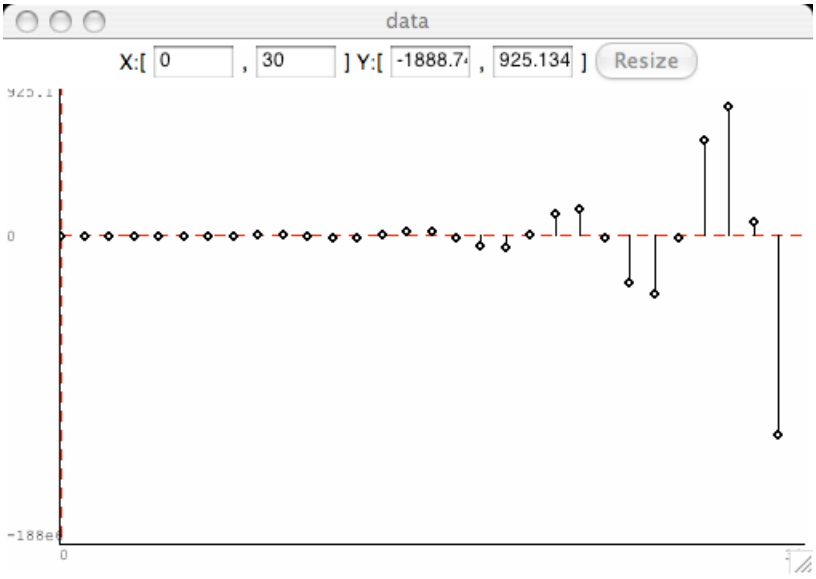

Figure 1: Behavior of system function `sf2`.

Question 5: Now, let's think of the controller that this robot might be executing. Assume the robot would like to stop at some distance $d_{Desired}$ from the wall. The controller can be modeled as a difference equation where the output, V is the robot's selected velocity, and the input D is its current measured distance from the wall. Assume that V at time t depends on D at time $t-1$. For now, assume there's no inertia involved, and so the controller can choose any new velocity it wants, independent of its previous velocity. In particular, let's assume it selects a new velocity that is equal to $k(d - d_{Desired})$, where d is the current distance to the wall, $d_{Desired}$ is the desired distance to the wall, and k is a "gain" constant. Write a difference equation to model this controller. You can treat $d_{Desired}$ as another input signal. (Do this on paper)

Question 6: Observing that the sequences D and V in these two systems are actually the same, we'd like to understand how the whole system behaves. The first step is to take the two equations that you developed in the previous two problems, and convert them into a single equation that only involves values of D. The new equation will define d[t] in terms of d[t−1], d[t − 2], and $d_{Desired}$. (Do this on paper, by converting to operator equations with the R operator, and then combining).

Question 7: Write a function that takes a value for the gain k as input, and returns an instance of the `SystemFunction` class corresponding to the difference equation you derived in the previous problem. Pick a value of k, and find out what the poles of the corresponding system are.

Question 8: Write a simple Python program that computes the poles of the system for a wide range of k. Find the range of values of k for which the system will be stable.

Question 9: Find 4 different values of k, that satisfy the following conditions: 1. converges monotonically; 2. converges in oscillation; 3. diverges monotonically; 4. diverges in oscillation. Using the `plotSequence` method of `SystemFunction`, plot sequences of D for systems with each of those values of k. (Note that the `inputSource` argument to `plotSequence` is a function from n to x[n]; we assumed above that the input was $d_{Desired}$, so you can use `lambda x: dDesired` for that argument.) Discuss the relationship between the predictions based on the poles of the system for different k, and the observed behavior sequences.

Run it in SoaR

Now, you can do the demo we did in class on the first day. For now, we'll just do it in simulation, though feel free to get out a robot and try it, as well. The file `wallControllerSkeleton.py` contains code for a very simple brain to control the robot. It computes the distance to the wall using the sonars, and only needs for you to decide what velocity commands to make.

Question 10: Put in your control rule and see how the robot behaves. Does its behavior match the predicted behavior of the model?

Question 11: If you set the variable `makeGraph` to `True`, then after `runLength` steps, a graph of the distance values will be printed out. Compare those to the graphs you got in the previous section for the same gain. How do they relate?

Exploration: Symbolic Algebra

This is worth 5 out of 10 points for the Week 5 exploration.

When we solve problems like trying to find a good gain for a system, we typically do it algebraically, leaving `k` as a variable, working out the answer, and then inferring values of `k` that have some desirable property. In this exploration, we'll ask you to construct a set of classes for symbols and algebraic expressions, together with overloadings of the usual Python arithmetic operators, so that it will support an interaction like this:

```
>>> p1 = Polynomial([S('a'), S('b'), 7, S('c')])
>>> p2 = Polynomial([2, S('a')])
>>> p1 + p2
a z**3 + b z**2 + 9.000z + (a + c)

>>> p1 * p2
(2.0 * a) z**4 + ((a * a) + (2.0 * b)) z**3 +
  ((a * b) + 14.0) z**2 + ((7.0 * a) + (2.0 * c))z + (a * c)

>>> sf1 = systemFunctionFromDifferenceEquation([1, -1, S('k')], [S('k')])
>>> sf1
SF(k/k R**2 + -1.000R + 1.000)
>>> sf2 = systemFunctionFromDifferenceEquation([1, S('k'), -3], [1, S('k')])
>>> sf2
SF(kR + 1.000/-3.000 R**2 + kR + 1.000)

>>> sf1.cascade(sf2)
SF((k * k)R + k /
(-3.0 * k) R**4 + ((k * k) + 3.0) R**3 + (k + (-1.0 * k) + -3.0) R**2
  + (-1.0 + k)R + 1.000)
```

We generated that listing *without changing anything about the* `Polynomial` *or* `SystemFunction` *classes*, relying entirely on Python's operator overloading.

You can approach this problem any way you'd like (though you shouldn't need or want to touch those classes), but ultimately, it should allow you to do basic arithmetic operations with mixed symbolic and numeric expressions. When we use the phrase "symbolic expressions", we don't mean just primitive symbols like `S('a')`, but also expressions involving them, like `S('a') * S('b')`. Your operations should be compositional, so that you can make ever more complicated expressions.

Exploration 1:	Demonstrate that you can create and print (nicely) symbolic expressions.
Exploration 2:	Show that you can add symbolic expressions.
Exploration 3:	Show that you can subtract symbolic expressions.
Exploration 4:	Show that you can multiply symbolic expressions.
Exploration 5:	Show that you can divide symbolic expressions.

Ideally, you'll do some basic simplification (e.g., noticing that $0 * a$ is 0, and maybe even simple gathering of terms, so that $a + a$ is $2 * a$), but doing a really good simplifier is quite difficult, and not necessary for this exploration.

If you want to go a bit above and beyond, supply a `val` method that takes a dictionary that maps strings to values, and evaluates an expression. So, for example, you'd get

```
>>> (S('a') + S('b')).val({'a':4, 'b':7})
11
```

If you want to go *way* above and beyond, push the symbols through the quadratic root computation, and find algebraic expressions for the bounds of the region where the system is stable.

Below, we outline the approach we took to this problem. You are welcome to approach it the same way, or in some other way.

```
# Generic superclass.  Nothing is generically an AlgExp, but we put
# the operators here.  You can put a lot of simplification here, as
well.
class AlgExp:
    def __add__(a, b):
        if isNumber(b):
            return a + N(b)
        return Sum(a, b)

    # Called when a + b, b isn't an algexp, but b is
    def __radd__(b, a):
        return N(a) + b

    def __mul__(a, b):
        if isNumber(b):
            return a * N(b)
        return Prod(a, b)

    def __rmul__(b, a):
        return N(a) * b

    def __div__(a, b):
        if isNumber(b):
            return a / N(b)
        return Frac(a, b)

    def __rdiv__(b, a):
        return N(a) / b

    def __neg__(a):
        return Neg(a)
    def __sub__(a, b):
```

```
        if isNumber(b):
            return a - N(b)
        return a + Neg(b)
    def __rsub__(b, a):
        return N(a) - b

    def __str__(self):
        return repr(self)

# Negation of an expression
class Neg(AlgExp):
    # your code here
    pass

# Sum of two expressions
class Sum(AlgExp):
    # your code here
    pass

# Product of two expressions
class Prod(AlgExp):
    # your code here
    pass

# Quotient of two expressions
class Frac(AlgExp):
    # your code here
    pass

# A number
class N(AlgExp):
    # your code here
    pass

# A symbol
class S(AlgExp):
    # your code here
    pass

def isNumber(x):
    return isinstance(x, int) or isinstance(x, float)
```

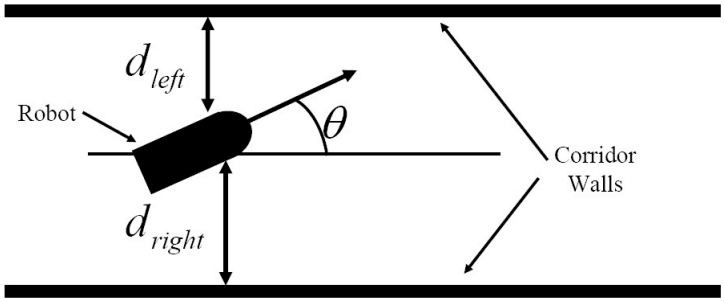

Figure 2: Robot in corridor.

Design Lab: Driving down a hall

Up to this point in the course, you have developed progressively more intelligent approaches to controlling the robot's actions based on sensory measurements. In those labs, the primary goal was to develop systematic ways of organizing software for construction complex systems of robot behaviors.

In this lab, you will be using measurements to control the robot in a feedback system, and will be investigating a situation in which detailed analysis is needed to achieve a desired performance. More specifically, we are asking you to investigate how to control the robot so that it drives down the center of a narrow corridor at a constant forward velocity. Such a task is similar to what an automobile driver does when keeping a car on the road. We will suggest a simple feedback scheme to keep the robot in the center of corridor, and ask you to develop and analyze a mathematical model based on difference equations that explains the behavior of the suggested simple feedback system.

In this lab you will be controlling the robot to drive down a narrow corridor as shown in Figure 2.[1] Notice that in the figure, we have denoted the direction of the robot's forward motion, the distance to the left wall, $\mathtt{d_{left}}$, the distance to the right wall, $\mathtt{d_{right}}$, and the angle of the robot with respect to the parallel walls, θ.

Consider the problem of trying to steer the robot down the center of the corridor while keeping the robot moving forward with a constant velocity. For example, if the robot is in the center of the corridor and pointing in a direction parallel to the walls, one could keep the robot moving forward down the center of the corridor with the command

```
motorOutput(0.1, 0.0)
```

which will keep the robot moving forward 0.1 meters per second. If the robot is too close to the right wall, one could issue the command

```
motorOutput(0.1, 0.2)
```

which will keep the robot moving 0.1 meters per second but will cause the robot to rotate to the left (increasing θ). If the robot is too close to the left wall, one could issue the command

```
motorOutput(0.1, -0.2)
```

[1] The optical sensing strategy and software used in this lab was developed by Karim Liman-Tinguiri, a former 6.01 student.

which will keep the robot moving forward at `0.1` meters per second but will cause the robot to rotate to the right.

The above observations suggest a very simple feedback system that keeps the robot moving forward at `0.1` meters per second and tries to keep the robot in the center of the corridor. One could set the forward velocity in the `motorOutput` command to `0.1` and set the rate of rotation in the `motorOutput` command to be proportional to the distance from the center of the corridor.

Measuring Distances

In previous labs, you measured distances to objects using the robot's built-in sonars. For this lab, we have added optical range sensors to the robots; these sensors are somewhat less noisy than sonars (especially when looking at surfaces that aren't covered in bubble wrap), and they can generate readings with less lag (light is faster than sound).

You will be using a set of four optical sensors mounted in a diamond configuration on the top of the robot (near the robot's center of rotation). There are four sensors so that triangulation can be used to determine the shortest distance from the robot center to each of the corridor walls, even if the robot is pointed in a direction that is not perfectly parallel to the wall.

The four optical sensors encode distance information as analog voltages, and these analog voltages are converted to digital numbers that are transmitted to your laptop via a USB connection. A National Instruments digital-to-analog converter (NIDAQ) (the white box on top of the robot) performs the conversion, and the NIDAQ is managed by running a server program on your laptop. This server program makes the converted analog voltage data available to Python programs.

In order to use the optical sensors, **you must turn on the power to the robot**, then open a terminal window and change to the `lab5` directory. Then make sure the USB cable from the NIDAQ is plugged into the USB port of your laptop (if so, the green light on the NIDAQ will blink a few times, and then quit). In the terminal window, type

```
./NIDAQserver
```

After a few seconds (possibly even a minute), the terminal screen will start filling with numbers. You may minimize this terminal window, but do **NOT** kill the window. If it prints an error message, but starts to spew numbers, everything is fine; don't worry about the error message. If it can't connect, check to be sure your USB cable is plugged into the laptop correctly.

Start soar, and then start the virtual oscilloscope program by clicking on the cool new oscilloscope icon on the far right of the icon bar.

An oscilloscope window should appear on your laptop screen, and the first four channels (Ai0 through AI3) from the NIDAQ should be plotted in the oscilloscope window. Try blocking one of the optical sensors and notice what happens in the oscilloscope window. The blue line is the front left sensor reading, red line is the rear left, magenta line is the rear right, and the green line is the front right. If the oscilloscope does not seem to be working (or if you've unplugged the NIDAQ box or turned off the robot), close the oscilloscope window, unplug the NIDAQ USB connection, wait three seconds, replug the USB connection and restart the server (you might need to type Control-C in the window where it was running before you can restart it) and hit the oscilloscope button. Please ensure that all four distance sensors are functional by waving an object in front of each of them in turn. You should observe a spike in voltage readings for that channel. In case

you're interested, the figure in the original handout contains a plot of the nominal voltage readings for these sensors, as a function of distance to an obstacle.

Now, connect your laptop to a robot with a serial cable (both the USB cable to the NIDAQ and the serial cable to the robot are needed for this lab).

PLEASE NOTE: It is very easy to accidentally block one of the optical sensors with either of the two cables connecting the robot and the NIDAQ to the laptop. Typically, a single sensor will be blocked, and then very odd things happen. This problem is best avoided by carrying the laptop behind the robot and making sure the cables do not cross the line of sight of the optical sensors.

Now, open the SoaR brain in `feedbackBrain.py`. We have provided a function `GetLR`, which returns the distances from the center of the optical sensor mount to the left and right walls, under the assumption that both of the optical sensors on a given side of the robot are "seeing" the same straight wall. Converting the voltages from the sensors to distances requires the use of the function plotted in the original handout, followed by some geometric calculations to recover the shortest distance between wall and robot from two distance readings on a given side of the robot. Writing such a conversion procedure is an interesting problem, but we have provided a program that returns the needed distances, in the interest of time.

You can test your robot's sensors further by changing all the velocities in the motor commands to 0, and printing out the left and right sensor readings. Now, put the robot in a hallway. Move it (by picking it up) from side to side and be sure the readings remain reasonable. Rotate the robot. Be sure you understand why the sensor readings do what they do.

Checkpoint: 30 minutes

- Demonstrate that your sensors are working; explain the way that they do or do not change when you displace your robot sideways or rotate it within a hallway.

Trying the robot control system

We're going to design a controller that sets the rotational velocity to be proportional to the robot's signed distance from the corridor center. The definition of the distance from the corridor center must be thought about carefully. It will turn out that it is convenient to measure the distance of the robot *to the left of the center line.*

Question 12: Determine the formula for `d`, the distance to the left of center, in terms of the distances to the right and left walls (which is what the robot senses), and be sure your formula works when the robot is against the left wall or up against the right wall.

Now, we'd like the robot to travel straight down the corridor, some distance $\texttt{d}_{Desired}$ to the left of center. Change the program in `feedbackBrain.py` so that the rotational velocity is proportional to the difference between the robot's actual distance `d` from the corridor center and $\texttt{d}_{Desired}$. Try your brain on your robot in one of the narrow corridors we have set up in the lab. Experiment with different gains in your feedback system, and try starting the robot both near and far from the center of the corridor. You can just let $\texttt{d}_{Desired}$ be 0 for now, so the robot tries to go right down the middle.

Question 13: Include the code for your brain. What is the effect on robot behavior of changing the feedback gain and the initial position? Try several gains and several initial positions and angles for the robot and describe the results.

Checkpoint: 90 minutes

- Demonstrate your robot moving in the corridor with several gains and initial positions.

Difference Equation Derivation

When soar is running your brain, it executes the step function about 5 times per second. If you wrote your brain correctly, this means that 5 times every second, your brain is taking sensor readings to determine the offset from center, and then updating the robot's rate of rotation. Now, our job is to develop a set of difference equations that describe the way the robot interacts with the world. Ultimately, we'd like to develop a single difference equation that describes a system with $d_{Desired}$ as the input and d as the output. But we'll do it in pieces.

First, let's think about how d (the distance to the left of center) depends on θ. The robot is displaced to the left of the center by an amount $d[n]$, and is moving with a speed V in a direction that makes an angle $\theta[n]$ as shown in Figure 2. So

$$\begin{aligned} d[n] &= d[n-1] + V\delta_T \sin\theta[n-1] \\ &\approx d[n-1] + V\delta_T\theta[n-1] \end{aligned}$$

where the approximation holds if the angle $\theta[n-1]$ is small.

Now, we'll ask you to write down difference equations to describe the rest of the system. We found it convenient to break it down into two parts.

- Let $\Omega[n]$ be the robot's *rotational* velocity at time n. Start by writing a difference equation that describes how θ depends on Ω (this is much like the dependence of between d and v in the example we did in software lab).
- Now, we have to relate Ω to d and $d_{Desired}$. This is exactly what your brain does. You can assume that the relationship between the sensor values and the rotational velocity command is instantaneous. (This assumption is different from the one we made in software lab.)
- Finally, combine these three difference equations (by converting them into operator equations and doing algebra) into a single difference equation relating d to $d_{Desired}$.

You'll need to switch to using the `SystemFunction` software in IDLE, rather than playing with the robot in soar. See the reference section at the end of this handout for some instructions on how to use it smoothly.

Question 14: Show your individual difference equations from above and the derivation of the difference equation for `d[n]`.

Question 15: Think about whether `K` should be positive or negative: Which way makes the robot turn in the correct direction in response to a deviation from the center of the corridor? Write a simple Python program that computes the poles of the system for a range of `k` with the correct sign. (Note that the Python range function always increments by integers; a while loop works best for this).

Question 16: What does your answer to the above question tell you about the solution to the difference equation model of the robot feedback system?

Question 17: Show plots of the difference equation output sequence for different feedback gains and a nonzero initial offset from the corridor center. Relate what you see in your plots to the poles you found in question 15.

Question 18: What does the model tell you about your ability to find a feedback gain for which the robot will not eventually hit the wall of the corridor?

Checkpoint: 2 hours and 30 minutes

- Show your difference equation model for the robot displacement from center to an LA.
- Describe your answers to the above questions to an LA.

Validating your model

Question 19: Determine and execute experiments to validate your difference equation model. That is, show that your model predicts the actual robot's behavior. What does the model predict about how the robot will behave as you change the feedback gain? Is the model accurate?

There are a number of constants to determine, and units must be verified as consistent. Do the optical sensors return distances in meters? Does the `motorOutput` command set the forward velocity in meters per second and set the rotational velocity in radians per second? What is the sample rate of your feedback system?

To gather data from your robot, be sure that the variable `makeGraph` in `feedbackBrain.py` is set to `True`. It will store the left distance values, and after a fixed number of steps (currently set to be 100, but you can change it by changing the value of `runLength`), it will write the values out to a file called `leftValues.py`, and draw a graph of the series.

If you want to read the values in from the file (for example, to plot them on the same axes as another graph), then you can use the following commands in `idle`:

```
import pickle
f = open("foo.py", "r")
data = pickle.load(f)
f.close()
```

Checkpoint: end of lab

- Show data from actual robot run and the model using the same controller and gains.

Troubleshooting

- Bad magic number means you have an old version of Python. On athena, do `add -f 6.01`.
- Virtual oscilloscope is slow: Be sure you have the latest version of Python.
- All the optical sensors move together. Maybe robot is not turned on, or there's a loose power wire.
- When you start the NIDAQ server, it generates an error, but it's okay to ignore.

Using SystemFunction software

- Make a new file, called `ps5work.py` in the `lab5` directory, and put this line at the top:

  ```
  from SystemFunction import *
  ```

  Now, run this file in idle. At this point, you can type expressions like

  ```
  > sf = systemFunctionFromDifferenceEquation([9, -4, 5], [0, 2])
  ```

  at the Python terminal.

- Consider the difference equation

$$y[n] = y[n-1] + y[n-2] + x[n] + x[n-1] \ .$$

  When we want to plot, or to compute the value at a particular point in the sequence, we need to know some previous values. In order to get this process started, we have to provide initial conditions. In this example, to compute $y[0]$, we need to know $y[-1]$, $y[-2]$, and $x[-1]$. We're assuming that the input source $x[n]$ is available starting at $n = 0$. Imagine that `sf6` is the name of the system function above. Then, to make a plot of its values, with $y[-1] = 10$, $y[-2] = 20$, $x[-1] = 100$, and $x[n] = n + 1$ for n greater than or equal to 0, you'd need to make a call

  ```
  > sf6.plotSequence([100], [20, 10], lambda n: n + 1)
  ```

Post-lab Homework: systems and stability

A crucial thing to understand about a system is whether, even in the absence of input, its output will go to zero, or whether it will grow unboundedly. We can find this out by determining the *poles* of the system. If the polynomial in the denominator of the system function has the form

$$a_0 + a_1 R + a_2 R^2 + ... + a_k R^k \ ,$$

then the poles are the roots of the equation

$$a_0 + a_1 \frac{1}{z} + a_2 \frac{1}{z}^2 + ... + a_k \frac{1}{z}^k = 0 \ .$$

Multiplying through by z^k, we find that we are interested in the roots of the equation

$$a_0 z^k + a_1 z^{k-1} + a_2 z^{k-2} + ... + a_k = 0 \ .$$

The polynomial can be constructed from the one in `R` simply by constructing a new polynomial with a reversed coefficient list (remember that the first element in the list of coefficients to the `Polynomial` initializer is the coefficient of the variable raised to the highest power.)

Now, the roots of this polynomial may be either real or complex. What matters most, in the question of stability, is whether their magnitude is greater or less than one. If the magnitude of any root is greater than 1, then the system is unstable. You can compute the magnitude of a complex number, `x` using `abs(x)`.

Question 20: Fill in the body of the `SystemFunction.poles` method in the file `SystemFunctionSkeleton.py`. It should do the following things:

- Return the roots of the polynomial, as long as the polynomial is first, second, or third order.
- Print out the magnitudes of the poles
- Print a warning if the system is unstable

Concepts covered in this lab

- Difference equations as models of input/output systems.
- Derivation of difference equation models of coupled systems from difference equation models of their components.
- Relationship between behavior of the system and the poles of the system function.

Exploration: Buying Toothbrushes

This is worth 5 out of 10 points for the Week 5 exploration.

As the owner of a local convenience store, you must not only sell products to your customers, you must also purchase products to fill your shelves. Being a graduate of EECS, you decide to automate the latter task. To limit the number of purchases that you make, you decide to place orders just once a week: on Mondays at 9am. Such orders are filled within two weeks, so that by Monday at 9am you can add to your shelves the products that were ordered two weeks ago.

The point of this exploration is to analyze the performance of several different algorithms for placing orders. Let $y[n]$ represent the "excess inventory" at the beginning of the n^{th} week. The excess inventory is the number of toothbrushes on your shelves minus the number that you believe is optimum. (If your inventory is currently 90 and your target is 100, then the excess is -10.) Let $x[n]$ represent the number of toothbrushes sold during the previous week.

- Algorithm 1: Each Monday, place a new order for $x[n]$ toothbrushes.
- Algorithm 2: Each Monday, place a new order for $x[n] - y[n]$ toothbrushes.
- Algorithm 3: Each Monday, place a new order for $x[n] - ky[n]$ toothbrushes.

Exploration 6: Analyze each of these algorithms as follows.

Exploration 7: Draw a block diagram of the transformation from $x[n]$ to $y[n]$.

Exploration 8: Determine the system functional and poles (if any) for each of the block diagrams.

Exploration 9: Characterize the temporal behavior of these algorithms by calculating their step responses, as follows. Assume that on your first day of business (which is a Monday), your excess inventory is zero. Make a plot of your excess inventory as a function of n, assuming that customers buy 10 toothbrushes each week.

Exploration 10: Identify the primary weakness of algorithm 1.

Exploration 11: Explain how algorithm 2 addresses the primary weakness of algorithm 1.

Exploration 12: What is the primary weakness of algorithm 2?

Exploration 13: For algorithm 3, determine an "optimal" value of k, and explain the sense in which your choice of k is optimal.

Exploration 14: Does algorithm 3 address the primary weakness of algorithm 2?

Exploration 15: How is algorithm 3 better/worse than algorithm 1?