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

# Practice Final Solutions (Revised May 16)

## 1. Robot in a corridor

(a) After two move-right commands and observing black, the belief state is:

$$0.32, 0.0, 0.66, 0.0, 0.02$$

After a single move and no observation, the belief state is:

$$0.4, 0.5, 0.1$$

After another move, it is:

$$0.4 * 0.4, 0.4 * 0.5 + 0.5 * 0.4, 0.4 * 0.1 + 0.5 * 0.5 + 0.1 * 0.4, 0.5 * 0.1 + 0.1 * 0.5, 0.1 * 0.1$$

that is:

$$0.16, 0.4, 0.33, 0.1, 0.01$$

Given the observation is "black" and we have a perfect sensor, the probability of the white squares goes to zero:

$$0.16, 0.0, 0.33, 0.0, 0.01$$

We divide by the sum of these probabilities (which is $P(O = \mathsf{black})$) and get:

$$0.32, 0.0, 0.66, 0.0, 0.02$$

(b) We can look to see if the command to move resulted in a change of color, if it didn't, try again. This can fail if we end up moving two steps since it will appear that we didn't move. But, there's only a 0.1 chance of that happening.

(c)
```
def move(n):
   oppositeColor = {'white':'black', 'black':'white'}
   nextColor = 'white'          # we start in black square
   for i in range(n):
      while look() != nextColor:
         moveright()
      nextColor = oppositeColor[nextColor]

move(10)
```

(d) After two move commands we have (as before):

$$0.16, 0.4, 0.33, 0.1, 0.01$$

Given the new sensor model ($P(O = \mathsf{black}|\mathsf{black}) = 0.8$ and $P(O = \mathsf{black}|\mathsf{white}) = 0.35$) and an observation of "black":

$$0.16 * 0.8, 0.4 * 0.35, 0.33 * 0.8, 0.1 * 0.35, 0.01 * 0.8$$

that is

$$[0.128, 0.140, 0.264, 0.035, 0.008]$$

normalizing (diving by 0.575)

$$[0.223, 0.243, 0.460, 0.061, 0.014]$$

## 2. Motor controller

(a) The motor speed could be off by 10%. For example if the real $J$ is 11, then $S/11 = S_{\text{desired}}/10$, that is, $S/S_{\text{desired}} = 11/10$.

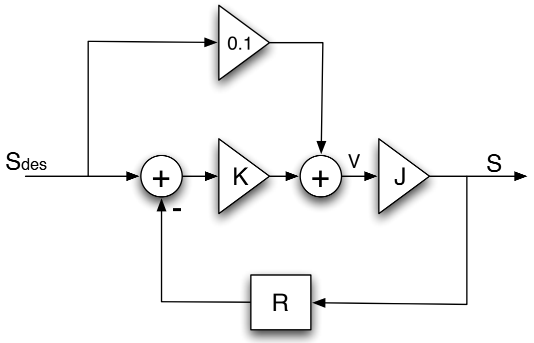
(b) The control is:

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

and the motor model is:

$$S[n] = JV[n]$$

Combining this gives us the following block diagram:



(c) The combined difference equation is:

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

or

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

so

$$H = \frac{S}{S_{\text{desired}}} = \frac{J(K + \frac{1}{10})}{1 + JKR}$$

therefore the single system pole is

$$-JK$$

For monotonic convergence we want

$$0 < -JK < 1$$

that is,

$$0 > K > -1/J$$

since $J \in [9, 11]$, we want $K > -\frac{1}{11}$.

(d) The following Python statement computes the controller voltage, given the desired velocity and a gain:

```
def control(Sdes, K):
    setVoltage(0.1*Sdes + K*(Sdes - readSpeed())
```

(e) Here's a simple, modular circuit to implement the control, it assumes that we have two signals $S$ and $S_{\text{desired}}$, one coming from the tachometer and the other representing the desired speed, the output is a voltage that can be used to drive the motor. We've assumed that we have +10V and -10V supplies so we don't need a virtual ground. We've assumed that the gain is $K = -1/11$, so the controller is

$$V = (1/10 - 1/11)S_{\text{desired}} + S/11$$

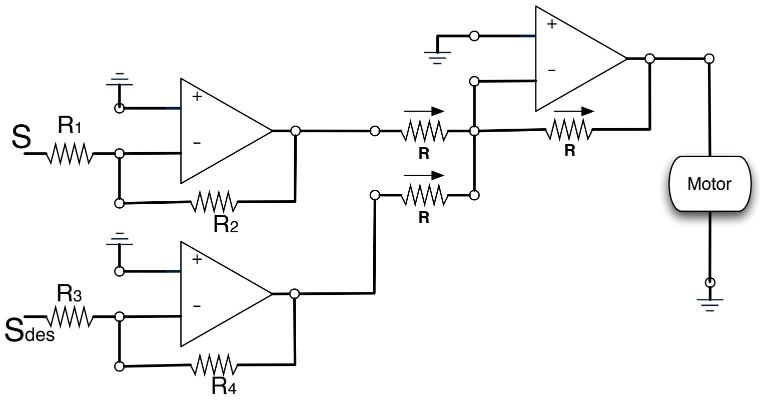
that is

$$V = S_{\text{desired}}/110 + S/11$$

The circuit components are:

- one inverting amplifier to make $-S/11$ - pick $R2/R1 = 1/11$.
- another inverting amplifier to make $-Sdes/110$ - pick $R4/R3 = 1/110$.
- an inverting summer to get $-(-S/11 + (-S_{\text{desired}}/110))$ - pick all the R's to be equal.
- connect that to one side of the motor and OV to the other side

We chose inverting amplifiers because it is easier to choose the resistors for the gains, but it could just as well be done with non-inverting amplifiers and a non-inverting summer.



(f) The analog controller has much lower delays than the computer controller and can therefore have a higher gain, leading to better performance, however the computer controller can be easily modified.

(g) If we change the controller to:

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

The system function is:

$$H = \frac{S}{S_{\text{desired}}} = \frac{J(K_1 + K_2 + \frac{1}{10})}{1 + JK_1R + JK_2R^2}$$

(h) The poles of this system function are the roots of:

$$z^2 + JK_1 z + JK_2$$

that is,

$$z = \frac{-JK_1 \pm \sqrt{J^2K_1^2 - 4JK_2}}{2}$$

To have the motor speed oscillate and eventually settle, we want the roots to be negative but of magnitude less that 1. Note that for $J = 10$, the choice of $K_1 = 0.1$ and $K_2 = 0$ produce roots of 0 and -1, increasing $K_2$ a little gives us the desired roots. For example, $K_1 = 0.1, K_2 = 0.01$ produce such roots, which also work for $J$ in the range between 9 and 11.

## 3. Search

(a) This creates the graph dictionary once and keeps it in the environment created by `makeSuccessors`:

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

(b) Here are a couple of implementations, one using a dictionary, the other just a list. Both construct a list of pairs `(goal, path)`.

```
def searchAllGoals(initial, goals, successor):
   paths = dict([(g, search(initial, lambda x: x==g, successor)) for g in goals])
   return lambda goal: paths[goal]

def searchAllGoals(initial, goals, successor):
   paths = [(g, search(initial, lambda x: x==g, successor)) for g in goals]
   def lookup(goal):
       for (g, p) in paths:
           if g == goal: return p
   return lookup
```

## 4. Circuits and Programming

(a) Change to add each equation in the list and the variables.

```
    def addConstraint(self, eqnList):
        # Adds a constraint where eqn is a list of (number string) or
        # (number None); the latter one is for constants.
        for eqn in eqnList:
           self.equations.append(eqn)
           for (n, var) in eqn:
               if var:
                   self.n2n.insert(var)
```

(b) Just change to return a list of one equation.

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

(c) Basically, a potentiometer defines a resistor between its first and second node (the wiper), with resistance `setting*totalR`, and another resistor between its second node (the wiper) and the third node with resistance `(1 - setting)*totalR`.

```
def potentiometer(totalR, setting, x):
    [vn1, vn2, vn3, i12, i23] = x
    return resistor(setting*totalR, [vn1, vn2, i12]) +
           resistor((1-setting)*totalR, [vn2, vn3, i23])
```