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

Final exam announcement and practice final, Revised May 12

This is a sample final to help you study and to give you a sense of what kinds of problems to expect on the final. The final will be given the afternoon of Wednesday, May 21. You can pick it up in 34-501 (the 6.01 lab) at 1:30PM, and you must turn it in by 7:30PM. You may turn in your solution early. You may *not* turn it in late.

Ground rules for taking the final

These instructions apply to taking the real fine—not this practice exam:

- Read through the entire final and ask any questions you have before you leave. If you find that you have a question after that, try to pick your own answer, and write down what assumption you're making.
- Your paper *must* have all the sheets stapled together, and *must* have your name on each sheet. In the midterm, some papers came in as separated sheets and without names. We took the trouble then to identify the anonymous papers and reassemble the separated pieces. We will not do that in grading the final: we'll simply not grade any unidentified work.
- For the problems that ask you to write Python code, you should just think it through and write down the code. Don't try to actually debug it (it will take too long). Don't spend time trying to be sure the syntax is correct. It's more important to communicate your ability to think algorithmically than to worry about the fine-grained details of syntax. Be sure to add comments that explain what you're trying to do.
- Your answers may be handwritten or typed; they should be legible in either case. We reserve the right to refuse to grade illegible papers.
- You answers must include *coherent explanations*. Simply writing down equations with no explanations, or attaching pages of uncommented program output, will get no credit. In grading, we will make no attempt to guess what you are thinking by trying to piece together fragments of phrases and equations and printout.
- You may read anything you like (labs, books, the Web) during the exam; you may not communicate with anyone else.
- ***Take above prohibition on collaboration seriously.*** On the midterm, we found some papers (a very small number) where people had collaborated on problems. We handled this by giving the students involved grades of zero for the midterm. The consequences of collaborating on the final would be at least as severe and, at a minimum, result in failing the course. Avoid temptation: find some place to work on the final without other people in around.
- You can use any of the software we've provided or you've written for lab. But it's your responsibility to make sure that your computer runs the code. Tales of woe of the form "I couldn't get the code to run," may receive sympathetic sighs, but that won't make any difference in grading.

1. Robot in a corridor

For this problem, please consider a long corridor of 100 alternating black and white squares, labeled 0 through 99. Your robot is initially known to be in square 0, which is black and at the far left. The robot can move only right. When you ask the robot to move right one square, it will actually move right one square only 50% of the time; 40% of the time it will not move at all; and 10% of the time it will move right two squares.

Suppose first that the robot can perfectly reliably observe whether it is on a black square or a white square.

(a) Suppose you ask the robot to move one square to the right twice, but do not make any observations until after the second move is completed, at which point the robot observes that it is on a black square. What is the probability that the robot is on square 0? What is the entire belief state distribution for the robot's position?

(b) Suppose you want the robot to end up on square 10. A simple strategy would be to command the robot to move right ten times and hope for the best. Devise a more reliable strategy that has the robot perform an observation after every move. Describe your strategy in English.

(c) Write a Python program that implements your strategy from part (b). Assume you have procedures `moveRight()`, which commands the robot to move right, and `look()`, which returns either "black" or "white". You do *not* need to debug your code.

(d) Suppose now that the robot's vision is faulty. The observation model is this: If the robot is actually on a black square, it will see black with 80% probability and white with 20% probability. If the robot is actually on a white square, it will see black with 35% probability and white with 65% probability.

Suppose as in part (a) above, the robot starts on square 0 and you command it to move one square to the right twice, not making any observations until after the second move is completed. If the robot now sees that it is on a black square, what is the the probability that the robot is on square 0? What is the entire belief state distribution for the robot's position?

2. Motor controller

This problem is about using a computer to precisely control the speed of a spinning motor. The motor has a tachometer that provides very precise speed measurements, and you can control the voltage across the motor. The motor speed S is given by

$$S = JV$$

where J is a constant associated with the motor and V is the voltage across the motor. Unfortunately, not all all motors have the same J: the nominal value of J is 10, but J can actually be as low as 9 and as high as 11.

(a) Your computer samples the motor velocity 10 times a second and can change the motor voltage based on readings from previous samples. One suggested control strategy is to determine the motor voltage using the nominal value of J ($J = 10$):

$$V[n+1] = \frac{1}{10} S_{\text{desired}}$$

where S_{desired} is a given desired motor speed. In the worst case, considering the variation in J, how far off will the motor speed be from the desired speed?

(b) Suppose we use a feedback control system to reduce the difference between the actual motor speed and the desired speed as follows:

$$V[n] = \frac{1}{10} S_{\text{desired}} + K\left(S_{\text{desired}} - S[n-1]\right)$$

where S is the actual motor speed as measured by the tachometer and K is the feedback gain. Draw a delay-adder-gain block diagram that describes this control scheme as a system with input S_{desired} and output S.

(c) What is the system function

$$H = \frac{S}{S_{\text{desired}}}$$

for this system? What is the largest value of K that can be used, and why?

(d) Implement the controller in part (b) as a Python program. Assume there are procedures `setVoltage(v)` that sets the motor voltage, and `readSpeed()` that returns the value from the tachometer.

(e) Rather than using a computer digital controller, design a a circuit that implements the control scheme in part (b) using resistors and op-amps and a power supply. Label your circuit to clearly explain the purpose of each part.

(f) Write a couple of sentences about the relative merits of the digital vs. the analog controller. Under what circumstances would you choose to use one vs. the other?

(g) Suppose that we use a different control law: one of the form

$$V[n] = \frac{1}{10} S_{\text{desired}} + K_1\left(S_{\text{desired}} - S[n-1]\right) + K_2\left(S_{\text{desired}} - S[n-2]\right)$$

Where K_1 and K_2 are constants. What is the system function for this system?

(h) For the system in part (g), find values of K_1 and K_2 that make the motor speed oscillate and eventually settle to a stable value.

3. Search

This question involves modifying or writing Python programs to search graphs.

(a) The following procedure returns a successor function for a simple graph:

```
>>> def makeSuccessors():
        def succ (s)
            graph = {'A': [(0, 'B'), (0, 'C')],
                     'B': [(0, 'D')],
                     'C': [(0, 'D')]}
            return graph[s]
        return succ
```

Write a simple variation of this procedure that does not have to recreate the graph dictionary every time that the successor function is called.

(b) Write a Python procedure `searchAllGoals` that is given:

- an initial state,
- a list of state names, and
- a successor function for the graph

The `searchAllGoals` procedure should return a procedure of one argument. This returned procedure, when called with a state name, should return the shortest path from the initial state to that state.

`searchAllGoals` should do all the searching once, when it is called, the returned procedure should be able to simply "look up" the shortest path.

For example:

```
>>> s = searchAllGoals('A', ['B','C','D'], makeSuccessors())
>>> s('D')
[(None, 'A'), (0, 'B'), (0, 'D')]
```

You may assume that all the search procedures in the file `search.py`, which you used in week 10, are available to you. Explain your approach to solving the problem in words and well as in code. Write your code clearly and use comments.

4. Circuits and programming (Revised)

Earlier this semester, you worked with two versions of a circuit solver. In the second one, you specified a circuit by defining a constraint set, e.g.,

```
ckt = ConstraintSet()
```

Then you added constraints for the various circuit elements, e.g.,

```
ckt.addConstraint(resistor(20, ['n1', 'n2', 'ir1']))
```

Then you added `kcl` constraints for each node. Finally, you called the constraint solver, and displayed the result, e.g.,

```
solution = ckt.solve()
ckt.display(solution)
```

Recall that the names used with the resistor are variables whose values are voltages and currents: `n1` designates the voltage between node n_1 and ground, `n2` designates the voltage between node n_2 and ground, and `ir1` the current through the resistor.

The `resistor` procedure generates the appropriate $v = iR$ constraint:

```
def resistor(R, x): # R is resistance
    [vn1, vn2, i12] = x
    return [(1.0, vn1), (-1.0, vn2), (-float(R), i12), (0, None)]
```

The constraint itself is a list of pairs (c_k, v_k) where each pair is a coefficient and a variable name, representing the linear equation, in this case: $v_{n1} - v_{n2} - Ri_{1,2} = 0$.

In lab, you used the solver with resistors and voltage sources, and you extended the system to include op-amps. We now want to extend the system to include potentiometers. A potentiometer, as you know, is 3-terminal device. It has two parameters, a total resistance `totalR` and a shaft `setting`, which is a number between 0 and 1, to indicate the fraction that the potentiometer has been turned between its minimum and maximum settings.

Modeling the potentiometer will require adding two linear equations to the circuit constraints. However, we have assumed that component procedures (such as `resistor` above) return only one equation. So, we will need to extend the system from Software Lab 9 to allow components to return a list of equations and we will need to extend the existing component procedures to conform to this convention.

(a) Show any changes to the `ConstraintSet` class you worked on for software lab 9 (in `solveLinearConstraintsShell.py`) that need to be made so that it handles components returning lists. This is a small change to what you did in Software Lab 9, show **only** the required changes. Explain the change.

(b) Re-write the `resistor` procedure so that it will work in this new framework. Indicate the changes.

(c) Give the Python implementation for the potentiometer constraint procedure. The procedure takes `totalR` and `setting` as inputs and a list of nodes and currents. It returns a list of equations. Explain the code.