PH6130 - Data Science Analysis

Kalman Filter and its Applications

Sahukari Chaitanya Varun

EE19BTECH11040

Pakala Srijith Reddy

EE19BTECH11041

2nd May, 2023

Contents

Chapter 1

Theoretical Perspective

Kalman filtering, Li et al. 2015, is a state estimation technique invented in 1960 by Rudolf E. Kálmán. Because of its ability to extract useful information from noisy data and its small computational and memory requirements, it is used in many application areas, including spacecraft navigation, motion planning in robotics, signal processing, and wireless sensor networks. This algorithm is especially useful when one has some uncertain information about some linear dynamic system and has to give an educated guess about its future based on previous states and observations.

The need for such an algorithm to extract data from noise stems from the fact that the traditional Wiener filters require prior information about the statistics of data. Adaptive filters were then used to counteract the problem. These filters recursively adjust themselves to the statistics of the data. They start from an arbitrary initial condition and update iteratively. Estimation of the filter statistics is built into the filtering process. In nonstationary conditions, the adaptive filter tracks the statistics over time. The specialty of the Kalman filter algorithm from other traditional adaptive filters is that it considers the constraints on the production model. The variation of statistics adhering to a specific time-varying form can be captured by this algorithm for a polished estimate.

To understand the algorithm better, we illustrate the proof of working for the Kalman Filter. The proof stems from the basics of linear algebra and statistical theory for random processes.

1.1 Proof of Working

A typical linear dynamical system is given by

$$\sum_{k=0}^{M} a_n[k]x[n-k] = \sum_{k=0}^{N} b_n[k]u[n-k] + v_p[n] \tag{1.1}$$

$$\implies x[n] = -\sum_{k=1}^{M} a_n[k]x[n-k] + \sum_{k=0}^{N} b_n[k]u[n-k] + v_p[n] \tag{1.2}$$

$$\implies x[n] = \mathbf{a^T x[n-1]} + \mathbf{b_n u[n]} + \mathbf{v_p[n]} \tag{1.3}$$

$$\implies x[n] = \mathbf{A_n x[n-1]} + \mathbf{B_n u_n} + \mathbf{v_p[n]} \tag{1.4}$$

The equation 1.4 is referred as the Production model. The terms are defined as

- $\mathbf{x_n}$: State Matrix
- $\mathbf{A_n}$: Transition Matrix
- $\mathbf{u_n}$: Control Inputs
- $\mathbf{B_n}$: Input Control Matrix
- $\mathbf{v_p[n]}$: Channel/Production Noise $\sim \mathcal{N}(0, \mathbf{\Sigma_p})$

Similarly the Measurement model is given by

$$\mathbf{y_n} = \mathbf{H_n x_n} + \mathbf{v_m[n]} \tag{1.5}$$

where,

- $\mathbf{y_n}$: Measurement Vector
- $\mathbf{H_n}$: Transformation Matrix
- $\mathbf{v_m[n]}$: Measurement Noise $\sim \mathcal{N}(0, \mathbf{\Sigma_m})$
- Assume $\mathbf{v}_m \perp \mathbf{v}_p$

The complete end-to-end diagram of the Kalman Estimation problem is as shown in figure.1.1.

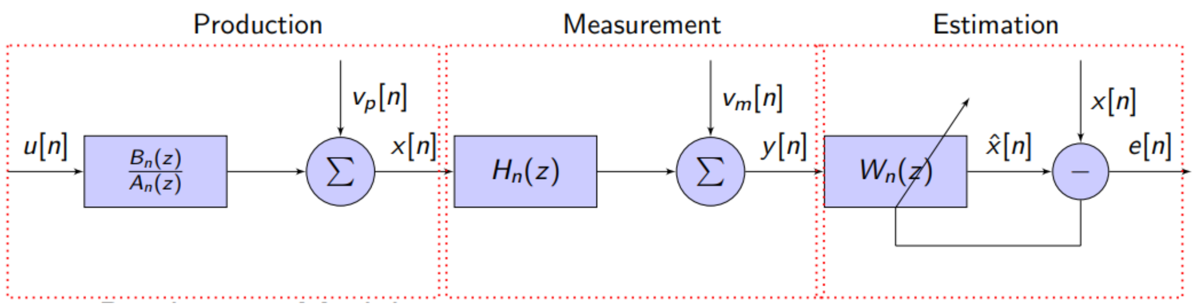


Figure 1.1: Basis of Kalman Filtering

Assuming the problem is to find the estimates ignoring the control parameters, the prediction model is given by

$$x[n] = \mathbf{A}[n]\mathbf{x}[n-1] + v_p[n] \tag{1.6}$$

Hence, the apriori estimate, i.e, the prediction for the future state based on the previous posterior estimate (previous observation included), is given by

$$\hat{x}[n/n-1] = \mathbf{A}[n]\hat{\mathbf{x}}[n-1/n-1] \tag{1.7}$$

And the corresponding posterior estimate is given by

$$\hat{\mathbf{x}}[n/n] = \mathbf{K}'[n]\hat{\mathbf{x}}[n/n-1] + \mathbf{K}[n]\hat{\mathbf{y}}[n] \tag{1.8}$$

where $\mathbf{K}$ and $\mathbf{K}'$ correspond to the Kalman gain. To find these gains, we minimize the Mean Square Estimate (MSE) of the a posteriori error given by

$$\mathbf{e}[n/n] = \mathbf{x}[n] - \hat{\mathbf{x}}[n/n] \tag{1.9}$$

$$\mathbf{J}[n] = \mathbb{E}[||\mathbf{e}[n/n]||^2] \tag{1.10}$$

The relation between $\mathbf{K}$ and $\mathbf{K}'$ can be found as follows,

$$\begin{aligned}
\mathbf{e}[n/n] &= \mathbf{x}[n] - \hat{\mathbf{x}}[n/n] \\
&= \mathbf{x}[n] - \mathbf{K}'[n]\hat{\mathbf{x}}[n/n-1] - \mathbf{K}[n]\mathbf{y}[n] \\
&= \mathbf{x}[n] - \mathbf{K}'[n](\mathbf{x}[n] - \mathbf{e}[n/n-1]) - \mathbf{K}[n]\left(\mathbf{H}[n]\mathbf{x}[n] + \mathbf{v}_m[n]\right) \\
&= \left(\mathbf{I} - \mathbf{K}'[n] - \mathbf{K}[n]\mathbf{H}[n]\right)\mathbf{x}[n] + \mathbf{K}'[n]\mathbf{e}[n/n-1] - \mathbf{K}[n]\mathbf{v}_m[n]
\end{aligned} \tag{1.11}$$

For unbiased estimation of state vector, $\mathbb{E}[\mathbf{e}[n/n]] = 0$. Hence

$$\mathbf{K}'[n] = \mathbf{I} - \mathbf{K}[n]\mathbf{H}[n] \tag{1.12}$$

The a posteriori estimate of the state is given by

$$\mathbf{x}[n/n] = \hat{\mathbf{x}}[n/n-1] + \mathbf{K}[n](\mathbf{y}[n] - \mathbf{H}[n]\hat{\mathbf{x}}[n/n-1]) \tag{1.13}$$

The posterior error estimate becomes

$$\mathbf{e}[n/n] = (\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])\mathbf{e}[n/n-1] - \mathbf{K}[n]\mathbf{v}_m[n] \tag{1.14}$$

The posterior covariance is given by

$$\begin{aligned}\mathbf{P}[n/n] &= \mathbb{E}\left[\mathbf{e}[n]\mathbf{e}^\top[n]\right] \\ &= (\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])\mathbf{P}[n/n-1](\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])^\top + \mathbf{K}[n]\mathbf{\Sigma}_m\mathbf{K}^\top[n]\end{aligned} \tag{1.15}$$

The corresponding loss estimate can be written as

$$\begin{aligned}J[n] &= \mathbb{E}\left[\|\mathbf{e}[n/n]\|^2\right] = \mathbb{E}\left[\mathbf{e}^\top[n]\mathbf{e}[n]\right] = \mathrm{Tr}\{\mathbf{P}[n/n]\} \\ \nabla_{\mathbf{K}} J[n] &= -2(\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])\mathbf{P}[n/n-1]\mathbf{H}^\top[n] + 2\mathbf{K}[n]\mathbf{\Sigma}_m = \mathbf{0}\end{aligned} \tag{1.16}$$

The Kalman gain is given by

$$\mathbf{K}[n] = \mathbf{P}[n/n-1]\mathbf{H}^\top[n]\left[\mathbf{H}[n]\mathbf{P}[n/n-1]\mathbf{H}^\top[n] + \mathbf{\Sigma}_m\right]^{-1} \tag{1.17}$$

where the posterior covariance turns out to be

$$\mathbf{P}[n/n] = (\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])\mathbf{P}[n/n-1] \tag{1.18}$$

1.2 Implementing Kalman Filter

Kalman Filter is an iterative algorithm. For any problem, to implement the Kalman filter, we have to first find the observations, state matrix, transformation matrix, the covariances (i.e., confidence) in production and measurement. The implementation of Kalman filter happens as follows:

- Observation vectors: $(\mathbf{y}[1], \mathbf{y}[2], \ldots, \mathbf{y}[n], \ldots)$
- Known parameters: $\mathbf{A}[n], \mathbf{H}[n], \boldsymbol{\Sigma_p}, \boldsymbol{\Sigma_m}, \mathbf{x}[0]$
- Initial conditions: $\hat{\mathbf{x}}[0/0] = \mathbb{E}[\mathbf{x}[0]] \quad \mathbf{P}[0/0] = \mathbb{E}\left[\mathbf{e}[0/0]\mathbf{e}^\top[0/0]\right]$
- Iterations: For $n = 1, 2, 3, \cdots$

$$\begin{aligned}
\hat{\mathbf{x}}[n/n-1] &= \mathbf{A}[n]\hat{\mathbf{x}}[n-1/n-1] \\
\mathbf{P}[n/n-1] &= \mathbf{A}[n]\mathbf{P}[n-1/n-1]\mathbf{A}^\top[n] + \boldsymbol{\Sigma_p} \\
\mathbf{K}[n] &= \mathbf{P}[n/n-1]\mathbf{H}^\top[n]\left(\mathbf{H}[n]\mathbf{P}[n/n-1]\mathbf{H}^\top[n] + \boldsymbol{\Sigma}_m\right)^{-1} \\
\hat{\mathbf{x}}[n/n] &= \hat{\mathbf{x}}[n/n-1] + \mathbf{K}[n](\mathbf{y}[n] - \mathbf{H}[n]\hat{\mathbf{x}}[n/n-1]) \\
\mathbf{P}[n/n] &= (\mathbf{I} - \mathbf{K}[n]\mathbf{H}[n])\mathbf{P}[n/n-1]
\end{aligned}$$

- For stationary process, as $n \to \infty, \mathbf{P}[n/n-1] \to \mathbf{P}$ (steady-state).

$$\mathbf{P} = \mathbf{APA}^\top + \boldsymbol{\Sigma_p} - \mathbf{APH}^\top\left(\mathbf{HPHH}^\top + \boldsymbol{\Sigma_m}\right)^{-1}\mathbf{HPA}^\top$$

- The Kalman gain converges to

$$\mathbf{K}[n] \to \mathbf{PH}^\top\left(\mathbf{HPH}^\top + \boldsymbol{\Sigma_m}\right)^{-1} \tag{1.19}$$

- With the covariance converging to

$$\mathbf{P}[n/n] \to (\mathbf{I} - \mathbf{KH})\mathbf{P} \tag{1.20}$$

This procedure is followed for the estimation of signal from noise or predict the future states from the given observations. Kalman filter is only suitable for linear systems, and requires that the observation equation is linear. Most of the practical applications are nonlinear systems; therefore the research of nonlinear filter is very important. There are different variants of Kalman filter algorithms to tackle this challenge such as *Extended Kalman Filter.* Similarly *Unscented Kalman Filter* is used for dealing with heavily non-uniform distributions. In the coming chapter, we illustrates its general applications in speech, image and navigation problems.

Chapter 2

General Applications

2.1 Speech Enhancement

Speech enhancement, Das 2016, removes noise from corrupted Speech and has applications in cellular and radio communication, voice-controlled devices, and a preprocessing step in automatic speech/speaker recognition. Speech can be modeled as the output of a linear time-varying filter, excited by either quasi-periodic pulses or noise. Figure 2.1 gives a schematic of the speech production model.

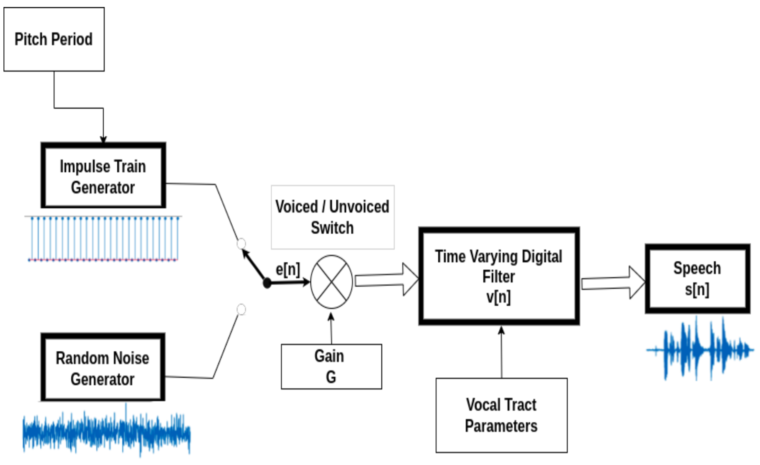


Figure 2.1: Speech Production System

This is often referred to as the auto-regressive model of Speech, where the present sample, call it *x[k]*, depends on the linear combination of the past p samples added with stochastic noise.

$$x[k] = -\sum_{i=1}^{p} a_i x[k-i] + u[k] \tag{2.1}$$

The coefficients a_i are the linear prediction coefficients (LPCs), and *u[k]* is the process noise. From the Yule-Walker equations, these coefficients can be found from the autocorrelation function, R_{xx} as

$$a = -R^{-1}r \tag{2.2}$$

where

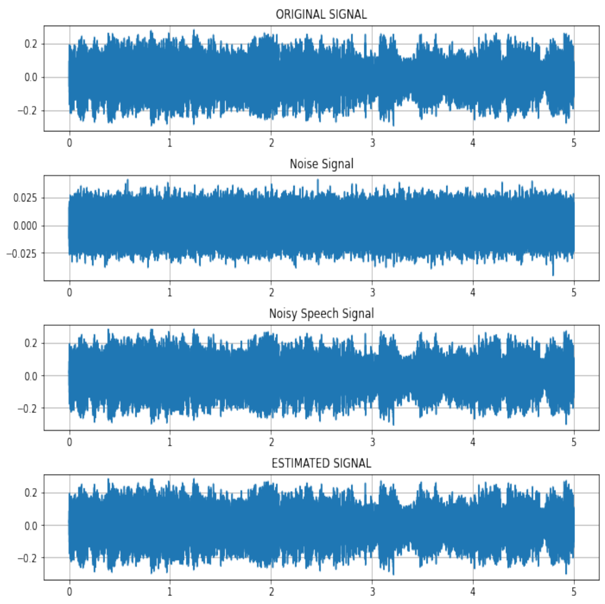
$$R = \begin{bmatrix} R_{xx}(0) & R_{xx}(-1) & ... & R_{xx}(1-p) \\ R_{xx}(-1) & R_{xx}(0) & ... & R_{xx}(2-p) \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ R_{xx}(p-1) & R_{xx}(p-2) & ... & R_{xx}(0) \end{bmatrix} \tag{2.3}$$

and

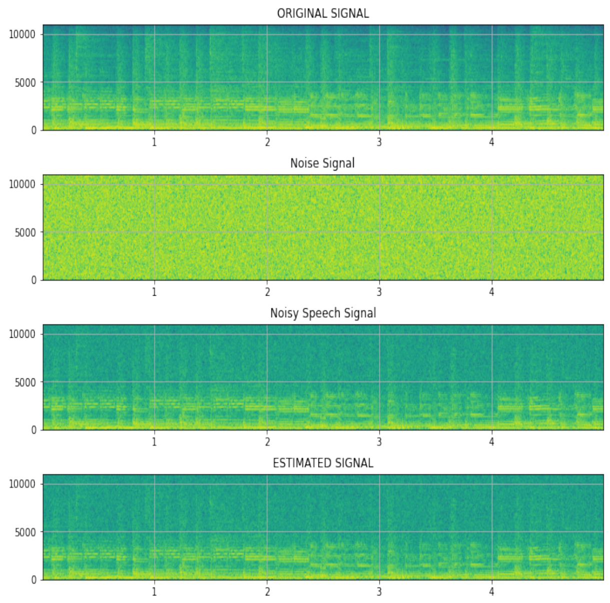
$$r = [R_{xx}(1-p), R_{xx}(2-p), ..., R_{xx}(0)]^T \tag{2.4}$$

From these relations, we can find the state vector matrix, the state transition matrix, and the observation matrix. Estimating measurement noise covariance and the process/production covariance matrix is a separate study. Based on the *Power Spectral Density (PSD)* and *Zero Crossing Rate (ZCR)*, the voiced and unvoiced regions are found, which helps to find the spectral flatness. The mean of variances over the silence regions is the measurement noise covariance. The production noise covariance estimation is a much more difficult task. The compromise value of Q or process covariance is based on a few sensitivity and robustness metrics. Q is generally varied for silent and non-silent regions for better estimation. The model order for the auto-regressive model is a practical choice one makes based on the data. We take a speech sample and add an additive white Gaussian noise with known variance to understand how Kalman filters in speech enhancement. We get the following results by taking the model order to be around 3 and iterating over the speech sample updating the Kalman gain and filter coefficients.

The iteration procedure is as followed as shown in Section 1.2. The corresponding python code is followed right below the results.

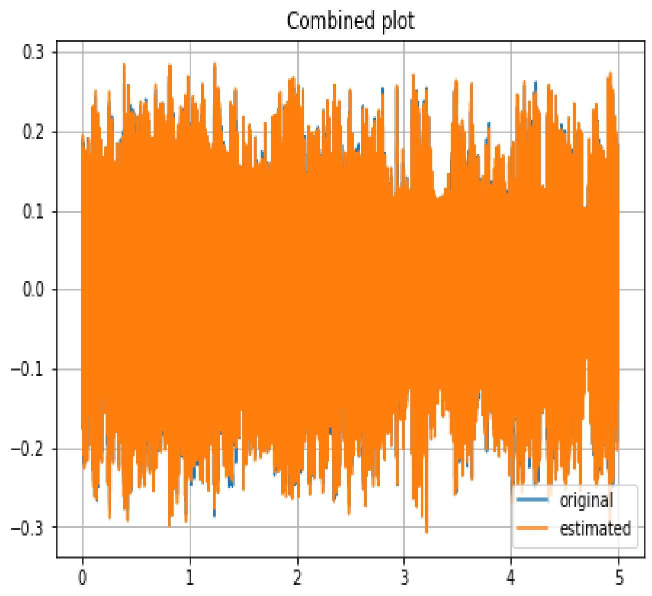


(a) Time Signal



(b) Spectograms, green color represents the magnitude

Figure 2.2: Results Observed



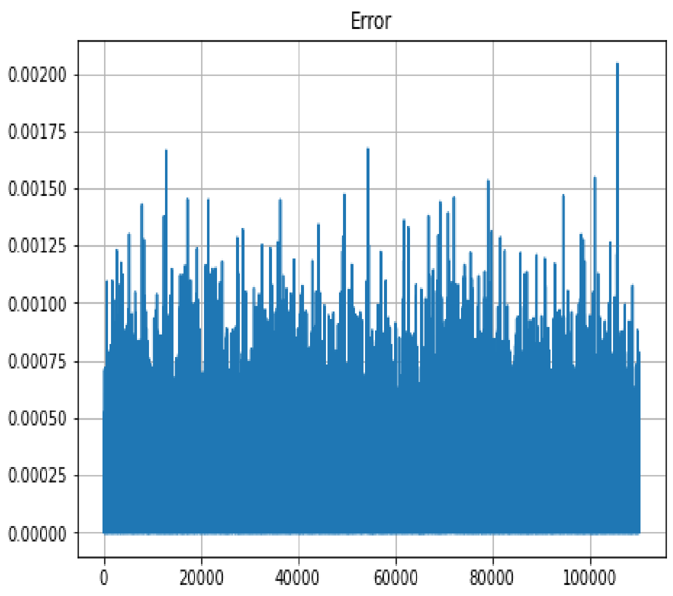


Figure 2.3: Comparing the Estimated Result

The observed MSE comes to be around 9.96×10^{-5}. It shows that the error is considerably low. On listening to the signal, one can observe the removal of white noise. As speech is a complicated signal and involves high randomness, we can observe that iterations errors at the end seem to be on par with the beginning. The code for reproducing the results is as follows.

Listing 2.1: SpeechEnhancement

```
import numpy as np
import librosa
import scipy.io.wavfile as wav
# Close all figures
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

file_name = 'input.wav'
# Read input wav file
x, fs = librosa.load(file_name)
t = np.arange(len(x))*(1/fs)

# Add random Gaussian noise
STD_n= 0.01
v=np.random.normal(0, STD_n, x.shape[0])

# Noisy Speech Signal
orig = x.astype('float64') + v.astype('float64')
noised_signal = orig.copy()

# Initialize variables
N = len(x) #length of the input signal

F = np.zeros((5, N)).astype('float64') #initialization of standard transition
    matrix
I = np.eye(5).astype('float64')
H = np.zeros((5, N)).astype('float64') #Transformation Matrix
sig = np.zeros((5, 5*N)).astype('float64') # priori or posteri covariance
    matrix.
K = np.zeros((5, N)).astype('float64') #kalman gain.
XX = np.zeros((5, N)).astype('float64') #kalman coefficient for yy.
y = np.zeros((N,1)).astype('float64') #desired signal
vv = np.zeros((N,1)).astype('float64') #predicted state error vector
yy = np.zeros((N,1)).astype('float64') #Estimated error sequence

Q = 1e-6 * np.eye(5).astype('float64') #Process Noise Covariance.
R = 0.1 #Measurement Noise Covariance

y[:N,:] = x.astype('float64').reshape((-1,1)) #Measurement Noise Covariance
sig[:5, :5] = 0.1 * I

for k in range(6, N):
    F[0:5, k] = -np.array([y[k-1], y[k-2], y[k-3], y[k-4], y[k-5]]).reshape(F
    [0:5, k].shape)
    H[0:5, k] = -np.array([yy[k-1], yy[k-2], yy[k-3], yy[k-4], yy[k-5]]).
    reshape(H[0:5, k].shape)
    K[0:5, k] = sig[0:5, 5*k-30:5*k-25]@F[0:5, k]/(F[0:5, k].T @ sig[0:5, 5*k
    -30:5*k-25] @ F[0:5, k] + R) # Kalman Gain
    sig[0:5, 5*k-24:5*k-19] = sig[0:5, 5*k-30:5*k-25] - K[0:5, k] @ (F[0:5, k].
    T @ sig[0:5, 5*k-30:5*k-25]) + Q # error covariance matrix

    XX[0:5, k] = (I - K[0:5, k] @ F[0:5, k].T) @ XX[0:5, k-1] + (K[0:5, k] * y[
    k]) # posteriori value of estimate X(k)
    orig_k = y[k] - (F[0:3, k].T @ XX[0:3, k]) # estimated speech signal
    yy[k] = (H[0:5, k].T @ XX[0:5, k]) + orig_k # no. of coefficients per
    iteration
# Calculate mean squared error
Er = orig - x
# print(Er)
```

```
MSE_kalman = np.sum(Er ** 2) / len(Er)
MSE_kalman


# Plot signals
tt = np.arange(len(x))

fig, axs = plt.subplots(4, 1,figsize=(10,8))
fig.tight_layout(pad=2)

axs[0].plot(t,x)
axs[0].set_title('ORIGINAL SIGNAL')
axs[0].grid()
axs[1].plot(t,v)
axs[1].set_title('Noise Signal')
axs[1].grid()
axs[2].plot(t,noised_signal)
axs[2].set_title('Noisy Speech Signal')
axs[2].grid()
axs[3].plot(t,orig)
axs[3].set_title('ESTIMATED SIGNAL')
axs[3].grid()
plt.show()


fig, axs = plt.subplots(4, 1,figsize=(10,8))
fig.tight_layout(pad=2)

axs[0].specgram(x,Fs=fs)
axs[0].set_title('ORIGINAL SIGNAL')
axs[0].grid()
axs[1].specgram(v,Fs=fs)
axs[1].set_title('Noise Signal')
axs[1].grid()
axs[2].specgram(noised_signal,Fs=fs)
axs[2].set_title('Noisy Speech Signal')
axs[2].grid()
axs[3].specgram(orig,Fs=fs)
axs[3].set_title('ESTIMATED SIGNAL')
axs[3].grid()
plt.show()


plt.figure(figsize=(16,5))
plt.subplot(1,2,1)
plt.plot(t, x, t, orig)
plt.title('Combined plot')
plt.legend(['original', 'estimated'])
plt.grid()
plt.subplot(1,2,2)
plt.plot(Er**2)
plt.title('Error')
plt.grid()
plt.show()
```

2.2 Video Stabilization

Kalman filter can also be used to remove the jitter in the video caused by the irregular camera motion, as studied in the work of Yu and Zhang 2014. The general stabilization algorithms used to address this problem include the gray projection algorithm, block matching algorithm, phase correlation algorithm, corner matching algorithm, and optical flow techniques. The underlying principle for these algorithms is detailed as shown in the figure.2.4.

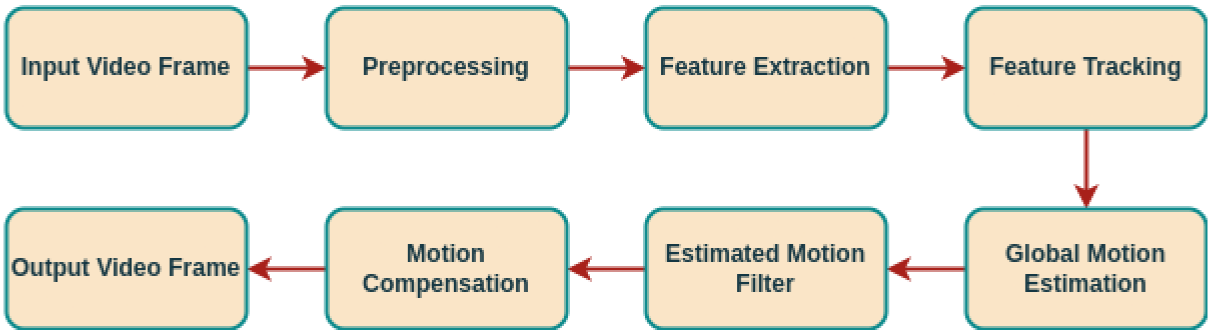


Figure 2.4: Generic Video Stabilization System

The preprocessing is done before tracking to avoid the noise points, if present any. This is achieved by passing the video frame through a Gaussian filter. The above-referred algorithms use different features to track, each having its own benefits and limitations. In the present illustration, we will use the *cv2.goodFeaturesToTrack* function, which is the construction function for the Harris corner extraction algorithm with an enhanced scoring function suggested by Shi-Tomasi that finds N strongest corners in the image. We use a matching algorithm called the Diamond Search algorithm to track these points. The tracked point's motion is estimated by calculating the optical flow with the Lucas-Kanade method. Using an inverse transformation, we estimate a filter that relates the points in the previous frame to the current frame. The Kalman filtering is used to identify the adjustments needed to this filter coefficient to track the positioning of the frame. The analysis suggests that the Kalman filter acts like a low-pass filter, which silences the sudden high-frequency jittered motion in the video. For the current problem, the Kalman filtering is applied as follows. The standard notation used in chapter one is retained. The aim of the Kalman filter is to regain X_k when A, H, the covariance matrix of process noise Q, and the covariance matrix of observation noise R are known. In this video stabilization system, set the position of center point of the first frame coordinate origin, assume the position of the center point of each frame without jitter is (x_k, y_k), corresponding observed

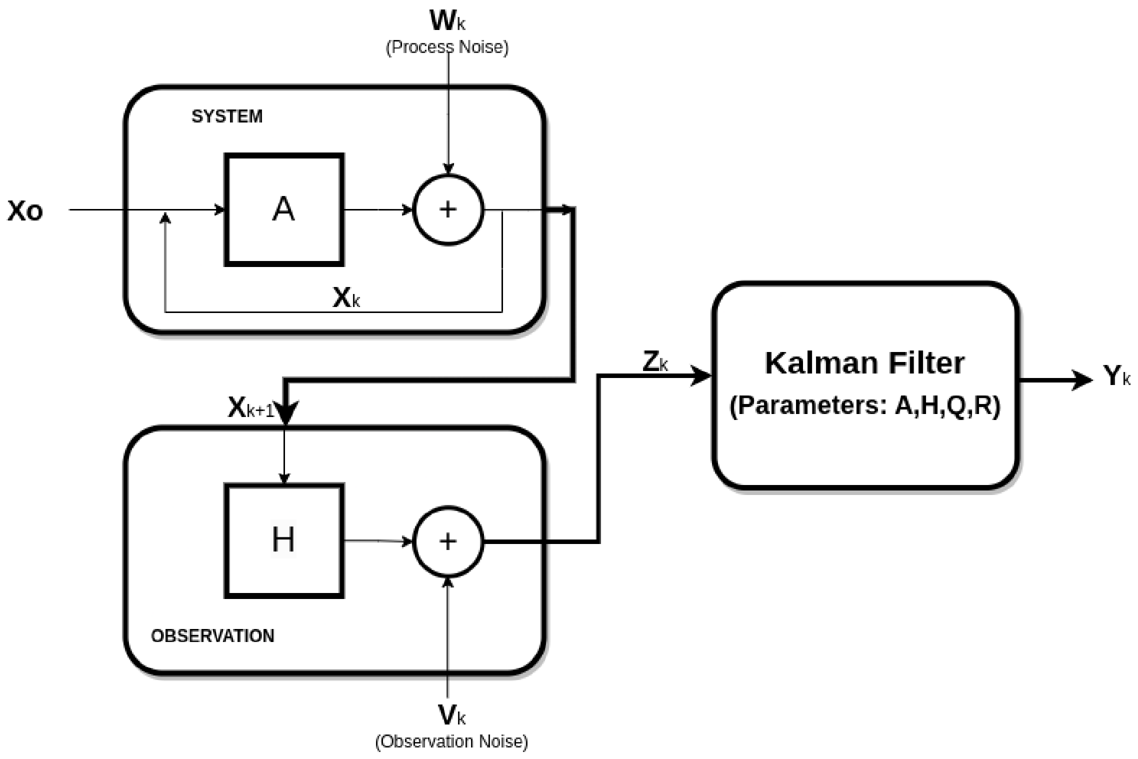


Figure 2.5: Kalman Filtering in Video Stabilization System

value is (x_{ok}, y_{ok}), with variation rate of *(dx,dy)*. The state equation can be written as

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ dx \\ dy \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ dx \\ dy \end{bmatrix} + W_k \tag{2.5}$$

$$\begin{bmatrix} x_{ok} \\ y_{ok} \\ dx \\ dy \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \\ dx \\ dy \end{bmatrix} + V_k \tag{2.6}$$

Hence, we can get the transfer matrix A and observation matrix H. The process noise Q and R can be set reasonably. We take the process variance default to 0.1 and measurement variance close to 2. As per the iteration procedure shown in Section 1.2, we get the following results.

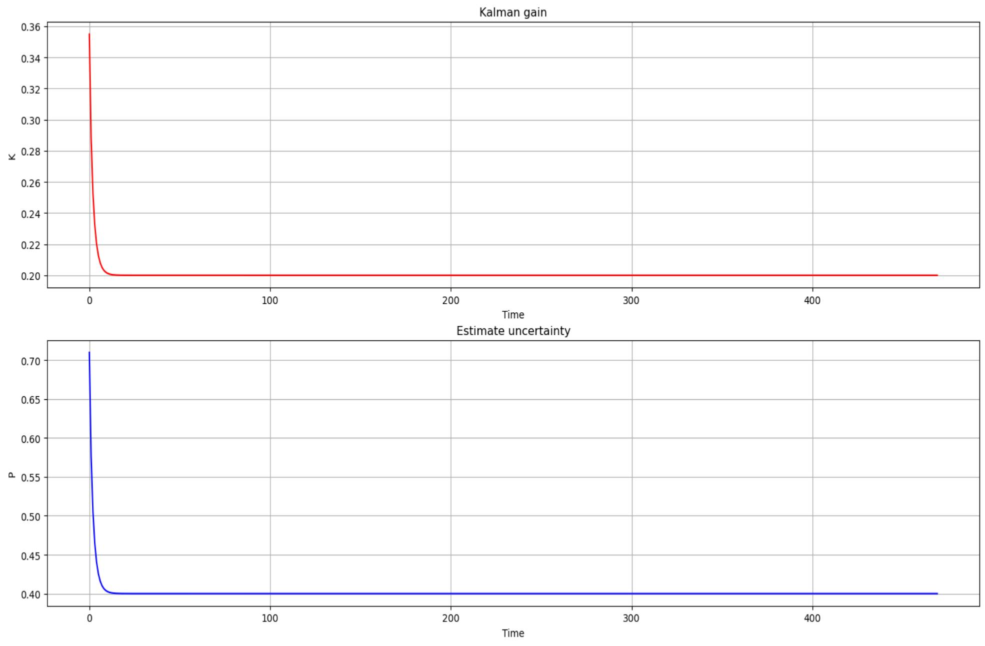


Figure 2.6: Video Stabilization Kalman Results

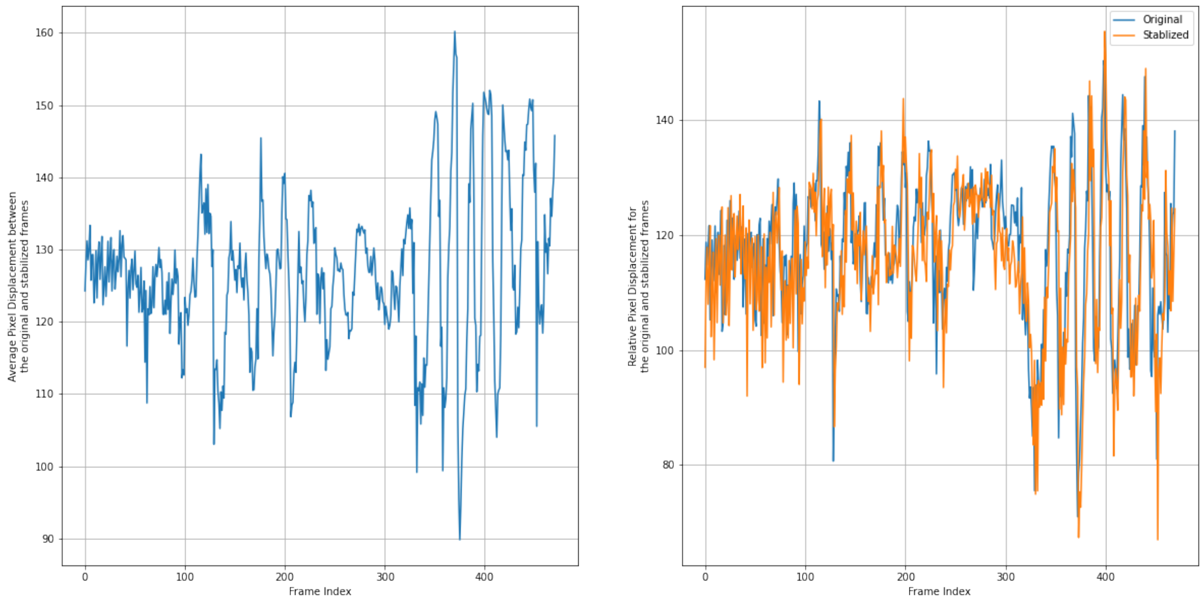


Figure 2.7: Video Stabilization Kalman Metrics of Study

It can be observed that the initial transformation doesn't deform the initial sequence but is later adjusted to impact the video frame. The relative displacement plots show that the flow is much smoother for the Kalman stabilized video compared to the original video. The corresponding code to replicate the results is as follows:

Listing 2.2: SpeechEnhancement

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

class VideoStabilizer:
  def __init__(self, source, *, size=(640,480), processVar=0.1, measVar=2):
    self.stream = source
    self.frameSize = size
    self.count = 0

    self.a = 0
    self.x = 0
    self.y = 0

    self.processVar = processVar
    self.measVar = measVar

    self.Q = np.array([[self.processVar]*3])
    self.R = np.array([[self.measVar]*3])

    grab, frame = self.stream.read()
    if not grab:
      print("[VideoStabilizer] No frame is captured. Exit")
      exit(1)

    self.prevFrame = cv2.resize(frame, self.frameSize)
    self.prevGray = cv2.cvtColor(self.prevFrame, cv2.COLOR_BGR2GRAY)
    self.lastRigidTransform = None

    self.K_collect = []
    self.P_collect = []


  def read(self):
    grab, frame = self.stream.read()
    if not grab:
      print("[VideoStabilizer] No frame is captured.")
      return False, None, None

    currentFrame = cv2.resize(frame, self.frameSize)
    currentGray = cv2.cvtColor(currentFrame, cv2.COLOR_BGR2GRAY)

    self.prevPoints = cv2.goodFeaturesToTrack(self.prevGray,
                      maxCorners=200,
                      qualityLevel=0.01,
                      minDistance=30,
                      blockSize=3)

    currentPoints, status, err = cv2.calcOpticalFlowPyrLK(self.prevGray,
                            currentGray,
                            self.prevPoints,
                            None)

    assert self.prevPoints.shape == currentPoints.shape

    idx = np.where(status == 1)[0]
    self.prevPoints = self.prevPoints[idx]
    currentPoints = currentPoints[idx]

    m, inliers = cv2.estimateAffinePartial2D(self.prevPoints, currentPoints)
    if m is None:
      m = self.lastRigidTransform

    dx = m[0, 2]
    dy = m[1, 2]

    da = np.arctan2(m[1, 0], m[0, 0])

    self.x += dx
    self.y += dy
    self.a += da

    Z = np.array([[self.x, self.y, self.a]], dtype="float")

    if self.count == 0:
      # initialization
      self.X_estimate = np.zeros((1,3), dtype="float")
      self.P_estimate = np.ones((1,3), dtype="float")
    else:
      # extrapolation
      X_predict = self.X_estimate
      P_predict = self.P_estimate + self.Q

      # update state
      K = P_predict / (P_predict + self.R)
      self.X_estimate = X_predict + K * (Z - X_predict)
      self.P_estimate = (np.ones((1,3), dtype="float") - K) * P_predict

      self.K_collect.append(K)
      self.P_collect.append(self.P_estimate)

    diff_x = self.X_estimate[0,0] - self.x
    diff_y = self.X_estimate[0,1] - self.y
    diff_a = self.X_estimate[0,2] - self.a

    dx += diff_x
    dy += diff_y
    da += diff_a

    m = np.zeros((2,3), dtype="float")
    m[0,0] = np.cos(da)
    m[0,1] = -np.sin(da)
    m[1,0] = np.sin(da)
    m[1,1] = np.cos(da)
    m[0,2] = dx
    m[1,2] = dy

    frame_stabilized = cv2.warpAffine(self.prevFrame, m, self.frameSize)
    frame_stabilized = self.fixBorder(frame_stabilized)

    self.prevGray = currentGray
    self.prevFrame = currentFrame
    self.lastRigidTransform = m

    self.count += 1

    return True, currentFrame, frame_stabilized


  def fixBorder(self, frame):
    s = frame.shape
    # Scale the image 10% without moving the center
    T = cv2.getRotationMatrix2D((s[1]/2, s[0]/2), 0, 1.1)
    frame = cv2.warpAffine(frame, T, (s[1], s[0]))
    return frame


  def showGraph(self):
    self.K_collect = np.array(self.K_collect)
    self.P_collect = np.array(self.P_collect)
    plt.figure(figsize=(20,10))
    plt.subplot(2,1,1)
    plt.grid()
    plt.xlabel("Time")
    plt.ylabel("K")
    plt.plot(range(self.K_collect.shape[0]), self.K_collect[...,0], color='r')
    plt.title("Kalman gain")
    plt.subplot(2,1,2)
    plt.grid()
    plt.plot(range(self.P_collect.shape[0]), self.P_collect[...,2], color='b')
    plt.title("Estimate uncertainty")
    plt.ylabel("P")
    plt.xlabel("Time")
    plt.show()
            # Metric Plots
            # print(original_frames[0].shape,stable_frames[0].shape)
            difference_frames = [original_frames[i]-stable_frames[i] for i in
    range(len(original_frames)-1)]
            diff = np.zeros((len(difference_frames),1))
            for i in range(len(original_frames)-1):
                diff[i] = np.average(np.abs(difference_frames[i]))

            relative_displacement_orig = [original_frames[i+1]-original_frames[
    i] for i in range(len(original_frames)-2)]
            relative_displacement_stable = [stable_frames[i+1]-stable_frames[i]
     for i in range(len(stable_frames)-2)]
            disp_ori = []
            disp_stable = []

            for i in range(len(relative_displacement_orig)):
                disp_ori.append(np.average(np.abs(relative_displacement_orig[i
    ])))
                disp_stable.append(np.average(np.abs(
    relative_displacement_stable[i])))

            plt.figure(figsize=(20,10))
            plt.subplot(1,2,1)
            plt.plot(np.arange(len(diff)),diff)
            plt.xlabel("Frame Index")
            plt.ylabel("Average Pixel Displacement between\nthe original and
    stabilized frames")
            plt.grid()
            plt.subplot(1,2,2)
            plt.plot(np.arange(len(disp_ori)),disp_ori,label='Original')
            plt.plot(np.arange(len(disp_stable)),disp_stable,label='Stablized')
            plt.xlabel("Frame Index")
            plt.ylabel("Relative Pixel Displacement for\nthe original and
    stabilized frames")
            plt.grid()
            plt.legend()
            plt.show()


if __name__ == "__main__":
  video = cv2.VideoCapture("./Running/1.avi")
  stabilizer = VideoStabilizer(video)
```

```
while True:
  success, _, frame = stabilizer.read()
  if not success:
    print("No frame is captured.")
    break

  cv2.imshow("frame", frame)

  if cv2.waitKey(20) == 27:
    break
print("showing Graph!")
stabilizer.showGraph()
```

The dataset is taken from the open-sourced GitHub repo, which can be found at `https://alex04072000.github.io/FuSta/index.html`. And the base source for the code can be found at `https://github.com/trongphuongpro/videostabilizer`. As the Kalman filter relies only on the past frame for processing, it is often convenient to implement it in real time. For better processing, Least Squares Matching Methods, Convolutional Networks, etc, can be used for deployment.

2.3 Vehicle Tracking

With increasing congestion on urban roads, authorities need better real-time traffic information to manage traffic as studied in the work of Aydos et al. 2009. Kalman Filters are efficient algorithms that can be adapted to track vehicles in urban traffic given noisy sensor data. A Kalman Filter process model that approximates dynamic vehicle behavior is a reusable subsystem for modeling the dynamics of a multi-vehicle traffic system. We illustrate the basic Kalman path recovery from the assumed Gaussian noise on the sensor data. We consider a state vector $x_t \in \mathcal{R}^4$, where (x_{t0}, x_{t1}) represent the position of the vehicle in two dimensions and (x_{t2}, x_{t3}) represent the vehicle velocity correspondingly. Let the driving force be denoted as w_t and the observed noisy measurements for the vehicle's position be denoted as $y_t \in \mathcal{R}^4$.

The dynamics of the system are captured as follows

- For the state vector, let x_t, v_t, and w_t be the vehicle's position, velocity, and input drive force at a certain instant.
- The resulting acceleration of the vehicle is

$$a_t = w_t - \gamma v_t \tag{2.7}$$

where $-\gamma v_t$ is the damping term with parameter γ.

- The discretized dynamics are then obtained by numerically integrating as

$$x_{t+1} = x_t + \left(1 - \frac{\gamma \Delta t}{2}\right) v_t \Delta t + \frac{1}{2} w_t \Delta t^2 \tag{2.8}$$

$$v_{t+1} = (1-\gamma) v_t + w_t \Delta t \tag{2.9}$$

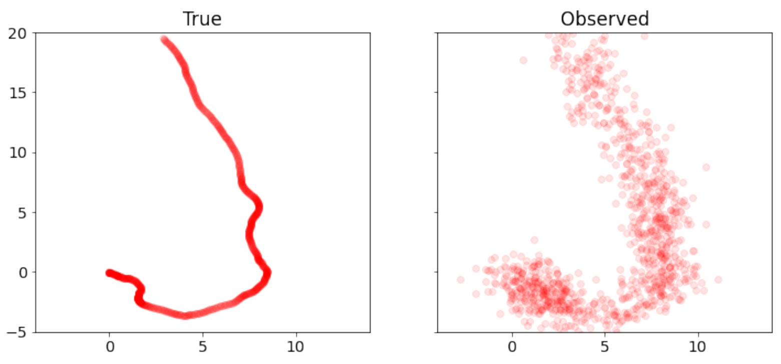
Based on the dynamics, extrapolating to the two dimensions we get the matrices to be

$$A = \begin{bmatrix} 1 & 0 & (1 - \frac{\gamma}{2}\Delta t)\Delta t & 0 \\ 0 & 1 & 0 & (1 - \frac{\gamma}{2}\Delta t)\Delta t \\ 0 & 0 & 1 - \gamma \Delta t & 0 \\ 0 & 0 & 0 & 1 - \gamma \Delta t \end{bmatrix} \tag{2.10}$$

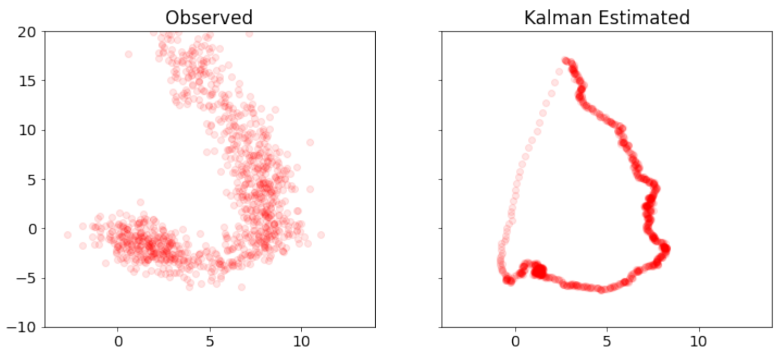
$$B = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \tag{2.11}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.12}$$

Taking the measurement noise(Q) to be 10^{-3} and R to be identity, following the iteration method explained in Chapter1.2, we get the following results. Note that the Kalman filter is effectively able to trace the path from the noise but also its correspondence to the ground truth. The initial estimates are a bit off as we started from a random starting point. This difference quickly reduces for the coming time steps. Though numerically, there seems to be some bias in the estimates, this could be arising from the incorrect accounting for the process noise. However, the trend of the parameters is effectively captured.



(a) Actual Traced Path and the Observed Sensor path



(b) Observed Path and the Kalman Recovered Path

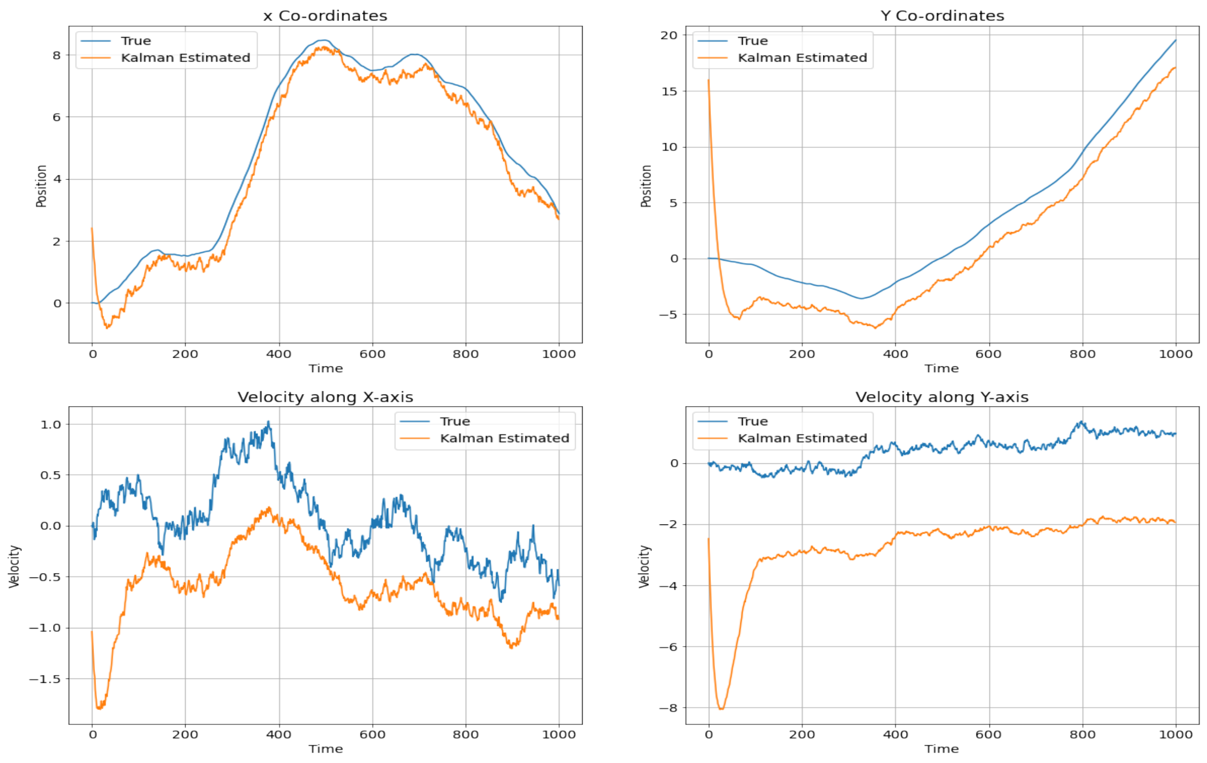


Figure 2.9: Comparing the Estimates with the ground truth

The code to replicate the above simulation is as follows:

Listing 2.3: SpeechEnhancement

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

def plot_state(t,actual, estimated=None):
    '''
    plot position, speed, and acceleration in the x and y coordinates for
    the actual data, and optionally for the estimated data
    '''
    trajectories = [actual]
    if estimated is not None:
        trajectories.append(estimated)

    fig, ax = plt.subplots(3, 2, sharex='col', sharey='row', figsize=(8,8))
    for x, w in trajectories:
        ax[0,0].plot(t,x[0,:-1])
        ax[0,1].plot(t,x[1,:-1])
        ax[1,0].plot(t,x[2,:-1])
        ax[1,1].plot(t,x[3,:-1])
        ax[2,0].plot(t,w[0,:])
        ax[2,1].plot(t,w[1,:])

    ax[0,0].set_ylabel('x position')
    ax[1,0].set_ylabel('x velocity')
    ax[2,0].set_ylabel('x input')

    ax[0,1].set_ylabel('y position')
    ax[1,1].set_ylabel('y velocity')
    ax[2,1].set_ylabel('y input')

    ax[0,1].yaxis.tick_right()
    ax[1,1].yaxis.tick_right()
    ax[2,1].yaxis.tick_right()

    ax[0,1].yaxis.set_label_position("right")
    ax[1,1].yaxis.set_label_position("right")
    ax[2,1].yaxis.set_label_position("right")

    ax[2,0].set_xlabel('time')
    ax[2,1].set_xlabel('time')

def plot_positions(traj, labels, axis=None,filename=None):
    '''
    show point clouds for true, observed, and recovered positions
    '''
    matplotlib.rcParams.update({'font.size': 14})
    n = len(traj)

    fig, ax = plt.subplots(1, n, sharex=True, sharey=True,figsize=(12, 5))
    if n == 1:
        ax = [ax]

    for i,x in enumerate(traj):
        ax[i].plot(x[0,:], x[1,:], 'ro', alpha=.1)
        ax[i].set_title(labels[i])
        if axis:
            ax[i].axis(axis)
    if filename:
        fig.savefig(filename, bbox_inches='tight')


n = 1000 # number of timesteps
T = 50 # time will vary from 0 to T with step delt
ts, delt = np.linspace(0,T,n,endpoint=True, retstep=True)
gamma = .05 # damping, 0 is no damping

A = np.zeros((4,4))
B = np.zeros((4,2))
C = np.zeros((2,4))

A[0,0] = 1
A[1,1] = 1
A[0,2] = (1-gamma*delt/2)*delt
A[1,3] = (1-gamma*delt/2)*delt
A[2,2] = 1 - gamma*delt
A[3,3] = 1 - gamma*delt

B[0,0] = delt**2/2
B[1,1] = delt**2/2
B[2,0] = delt
B[3,1] = delt

C[0,0] = 1
C[1,1] = 1
sigma = 20
p = .20
np.random.seed(6)

x = np.zeros((4,n+1))
x[:,0] = [0,0,0,0]
y = np.zeros((2,n))

# generate random input and noise vectors
w = np.random.randn(2,n)
v = np.random.randn(2,n)

# add outliers to v
np.random.seed(0)
inds = np.random.rand(n) <= p
# v[:,inds] = sigma*np.random.randn(2,n)[:,inds]

# simulate the system forward in time
for t in range(n):
    y[:,t] = C.dot(x[:,t]) + v[:,t]
    x[:,t+1] = A.dot(x[:,t]) + B.dot(w[:,t])

x_true = x.copy()
w_true = w.copy()

plot_state(ts,(x_true,w_true))
plot_positions([x_true,y], ['True', 'Observed'],[-4,14,-5,20],'rkf1.pdf')

Q = 0.001*np.eye(4)
R = np.eye(2)
P = np.eye(4)
x_k = np.zeros((4,n))
x_prev = np.zeros((4,1))
iterations = 10
for iteration in range(iterations):
    for t in range(n):
        #Prediction
        x_est = A@x_prev + B@w[:,0].reshape((-1,1))
        P_est = A@P@A.T + Q
        #Update
        V = y[:,t].reshape((-1,1))-C@x_est
        S = C@P_est@C.T + R
        K = P_est@C.T@np.linalg.inv(S)
        x_k[:,t] = (x_est + K@V).reshape(x_k[:,t].shape)
        P = P_est - K@S@K.T
        x_prev = x_k[:,t].reshape((-1,1))
plot_positions([y,x_k], ['Observed', 'Kalman Estimated'], [-4,14,-10,20])
plt.figure(figsize = (20,16))
plt.subplot(2,2,1)
plt.plot(x_true[0,:],label='True')
plt.plot(x_k[0,:],label='Kalman Estimated')
plt.grid()
plt.xlabel('Time')
plt.ylabel('Position')
plt.legend()
plt.title('x Co-ordinates')
plt.subplot(2,2,2)
plt.plot(x_true[1,:],label='True')
plt.plot(x_k[1,:],label='Kalman Estimated')
plt.grid()
plt.xlabel('Time')
plt.ylabel('Position')
plt.title('Y Co-ordinates')
plt.legend()
plt.subplot(2,2,3)
plt.plot(x_true[2,:],label='True')
plt.plot(x_k[2,:],label='Kalman Estimated')
plt.grid()
plt.xlabel('Time')
plt.ylabel('Velocity')
plt.title('Velocity along X-axis')
plt.legend()
plt.subplot(2,2,4)
plt.plot(x_true[3,:],label='True')
plt.plot(x_k[3,:],label='Kalman Estimated')
plt.grid()
plt.xlabel('Time')
plt.ylabel('Velocity')
plt.title('Velocity along Y-axis')
plt.legend()
plt.show()
```

The base source code on top of which the algorithm is applied is as shown in the *CVXPY* documentation under the article `https://www.cvxpy.org/examples/applications/robust_kalman.html`. Here we have tweaked the algorithm from a convex optimization problem perspective to a simpler iterative linear dynamic system. However, the algorithm performs worst when processed with Gaussian noised data and would need the use of Robust Kalman filtering, which uses the Huber Loss metric. The final answer should overlap after the significant number of iterations as shown in the proof of working.

References

Aydos, Carlos, Hengst, Bernhard, and Uther, William (2009). “Kalman filter process models for urban vehicle tracking”. In: *2009 12th International IEEE Conference on Intelligent Transportation Systems*, pp. 1–8. DOI: `10.1109/ITSC.2009.5309752`.

Das, Orchisama (May 2016). “Kalman Filter in Speech Enhancement”. In: DOI: `10.13140/RG.2.2.27954.20160`.

Li, Qiang, Li, Ranyang, Ji, Kaifan, and Dai, Wei (2015). “Kalman Filter and Its Application”. In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pp. 74–77. DOI: `10.1109/ICINIS.2015.35`.

Yu, Huan and Zhang, Wenhui (2014). “Moving camera video stabilization based on Kalman filter and least squares fitting”. In: *Proceeding of the 11th World Congress on Intelligent Control and Automation*, pp. 5956–5961. DOI: `10.1109/WCICA.2014.7053740`.