ECE 383/MEMS 442/ECE 590 Lab 5: State Estimation

Due date: 12/11/2017 at 5pm

Instructions: This assignment is to be completed *individually*. You are allowed to discuss the problems with your classmates, but all work must be your own. You are not permitted to copy code, take written notes, or look up online solutions.

All coding will be done on the Klamp't Docker container on http://vm-manage.oit.duke.edu/.

To submit, copy your code from the browser to a file lab5.py, and submit it on Sakai.

Problem Description

Your task is to estimate the position and velocities of objects thrown toward a goal using a simulated camera sensor. The objects are launched one by one and follow ballistic trajectories with no air resistance.

The camera is located at a fixed location near the goal. The update() function in your code chooses between different state estimators for different sensors. 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.

The predicted ballistic trajectories of the objects as given by your state estimator are drawn as lines.

Omniscient mode. By default, we provide an "omniscient" sensor that provides precise readings of object positions and velocities. Each object is indexed by its "name" which is actually the (r,g,b,a) tuple giving its color. In this mode, the `OmniscientStateEstimator` class processes those raw measurements (trivially) into a `MultiObjectStateEstimate` object.

Position mode. The position sensor provides noisy position estimates for each ball, which are within +/- 0.1 unit of the true position. The camera's characteristics has no bearing on the sensor readings. In this mode, the `PositionStateEstimator` class processes a stream of `ObjectPositionOutput` objects to produce an updated `MultiObjectStateEstimate` object. You will need to fill out this function in part A.

Blob detector mode. 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. You will use these blobs to estimate object positions and velocities in 3D space. These attributes are stored in the `CameraColorDetectorOutput` class.

The "blobs" that it detects will be drawn near the yellow camera widget. These are for your own reference, only.

In the `BlobStateEstimator` class, your job will be to process a stream of `CameraColorDetectorOutput` objects to produce an updated `MultiObjectStateEstimate` objects. You will need to fill out this function in part B.

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. Notably, the `kalman_filter_x` functions are available for you to use.

Numpy

You will need to use the Numpy module to set up matrices and vectors for the Kalman filter. A tutorial on Numpy can be found here. If you are a Matlab user, this Matlab to Numpy Guide may be helpful.

A. Kalman Filter with Position Sensor

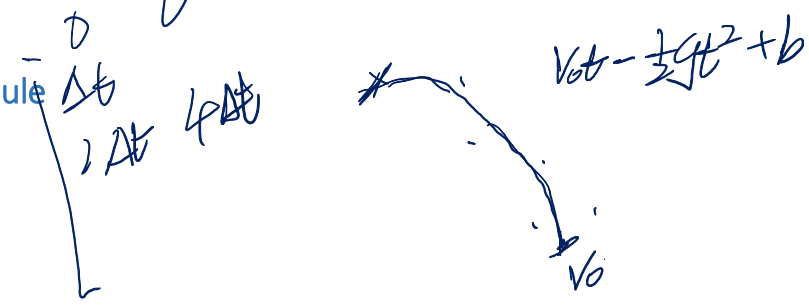
In this problem, your task is to implement a Kalman filter for the Position sensing mode.

1. Using the information provided to you, build a linear Gaussian forward dynamics model and observation model. Implement the Kalman filter to properly maintain a state estimate so that the estimated position and velocity estimate the correct ballistic trajectory quickly after the object is launched.
   (You may store whatever information you need to keep from time step to time step in `self`. You may also change the random seed to test the performance of your method for different throwing targets)
2. What quantities of your models came directly from the problem description? What assumptions did you need to make to fill in the missing information?
3. When a ball is launched, there is a discontinuity in its velocity that is not modeled well by a Gaussian white noise. Compared to setting the velocity disturbance variance to be large, how might you handle this phenomenon more robustly?

B. Extended Kalman Filter with the Blob Detection Sensor

In this problem, your task is to implement an Extended Kalman Filter (EKF) with the Blob Detector sensing mode.

1. Using the information provided to you as well as in the camera extrinsic and intrinsic parameters Tsensor, fov, w, h, and dmax, build a forward dynamics model and observation model. (These should be similar to your answer in HW5.)
2. Implement an EKF by linearizing each update about the current state estimate. You will need to perform initialization of each tracked object when a new blob appears.
3. How well does this perform compared to the Position Sensor?



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

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


#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)
```