MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.01—Introduction to EECS I
Fall Semester, 2008

Midterm solutions

Signals and systems (16 points)

Heating system

Consider a system for keeping a room at a desired temperature (in degrees), D_i. The system has three subsystems:

- a controller that specifies the rate at which the temperature in the room changes, C, which is in degrees/second,
- a plant, which is a model of how to temperature of the room (in degrees), D_o, changes,
- a sensor that will give the sensed room temperature (in degrees), D_s.

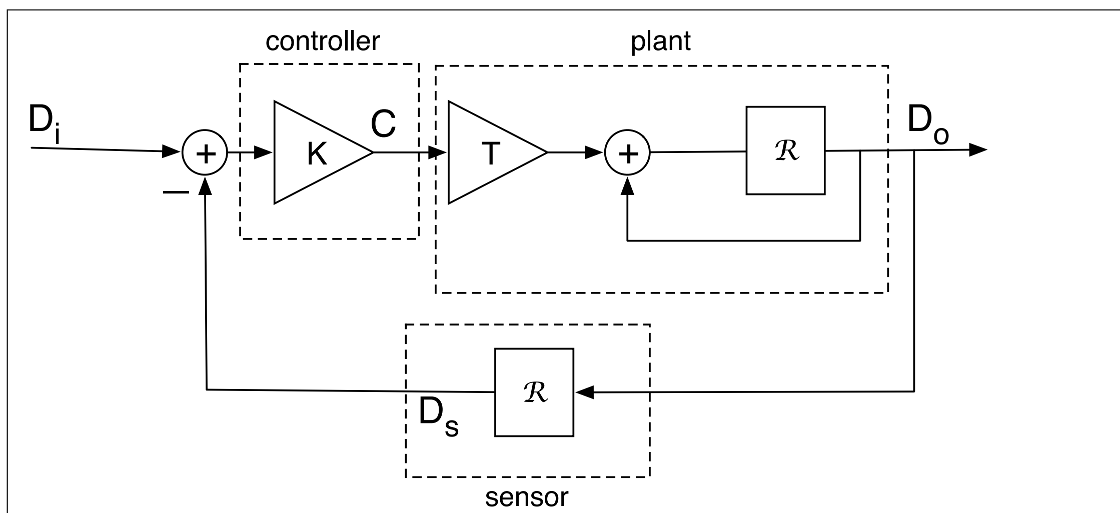
Here are some more details:

- The controller is a proportional controller based on the difference between the desired and sensed temperatures. There is no delay in the controller.
- The sensed room temperature is a one-time-step delayed version of the room temperature.
- There is a one-time-step delay in the plant.
- The time between timesteps is T.

Question 1: (**6 points**) Write a difference equation for each of the subsystems:

1. **The controller**
   $C[n] = k(D_i[n] - D_s[n])$
2. **The plant**:
   $D_o[n] = D_o[n-1] + TC[n-1]$
3. **The sensor**
   $D_s[n] = D_o[n-1]$

Question 2: (**4 points**) Draw a block diagram of the system above. Your diagram should contain all of the necessary delays and gains. Put a box around, and label, the parts of the system that correspond to the three subsystems.



System function poles

The system function for the temperature control system described above is

$$\frac{D_o}{D_i} = \frac{Tk\mathcal{R}}{Tk\mathcal{R}^2 - \mathcal{R} + 1}$$

.

Question 3: (**2 points**) Find an expression for the poles of this system in terms of T and k. Be sure to show your work.

$$\frac{1}{2} \pm \frac{\sqrt{1 - 4Tk}}{2}$$

Consider a system (not the one from above) whose poles are

$$\frac{1}{2} \pm \sqrt{\frac{1}{4} - 2k}$$

.

Question 4: (**4 points**) For each of the following values of k, say whether the system would be converging or diverging and oscillating or monotonic.

k	converging/diverging	oscillating/monotonic
$k = 1$	diverging	oscillating
$k = \frac{1}{4}$	converging	oscillating
$k = \frac{1}{8}$	converging	monotonic
$k = -\frac{1}{2}$	diverging	monotonic

Python and objects (16 points)

In the next few problems, we are going to build a data structure for storing points, along with some information about each point. Once we have stored several (point, information) pairs, if we encounter a new point, for which the information is missing, we can make an educated guess about what the missing information might be. We do this by finding the stored pairs with points that are closest to our new point, and then combining the information associated with these points in some way to generate an estimate.

For example, say we are storing x, y `Point` objects that correspond to locations in a room, and the information associated with each point is the light level at that location, measured with a light sensor on a robot. Now imagine that, after storing a number of points and light readings, the robot's light sensor has broken, and the robot needs to know the light level at some new location in the room. Since we have stored previous readings, along with their locations, we might make an educated guess about the likely light level at our new location, based on the reading that we collected at the stored location closest to our current location. This approach is called *nearest neighbor*, for obvious reasons.

First we need something to store in our structure. Below is a `LightReading` class that we will use to store a single light reading, as in out example above. The location will be a `Point` instance, like the ones we have used in class.

```
class LightReading:
    def __init__(self, location, reading):
       self.location = location
       self.reading = reading
    def distance(self, newLocation):
       return self.location.distance(newLocation)
```

Below is a skeleton of a `Neighborhood` class that we will use to store a collection of `LightReading` instances. To be useful, it will need a `findNearest` function, which takes a `Point`, `newLocation`, and a number, `r`, and finds all of the stored `LightReading` instances that have distance less than `r` from `newLocation`. Then `findNearest` should return a list of tuples of the form `(distance, neighbor)`, one tuple for each neighbor that is close enough, where `distance` is the distance between the neighbor's `location` and `newLocation`.

Here is a transcript of the `findNearest` function being called on an instance of the `Neighborhood` class that has had some points stored in it:

```
>>> nhood.findNearest(Point(0.1, 2.0), 1.0)
[(0.40000000000000002, <nn3.LightReading instance at 0x76620>)]
>>> nhood.findNearest(Point(0.1, 2.0), 2.0)
[(1.004987562112089, <nn3.LightReading instance at 0x76e90>),
(0.40000000000000002, <nn3.LightReading instance at 0x76620>)]
```

Question 5: (**4 points**) Implement the `findNearest` function.

```
class Neighborhood:
    def __init__(self):
       self.neighbors = []

    def addNeighbor(self, neighbor):
       self.neighbors.append(neighbor)

    def findNearest(self, newLocation, r):
       # write your code here
```

```
      tuples = [(n.distance(newLocation),n) for n in self.neighbors]
      return [t for t in tuples if t[0]<r]
```

Question 6: (**6 points**) When the robot is at a new location with a broken sensor, we will estimate the light level at that location to be the **mean** of the light levels of the stored neighbors within a radius `r` from our current location. Write a new method for the `Neighborhood` class, `estimateValue`, which takes a location (which is a `Point`) and a radius `r`, and returns an estimated light level for that new location. This method should call `findNearest`. You may assume that `r` is large enough that there is always at least one near neighbor.

```
    def estimateValue(self, location, r):
       ns = self.findNearest(location, r)
       readings = [n[1].reading for n in ns]
       return sum(readings)/float(len(readings))
```

Question 7: (**2 points**) Assume `nhood` is an instance of the `Neighborhood` class. Imagine that the following operations (and only these operations) have been performed on it.

```
>>> nhood.addNeighbor(LightReading(Point(1.0, 2.0), 0.2))
>>> nhood.addNeighbor(LightReading(Point(1.5, 2.5), 0.1))
>>> nhood.addNeighbor(LightReading(Point(1.0, 1.5), 0.3))
```

Now the robot is at location $\langle 1.4, 2.4 \rangle$ with a broken light sensor. Using a radius of 0.2, show how to call the `estimateValue` method, and say what it will return.

```
nhood.estimateValue(Point(1.4, 2.4), 0.2)
```

It will return 0.1, because only the second point is within 0.2 of the robot's current location.

Question 8: (**4 points**) *Note: This problem is somewhat trickier than the others. You might consider doing it last.* Instead of just taking the mean of our nearby data, we might like to do a weighted average, where the weights depend on the distance in some way. This would allow us to care more about the closer points than the farther ones. Consider a new point p which has N neighbors where the ith neighbor has distance d_i and value v_i (in our example, light reading v_i). Given a weighting function w which maps distances into weights, we can compute an estimated value $\hat{v}$ for our new point:

$$\hat{v} = \frac{\sum_{i=0}^{N-1} w(d_i) v_i}{\sum_{i=0}^{N-1} w(d_i)}$$

Create a subclass of the `Neighborhood` class, called `WeightedNeighborhood` to do this. Its constructor should take a weighting function, and then `estimateValue` should use the weighting function to generate the estimate as described above. Write as little new code as possible.

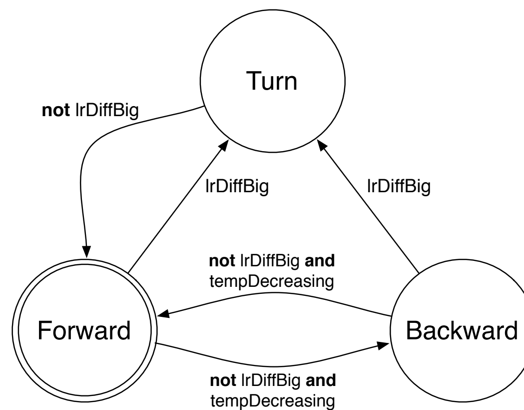
```
class WeightedNeighborhood(Neighborhood):
    def __init__(self, weightingFunc):
       Neighborhood.__init__(self)
       self.w = weightingFunc
    def estimateValue(self, location, r):
       ns = self.findNearest(location, r)
       return (sum([n[1].reading*self.w(n[0]) for n in ns])/
               sum([self.w(n[0]) for n in ns]))
```

State machines (16 points)

Imagine that our robots are outfitted with two temperature sensors, one on the left of the robot, and one on the right. We will add a `temperature` variable to the normal sensor input. So `inp.temperature` is a tuple of the left and right sensor values (in that order).

Let `lrDiffBig` be a Boolean that says whether the absolute value of the difference between the left and right temperatures is larger than some `lrDiffTolerance` variable. Let `tempDecreasing` be a Boolean that says whether the average temperature at this time step is less than the average temperature before (meaning the robot is moving away from the heat source). With the relevant information in the state, these variables can be computed based on the state and the input.

Below is a state transition diagram and its output function. If there is no transition out of a state for a particular input, then the machine stays in the same state. The starting state for the machine is the "Forward" state. Not all of the required state is captured in the diagram.



The output of the machine is as follows:

State	Input	Output
Forward	any	go forward
Backward	any	go backward
Turn	left temp. > right temp.	turn in place, left
Turn	left temp. < right temp.	turn in place, right

Note: There is a "tear-away" sheet containing the transition diagram and output function at the end of the exam, that you can tear out and refer to while working on the problem if you like.

Question 9: (**2 points**) Describe in English how this state machine will cause the robot to behave.

It will cause the robot to move toward the heat.

Question 10: (**2 points**) What should the state consist of?

It should have the state shown in the transition diagram, as well as the previous temperature.

Question 11: (**4 points**) Assume that you have a functional `TemperatureSM` class that implements the state machine described above (a skeleton for such a class is on the next page). Without changing `TemperatureSM` in any way, show how you would specify that the robot should stop when it detects an obstacle directly in front of it, at a distance of 0.2 meters or less (using the sonars). The robot should resume its motion when the obstacle moves out of its way. You can use the `Stop` state machine that we gave you in lab 4 if you'd like.

```
def obstacleInFront(inp): return (min(inp.sonars[3:4]) < 0.2)

behavior = StepwiseIfTSM(obstacleInFront, Stop(), TemperatureSM())
```

Question 12: (**8 points**) Fill in the getNextValues method in the following state machine skeleton to implement the state machine described above.

```
class HeatseekerSM(sm.SM):
    startState = ('forward', 0.0)
    def __init__(self):
       self.lrDiffTolerance = 0.001

    def getNextValues(self, state, inp):
       (leftT, rightT) = inp.temperature
       aveT = (leftT+rightT)/2.0
       lrDiffBig = abs(leftT-rightT) > self.lrDiffTolerance
       tempDecreasing = aveT < state[1]
       stateType = state[0]
       if state[0] == 'forward':
           if lrDiffBig:
              stateType = 'turn'
           elif tempDecreasing:
              stateType = 'backward'
       elif state[0] == 'backward':
           if lrDiffBig:
              stateType = 'turn'
           elif tempDecreasing:
              stateType = 'forward'
       elif state[0] == 'turn' and not lrDiffBig:
           stateType = 'forward'

       if stateType == 'forward':
           a = io.Action(fvel=0.1, rvel=0.0)
       elif stateType == 'backward':
           a = io.Action(fvel=-0.1,rvel=0.0)
       else:
           (leftT, rightT) = inp.temperature
            print inp.temperature
           if leftT > rightT:
              a = io.Action(fvel=0, rvel=0.1)
           else:
              a = io.Action(fvel=0, rvel=-0.1)
       return ((stateType, aveT), a)
```