

Filtrelemek

Filtreler dis dunyadaki bir aksiyon hakkında elde edilen gurultulu sinyalleri, tersine cevirecek arka plandaki aksiyon hakkında hesaplama yapabilmemizi saglar. Mesela Kalman Filtreleri (KF) icin gizlenmis konum bir robotun nerede oldugu, bir senetin fiyati gibi bir sey olabilir, gizli konum bilgisi x_t degiskeninde o konum hakkindaki gurultulu olcum y_t icindedir. Hem gizli konumlar arasindaki gecis, hem de olcumun gurultusu lineer bir fonksiyon uzereindir.

$$x_{t+1} = Ax_t + v$$

$$y_t = Hx_t + w$$

v ve w 'in dagilimi Gaussian'dir ve kovaryans sirasiyla Q ve R icindedir. Cogu filtre hesaplamasinda A, H, Q, R matrisleri onceden bilinir.

Zaman faktorunu de dahil etmek gerekirse;

$$\hat{x}_t^t = E[x_t|y_0, \dots, y_t]$$

$$P_t^t = E[(x_t - \hat{x}_{t|t})(x_t - \hat{x}_{t|t})'|y_0, \dots, y_t]$$

Filtremenin amaci x_{t+1} ve P_{t+1} hesabini yeni bir olcum y_{t+1} uzereinden yapmak olacak. “Gizli” x_t derken bunu kastediyorduk, bu deger bize verilmiyor, sadece x_t ve x_{t+1} arasindaki gecisin nasil oldugunu biliyoruz, gurultunun nasil eklendigini biliyoruz, ama bunlari bilsek bile elde bir suru bilinmeyen var. Filtrelemenin matematiksel numaralari sayesinde bunu hesaplayabiliyor olacagiz. Yani yapmamiz gereken “oku tersine cevirmek”, yani x_t 'nin y_t uzereindeki sartasal baglilikini (conditional dependence) ortaya cikartmak, bunu y_t 'nin x_t 'ye olan sartasal bagimlilikini tersine cevirecek yapmak. Ana denklemin iki tarafinin da beklentisini (expectation) alalim:

$$E[x_{t+1}] = E[Ax_t + v] = E[Ax_t] = AE[x_t] = A\mu_t = A\hat{x}_t$$

Notasyon olarak sunu soylesek iyi olur

$$E[x_{t+1}] = \hat{x}_{t+1}$$

Beklentiye alinca v yok oldu cunku bu rasgele degiskenin beklentisinin sifir oldugunu biliyoruz. Simdi iki tarafin kovaryansini alalim ve P_t 'yi $cov x_t$ olarak belirtelim:

$$P_{t+1} = AP_tA' + Q$$

Bu gecis “zaman guncellemesi” olarak adlandirilir. Normal dagilimleri t anindan $t + 1$ anina gecirmemizi saglar. y iceren formullerde benzer bir durum var.

$$\hat{x}_{t+1}^t = Ax_t^t$$

$$P_{t+1}^t = AP_t^tA' + Q$$

$$y_{t+1} = Cx_{t+1} + w_t$$

$$E[y_{t+1}|y_0, \dots, y_t] = E[Cx_{t+1} + w_t|y_0, \dots, y_t]$$

$$\hat{y}_{t+1}^t = C\hat{x}_{t+1}$$

Kovaryans icin benzer durum

$$E[(y_{t+1} - \hat{y}_{t+1}^t)(y_{t+1} - \hat{y}_{t+1}^t)' | y_0, \dots, y_t] = C_{t+1}^t C' + R$$

Simdi daha zor is olan oku tersini cevirmeye geelim. Eger amacimiz $p(x_t|y_t)$ denklemini elde etmek ise o zaman bu iki degiskeni iceren birlesik dagilimi (joint distribution) elde etmek zorundayiz. Iki Gaussian'in birlesiminin yeni bir Gaussian oldugunu biliyoruz, o zaman hem x_t hem de y_t 'in kendisi cok boyutlu birer Gaussian olduklari icin onların birlesimi $p(x_t|y_t)$ 'in hakikaten devasa bir Gaussian olacagini tahmin edebiliriz.

x_t ve y_t 'in birlesimi olan Gaussian'i bulmak demek, bu Gaussian'in ortalamasini (mean) ve kovaryansini bulmak demektir cunku bir Gaussian ortalama ve kovaryansi ile net bir sekilde tanimlanabilir bir seydir. Bir numara yapalim, ve $y_t = Cx_t + w_t$ 'yi $z = Hu$ seklinde yazalim. Sonra

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix}, H = \begin{bmatrix} I & 0 \\ C & I \end{bmatrix}, u = \begin{bmatrix} x_t \\ w_t \end{bmatrix}$$

Boylece daha basit bir denklemin kovaryansini alabiliriz

$$cov(z) = H cov(u) H'$$

$$cov(u) = \begin{bmatrix} P_t & 0 \\ 0 & R \end{bmatrix}$$

Tam carpim suna esit

$$\begin{bmatrix} I & 0 \\ C & I \end{bmatrix} \begin{bmatrix} P_t & 0 \\ 0 & R \end{bmatrix} \begin{bmatrix} I & C' \\ 0 & I \end{bmatrix}$$

bunun sonucu ise

$$\begin{bmatrix} P_t & P_t C' \\ C P_t & C P_t C' + R \end{bmatrix}$$

Bunu baglantisal denklem icin ve ortalamayi icerecek sekilde yazabiliriz

$$\begin{bmatrix} \hat{x}_t^t \\ C\hat{x}_t^t \end{bmatrix}, \begin{bmatrix} P_t^t & P_t^t C' \\ C P_t^t & C P_t^t C' + R \end{bmatrix}$$

Ayni sekilde x_{t+1}, y_{t+1} birlesik dagilim icin

$$\begin{bmatrix} \hat{x}_{t+1}^t \\ C\hat{x}_{t+1}^t \end{bmatrix}, \begin{bmatrix} P_{t+1}^t & P_{t+1}^t C' \\ C P_{t+1}^t & C P_{t+1}^t C' + R \end{bmatrix} \quad (1)$$

Simdi x_{t+1}^{t+1} 'in ortalama ve varyansi icin parcali Gaussian kavramini anlatmaliyiz. Bir n boyutlu Gaussian daha kucuk boyutlardaki p ve q alt Gaussian'lara parcalanabilir (tabii ki $n = p + q$). Yani su ifade kullanilabilir

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \quad (2)$$

$$p(x|\mu, \Sigma) = \frac{1}{(2\pi)^{(p+q)/2} |\Sigma|^{1/2}} \exp \left\{ -\frac{1}{2} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix}' \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix} \right\}$$

Uzun cebirsel işlemlerden sonra $p(x_1|x_2)$ ifadesini elde ederiz. Bu cebirsel türetimi görmek istiyorsanız, *Istatistik* ders notlarımızda Ders 3'e bakabilirsiniz.

Bundan sonra şartlanmış (conditioned) μ ve Σ alınır.

$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) \quad (3)$$

$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$$

\hat{x}_{t+1}^{t+1} ve P_{t+1}^{t+1} nasıl hesaplayacağız?

x_{t+1}, y_{t+1} birlesik dagilimini hesapladigimiz zaman ve y_{t+1} 'in verildiği kosulsal durumda aslında otomatik olarak t 'yi bir ileri atmış oluruz (çünkü y_{t+1} verili, o verilmiş ise, artık $t + 1$ 'deyiz demektir). O zaman x_{t+1}, y_{t+1} birlesik dagilimından y_{t+1} 'i verilmiş x_{t+1} kosulsal olasiligini hesaplırsak, \hat{x}_{t+1}^{t+1} ve P_{t+1}^{t+1} hesaplayabiliriz.

O zaman (3)'ü baz alıp (1)'in icine koydugumuzda ve (2)'deki yerlesim yapısını dikkate aldığımızda \hat{x}_{t+1}^{t+1} ve P_{t+1}^{t+1} formüllerini ortaya çıkartabiliriz.

$$\hat{x}_{t+1}^{t+1} = x_{t+1}^t + P_{t+1}^t C' (C P_{t+1}^t C' + \Sigma_w)^{-1} (y_{t+1} - C \hat{x}_{t+1}^t)$$

$$P_{t+1}^{t+1} = P_{t+1}^t - P_{t+1}^t C' (C P_{t+1}^t C' + R)^{-1} C P_{t+1}^t$$

Eğer $K = P_{t+1}^t C' (C P_{t+1}^t C' + \Sigma_w)^{-1}$ dersek

$$\hat{x}_{t+1}^{t+1} = \hat{x}_{t+1}^t + K_t (y_{t+1} - C \hat{x}_{t+1}^t)$$

$$P_{t+1}^{t+1} = P_{t+1}^t - K_t C P_{t+1}^t$$

Ornek: Veriye Düz Çizgi Uydurmak (Line Fitting)

Eğer elimizde bir çizgiye uydurmak için kullanacağımız tüm veri olsaydı, uydurma işlemi için en az kareler (least squares) yöntemini kullanabilirdik. Kalman Filtreleri bize yeni veri geldiği anda, her seferinde, azar azar bir çizgiyi uydurmamızı sağlıyor. Hatta matematiksel olarak ispalanmıştır ki eğer başlangıç noktası aynıysa, azar azar veriyi KF ile almanın sonunda, tüm veriyi bir kerede en az karesel yöntem ile uydurmak aynı sonucu verir.

Peki bu uydurma işlemini nasıl yaparız? Burada veriyi nasıl temsil ettiğimiz konusunda ufak bir numara kullanmamız lazım.

Kendimize bir soru soralım: bu sistemin konum bilgisi nedir? Bir robotu izliyorsak mesela soru cevabı basittir, onun x, y gibi koordinat bilgisi. Düz çizgi fit ederken takip edilen bunlar değil, bize gerekli olan bir çizginin “egimi (slope)”. Yani hem bir çizginin y eksenini kestigi nokta, hem de çizginin egimi x_t konum bilgisi içinde dahil edilecek. Burada KF literatüründen gelen x, y harfleri birbirine karışmasın diye çizginin değerlerini xx_t ve yy_t olarak tanımlayacağız. O zaman x_t vektörü suna

benzer:

$$x_t = \begin{bmatrix} yy_t \\ a \end{bmatrix}$$

ki burada a harfi egimi temsil etmektedir. a bir sabit olduğuna göre KF her zaman diliminde aynı kalacak bir değişkeni hesaplayacaktır. Cogunlukla KF ile her zaman diliminde değişik olan değerlerin hesaplandığını görürüz, bu uygulamaya göre değişen bir şeydir, matematiksel bir mecburiyet değildir. A matrisimiz ile de biraz numara yapmamız gerekli. Bu matris x_t 'yi dönüştürüp x_{t+1} 'i elde etmemizi sağlayan şey olduğuna göre A 'nın şöyle olması gerekir:

$$A_t = \begin{bmatrix} 1 & \Delta xx \\ 0 & 1 \end{bmatrix}$$

Bu matrisi x_t ile çarptığımızda $yy_t \cdot 1 + a \cdot \Delta xx$ değerini elde ediyoruz, ki bu değer bir çizgi üzerinde bir sonraki noktayı temsil ediyor. Dis ölçümü veren gürültü matrisi H ise

$$H = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

şeklinde. Bunu x_t ile çarptığımızda y_t 'yi (iki kere) elde ettiğimizi göreceğiz. Not: Niye iki kere? Kodlama sırasında boyutların uyumlu olması için böyle gerekti, çok büyük bir rahatsızlık değil. Kod altta görülebilir.

```
import numpy.linalg as lin
slope = 2

#
#  $x_{t+1} = A x_t + Q$ 
#  $y_t = Hx_t + R$ 
#
def Kalman(obs,x,mu_init,nsteps):

    ndim = shape(mu_init)[0]

    Q = np.zeros((ndim, ndim))
    A = np.eye(ndim)
    H = np.array([[1, 0], [1, 0]])

    mu_hat = mu_init
    cov = np.ones((ndim, ndim))
    R = np.eye(ndim) * 10

    m = np.zeros((ndim,nsteps),dtype=float)
    ce = np.zeros((ndim,ndim,nsteps),dtype=float)

    for t in range(1,nsteps):
```

```

# Tahmini yap
# A transofmrasyon matrisi ve suna esit
# / 1 delta_x /
# / 0      1   /
A = np.array([[1, x[t]-x[t-1]], [0, 1]])
mu_hat_est = np.dot(A,mu_hat)
cov_est = np.dot(A,dot(cov,transpose(A))) + Q

# tahmini guncelle
error_mu = obs[:,t] - dot(H,mu_hat_est)
error_cov = np.dot(H,np.dot(cov,np.transpose(H))) + R
K = np.dot(np.dot(cov_est,np.transpose(H)),lin.inv(error_cov))
mu_hat = mu_hat_est + np.dot(K,error_mu)
m[:,t] = mu_hat
cov = np.dot((np.eye(ndim) - np.dot(K,H)),cov_est)
ce[:,:,t] = cov
return mu_hat

N = 20

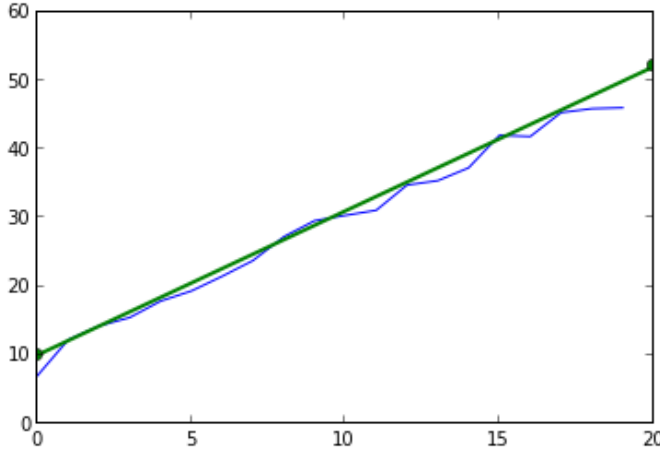
#
# ornek veri yarat
#

obs = zeros((2, N))
x = xrange(N)
for i in xrange(N):
    obs[0, i] = obs[1, i] = (slope*i)+random.normal(10)

mu_hat = Kalman(obs, x, mu_init=array([0, 0]),nsteps=N)

plt.plot(obs[0, :])
plt.plot([0,N], [10,N*mu_hat[1]], 'go-', label='line 1', linewidth=2)
plt.savefig('kalman-line-fit.png')

```



Ornek: Obje Takibi

Daha degisik bir ornekten bahsedelim. Bu ornekte OpenCV kutuphanesinden elde ettigimiz 2 boyutlu degerleri y_t icin kullanacagiz. Degerler OpenCV'nin bir satranc tahtasi seklinin kose noktalarini otomatik olarak bulabilen `cvFindChessboardCorners` cagrisinden gelecek (ayrica `cvDrawChessboardCorners` ile bu noktaları ekranda aninda gosterebilecegiz).

Elimizdeki “gurultulu” olcumler iki boyutlu noktasal degerler. Gurultulu cunku kamera bize bu imajlari aktarirken hata eklemis olabilir, OpenCV fonksiyonu hesabi yaparken hata eklemis olabilir, bir suru olasilik var.

Bu ornekte, ayrica, ilk kez KF ortaminda boyut degisikligi olasiligini net bir sekilde gorebiliyoruz. Gizli konum bilgisi x_t 3 boyutlu bir nokta, ama elimizdeki olcum 2 boyutlu bir “yansima”. Yansima sirasinda kacinilmaz olarak deger kaybediliyor, bir boyutun bilgisi ortadan yokoluyor. Ama tum bu bilinmezlerle ragmen Kalman filtresinin bizim icin gizli bilgiyi hesaplamasini istiyoruz.

Bu problemde A matrisi ne olacaktir? Obje takibi konularinda A 'nin ne oldugunu hayal etmek daha kolay, A matrisi iki zaman dilimi arasindaki “hareketi” temsil edecek. Bu problemdeki ek bir kolaylik bu hareketi onceden bildigimiz, ve hareketin tek yonde oldugu. Yani resimde benim tuttugum kartonu ne kadar hizla hareket ettirdigimi ben onceden probleme bildiriyorum. Yer degisikligini d olarak betimledim, ve A soyle oldu:

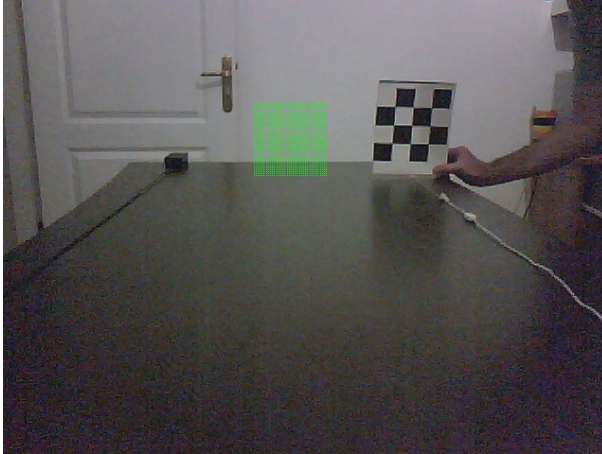
$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dikkat edersek A 4x4 boyutunda, 3x3 degil. 3 boyutlu kordinatlari temsil etmek icin homojen kordinat sistemini kullandigimiz icin boyle oldu, o sebeple zaten x_t de 4x1 oldu, ona uymak icin A 'nin degismesi gerekiyordu. Ax_t carpiminin hakikaten kartonu hareket ettirdigini gostermek icin bu carpimi bir ornek uzerinde yapalim:

Diyelim ki $x_t = [a_1 \ a_2 \ a_3 \ a_4]$ o zaman Ax_t ya da x_{t+1} su hale gelir: $[a_1 \ a_2 \ a_3 + d \ a_4]$.

Bakiyoruz, hakikaten de d kadarlık bir yer degisimi z kordinati, yani derinlik uzerinde eklenmis. Test amaclarimiz icin $d = -0.5$ aldik, yani satranc tahta kartonunun her zaman diliminde kameraya dogru 0.5 cm ilerledigini belirttik. Tabii bu da kabaca bir tahmindir (her ne kadar hareketi yaptiran ben olsam bile!), ama filtrelemenin gucunu burada goruyoruz. Benim tahminimde “gurultu” yani “hata payi” var, olcumde gurultu var, tum bunlar ust uste konsa bile filtre yine de gizli konumu bulacak.

Olcumsal donusumu temsil eden H 'e ben onun temeli olan yansima (projection) kelimesinden gelen P matrisinden bahsedelim. Yansima matrisi goruntu (vision) literaturunde tek delikli kamera (pinhole camera) modelinden ileri gelen bir matriştir ve bu matrisi hesaplamak ayarlama / kalibrasyon (calibration) denen apayri bir islemin parcasidir. OpenCV icinde kalibrasyon icin fonksiyonlar var, biz de bunlari denedik, kalibrasyon icin kullandigimiz resimlerle alakali olmali, elde edilen sonuclardan memnun kalmadik. Alternatif olarak sunu yaptik; resimde gorulen yesil yuzey bizim programin olusturdugu hayali bir yuzey. Filtrenin o anki tahminini P uzerinden goruntuye yansitarak bu yuzeyi olusturduk, Boylece deneme / yanilma yontemiyle pek cok P degerini deneyerek, yuzeyin resimde gorulen masanin sonunda cikacak sekilde olmasini sagladik. Projeksiyon icin kullanan K matrisi ve baslangic imaji altta:



```
from numpy import *
```

```
K = array([[700., 0., 300.],  
           [0., 700., 330.],  
           [0., 0., 1.]])
```

O noktaya gelince istedigimiz P degerini bulmus oluyorduk. Yansitma matrisleri 3×3 olur, KF buna bir dorduncu $[0 \ 0 \ 0]$ satiri ekleyerek onu 4×3 H haline getiriyor.

KF 'in baslangic noktası olarak P 'yi bulmak icin kullandigimiz masa sonunu kullandik. Kararsizlik olcutu Q icin, ki bu degisken bir Gaussian kovaryansidir, $Q = I \cdot 150cm$ degerini kullandik, yani oldukca buyuk bir kararsizlik degeri kullandik.

Sebeup baslangic degeri olan masa ortasini sectik, ve takip edecegimiz satranc tah- tasinin nerede oldugunu bilmiyoruz, “emin degiliz”. Bu kararsizligi sayisal olarak programa bildirmis olduk.

Alttaki resimlerde filtrenin tahminini temsil eden yesil yuzeyin satranc tah- tasini basariyla takip ettigini goreceksiniz.

```
import cv
import sys
from kalman_3d import *
from K import *

def proj_board(im, xl, yl, z):
    color = cv.CV_RGB(0, 255, 0)
    image_size = cv.GetSize(im)
    for x in arange(xl-9, xl+9, 0.5):
        for y in arange(yl-9, yl+9, 0.5):
            X = array([x, y, z])
            q = dot(K, X)
            q = [int(q[0]/q[2]), int(q[1]/q[2])]
            cv.Set2D(im, im.height-q[1], q[0], color)

def show_data(image, mu_x):
    line_type = cv.CV_AA
    pt1 = (30, 400)
    font = cv.InitFont (cv.CV_FONT_HERSHEY_SIMPLEX,
                        0.8, 0.1, 0, 1, cv.CV_AA)
    cv.PutText (image, "Kalman Filter " + str(mu_x),
                pt1, font, cv.CV_RGB(255,255,0))

def detect(image):
    image_size = cv.GetSize(image)

    # create grayscale version
    grayscale = cv.CreateImage(image_size, 8, 1)
    cv.CvtColor(image, grayscale, cv.CV_BGR2GRAY)
    storage = cv.CreateMemStorage(0)

    im = cv.CreateImage (image_size, 8, 3)

    status, corners = cv.FindChessboardCorners( grayscale, (dim,dim))
    if status:
        cv.DrawChessboardCorners( image, (dim,dim), corners, status)
        is_x = [p[0] for p in corners]
        is_y = [p[1] for p in corners]
        return is_x, is_y
```



```

    return [], []

if __name__ == "__main__":

    snap_no = 1
    frame_no = 0

    # create windows
    cv.NamedWindow('Camera', cv.CV_WINDOW_AUTOSIZE)

    # create capture device
    device = 0 # assume we want first device

    capture = cv.CreateFileCapture (sys.argv[1])
    #capture = cvCreateCameraCapture (0)

    dim = 3

    pts = dim * dim
    mid = int(pts / 2)

    cv.SetCaptureProperty(capture, cv.CV_CAP_PROP_FRAME_WIDTH, 640)
    cv.SetCaptureProperty(capture, cv.CV_CAP_PROP_FRAME_HEIGHT, 480)

    # check if capture device is OK
    if not capture:
        print "Error opening capture device"
        sys.exit(1)

    kalman = Kalman(K, mu_init=array([1., 1., 165., 2.]))

    frame = cv.QueryFrame(capture)
    proj_board(frame, 1, 1, 160)
    # ilk resimde ilk projeksiyon var, hemen kaydedelim
    cv.SaveImage('first-projection.jpg', frame)
    cv.ShowImage('Camera', frame)

    while True:
        frame_no += 1

        # capture the current frame
        frame = cv.QueryFrame(capture)
        image_size = cv.GetSize(frame)
        if frame is None:
            break

```

```

is_x, is_y = detect(frame)

if len(is_x) > 0 :
    kalman.update(array([is_x[5], frame.height-is_y[5], 1.]))
    proj_board(frame,
                kalman.mu_hat[0],
                kalman.mu_hat[1],
                kalman.mu_hat[2])

    show_data(frame, kalman.mu_hat)

# display webcam image
cv.ShowImage('Camera', frame)

if snap_no == 12: break
if frame_no % 10 == 0:
    cv.SaveImage('cb-kf-' + str(snap_no) + '.jpg', frame)
    snap_no += 1

# handle events
k = cv.WaitKey(40)
if k == 27: # ESC
    print 'ESC pressed. Exiting ...'
    break

from numpy import *

#  $x_{t+1} = A x_t + \text{Sigma}_x$ 
#  $y_t = Hx_t + R$ 
class Kalman:
    # T is the translation matrix
    # K is the camera matrix calculated by calibration
    def __init__(self, K, mu_init):
        self.ndim = 3
        self.Sigma_x = eye(self.ndim+1)*150
        self.A = eye(4)
        self.A[2,3] = -0.5
        self.H = append(K, [[0], [0], [0]], axis=1)
        self.mu_hat = mu_init
        self.cov = eye(self.ndim+1)
        self.R = eye(self.ndim)*1.5

    def normalize_2d(self, x):
        return array([x[0]/x[2], x[1]/x[2], 1.0])

```

```

def update(self, obs):

    # Make prediction
    #print "self.mu_hat=" + str(self.mu_hat)
    self.mu_hat_est = dot(self.A, self.mu_hat)
    prod = dot(self.A, dot(self.cov, transpose(self.A)))
    self.cov_est = prod + self.Sigma_x
    #print "self.mu_hat_est=" + str(self.mu_hat_est)
    #print "self.cov_est=" + str(self.cov_est)

    # Update estimate
    prod = self.normalize_2d(dot(self.H, self.mu_hat_est))
    self.error_mu = obs - prod

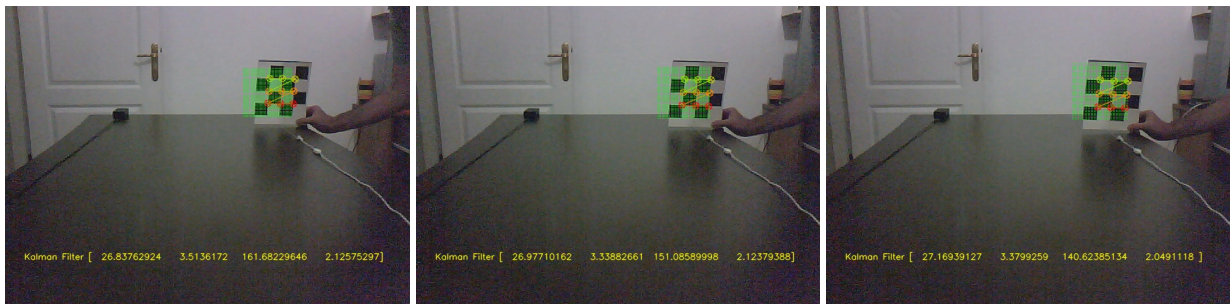
    prod = dot(self.cov, transpose(self.H))
    prod = dot(self.H, prod)
    self.error_cov = prod + self.R
    prod = dot(self.cov_est, transpose(self.H))
    self.K = dot(prod, linalg.inv(self.error_cov))
    self.mu_hat = self.mu_hat_est + dot(self.K, self.error_mu)

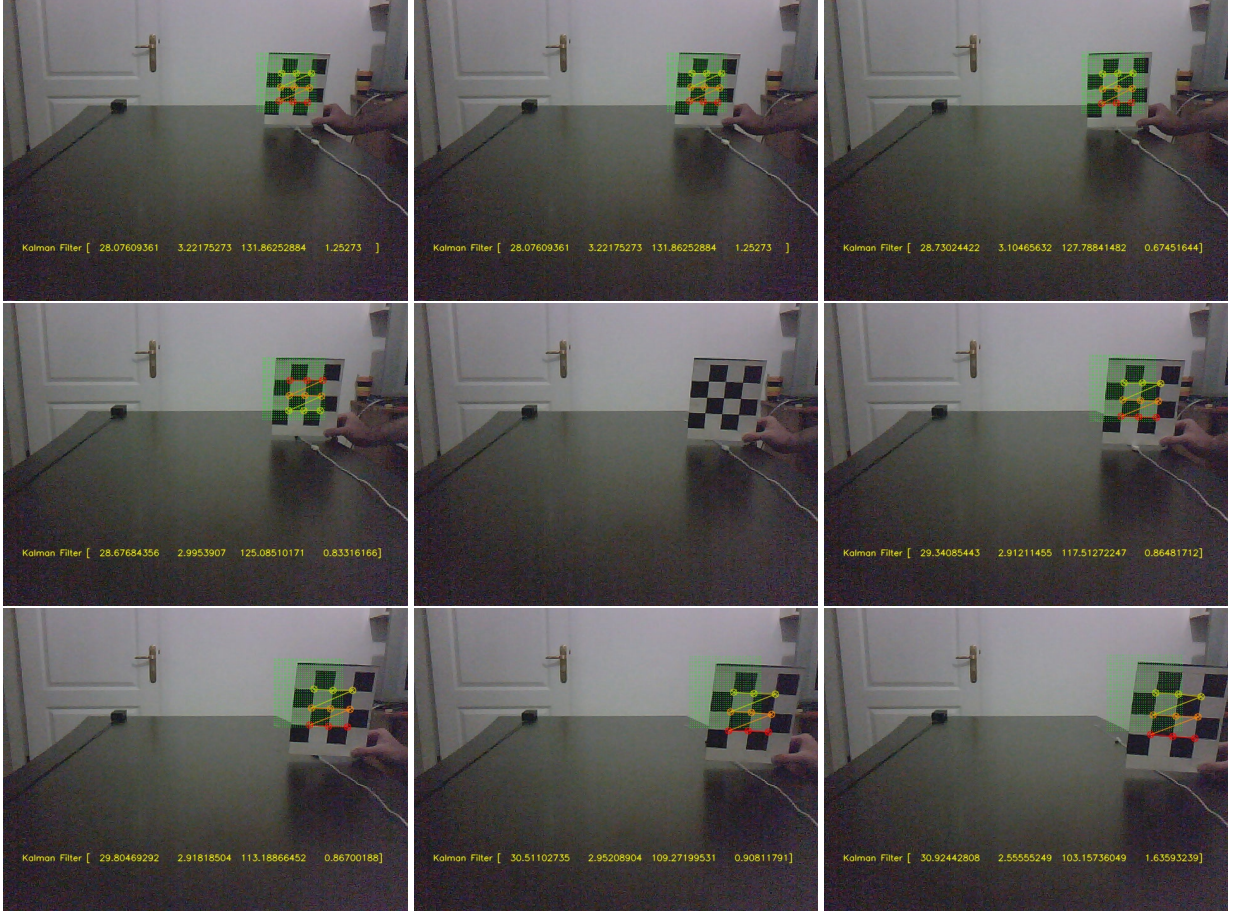
    prod = dot(self.K, self.H)
    left = eye(self.ndim+1)
    diff = left - prod
    self.cov = dot(diff, self.cov_est)

import sys, os
if sys.argv[1] == "kf":
    file = "$HOME/Dropbox/Public/skfiles/campy/chessb-right.avi"
    os.system("python track-chess-kf.py %s" % file)
if sys.argv[1] == "pf":
    file = "$HOME/Dropbox/Public/skfiles/campy/chessb-right.avi"
    os.system("python track-chess-pf.py %s" % file)

```

!python run_filter.py kf





Parcacik Filtreleri

Filtrelemede tek yontem Kalman filtreleri degil. KF kararsizlik Gaussian olarak gosterilebiliyorsa cok faydali, ve hizli bir yontem. Bir KF bellekte cok az yer tutar, 3 boyutlu bir Gaussian icin 3x1 boyutunda bir ortalama vektoru, ve 3x3 boyutunda bir kovaryans matrisi yeterlidir, yani $3 + 9 = 12$ sayi.

Parcacik filtreleri (PF) bir dagilimi “ayriksal” olarak temsil ederler. Yani diyelim ki tek boyutlu bir dagilimi 100 eleman iceren bir dizin ile temsil edebiliriz, o zaman dagilimin degerlerini 100 tane noktada tasimamiz gerekir. Bunun faydalari her turllu dagilim seklini temsil edebilmemiz. Gaussian sadece belli bir sekilde olabilir, tek bir tepe noktasi olmalidir, vs. Ayriksal temsil ile 2, 3, istedigimiz kadar tepe noktasi olan (ya da hic olmayan) bir dagilim kullanabiliriz.

Bu neye yarar? Birden fazla hipotezi ayni anda isletebilmemize yarar. KF ile tepe noktasi en iyi tahminimizdir (mesela.. satranc kartonu masa ortasinda), PF ile birkac tahmini ayni anda hesaplatmak mumkun olabilir.

Daha detaylandirmak gerekirse, PF kodlamasi x_t icin iki tane veri yapisi gerektirir. Bir veri yapisi dagilimdaki degerleri temsil eden parcaciklardir, digeri ise bu parcaciklari dagilimdaki onemini temsil eden agirliklardir. Filtreleme sistemi KF’ye benzer, once bir gecis uygulanir, ki bu gecis kararsizligi arttiracaktir, fakat ardindan gozlem verisi bir hata fonksiyonu uzerinden dagilim guncellenir. Bu islem sirasinda

hataşı yuksek olan parcacıklar cezalandırılır, onların ağırlığı azalır, otekilerinki yuk-
selir. Her parcacık için hata fonksiyonu şudur:

$$w^{[i]} = \frac{1}{1 + (y^{[i]} - p^{[i]})^2}$$

$y^{[i]}$ gözlem değeri, $p^{[i]}$ gecis uygulandıktan sonra elimizdeki tahminimizdir, ki bu KF dünyasındaki $Ax_t + Q$ 'nın karşılığıdır. PF için hareket gecisi şöyle hesaplanır: Bir uniform dağılımdan örnekleme yapılır, ve bu örneklenen değerler x 'e eklenir. Örnekleme için z-kordinati için $Unif(-0.1, -1)$ 'i, x kordinati için $Unif(-40, 40)$ 'i kullandık. Yani ileri doğru 0.1 ve 1 santimetre arasında bir hareket ekliyoruz, ve sağa ve sola donuk olarak 80 santimetrelık bir kararsızlığı hesaplara ekliyoruz.

Ustteki formülde $(y^{[i]} - p^{[i]})^2$ e niye 1 değeri ekledigimiz aciktir herhalde, bu sayede hata fonksiyonunun olasılık değerlerini andıran bir sonuc don- durmesini istiyoruz. Çok ufak hatalar için $1 + hata$ bölünendeki 1'i böylecek, ve 1'e yakın bir değer geri getirecek. İstedigimiz de bu zaten, küçük hataların daha büyük ağırlığa sebebiyet vermeleri, büyük hataların ise tam tersi sonuca sebep olmaları.

Tekrar örnekleme (resampling) sürecinde parcacıklar tekrar düzenlenerek ağırlığı çok olan parcacıkların ağırlığı az olanlara göre daha fazla tekrarlanmasını istiyoruz. Dikkat: tekrar örnekleme süreci yeni parcacık değerleri yaratmıyor, sadece mevcut olanları tekrarlıyor ya da onları atlıyor.

```
from numpy import *
from numpy.random import *

class PF:

    def __init__(self, K, n):
        self.H = append(K, [[0], [0], [0]], axis=1)
        self.n = n
        self.x = zeros((self.n, 4))
        self.x[:, :] = array([1., 1., 165., -1])

    def normalize_2d(self, x):
        return array([x[0]/x[2], x[1]/x[2], 1.0])

    def resample(self, weights):
        n = len(weights)
        indices = []
        C = [0.] + [sum(weights[:i+1]) for i in range(n)]
        u0, j = random(), 0
        for u in [(u0+i)/n for i in range(n)]:
            while u > C[j]:
                j+=1
            indices.append(j-1)
        return indices
```



```

def update(self, y):
    u = uniform(-0.1, -1, self.n) # forward with uncertainty
    self.x[:,2] += u
    u = uniform(-40,40, self.n) # left right uncertainty
    self.x[:,0] += u
    p = dot(self.x,self.H.T)
    for i, item in enumerate(p): # modify in place
        p[i,:] = self.normalize_2d(item)
    self.w = 1./(1. + (y-p)**2)
    self.w = self.w[:,0]+self.w[:,1]
    #self.w = self.w[:,0]
    self.w /= sum(self.w)
    self.x = self.x[self.resample(self.w),:]

def average(self):
    return sum(self.x.T*self.w, axis=1)

import sys
import cv
from numpy import *
from K import *
from PF import *

def proj_board(im, xl, yl, z):
    color = cv.CV_RGB(0, 255, 0)
    image_size = (im.width, im.height)
    for x in arange(xl-9, xl+9, 0.5):
        for y in arange(yl-9, yl+9, 0.5):
            X = array([x, y, z])
            q = dot(K, X)
            q = [int(q[0]/q[2]), int(q[1]/q[2])]
            cv.Set2D(im, im.height-q[1], q[0], color)

def detect(image):
    image_size = cv.GetSize(image)

    # create grayscale version
    grayscale = cv.CreateImage(image_size, 8, 1)
    cv.CvtColor(image, grayscale, cv.CV_BGR2GRAY)
    storage = cv.CreateMemStorage(0)

    im = cv.CreateImage (image_size, 8, 3)

    status, corners = cv.FindChessboardCorners( grayscale, (dim,dim))

```

```

    if status:
        cv.DrawChessboardCorners( image, (dim,dim), corners, status)
        is_x = [p[0] for p in corners]
        is_y = [p[1] for p in corners]
        return is_x, is_y
    return [], []

def show_data(image, mu_x):
    line_type = cv.CV_AA
    pt1 = (30, 400)
    font = cv.InitFont (cv.CV_FONT_HERSHEY_SIMPLEX,
                        0.8, 0.1, 0, 1, cv.CV_AA)
    cv.PutText (image, "Particle Filter " + str(mu_x), pt1
                , font, cv.CV_RGB(255,255,0))

if __name__ == "__main__":

    snap_no = 0
    frame_no = 0

    # create windows
    cv.NamedWindow('Camera')

    # create capture device
    device = 0 # assume we want first device

    capture = cv.CreateFileCapture (sys.argv[1])
    dim = 3
    forward_step = -2.

    pts = dim * dim
    mid = int(pts / 2)

    cv.SetCaptureProperty(capture, cv.CV_CAP_PROP_FRAME_WIDTH, 640)
    cv.SetCaptureProperty(capture, cv.CV_CAP_PROP_FRAME_HEIGHT, 480)

    # check if capture device is OK
    if not capture:
        print "Error opening capture device"
        sys.exit(1)

    pf = PF(K, 200)

    frame = cv.QueryFrame(capture)
    proj_board(frame, 1, 1, 160)

```

```

cv.ShowImage('Camera', frame)

while 1:
    frame_no += 1
    frame = cv.QueryFrame(capture)

    image_size = cv.GetSize(frame)
    if frame is None:
        break

    is_x, is_y = detect(frame)

    if len(is_x) > 0:
        pf.update(array([is_x[5], frame.height-is_y[5], 1.]))
        mu_x = pf.average()
        proj_board(frame, mu_x[0], mu_x[1], mu_x[2])
        show_data(frame, mu_x)

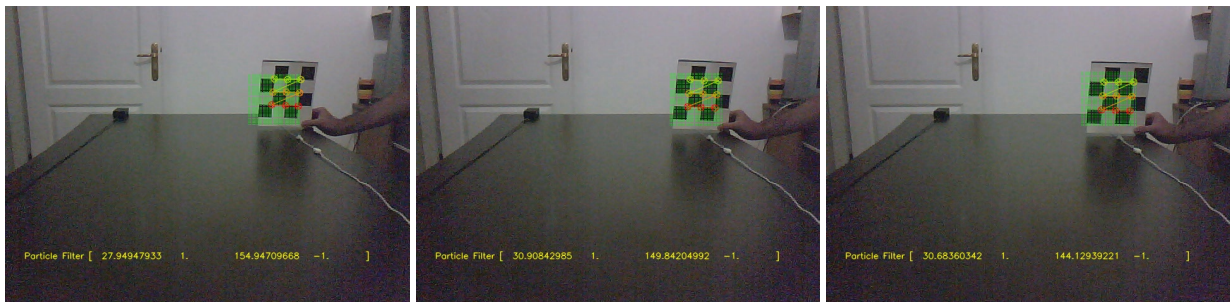
    # display webcam image
    cv.ShowImage('Camera', frame)

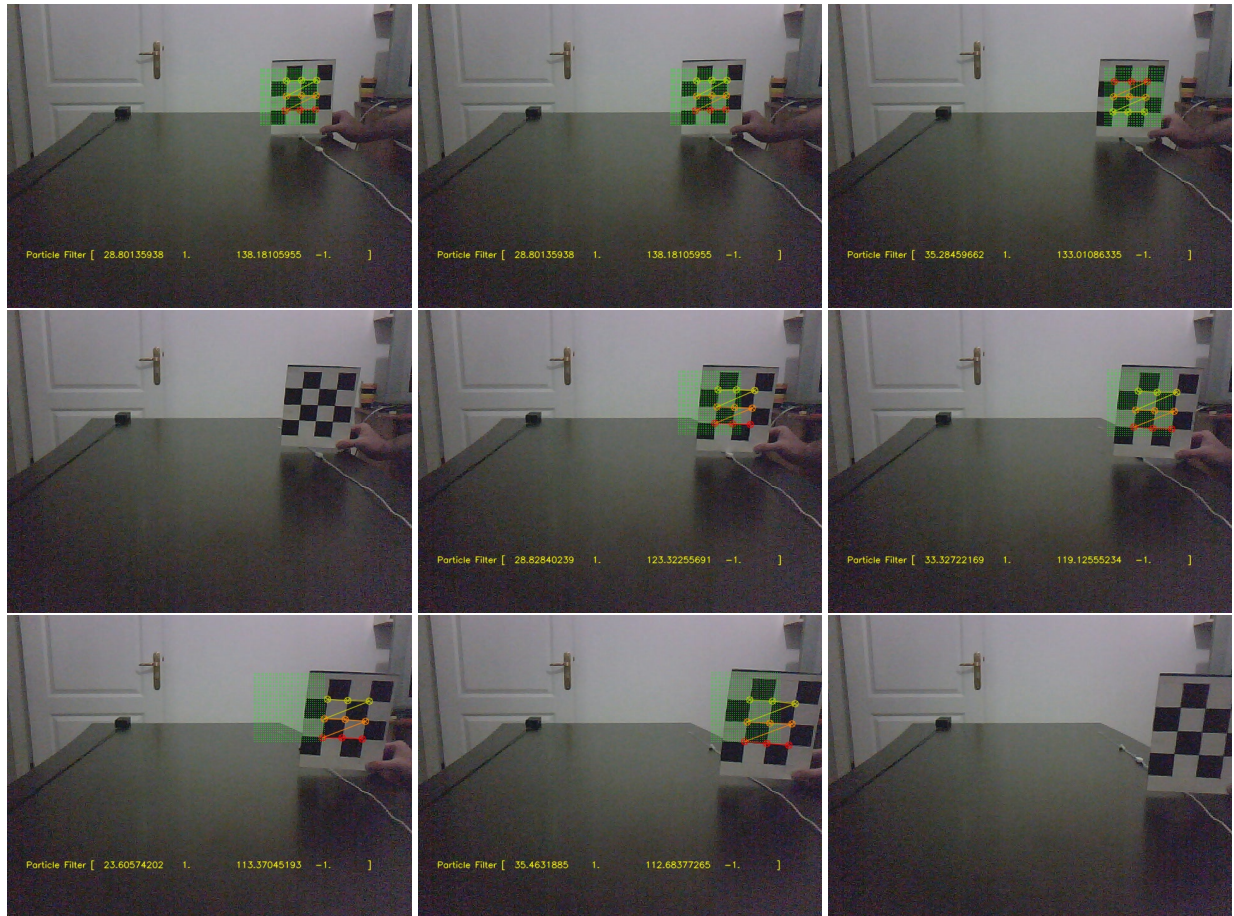
    if snap_no == 12: break
    if frame_no % 10 == 0:
        cv.SaveImage('cb-pf-' + str(snap_no) + '.jpg', frame)
        snap_no += 1

    # handle events
    k = cv.WaitKey(40)
    if k == 27: # ESC
        print 'ESC pressed. Exiting ...'
        break

```

! python run_filter.py pf





Kaynaklar

<http://dl.dropbox.com/u/1570604/skfiles/campy/chessb-left.avi>

<http://dl.dropbox.com/u/1570604/skfiles/campy/chessb-right.avi>

S. Marsland, Machine Learning: An Algorithmic Perspective, CRC Press, 2009.

S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics, MIT Press, Cambridge, MA, 2005

C. Bishop Pattern Recognition and Machine Learning , 2006.

Rabiner L. R. , A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Proceedings of IEEE vol. 77, no. 2, pp. 257-286, 1989.

<ftp://ftp.cs.toronto.edu/pub/zoubin/tr96-2.ps.gz>, Department of Computer Science, University of Toronto, 1996.

Shumway R., H. S. Stoffer Time series analysis and its applications 2nd Edition, New York, Springer, (Springer texts in statistics), 2000.

Jordan M. I. , C. Bishop An Introduction to Graphical Models, Not yet published, 2000.

Kalman R. E., A New Approach to Linear Filtering and Prediction Problems, Trans-

actions of the ASME-Journal of Basic Engineering, 82 (Series D): 35-45, 1960.

Kalman, R.E. and R.S. Bucy, New results in filtering and prediction theory, Trans. ASME J. Basic Eng., 83, 95-108, 1961.

Welling, M., The Kalman Filter - Lecture Tutorial, California Institute of Technology, 2008.

Lall, S., Modern Control 2 Lecture Notes, Stanford University, 2006.

Quantitative Economics Lecture Notebooks - <http://quant-econ.net/kalman.html>