Name:	**Cohort:**

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
Established in collaboration with MIT

10.009 The Digital World
2014 Term 3

Date: 2 May 2014
Time: 9:00 a.m.
Duration: 2 hours

Instructions to candidates:

1. Write your name and cohort number at the top of this page.
2. This paper consists of 7 questions and 16 printed pages.
3. For questions 1–2, you are required to submit your answers via Tutor. For questions 3–7, you are required to write your answers in this exam booklet using a blue or black pen. All answers will be manually graded.
4. You may refer to the Digital World Notes, your personal notes, your copies of lab handouts, your solutions to lab questions, all material on your laptop, and any reference book.
5. You may write and test your code in IDLE.
6. You may use a web browser to refer to your personal copy of the libdw documentation, and your solutions to Tutor questions. No other Internet access allowed.
7. You may **not** communicate via any means with anyone (aside from the proctors in the examination room).

Qs 3	/10
Qs 4	/13
Qs 5	/9
Qs 6	/6
Qs 7	/12
Qs 3–8 Total	**/50**

1 Polylines in Python *(30 points)*

Develop a Python class to represent polylines. A polyline is a piecewise linear curve – one can think of it as a series of connected line segments. The polyline class will build upon the two classes below, which represents points and vectors.

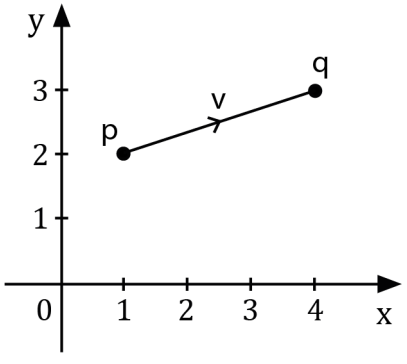
```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return 'Point2D(' + str(self.x) + ',' + str(self.y) + ')'

class Vector2D:
    def __init__(self, dx, dy):
        self.dx = dx
        self.dy = dy
    def length(self):
        return math.sqrt(self.dx**2 + self.dy**2)
```

1.1 *(5 points)*

Write a method `add` in the `Point2D` class that adds a `Vector2D` to a `Point2D`, resulting in another `Point2D`. Make sure that your method does not change the values of the arguments. Write your code and test it in IDLE, and **submit your answer via Tutor**.

```
>>> p =  Point2D(1,2)
>>> v = Vector2D(3,1)
>>> q = p.add(v)
>>> print q
Point2D(4,3)
```

The figure below illustrates `p` and `q`.



1.2 *(10 points)*

Implement the `Polyline2D` class, which has the following two methods. (Write your code and test it in IDLE, and **submit your answers via Tutor**.)

`__init__`: takes two arguments. The first argument is an instance of `Point2D` representing the start of the polyline. The second argument is a list of instances of Vector2D, which represents the consecutive line segments of the polyline.

`addSegment`: adds a line segment to the end of the polyline. It takes one argument, which is the new line segment represented as an instance of `Vector2D`. It does not need to return any value.

1.3 *(10 points)*

Implement the following methods of the `Polyline2D` class. (Write your code and test it in IDLE, and **submit your answer via Tutor**.)

`length`: returns the total length of the polyline, which is the sum of the lengths of the segments. This method has no arguments. You must use `Vector2D` 's `length` method in your implementation.

`vertex`: takes an integer index and returns the polyline vertex, represented as an instance of `Point2D`, at that index. The starting point of the polyline is vertex 0, and you can assume that the index you are given is legal.

Example code for `Polyline2D`:

```
>>> pline = Polyline2D(Point2D(1,2), [Vector2D(3,1)])
>>> pline.addSegment(Vector2D(1,0))
>>> pline.addSegment(Vector2D(0,2))
>>> print pline.length()
6.16227766017
>>> print pline.vertex(0)
Point2D(1,2)
>>> print pline.vertex(1)
Point2D(4,3)
>>> print pline.vertex(2)
Point2D(5,3)
>>> print pline.vertex(3)
Point2D(5,5)
```

1.4 *(5 points)*

Define a new class `ClosedPolyline2D` that has the same functionality as `Polyline2D`. Closed polylines form a closed loop in 2D. `ClosedPolyline2D` can have the same representation as `Polyline2D`, with segments added to it in exactly the same way. The difference is that, implicitly, the very last vertex is connected to the starting vertex, thus that last segment need not be added to the segment list explicitly.

The `ClosedPolyline2D` class must support all of the methods of the `Polyline2D` class, viz.,

`addSegment`, `vertex`, `length`. **This new class must make use of Polyline2D and should contain as little new code as possible.** (Hint: You should use inheritance and only the `length` method needs to be overridden in `ClosedPolyline2D`.)

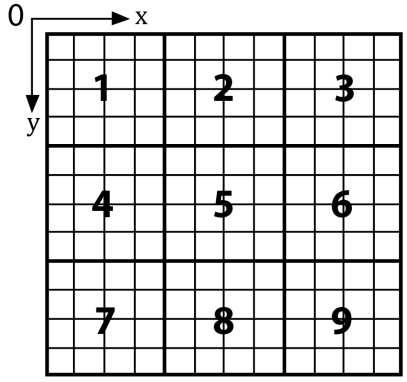
Write your code and test it in IDLE, and **submit your answer via Tutor**.

Example code for `ClosedPolyline2D`:

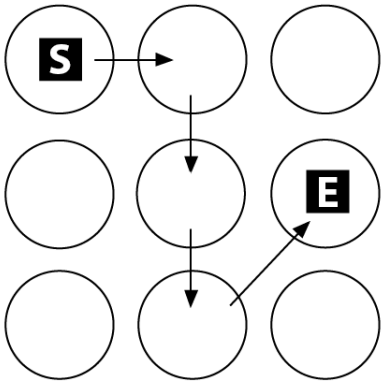
```
>>> cpline = ClosedPolyline2D(Point2D(1,2), [Vector2D(3,1)])
>>> cpline.addSegment(Vector2D(1,0))
>>> cpline.addSegment(Vector2D(0,2))
>>> print cpline.length()
11.1622776602
```

2 State Machine *(20 points)*

Touchscreen sensors indexed by their (x,y)-coordinates. Each (x,y) pair is mapped to a virtual point labeled 1 to 9.



Sliding from S(tart) to E(nd) through sequence indicated by arrows unlocks phone.



Implement a mechanism to unlock a cellphone's screen using finger movements. To unlock a phone, you have to move your finger through a grid of virtual points in a user-defined pattern (see diagram above). The phone is unlocked if at the time you lift your finger off the screen, the pattern of your finger swipe matches a pre-specified *key* pattern. One way to implement the unlocking mechanism is to treat each virtual point as an integer 1 to 9. The gesture pattern can then be modeled as a sequence of numbers. When a sequence of inputs matches the key sequence, the screen is unlocked and the machine is reset.

2.1 *10 points*

You shall complete the definition of a `CombLock` class, a subclass of state machine `sm.SM`, that implements a numerical combinational lock.

The *key* combination that unlocks the lock is passed to `CombLock` during initialization as a list of integers. The input to the `CombLock` machine is an integer. The machine should behave as follows.

- An input between 1 and 9 is to be remembered. The machine should output the string `'locked'`.
- A zero input should have no effect. The machine should output the string `'locked'`.
- When the input is -1, the machine should check if the remembered input sequence matches the key given at initialization. If the key matches, it outputs the string `'unlocked'`. Otherwise, it should output the string `'locked'`. In either case, the remembered sequence should be reset.

Fill in the code below at points marked "add your code here". Write your code and test it in IDLE, and **submit your answer via Tutor.**

```
import libdw.sm as sm
class CombLock(sm.SM):
    def __init__(self, keyList):
        #add your code here
    def getNextValues(self, state, inp):
        if inp = 0:
            #add your code here
        elif inp >= 1 and inp <= 9:
            #add your code here
        elif inp == -1:
            #add your code here
```

Example for `CombLock`:

```
>>> lock = CombLock([1,2,5])
>>> print lock.transduce([1,2,5,-1])
['locked', 'locked', 'locked', 'unlocked']
>>> print lock.transduce([1,0,2,5,-1])
['locked', 'locked', 'locked', 'locked', 'unlocked']
>>> print lock.transduce([3,2,5,-1])
['locked', 'locked', 'locked', 'locked']
>>> print lock.transduce([1,2,5,-1,1,2,5,-1])
['locked', 'locked', 'locked', 'unlocked', 'locked', 'locked', 'locked', 'unlocked']
>>> print lock.transduce([3,2,5,-1,1,2,5,-1])
['locked', 'locked', 'locked', 'locked', 'locked', 'locked', 'locked', 'unlocked']
```

2.2 *(10 points)*

To generate the correct key sequence from the touch screen, write a state machine class `TouchMap` that maps the touchscreen sensor reading to the corresponding virtual point. It uses a global helper function `mapT2P(x,y)` that maps a screen location `(x,y)` to the virtual point on the grid that it corresponds to, represented as an integer between 1 and 9. (Assume that the finger does not touch anywhere outside the grid.) `TouchMap` has the following behavior.

- Inputs to `TouchMap` are triples of the form `(e,x,y)`. `e` is one of three strings `'TouchDown'`, `'TouchUpdate'`, or `'TouchUp'`, which respectively correspond to a finger first touching the screen, swiping on the screen, and lifting off the screen. `x, y` refer to the `(x, y)` location on the touchscreen where the event `e` occurred.
- `TouchMap` outputs a virtual point number (1 to 9) when a user starts touching the screen at a virtual point, or when the finger moves to a different virtual point.
- `TouchMap` outputs 0 when the finger remains within the vicinity of the current nearest virtual point.
- `TouchMap` outputs -1 when the finger is lifted.

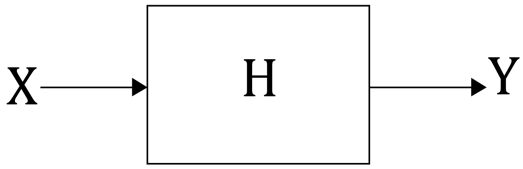
```
import libdw.sm as sm

def mapT2P(x,y):
    if 0 <= x and x <= 3:
        if 0 <= y and y <= 3:
            return 1
        if 4 <= y and y <= 7:
            return 4
        if 8 <= y and y <= 11:
            return 7
    if 4 <= x and x <= 7:
        if 0 <= y and y <= 3:
            return 2
        if 4 <= y and y <= 7:
            return 5
        if 8 <= y and y <= 11:
            return 8
    if 8 <= x and x <= 11:
        if 0 <= y and y <= 3:
            return 3
        if 4 <= y and y <= 7:
            return 6
        if 8 <= y and y <= 11:
            return 9

class TouchMap(sm.SM):
    startState = 0
    def getNextValues(self, state, inp):
        (e,x,y) = inp
        #add your code here
```

Example code for `TouchMap`:

```
>>> m = TouchMap()
>>> print m.transduce([('TouchDown',2,2), ('TouchUpdate',3,3), ('TouchUp',4,4)])
[1, 0, -1]
>>> print m.transduce([('TouchDown',3,3), ('TouchUpdate',4,3), ('TouchUp',4,4)])
[1, 2, -1]
```

Submit your answer via Tutor.

3 Systems *(10 points)*



Suppose the input X and ouput Y of the system H above are defined as follows.

$$x[n] = \delta[n] = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$y[n] = \begin{cases} (-2)^n & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

3.1 *(4 points)*

Let $y_1[n]$ be the output response to input $x_1[n]$ (specified below) into the same systen H.

$$x_1[n] = \delta[n] - 16\delta[n-4] = \begin{cases} 1 & \text{if } n = 0 \\ -16 & \text{if } n = 4 \\ 0 & \text{otherwise} \end{cases}$$

Express $y_1[n]$ in terms of $y[n]$: $y_1[n] =$ ☐

Fill in the values below.

$y_1[1] =$ ☐

$y_1[3] =$ ☐

$y_1[4] =$ ☐

$y_1[6] =$ ☐

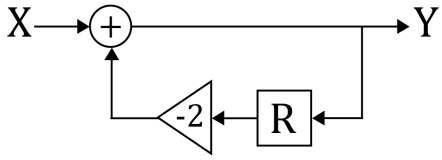
3.2 *(3 points)*

Specify H's pole(s): [ ]

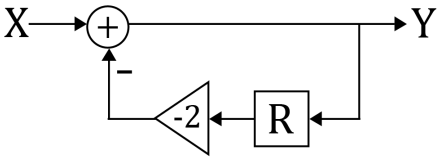
Is H stable? (Yes/No): [ ]

3.3 *(3 points)*

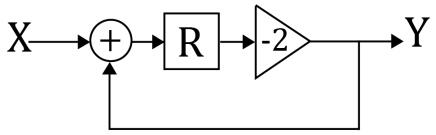
(a)



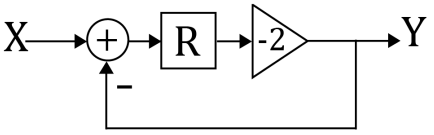
(b)



(c)

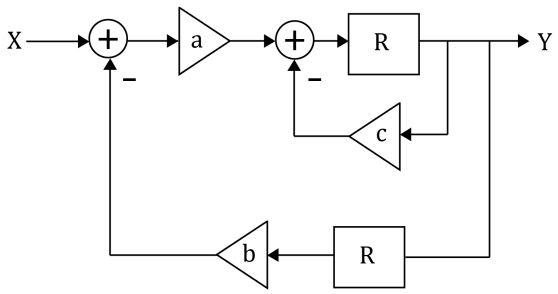


(d)



Which ONE of the above block diagrams is equivalent to H: [ ]

4 Representation *(13 points)*



4.1 *(4 points)*

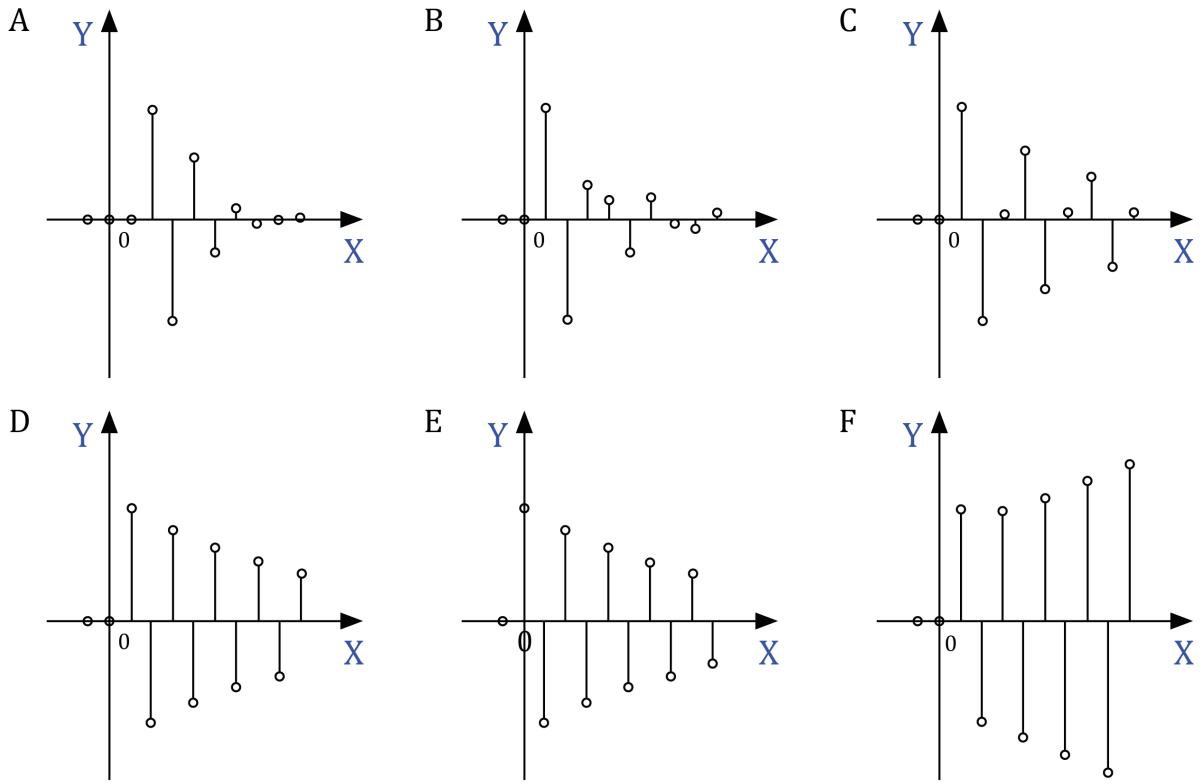
Write the system function for the above system: $H = \frac{Y}{X} =$

4.2 *(3 points)*

Write the difference equation for the above system: $y[n] =$

4.3 *(6 points)*

Below are responses that the system could produce (the system starts at rest and the input is a unit sample signal).



Assume $a = 0.5$ and $c = 0.9$. Which ONE of the above figures (A,B,C,D,E,F) best represents the system behavior

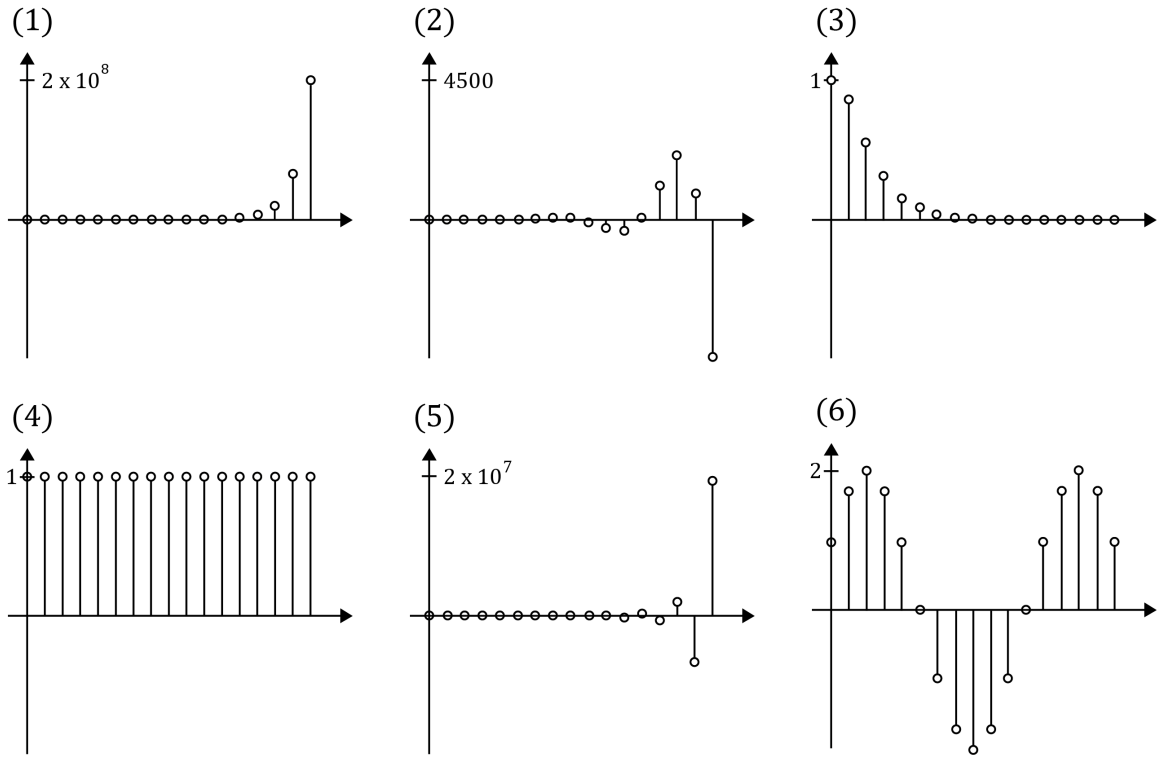
when $b = 0$: [ ]

when $b = -0.4$: [ ]

when $b = 1.5$: [ ]

5 Poles *(9 points)*

$$H = \frac{Y}{X} = \frac{1}{a + bR + cR^2 + dR^3} \quad (a, b, c, d \text{ are real constants})$$



For the above system function H, match one of its unit sample response graphs above to each set of poles below. Write "invalid" if the poles do not match any graph.

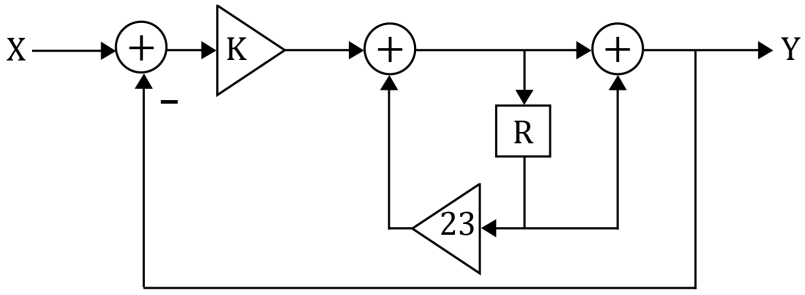
A. Poles: $3, 1 + \sqrt{2}j, 2 - \sqrt{2}j$; Graph: ☐

B. Poles: $0, \frac{\sqrt{3}}{2} \pm \frac{1}{2}j$ Graph: ☐

C. Poles: $1, 2, -33$ Graph: ☐

6 Stability Analysis *(6 points)*

Specify $K's$ range of values that makes the system below stable:



7 Lab: Hitting the Wall *(12 points)*

In Week 9's lab "Hitting the Wall", you wrote a controller to move a robot to a point that was a fixed distance from a wall. You made a model (that combined a controller, a plant, and a sensor) to explain the robot's behavior in terms of the system's parameters.

Suppose that the desired distance $d_i = 0.5$ m. Describe one possible combination of initial distance $d_o[0]$ and poles p_0, p_1 to account for each of the robot's behavior below. If the behavior is not possible, write "N.A." in the answer boxes. Briefly explain your answers.

1. The robot moves backward forever, and eventually gets arbitrary far from the wall.

$d_o[0] =$

$p_0 =$

$p_1 =$

Briefly explain:

2. The robot creeps forward slowly and eventually stops.

$d_o[0] =$

$p_0 =$

$p_1 =$

Briefly explain:

3. The robot moves backward fast, then moves forward and eventually crashes into the wall.

$d_o[0] =$

$p_0 =$

$p_1 =$

Briefly explain:

END OF PAPER