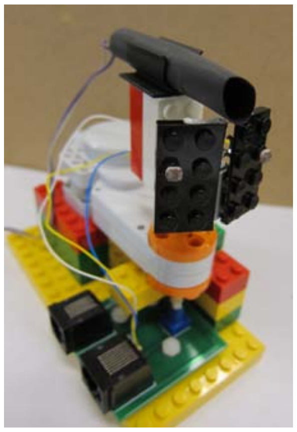
Homework 2: Heads Up!

This assignment includes all problems in **Wk.5.5**.x in the online tutor. For any of the sections of the assignment that require you to write code, please do your work in the file `hw2Work.py`.

You can discuss this problem, at a high level, with other students, but everything you turn in must be your own work.

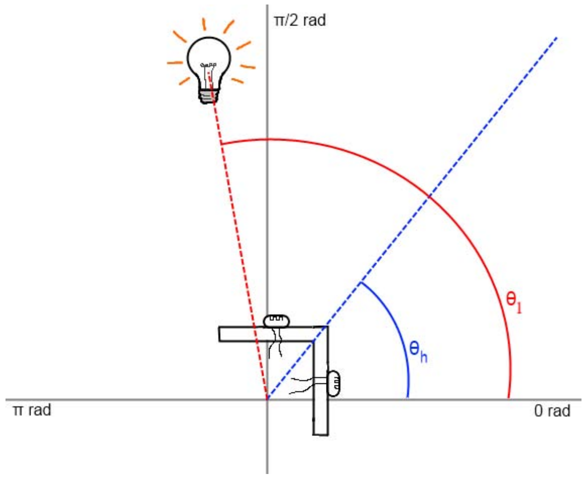
Introduction

In this problem, we use techniques and methods you practiced in Design Labs 4 and 5 to construct and analyze a model of a system which we will see again in Design Labs 7 through 9, which controls a motor to turn a robot "head" (pictured at right) to face a light. The system consists of three components:



- a light sensor, from which we can determine the position of the light relative to the head
- a motor, which we can use to turn the head toward the light
- a control circuit, which controls the motor (and which you will build in later labs)

We think of the system as follows: the light is somewhere in the world, with some absolute angular position θ_l. The head itself also has an absolute angular position, which we'll call θ_h. The goal is to make the head turn to face the light; that is, we want a system that makes θ_h become equal to θ_l. The relation between θ_h and θ_l is pictured below:



1 Preliminary Information

Note that in this course, we use capital letters to denote *signals*, and lowercase letters to denote *samples* of signals. For example, Θ_h is a signal, whose samples are $\theta_h[n]$.
Constants may be either capital letters or lowercase letters, depending on the convention for that particular constant.

The motor in this system produces a torque that accelerates the motor. The angular acceleration α_h (in $\frac{\text{rad}}{\text{sec}^2}$) of the motor is:

$$\alpha_h[n] = k_m i_m[n]$$

where i_m is the current flowing through the motor (in Amps), and k_m is a constant specific to the motor used.

All motors are generators. If you were to mechanically drive a DC motor (i.e., turn the shaft by hand), you could measure a voltage drop across the motor's contacts. In fact, even when we drive the motor using a battery, this voltage drop (known as back-electromotive force, or back-EMF) is still produced. This back-EMF v_b, measured in Volts, is proportional to the angular velocity of the motor:

$$v_b[n] = k_b \omega_h[n]$$

where ω_h is the angular velocity of the motor (in $\frac{\text{rad}}{\text{sec}}$), and k_b is a constant specific to the motor used.

Once we know the applied voltage, the back-EMF, and the resistance of the motor, we can find the current through the motor using **Ohm's law**. The current is equal to the total voltage drop across the motor, divided by the resistance of the motor:

$$i_m[n] = \frac{(v_c[n] - v_b[n])}{r_m}$$

where v_c is the voltage applied to drive the motor, and r_m is the resistance of the motor (r_m is constant for a given motor, and is measured in ohms).

We model the light direction sensor as producing a voltage v_s that is proportional to the angle between the light's angular position and that of the head. We call this difference the error e.

$$v_s[n] = k_s(\theta_l[n] - \theta_h[n]) = k_s e[n]$$

where k_s is also a constant. Finally, we use a simple proportional controller (with gain k_c) to generate the voltage v_c, which drives the motor:

$$v_c[n] = k_c v_s[n]$$

Notice that k_c is the only gain we can control; all of the others are predetermined by the parts we are using.

Check Yourself 1. What should the respective units of k_m, k_b, k_s, and k_c be?

2 Building the Model

This system is analogous to the **wall-follower** and **wall-finder** that you have seen in recent design labs. Because of this, we think of this system in terms of familiar pieces: a controller, a plant[1], and a sensor.

2.1 Modeling the Sensor and Controller

We use a simple proportional controller to generate the control voltage V_c, given the sensor signal V_s. V_s, in turn, is proportional to $E = (\Theta_l - \Theta_h)$.

We combine these two systems in our model to make one block that takes E as its input and generates V_c as its output.

Notice that, unlike the systems in design labs 4 and 5, we are modeling the light sensor as an unusual kind of "sensor" that measures the error directly, with **no time delay**.

Step 1. Find the system function for this control/sensor system, and draw the associated block diagram. Do these in whatever order you prefer, but make sure your system function matches your diagram.

Step 2. Using the primitives and combinators provided in the **`lib601.sf` module**, define a method `controllerAndSensorModel`, which takes as input a gain k_c, and outputs an instance of `sf.SystemFunction` which represents the controller/sensor system with this gain. You may use within your definition any of the variables defined in `hw2Work.py`.

Do **NOT** construct this instance diretcly using the numerator and denominator `Polynomials`.

Wk.5.5.1 Enter your system function and your code into this tutor problem, and upload a PDF of your block diagrams. Detailed instructions for entering your system function into the tutor are available on the tutor page for this problem. For information regarding creating PDF files on various platforms, consult Appendix 2 of this handout.

2.2 Modeling the Plant

Our plant takes as input V_c (the output of the controller), and generates output Θ_h.

As in design lab 5, we separate the plant into two smaller systems: the motor, and an integrator.

2.2.1 Integrator

We start by making an integrator which takes as its input the motor's angular velocity Ω_h, and outputs the motor's angular position Θ_h. Recall that all of the signals are sampled every T seconds.

Step 3. Find the system function for the integrator, and draw the associated block diagram.

Step 4. Using the primitives and combinators provided in the **`lib601.sf` module**, define a method `integrator`, which takes as input a timestep T (the length of time between samples) and returns an instance of `sf.SystemFunction` which represents an integrator with the appropriate timestep.

[1] The "plant" is the system we are trying to control. It is called the "plant" (as in factory, not as in leafy green thing) because a common early application of control theory was controlling manufacturing plants.

Note the difference between an integrator and an **accumulator**; an integrator, for example, transforms velocities to positions, in a manner such that the position at a given time step is determined by the previous position and the previous velocity, as was done in **Design Lab 4**

2.2.2 Modeling the Motor

Next, we model the motor, which takes V_c as input and generates Ω_h as output.

Step 5. Using the equations given on the previous page, find the system function for the motor, and draw the associated block diagram. Label A_h on your diagram.[2]

Step 6. Using the primitives and combinators provided in the `lib601.sf` module, as well as any relevant methods you have already created, define a method `motorModel`, which takes the same input as `integrator`, but returns an instance of `sf.SystemFunction` which represents the motor. You can assume that k_b, k_m, and k_s are already defined for you.

2.2.3 The Combined Plant

Step 7. Find the system function for the entire plant, and draw the associated block diagram. Label the parts of the diagram that correspond to the motor and the integrator.

Step 8. Using the pieces you've constructed above, as well as any relevant primitives and combinators provided in the `lib601.sf` module, define a method `plantModel`, which takes the same input as `integrator` and `motorModel`, and returns an instance of `sf.SystemFunction` which represents the entire plant.

Wk.5.5.2 Enter your system functions for both the motor model and the combined plant model, as well as your code for this section, into this tutor problem, and upload a PDF of your block diagram for the combined plant. Be sure to **label which part of your block diagram represents the motor, and which represents the integrator**.

2.3 Putting It All Together

Now we connect all of these systems together into one large system whose input Θ_l is the angular position of the light, and whose output Θ_h is the angular position of the head.

Step 9. Find the system function for this composite system, and draw the associated block diagram. Label the following signals on your diagram: Θ_l, Θ_h, E, V_s, I_m, Ω_h, A_h, V_c, V_b.

Again, do these in whatever order feels more natural to you, but make sure that your system function matches your diagram.

Step 10. What are the poles of this system? Express your answer as an algebraic expression involving k_c and any necessary constants.

Step 11. Using the pieces you've constructed above, as well as any relevant primitives and combinators provided in the `lib601.sf` module, define a method `lightTrackerModel`, which takes as input

[2] A is a capital "α," not a capital "a."

the gain to use for the controller (`k_c`) and the length of a timestep (`T`), and returns an instance of `sf.SystemFunction` which represents the entire light-tracking system with the specified gains and timesteps.

Wk.5.5.3 Enter your system function for the entire light tracker model, as well as your code for this section, into this tutor problem, and upload a PDF of your block diagram for the entire system. Be sure to label the signals Θ_l, Θ_h, E, V_s, I_m, Ω_h, A_h, V_c, and V_b in your diagram.

3 Analyzing the System

We are going to ask you to make some claims and argue that they are true, either theoretically or empirically:

- Theoretical – you can analyze particular systems by determining their poles, and finding optimal gains by analytically calculating optima of functions.
- Empirical – you can generate plots of the behavior of particular systems given initial conditions, and finding good gains by systematically sampling a set of candidate gains, evaluating them, and returning the best one.

Demonstrate the correctness of each of your answers below, either empirically or theoretically.

For the analysis, assume the following values for the given constants (these are defined for you in `hw2Work.py`):

- $k_m = 1000 \frac{\frac{\text{rad}}{\text{sec}^2}}{\text{Amp}}$
- $k_b = 0.5 \frac{\text{volts}}{\frac{\text{rad}}{\text{sec}}}$
- $k_s = 5 \frac{\text{volts}}{\text{rad}}$
- $r_m = 20$ ohms

Step 12. Given these values for the constants in the equation, find the value of k_c that makes the head react most quickly to changes in the light's position, while causing the head to converge to the desired angular position without oscillation. What are the poles for the best gain? For this step, use $T = 0.005$ seconds. (Suggestion: see Appendix 1 on Optimal Solutions).

Explain how you arrived at this answer, and plot the response of the light tracker with your chosen gain to a unit step input. This will model the behavior of the system when the head starts facing the light, but at time $t = 0$ the light is instantly moved to $\theta_l = 1$ radian, where it remains indefinitely.

Here is a handy procedure that takes a system function as input and generates a plot of the behavior resulting from the system starting at rest[3] with a unit step signal as input. This function is defined in `hw2Work.py` for your convenience.

```
def plotOutput(sfModel):
    smModel = sfModel.differenceEquation().stateMachine()
    outSig = ts.TransducedSignal(sig.StepSignal(), smModel)
    outSig.plot()
```

[3] When a system "starts from rest," this means that all inputs to and outputs from the system for time $t < 0$ have value 0.

Step 13. When $T = 0.005$ seconds, for what range of k_c is the system monotonically convergent? oscillatory and convergent? divergent? Plot the response of the light tracker to a unit step input for one gain in each of the ranges you found above.

Step 14. Fix k_c at the best value you found in **Step 12**. Now, try increasing T. Also try decreasing it. Experiment with different values of k_c and T; what can you say about how the length of the timestep T affects the behavior of the system?

Wk.5.5.4 Answer the questions in this problem and upload a PDF with your clearly labeled plots.

4 Variable Reference

For your reference, here is a summary of the symbols used in this assignment:

- Θ_l : the absolute angular position of the light, a signal whose samples are $\theta_l[n]$.
- Θ_h : the absolute angular position of the head, a signal whose samples are $\theta_h[n]$.
- E : the error signal $(\Theta_l - \Theta_h)$.
- V_s : the voltage produced by the light direction sensor, a signal whose samples are $v_s[n]$.
- I_m : the current flowing through the motor, a signal whose samples are $i_m[n]$.
- Ω_h : the motor's angular *velocity*, a signal whose samples are $\omega_h[n]$.
- A_h: the angular acceleration of the motor, a signal whose samples are $\alpha_h[n]$.
- V_c : motor control voltage, a signal whose samples are $v_c[n]$.
- V_b : the back-EMF of the motor, a signal whose samples are $v_b[n]$.

Appendix 1: Optimal Solutions

Recall that the speed of convergence of a system's response is improved by **reducing the magnitude of the dominant pole**. Thus, to find the optimal performance possible, you could construct a function $f(k)$ that computes the magnitude of the system's dominant pole, then find the value of k that produces the minimum value of this function.

Finding the minima of a function is a generic problem. 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.

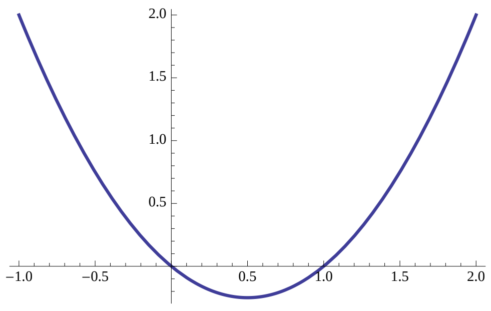
The procedure **`optimize.optOverLine`** does this. It is part of the **lib601** optimize **module**, and is called as follows:

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

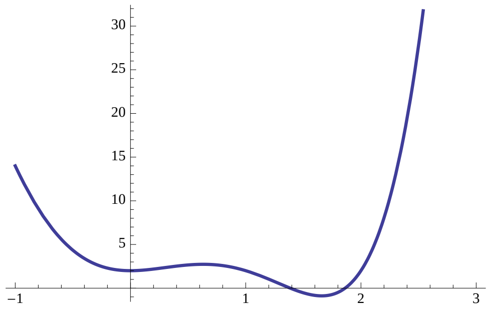
- `objective`: a procedure that takes a single numeric argument (x in our example) and returns a numeric value ($f(x)$ in our example),
- `xmin, xmax`: a range of values for the argument,
- `numXsteps`: how many points to test within the range,
- `compare`: an optional comparison function that defaults 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.



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



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

Appendix 2: Creating PDF Files

Perhaps the easiest way to create PDFs is to use a scanner.

Note: regardless of what method you use to create your PDF files, **always** double-check to make sure that they are *valid* and *complete* before you submit them; do this by opening them in a PDF reader and making sure they open correctly and contain everything you think they should contain.

Print-to-PDF

On Athena (or vanilla Ubuntu): Open the file you want to convert in an appropriate program. Choose `File` → `Print`. Select the "Print to File" option. Make sure "PDF" is selected under "Output Format." Choose where to save the resulting PDF file, and click "Print."

On Windows: Install a PDF printer (such as **doPDF**). Open the file you want to convert in an appropriate program. Choose `File` → `Print`. Choose to print to your recently-installed PDF printer; you will then be presented with options about where to save the resulting PDF file.

On Mac OSX: Open the file you want to convert in an appropriate program. Choose `File` → `Print`. Click on the PDF button in the lower left-hand corner of the "print" dialog. Select the option "Save as PDF...", choose the location where you want to save the resulting PDF file, and click "Save."

LibreOffice / OpenOffice.org

LibreOffice and **OpenOffice.org** allow exporting files directly to PDF.

Create a file which includes all graphics and text you want to include in your PDF. Choose `File` → `Export as PDF...`. Make sure the box next to "PDF/A-1a" is checked. For submitting graphs or scans, you probably want to choose the "Lossless compression" option under "Images." Click "Export," choose where to save the resulting PDF, and click "Save."

Online Drawing Tools

You're welcome to try online drawing tools for your block diagrams, such as the **Google Docs** presentation drawing tools, or **diagram.ly**, or **gliffy.com**. We do not endorse any of these; your mileage may vary.