ECE 383 / MEMS 442 / ECE 595 Fall 2017 Final Exam

Deadline: December 15 at 11pm

Instructions: collect your written answers in a final.pdf file, and collector your Klamp't Online code into a file named final.py. Zip and submit the package on Sakai before the deadline.

You are NOT permitted to consult other students or the TAs for help during this exam. Clarifying questions are allowed, as long as they are posted on the course Piazza site and are public.

Summary: the Klamp't Olympics

In this final exam, you will design and develop a robot's sensing and control system to perform in an "event" in the Klamp't Olympics. You will be assigned an event (A, B, or C) which will designate which task that you will work on.

Event A (Robot Goalie): In your event, balls will be hurled toward a goal, and your robot should block as many of them as possible from passing through the goal.

Event B (Robot Catcher): In your event, balls will be lobbed toward you, and your robot must catch as many of them as possible.

Event C (Robot Batter): In your event, you will need to hit as many balls into a goal as possible, while obstacles try to block the ball's path.

Your robot's performance on an event will be judged by an *event score*, which in event A subtracts -10 points for every ball scored, in event B adds +10 points for every ball caught, and in event C adds +10 points for every ball scored. Points are subtracted for various penalties, including collisions, joint limit violations, taking too long, etc.

Note: There is no "correct answer", as there are many possible approaches to address each task. It is unknown whether a perfect score is even possible.

The final will consist of a written portion (60 points), in which you describe the design and technical details of a system to address your assigned task, as well as a programming portion, in which you implement the proposed system (40 points). Furthermore, the student with the highest score in each event (in medium difficulty mode, with omniscient sensing off) will receive 5 points extra credit.

Note: if your implementation is incomplete you will not be penalized on the written portion of this exam. It is still possible to achieve full credit for a written portion that is complete, technically correct, and logically sound even without an implementation. However, it is likely to prove difficult to fully understand the logical issues involved in solving the problem without attempting the implementation.

Robot and Sensors

The robot used in this project is a standard 6DOF fixed-base industrial robot, affixed with a paddle, scoop, or bat attachment, depending on your event. The low-level controller drives its motors using a PID controller with gravity compensation. You will be penalized for collisions between the robot and the playing field (20 points per second of violation), and for violating the robot's joint limits or its torque limits (10 points per second of violation).

Two types of sensors are available for you to use: 1) the robot's joint encoders and 2) a simulated camera sensor. The camera is located at a fixed location off of the robot. The coordinates of the camera frame with respect to the world frame is known, and its origin is at the camera focal point, its x direction points in the "right" direction of the camera image, and the y direction points in the "down" direction of the camera image.

Rather than directly accessing the camera's pixels, you will have access to a "blob detector" which produces rectangular windows of detected "blobs" that indicate various objects of interest. In later stages of this project you will use these blobs to estimate object positions (and possibly velocities) in 3D space. These attributes are stored in the `CameraColorDetectorOutput` class. However, in early stages of this project we will provide "omniscient", precise readings of object positions and velocities. This will help you debug your high-level controller.

In Events A and B, each subsequent ball will be of a different color. (There may be multiple balls in play at once.) In Event C, the ball will be red (RGB color (1,0,0)) and the other obstacles will be of different non-white colors.

When the sensor is enabled with `omniscient_sensor = False`, the "blobs" that it detects will be drawn near the yellow camera widget. These are for your own reference, only.

High-Level Controller

Your primary job is to design a high-level controller to make your robot perform the event as best as possible. This involves implementing a control loop in which you read from sensors, process the information, and then send a command to the robot's low-level controller.

The processing will involve perception (in particular, state estimation of moving objects), control (in particular, managing the internal state of different control components) and planning (deciding where and how to move in response to detected objects, and the robot's current state).

You will be penalized for excessively long control loops. One point will be subtracted from your event score every time your controller takes more than 1s calculation per 1s of execution time. If it takes more than 5s calculation per 1s of execution time, 5 points will be subtracted from your score, and the run will be terminated.

Manual Controller and Robot Posing

When designing your system, it may be helpful to do manual control of your robot. To turn on manual control, use the "Mode" selection box to switch to User Mode. Your controller's commands will be ignored now. Now, the sliders will control the robot's joint angles.

A final useful tool is the "Print" button, which prints out the slider's configuration in the form of a Python object. You can then add this configuration as an object in your code.

State Estimation

When the omniscient sensor is provided, the `OmniscientStateEstimator` class processes those raw measurements into a `MultiObjectStateEstimate` object. In your `MyObjectStateEstimator` class, your job will be to process a stream of `CameraColorDetectorOutput` objects into a stream of `MultiObjectStateEstimate` objects.

Common Code

The file common.py, detailed in the Appendix, contains all definitions that are available to you regarding sensor outputs, expected state estimator outputs, and utility code.

1. System Design (Written)

In this portion you will describe your implementation of a software system to solve the task assigned to you. You will be judged on the logical correctness of your approach and the clarity of your description. You should describe in precise terms how the information and inputs that you are given are (or ought to be) processed to produce the desired output. "Information" can include (but is not limited to) problem assumptions, mathematical models, knowledge of physics, knowledge from personal observation, postures and motions designed by hand, or empirical data. DO NOT DOCUMENT COMPONENTS PROVIDED TO YOU.

A. (10 pts) **Summary**. Describe the overall design of your system, including perception, planning, and control components. Summarize the approach you have taken. As part of this summary, include a block diagram naming each component as well as information flow between each component. Each block is a component, and each arc (arrow) is a piece of information. Label each block and arc.

B. (15 pts) **Components**. For each component and subcomponent described above, describe in precise, technical language:
    a. Its purpose, including how it relates to neighboring components
    b. Its inputs, including the data type, format, and reference frame (if applicable)
    c. Its output, including the data type, format, and reference frame (if applicable)
    d. How it is invoked. (Is it run at a constant rate? On request?)
    e. How it is implemented. (What algorithm? Is it an off-the-shelf or custom implementation?)

f. Potential failure cases

C. (15 pts) **Planning and Control Strategy.** Describe your planning / control components in more detail. As part of this description, include a state machine diagram of how the high-level controller processes perception inputs and invokes planning components to produce commands for the low-level controller. How does it handle planning failures?

D. (15 pts) **Perception Strategy.** Describe your perception components in more detail. Do you need to estimate all components of 3D object positions and velocities in order to perform your task? To what accuracy do you expect to measure them? What strategy or algorithm do you use to implement these components? If there is a mathematical model you use, state this model. For example, if you are using a Bayesian filter, describe the transition model, observation model, and belief initialization.

E. (5 pts) **Reflection.** After implementing your system, write 1-2 paragraphs describing how well the system worked, what you could have improved, what was harder than expected, etc. Also, describe what additional challenges you might face when designing a robot to perform your event in real life rather than in simulation.

2. System Implementation

The programming portion of this exam consists of two independent subsystems. Part A is the high-level controller subsystem, and Part B is the perception (i.e., state estimation) subsystem.

It is possible to work on Part A without having Part B implemented. The "omniscient_sensing = True" flag at the top of the file provides your controller with access to the true state of the world, circumventing the need for state estimation.

A. (20 pts) Implement the high-level controller subsystem. You can test it on easy, medium, and hard instances of your problem by setting the "difficulty" flag to either "easy", "medium", or "hard".

   By default, the event is randomized with a fixed random seed. This tests your event deterministically which may help you eliminate bugs. To test your event more generally, you can uncomment the line "seed = random.seed()". (This call seeds the random number generator with the current time, ensuring a new random event every time the program is run).

B. (20 pts) Implement the perception subsystem. For object state estimation, you are free to use the sensor transform `Tsensor`, and you are free to use your HW5 solution and/or the Kalman filtering code provided for you in the `common` module (but you are under no obligation to use it).

When you are ready to test your state estimation system in practice, you should set "omniscient_sensing = False" at the top of the file and comment out the line "multiObjectStateEstimate = omniscientObjectState."

Appendix: common.py module

```
import numpy as np

#Raw sensing output

class OmniscientObjectOutput:
    """Stores the output of an OmniscientObjectSensor
    Attributes:
    names: a list of object names (actually, these will be the colors
      of the objects)
    positions: a list of object positions
    velocities: a list of object positions
    """
    def __init__(self,names,positions,velocities):
        self.names = names[:]
        self.positions = positions[:]
        self.velocities = velocities[:]

class CameraBlob:
    """A blob on the camera screen, in image coordinates.
    Attributes:
    - color: an (r,g,b) tuple indicating the blob's color.
    - x,y: the coordinates of the blob center
    - w,h: the width and height of the blob
    """
    def __init__(self,color,x,y,w,h):
        self.color = color
        self.x = x
        self.y = y
        self.w = w
        self.h = h

class CameraColorDetectorOutput:
    """Stores the output of a CameraColorDetectorSensor
    Attributes:
    blobs: a list of CameraBlobs specifying the regions of the detected
      blobs.
    """
    def __init__(self,blobs):
        self.blobs = blobs[:]

#State estimation code

class ObjectStateEstimate:
    """Attributes:
    - name: an identifier of the object
    - x: mean state (position / velocity) estimate, a 6-D vector
    - cov: state (position / velocity) covariance, a 6x6 numpy array
    """
    def __init__(self,name,x,cov=0):
        self.name = name
        self.x = x
        self.cov = cov
        if isinstance(cov,(int,float)):
            self.cov = np.eye(6)*cov
    def meanPosition(self):
        return self.x[0:3]
    def meanVelocity(self):
        return self.x[3:6]

class MultiObjectStateEstimate:
    """A list of ObjectStateEstimates.
```

```
    Attributes:
        - objects: a list of ObjectStateEstimates, corresponding to all
          of the objects currently tracked by the state estimator.
    """
    def __init__(self,objectEstimates):
        self.objects = objectEstimates[:]

    def get(self,name):
        """Retrieves an object's state estimate by name"""
        for o in self.objects:
            if o.name==name:
                return o
        return None

class OmniscientStateEstimator:
    """A hack state estimator that gives perfect state information from
    OmniscientObjectOutput readings."""
    def __init__(self):
        self.reset()
        return
    def reset(self):
        pass
    def update(self,o):
        """Produces an updated MultiObjectStateEstimate given an OmniscientObjectOutput
        sensor reading."""
        assert isinstance(o,OmniscientObjectOutput),"OmniscientStateEstimator only works with an omniscient
sensor"
        estimates = [ObjectStateEstimate(n,p+v) for n,p,v in zip(o.names,o.positions,o.velocities)]
        return MultiObjectStateEstimate(estimates)

#Kalman filtering code

def gaussian_linear_transform(mean,cov,A,b):
    """Given a prior x~N(mean,cov), returns the
    mean and covariance of the variate y=A*x+b.
    """
    ymean = np.dot(A,mean)+b
    ycov = np.dot(A,np.dot(cov,A.T))
    return (ymean,ycov)

def kalman_filter_predict(prior_mean,prior_cov,F,g,SigmaX):
    """For the Kalman filter model with transition model:
      x[t] = F*x[t-1] + g + eps_x
    with eps_x ~ N(0,SigmaX)
    and given prior estimate x[t-1]~N(prior_mean,prior_cov),
    computes the predicted mean and covariance matrix at x[t]

    Output:
    - A pair (mu,cov) with mu the updated mean and cov the updated covariance
      matrix.

    Note: all elements are numpy arrays.
    """
    if isinstance(SigmaX,(int,float)):
        SigmaX = np.eye(len(prior_mean))*SigmaX
    muprime = np.dot(F,prior_mean)+g
    covprime = np.dot(F,np.dot(prior_cov,F.T))+SigmaX
    return (muprime,covprime)

def kalman_filter_update(prior_mean,prior_cov,F,g,SigmaX,H,j,SigmaZ,z):
    """For the Kalman filter model with transition model:
      x[t] = F*x[t-1] + g + eps_x
    and observation model
      z[t] = H*x[t] + j + eps_z
    with eps_x ~ N(0,SigmaX) and eps_z ~ N(0,SigmaZ),
    and given prior estimate x[t-1]~N(prior_mean,prior_cov),
    computes the updated mean and covariance matrix after observing z[t]=z.

    Output:
    - A pair (mu,cov) with mu the updated mean and cov the updated covariance
```

```
  matrix.

Note: all elements are numpy arrays.

Note: can be applied as an approximate extended Kalman filter by setting
F*x+g and H*x+j to be the linearized models about the current estimate
prior_mean.  (The true EKF would propagate the mean update and linearize
the observation term around the mean update)
"""
if isinstance(SigmaX,(int,float)):
    SigmaX = np.eye(len(prior_mean))*SigmaX
if isinstance(SigmaZ,(int,float)):
    SigmaZ = np.eye(len(z))*SigmaZ
muprime = np.dot(F,prior_mean)+g
covprime = np.dot(F,np.dot(prior_cov,F.T))+SigmaX
C = np.dot(H,np.dot(covprime,H.T))+SigmaZ
zpred = np.dot(H,muprime)+j
K = np.dot(covprime,np.dot(H.T,np.linalg.pinv(C)))
mu = muprime + np.dot(K,z-zpred)
cov = np.dot(np.eye(covprime.shape[0])-np.dot(K,H), covprime)
return (mu,cov)
```