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

Design Lab and Homework 10, Issued Thursday, April 17

This handout contains design lab 10 as well as homework problems that are to be written up and handed in, together with questions from software lab 10, in your design lab on Thursday or Friday, April 24 or 25.

Please write up and hand in answers to the following questions: 7, 11, 12, 13, 14 and 15 from the design lab. Also, write up and hand in answers to homework questions 17 and 19.

Design Lab: Computer-based control

Do `athrun 6.01 update` and you will find the relevant files in the `lab10` directory.

This lab requires a laptop, a NIDAQ box, your head from last week, a power supply, and the usual circuit-building and testing stuff.

Warning: As you've seen in previous labs, some of the clip leads are bad. Make sure to test the ones you are using. You can just set your meter to the "sound" setting, and touch both ends of the clip-lead with the leads from the meter; the meter will beep if it's good.

Python to NIDAQ Input/Output

In this lab, and then in the final-project labs, we will be using the NIDAQ box for both input and output, between a Python program and a circuit. The NIDAQ box has the following ports that will be of interest to us:

- AIGND: analog input ground
- AI0—AI7: analog inputs 0 through 7
- AOGND: analog output ground
- AO0: analog output 0

The NIDAQ box can read voltage differences between AIGND and AI0, between AIGND and AI1, etc. It can actually set a voltage difference between AO0 and AOGND. It can actually read and set voltages between $-10V$ and $+10V$, but we will only use voltages in the range $0V$ to $+10V$.

You might wonder why, if the NIDAQ box can generate voltage differences, we need to use the power supply. The answer is that the NIDAQ box can only generate relatively small currents, but not necessarily large enough ones to drive our motor. Luckily, we know a way to convert a voltage sustained by a low-current source to that same voltage, but with more current: an op-amp connected to a voltage source that can supply more current. So, we'll use the $0V$ and $+12V$ from the power supply as the supply rails for op-amps in our circuit.

Staying Grounded

Things can get very confused here, very easily, because we have three wires labeled "ground": NIDAQ AIGND, NIDAQ AOGND, and 0V from the power supply. You cannot necessarily depend on them all being at the same voltage level (unless all of the devices are plugged into the the wall with a three-pronged plug). To keep all your "grounds" equal, you should start by connecting NIDAQ AIGND, NIDAQ AOGND, and power supply 0 together on the bottom board of your head. And you might as well connect the black lead of your meter to this value, while you're at it.

Read and writing

In lab 7, we read voltages from the NIDAQ box in a Python program. Now we'll see how we can use Python to command voltages to outputs on the NIDAQ box.

1. Connect a wire to the AO0 port of your NIDAQ box
2. Start the NIDAQ server by opening a new terminal window and typing

   ```
   > ./NIDAQserver --output
   ```

   Now, wait a somewhat uncomfortably long time, and numbers should start streaming out. Just leave that window alone.

   If you see a "fatal error"" message before the numbers start streaming, just ignore it.
3. Now, from IDLE, open the file `test.py`, and run it to load its definition. Type `writeTest()` at IDLE's shell prompt.

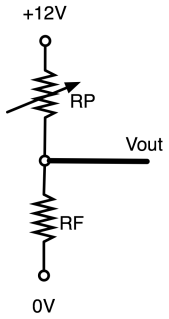
Here is the text of part of the `test.py` file. The `writeNIDAQ` procedure takes a voltage value and "writes" it out on the port AO0 of the NIDAQ box.

```
# Write values from 0 up to 9.5 volts to NIDAQ AO0
# Go in steps of 0.5, every 2 seconds
def writeTest():
    v = 0.0
    while True:
        print v
        writeNIDAQ(v)
        time.sleep(2.0)
        v = v + 0.5
        if v > 9.5: v = 0.0
```

You should see the voltage on the meter increment from 0V, in 0.5-Volt steps, every 2 seconds, until it reaches 9.5V, and then wraps around. You may notice that occasionally one of the commands is missed; this is a known problem, which won't affect subsequent parts of this lab, when we will be sending commands very frequently.

Python phototaxis

Now, let's see if we can replicate what we did in lab 9, having the head turn to follow the light, but with a Python program in control. It will be useful to put the photo-sensors in a resistor-divider configuration, like this:



Question 7: Design a circuit that will let you read the two values from the two photo-sensors on your head assembly, so they can be read in Python via the NIDAQ box. Remember that they need to be in the range 0V to +10V. You don't need more than the three wires that we have going up to the head; if you have a problem seeing how to do this, talk to an LA.

Checkpoint

Show your circuit design to your LA

It will simplify the process of doing this lab if you unwire the circuits that you have currently built on the bottom board of your head and start fresh. (You should leave the op-amp chips where they are; we'll be using those).

Question 8: Build your circuit on the bottom board of your head. Connect the AI0 and AI1 ports to the photo-sensor circuit outputs.

Question 9: The code below (included in `test.py`) repeatedly reads a list of 8 numbers from the NiDAQ box and prints the first two, which are the voltages between AI0 and AIGND and between AI1 and AIGND.

```
def readTest():
    while True:
        values = readNIDAQ()
        print values[0], values[1]
        time.sleep(1.0)
```

Run the program and try it. Observe how the output voltages change, and how they are affected when you shine a light on the head.

Checkpoint

Demonstrate reading the light sensors to your LA

Now that you can read the photo-sensors, it's time to turn to the task of driving the motor so that you can create feedback loop. Your program will read the sensors and use NIDAQ box to produce a voltage for the motor.

But there's a complication: The NIDAQ box can generate outputs in the range 0–+10V and the motor needs to turn in both directions. So, you'll need to generate a "virtual ground" again, this time, at +5V, and connect that to one side of the motor. Recalling tutor problem PS.10.1.5, you can use a 15KΩ resistor and a 10KΩ resistor to get the appropriate ratio in your divider.

Question 10: Build a circuit on the bottom board of your head that generates the required offset for the motor. Connect it to one terminal of the motor and connect the NIDAQ output voltage to the other terminal. Remember that you'll need to buffer the output of the NIDAQ box, because the box can't provide much current.

Question 11: Now, write a Python control loop that will control the head to turn toward the light. It should be a `while True` loop that reads the photo-sensor values, computes some error value, multiplies it by a gain and offsets it by an appropriate amount to produce a voltage for the motor and writes that voltage to AO0. The simplest error value to try is the difference between the voltages from the photo-sensors.

Make sure your commanded output to the motor is in the right range. Think carefully about the range of voltage values at each of the input photo-sensors. What is the range of values of the error value you are using? Make sure that you don't exceed the 10V limit on the NIDAQ box. Also, make sure that AO0 has the desired value when the photo-sensors have the same value.

Question 12: Find (by trial and error) the highest gain you can use that isn't unstable. How does this value compare to the gain values you found in lab 9? Introduce additional delays in your control loop, using `time.sleep(x)` for different values of `x` in seconds (`x` can be less than one). Explain what you see.

Checkpoint

Show your LA your light-seeking behavior. Demonstrate it in both a stable and an unstable regime. Explain what the difference is that makes it unstable.

Heading in the right direction

Question 13: Once the head is pointing towards the light, the robot might want to know the angle that the head makes to the front of the robot. Determine where you can put an input to the NIDAQ box so as to read the "neck" potentiometer. Determine which voltages from the "neck" potentiometer correspond to which angles of the robot's head. Write a Python procedure that can read such a voltage and print the associated angle. Hint: fit a line.

The angle printed should vary from almost $-\pi/2$ to almost $\pi/2$, where zero is printed when the head is parallel to the straight edge of the grey base plate, $\pi/4$ should be printed when the head is rotated 45 degrees counter-clockwise from parallel to base plate straight edge, and $-\pi/4$ should be printed when the head is rotated 45 degrees clockwise from parallel to base plate straight edge. Note that the motor stops the head from turning far enough to generate angles outside of this range.

Question 14: Write a Python program that can take a desired angle as an argument and stably turn the head to face that angle.

Question 15: Describe the relative advantages and disadvantages of the pure circuit approach to tracking a light versus the one using a computer in the loop. Discuss speed, stability, ease of construction and modification, and anything else that seems relevant to you.

Checkpoint

- Show that you can print out a reliable angle reading from the neck potentiometer.
- Show that your head can turn to a commanded angle.

Homework assignment: Preparation for planning

Do `athrun 6.01 update` *to get a directory* `lab11` *that contains the relevant files.*

Worlds as state machines

In software lab 10, we searched through a road network for a path between two cities, by directly writing a successor function and goal test, and calling a search algorithm. We will be interested in planning out courses of action in complex worlds that might, themselves, be combinations of simpler worlds. As these worlds become more complex writing the successor function directly will become more difficult.

Instead, we will explore a different approach, in which we model worlds as state machines. The "actions" that we can take to change the state of the world will be the inputs to the machine; the states of the machine will be the states of the world; for now, the output of the machine will just be its current state, but we will explore in future labs what happens when the underlying state of a world is not visible to the external world.

Modeling worlds as state machines will allow us to build more complex world descriptions out of simpler ones via serial and parallel state-machine composition methods. Once we have modeled a planning world as a state machine, we need only to apply a generic method for deriving a successor function from the definition of the state machine, and then use it in a call to a search algorithm. Given a state machine, a start state, and a goal test, we can thereby find a sequence of inputs to the machine such that, if we were to repeatedly call step with those inputs, the machine would end up in a state that satisfies the goal test.

The file `TSMFull.py` contains code for a terminating state machine class. We have modified it, based on your feedback from using it in week 4, and have extended it somewhat to support extra capabilities for this assignment. We'll document the relevant classes and methods below.

All terminating state machines must support the following methods:

- `legalInputs` : *state* $\rightarrow$ *list*(*input*) For each state, this method specifies which inputs can possibly be legally given to the machine. (In many cases, this set of inputs will be the same for every state.)
- `start` : *input* or `None`$\rightarrow$ None This method gets the state machine ready to run. It is distinct from the `__init__` method of a class, because we might need to depend on some aspect of the input at the moment the machine is started. (For example, the `moveForward` terminating state machine had to read the pose when it was started so it knew how much farther it needed to go.) *This method was formerly called* `reset`*.*
- `step`: *input* $\rightarrow$ *output* This method takes an input, causes the state machine to make a transition, and returns the output associated with the resulting state.
- `currentOutput`: `None` $\rightarrow$ `output` This method returns the output associated with the current state.
- `done`: `None` $\rightarrow$ *Boolean* This method returns `True` if the state machine has terminated and `False` otherwise.

A primitive state machine, `PrimitiveSM`, is created by specifying:

- `transitionfn` : (*state*, *input*) $\rightarrow$ *state*
- `outputfn` : *state* $\rightarrow$ *output*

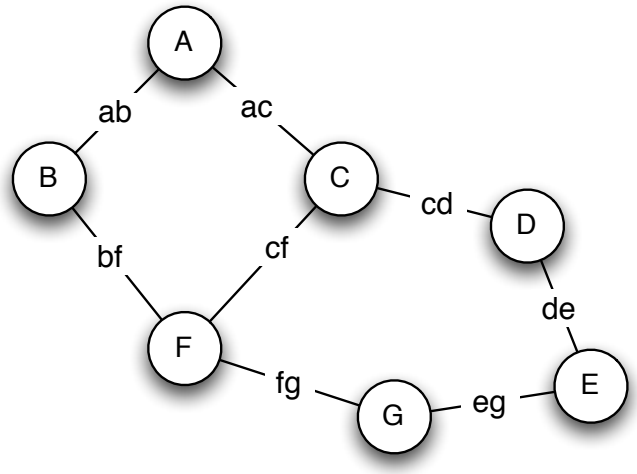


Figure 1: A simple map.

- `initStatefn` : `None` → *state*
- `terminationfn` : `state` → *Boolean*
- `legalInputs` : `state` → *list*(*input*)

A `PrimitiveSM` provides all of the methods above that every `SM` must provide, as well as the variable `self.currentState`, which is the current state of the primitive machine.

Question 16: The file `cityTest.py` contains definitions of a transition function and a legal input function for the world shown in figure 1. Read them and be sure you understand what they are doing.

Question 17: Create an instance of a `PrimitiveSM` that encodes the problem of moving from state `'A'` to state `'G'` in that world. Demonstrate that this model is good by first starting it by calling the `start` method, and then executing a series of calls to the `step` method with a sequence of inputs that will cause it to end in state `'G'`.

Planning

A `Planner` object can make plans to move through the states of a state machine. It is initialized with a state machine, `sm`:

```
def __init__(self, sm):
    self.sm = sm

def plan(self, initialState = None, goalTest = None, depthFirst = False):
```

The primary method, `plan`, will make a plan. You can specify an initial state and a goal test function, as for the search methods we've seen. The `depthFirst` argument does not have to be specified: we have said that if no value is passed in for it, then it should be given the default value of `False`, which means we'll do breadth-first search.

Question 18: Create an instance of `Planner` using the state machine you just made to represent the city graph in figure 1. Use this instance to plan a path from state `'A'` to state `'G'`. If you do this inside `cityTest.py` file, it will use the `Planner.pyc` file that we have given you.

We have given you a file called `PlannerShell.py` for you to edit. When you fill it in, be sure to change `cityTest.py` to import `PlannerShell` rather than `Planner`. The only work that has to be done is to construct a successor function for the planner using the transition function of the state machine.

```
def plan(self, initialState = None, goalTest = None, depthFirst = False):
    def successors(s):
       # your code here

    if initialState == None:
        initialState = self.sm.currentState()
    if goalTest == None:
        goalTest = self.sm.terminationFunction

    result = searchDP(initialState, goalTest, successors, depthFirst)
    if result:
        print "Path:", result
    else:
        print "Couldn't get from ", initialState, " to goal"
    return result
```

Question 19: Recalling that `successors` takes a state as an argument and returns a list of `(action, state)` pairs, that `self.sm.legalInputs` is a function from a state to the list of possible actions in that state, and that `self.sm.transitionFunction` takes a state and an action as arguments and returns the resulting next state, fill in the definition of the `successors` function. It only needs a single line of code. Test this on your city state machine. You should get the same results as with our version of `planner.pyc`.