

1 Kalman Filter

Lab Objective: *Understand how to implement the standard Kalman Filter. Apply to the problem of projectile tracking.*

Measured observations are often prone to significant noise, due to restrictions on measurement accuracy. For example, most commercial GPS devices can provide a good estimate of geolocation, but only within a dozen meters or so. A Kalman filter is an algorithm that takes a sequence of noisy observations made over time and attempts to get rid of the noise, producing more accurate estimates than the original observations. To do this, the algorithm needs information about the system being observed.

Consider the problem of tracking a projectile as it travels through the air. Short-range projectiles approximately trace out parabolas, but a sensor that is recording measurements of the projectile's position over time will likely show a path that is much less smooth. Because we know something about the laws of physics, we can filter out the noise in the measurements using basic Newtonian mechanics, recovering a more accurate estimate of the projectile's trajectory. In this lab, we will simulate measurements of a projectile and implement a Kalman filter to estimate the complete trajectory of the projectile.

Linear Dynamical Systems

The standard Kalman filter assumes that: (1) we have a linear dynamical system, (2) the state of the system evolves over time with some noise, and (3) we receive noisy measurements about the state of the system at each iteration. More formally, letting \mathbf{x}_k denote the state of the system at time k , we have

$$\mathbf{x}_{k+1} = F_k \mathbf{x}_k + G_k \mathbf{u}_k + \mathbf{w}_k \quad (1.1)$$

where F_k is a state-transition model, G_k is a control-input model, \mathbf{u}_k is a control vector, and \mathbf{w}_k is the noise present in state k . This noise is assumed to be drawn from a multivariate Gaussian distribution with zero mean and covariance matrix Q_k . The control-input model and control vector allow the assumption that the state can be additionally influenced by some other factor than the linear state-transition model.

We further assume that the states are “hidden,” and we only get the noisy observations

$$\mathbf{z}_k = H_k \mathbf{x}_k + \mathbf{v}_k \quad (1.2)$$

where H_k is the observation model mapping the state space to the observation space, and \mathbf{v}_k is the observation noise present at iteration k . As with the aforementioned error, we assume that this noise is drawn from a multivariate Gaussian distribution with zero mean and covariance matrix R_k .

The dynamics stated above are all taken to be linear. Thus, for our purposes, the operators F_k , G_k , and H_k are all matrices, and \mathbf{x}_k , \mathbf{u}_k , \mathbf{z}_k , and \mathbf{v}_k are all vectors.

We will assume that the transition and observation models, the control vector, and the noise covariances are constant, i.e. for each k , we will replace F_k , H_k , \mathbf{u}_k , Q_k , and R_k with F , H , \mathbf{u} , Q , and R . We will also assume that $G = I$ is the identity matrix, so it can safely be ignored.

Problem 1. Begin implementing a `KalmanFilter` class by writing an initialization method that stores the transition and observation models, noise covariances, and control vector. We provide an interface below:

```
class KalmanFilter(object):
    def __init__(self, F, Q, H, R, u):
        """
        Initialize the dynamical system models.

        Parameters
        -----
        F : ndarray of shape (n,n)
            The state transition model.
        Q : ndarray of shape (n,n)
            The covariance matrix for the state noise.
        H : ndarray of shape (m,n)
            The observation model.
        R : ndarray of shape (m,m)
            The covariance matrix for observation noise.
        u : ndarray of shape (n,)
            The control vector.
        """
        pass
```

We now derive the linear dynamical system parameters for a projectile traveling through \mathbb{R}^2 undergoing a constant downward gravitational force of 9.8 m/s^2 . The relevant information needed to describe how the projectile moves through space is its position and velocity. Thus, our state vector has the form

$$\mathbf{x} = \begin{pmatrix} s_x \\ s_y \\ V_x \\ V_y \end{pmatrix},$$

where s_x and s_y give the x and y coordinates of the position (in meters), and V_x and V_y give the horizontal and vertical components of the velocity (in meters per second), respectively.

How does the system evolve from one time step to the next? Assuming each time step is 0.1

seconds, it is easy enough to calculate the new position:

$$\begin{aligned}s'_x &= s_x + 0.1V_x \\ s'_y &= s_y + 0.1V_y.\end{aligned}$$

Further, since the only force acting on the projectile is gravity (we are ignoring things like wind resistance), the horizontal velocity remains constant:

$$V'_x = V_x.$$

The vertical velocity, however, does change due to the effects of gravity. From basic Newtonian mechanics, we have

$$V'_y = V_y - 0.1 \cdot 9.8.$$

In summary, over one time step, the state evolves from \mathbf{x} to \mathbf{x}' , where

$$\mathbf{x}' = \begin{pmatrix} s_x + 0.1V_x \\ s_y + 0.1V_y \\ V_x \\ V_y - 0.98 \end{pmatrix}.$$

From this equation, you can extract the state transition model F and the control vector u .

We now turn our attention to the observation model. Imagine that a radar sensor captures (noisy) measurements of the projectile's position as it travels through the air. At each time step, the radar transmits the observation $z = (z_x, z_y)$ given by

$$\begin{aligned}z_x &= s_x + v_x \\ z_y &= s_y + v_y,\end{aligned}$$

where (v_x, v_y) is a noise vector assumed to be drawn from a multivariate Gaussian with mean zero and some known covariance. These equations indicate the appropriate choice of observation model.

Problem 2. Work out the transition and observation models F and H , along with the control vector \mathbf{u} , corresponding to the projectile. Assume that the noise covariances are given by

$$\begin{aligned}Q &= 0.1 \cdot I_4 \\ R &= 5000 \cdot I_2.\end{aligned}$$

Instantiate a `KalmanFilter` object with these values.

We now wish to simulate a sequence of states and observations from the dynamical system. In addition to the system parameters, we need an initial state \mathbf{x}_0 to get started. Computing the subsequent states and observations is simply a matter of following equations 1.1 and 1.2. Bear in mind that the initial state \mathbf{x}_0 does not have added noise, but the initial observation and all subsequent states and observations take noise into account.

Problem 3. Add a method to your `KalmanFilter` class to generate a state and observation sequence by evolving the system from a given initial state (the function `numpy.random.multivariate_normal` will be useful). To do this, implement the following:

```
def evolve(self, x0, N):
```

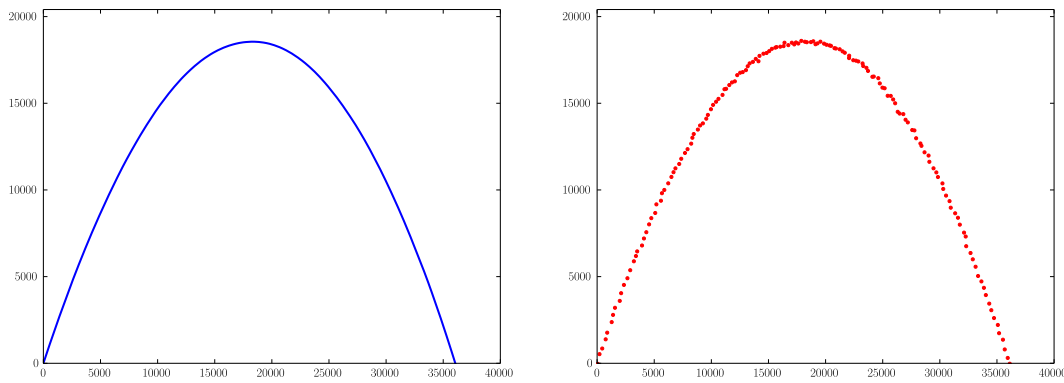


Figure 1.1: State sequence (left) and sampling of observation sequence (right).

```

"""
Compute the first N states and observations generated by the Kalman ↔
system.

Parameters
-----
x0 : ndarray of shape (n,)
    The initial state.
N : integer
    The number of time steps to evolve.

Returns
-----
states : ndarray of shape (n,N)
    States 0 through N-1, given by each column.
obs : ndarray of shape (m,N)
    Observations 0 through N-1, given by each column.
"""
pass

```

To test your code, simulate the true and observed trajectories of a projectile with initial state

$$\mathbf{x}_0 = \begin{pmatrix} 0 \\ 0 \\ 300 \\ 600 \end{pmatrix}.$$

Approximately 1250 time steps should be sufficient for the projectile to hit the ground (i.e. for the y coordinate to return to 0). Your results should qualitatively match those given in Figure 1.1.

State Estimation with the Kalman Filter

The Kalman filter is a recursive estimator that smooths out the noise in real time, estimating each current state based on the past state estimate and the current measurement. This process is done by repeatedly invoking two steps: Predict and Update. The predict step is used to estimate the current state based on the previous state. The update step then combines this prediction with the current observation, yielding a more robust estimate of the current state.

To describe these steps in detail, we need additional notation. Let

- $\hat{\mathbf{x}}_{n|m}$ be the state estimate at time n given only measurements up through time m ; and
- $P_{n|m}$ be an error covariance matrix, measuring the estimated accuracy of the state at time n given only measurements up through time m .

The elements $\hat{\mathbf{x}}_{k|k}$ and $P_{k|k}$ represent the state of the filter at time k , giving the state estimate and the accuracy of the estimate.

We evolve the filter recursively, as follows:

Predict

$$\hat{\mathbf{x}}_{k|k-1} = F\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{u}$$

Update

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q$$

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - H\hat{\mathbf{x}}_{k|k-1}$$

$$S_k = HP_{k|k-1}H^T + R$$

$$K_k = P_{k|k-1}H^TS_k^{-1}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k\tilde{\mathbf{y}}_k$$

$$P_{k|k} = (I - K_kH)P_{k|k-1}$$

The more observations we have, the greater the accuracy of these estimates becomes (i.e the norm of the accuracy matrix converges to 0).

Problem 4. Add code to your `KalmanFilter` class to estimate a state sequence corresponding to a given observation sequence and initial state estimate. Implement the following class method:

```
def estimate(self, x, P, z):
    """
    Compute the state estimates using the Kalman filter.
    If x and P correspond to time step k, then z is a sequence of
    observations starting at time step k+1.

    Parameters
    -----
    x : ndarray of shape (n,)
        The initial state estimate.
    P : ndarray of shape (n,n)
        The initial error covariance matrix.
    z : ndarray of shape(m,N)
        Sequence of N observations (each column is an observation).
```

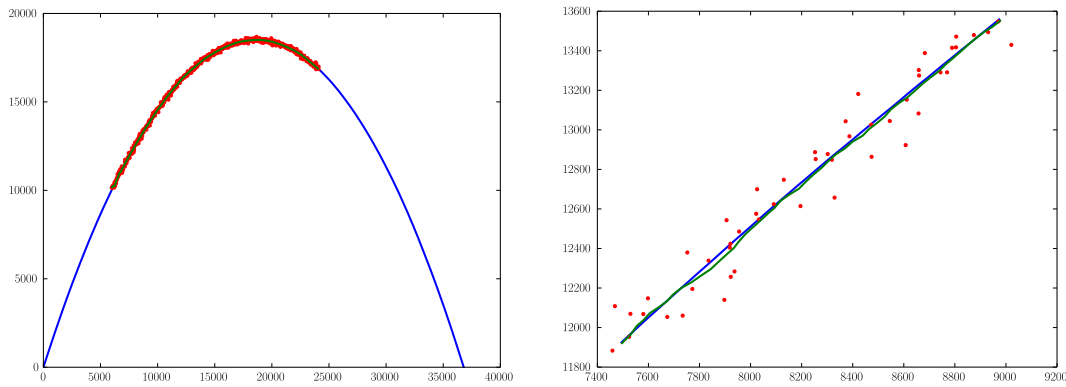


Figure 1.2: State estimates together with observations and true state sequence (detailed view on the right).

```

Returns
-----
out : ndarray of shape (n,N)
      Sequence of state estimates (each column is an estimate).
"""
pass

```

Returning to the projectile example, we now assume that our radar sensor has taken observations from time steps 200 through 800 (take the corresponding slice of the observations produced in Problem 3). Using these observations, we seek to estimate the corresponding true states of the projectile. We must first come up with a state estimate $\hat{\mathbf{x}}_{200}$ for time step 200, and then feed this into the Kalman filter to obtain estimates $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$.

Problem 5. Calculate an initial state estimate $\hat{\mathbf{x}}_{200}$ as follows: For the horizontal and vertical positions, simply use the observed position at time 200. For the velocity, compute the average velocity between the observations \mathbf{z}_k and \mathbf{z}_{k+1} for $k = 200, \dots, 208$, then average these 9 values and take this as the initial velocity estimate. (Hint: the NumPy function `diff` is useful here.)

Using the initial state estimate, $P_{200} = 10^6 \cdot Q$, and your Kalman filter, compute the next 600 state estimates, i.e. compute $\hat{\mathbf{x}}_{201}, \dots, \hat{\mathbf{x}}_{800}$. Plot these state estimates as a smooth green curve together with the radar observations (as red dots) and the entire true state sequence (as a blue curve). Zoom in to see how well it follows the true path. Your plots should be similar to Figure 1.2.

In the absence of observations, we can still estimate some information about the state of the system at some future time. We can do this by recognizing that the expected state noise $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$ at any time k . Thus, given a current state estimate $\hat{\mathbf{x}}_{n|m}$ using only measurements up through time m , the expected state at time $n + 1$ is

$$\hat{\mathbf{x}}_{n+1|m} = F\hat{\mathbf{x}}_{n|m} + \mathbf{u}$$

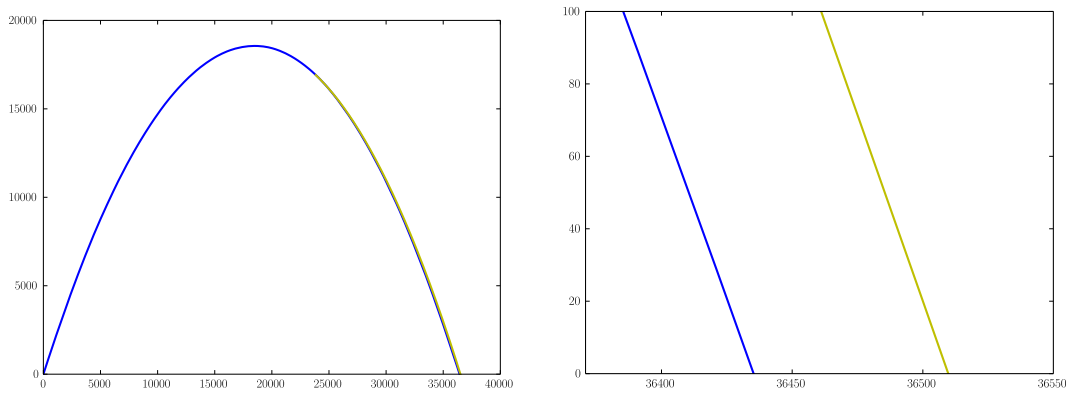


Figure 1.3: Predicted vs. actual point of impact (detailed view on right).

Problem 6. Add a function to your class that predicts the next k states given a current state estimate but in the absence of observations. Do so by implementing the following function:

```
def predict(self,x,k):
    """
    Predict the next k states in the absence of observations.

    Parameters
    -----
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of states to predict.

    Returns
    -----
    out : ndarray of shape (n,k)
        The next k predicted states.
    """
    pass
```

We can use this prediction routine to estimate where the projectile will hit the surface.

Problem 7. Using the final state estimate $\hat{\mathbf{x}}_{800}$ that you obtained in Problem 5, predict the future states of the projectile until it hits the ground. Predicting approximately the next 450 states should be sufficient.

Plot the actual state sequence together with the predicted state sequence (as a yellow curve), and observe how near the prediction is to the actual point of impact. Your results should be similar to those shown in Figure 1.3. The following code will help you plot both the

actual and predicted state sequences together in a nice plot.

```
>>> s,o = kal.evolve(x0,1250)
>>> predicted = kal.predict(estimates[:,-1],450)

>>> # create a dynamic xlim to always plot both sequences together
>>> x1 = s[0,:][np.where(s[1,:]>=0)][-1]
>>> x2 = predicted[0,:][np.where(predicted[1,:]>=0)][-1]
>>> plt.xlim(min(x1,x2)-50, max(x1,x2)+50)
```

In the absence of observations, we can also reverse the system and iterate backward in time to infer information about states of the system prior to measured observations. The system is reversed by

$$\mathbf{x}_k = F^{-1}(\mathbf{x}_{k+1} - \mathbf{u} - \boldsymbol{\varepsilon}_{k+1}).$$

Considering again that $\mathbb{E}[\boldsymbol{\varepsilon}_k] = 0$ at any time k , we can ignore this term, simplifying the recursive estimation backward in time.

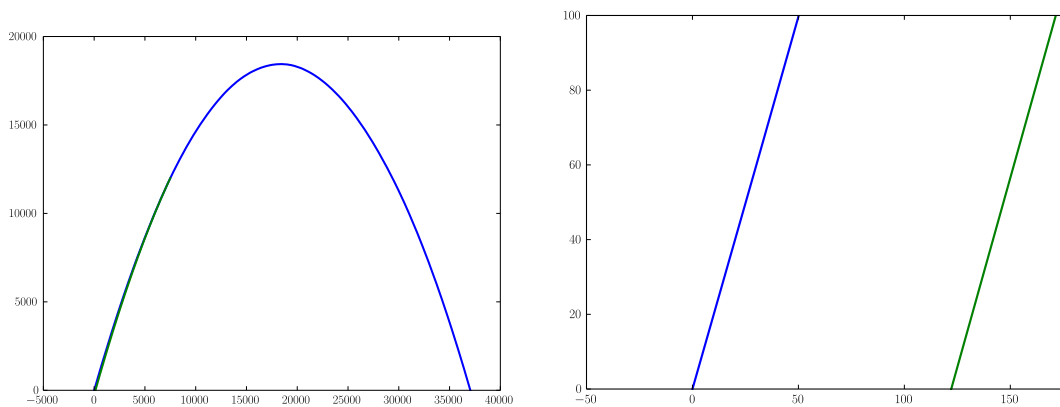


Figure 1.4: Predicted vs. actual point of origin (detailed view on right).

Problem 8. Add a function to your class that rewinds the system from a given state estimate, returning predictions for the previous states. Do so by implementing the following function:

```
def rewind(self,x,k):
    """
    Predict the k states preceding the current state estimate x.

    Parameters
    -----
    x : ndarray of shape (n,)
        The current state estimate.
    k : integer
        The number of preceding states to predict.
```



```

Returns
-----
out : ndarray of shape (n,k)
      The k preceding predicted states.
"""
pass

```

Returning to the projectile example, we can now predict the point of origin.

Problem 9. Using your state estimate $\hat{\mathbf{x}}_{250}$, predict the point of origin of the projectile along with all states leading up to time step 250. Note that you may have to take a few extra time steps to predict the point of origin. (The point of origin is the first point along the trajectory where the y coordinate is 0.) Plot these predicted states (in green) together with the original state sequence. Zoom in to see how accurate your prediction is. Your plots should be similar to Figure 1.4.

Repeat the prediction starting with $\hat{\mathbf{x}}_{600}$. Compare to the previous results. Which is better? Why?