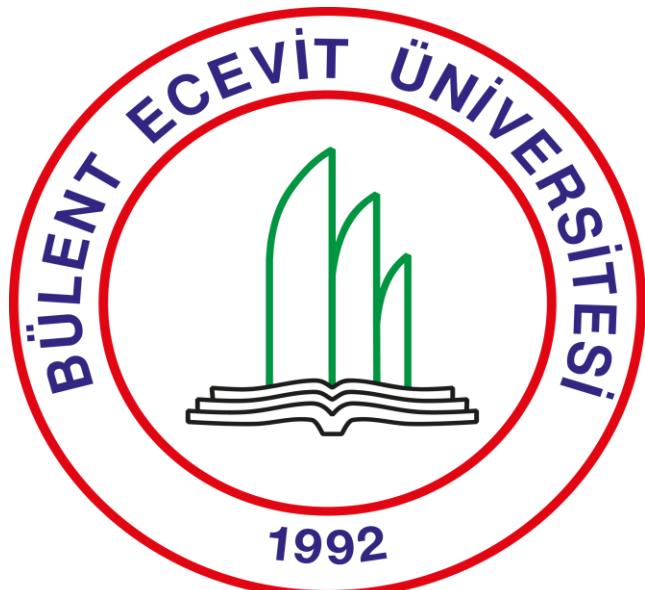


PYTHON İLE OPENCV

ZONGULDAK BÜLENT ECEVİT ÜNİVERSİTESİ



MÜHENDİSLİK FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

SALİH ÖZTÜRK

HAZİRAN 2020

1-)OpenCV-Python Eğiticilerine Giriş OpenCV

OpenCV, 1999 yılında Intel'de **Gary Bradsky** tarafından başlatıldı ve ilk sürüm 2000 yılında çıktı. **Vadim Pisarevsky**, Intel'in Rus yazılımı OpenCV ekibini yönetmek için **Gary Bradsky'ye** katıldı. 2005 yılında, 2005 DARPA Büyük Mücadelesini kazanan araç Stanley'de OpenCV kullanıldı. Daha sonra Willow Garage ve Gary Bradsky ve Vadim Pisarevsky'nin projeye liderlik etmesiyle aktif gelişimi devam etti. OpenCV artık Bilgisayarla Görme ve Makine Öğrenimi ile ilgili çok sayıda algoritmayı destekliyor ve her geçen gün genişliyor.

OpenCV, C++, Python, Java, vb. Gibi çok çeşitli programlama dillerini destekler ve Windows, Linux, OS X, Android ve iOS gibi farklı platformlarda kullanılabilir. CUDA ve OpenCL tabanlı yüksek hızlı GPU işlemleri için arayüzler de aktif geliştirme aşamasındadır.

OpenCV-Python, OpenCV için Python API'sı olup OpenCV C++ API'sinin ve Python dilinin en iyi özelliklerini bir araya getirir.

OpenCV-Python

OpenCV-Python, bilgisayar görme sorunlarını çözmek için tasarlanmış bir Python bağlamaları kütüphanesidir.

Python, **Guido van Rossum** tarafından başlatılan, basitliği ve kod okunabilirliği nedeniyle çok hızlı bir şekilde popüler hale gelen genel amaçlı bir programlama dilidir. Programcının, okunabilirliği azaltmadan fikirleri daha az kod satırında ifade etmesini sağlar.

C / C++ gibi dillerle karşılaşıldığında, Python daha yavaştır. Bununla birlikte, Python, C / C++ 'da hesaplama yoğun kod yazmamızı ve Python modülleri olarak kullanılabilen Python sarmalayıcıları oluşturmamızı sağlayan C / C++ ile kolayca genişletilebilir. Bu bize iki avantaj sağlar: birincisi, kod orijinal C / C++ kodu kadar hızlıdır (arka planda çalışan gerçek C++ kodu olduğu için) ve ikincisi, Python'da kodlamak C / C++ 'dan daha kolaydır. OpenCV-Python, orijinal OpenCV C++ uygulaması için bir Python sarıcıdır.

OpenCV-Python, MATLAB stili bir sözdizimiyle sayısal işlemler için oldukça optimize edilmiş bir kitaplık olan **Numpy'yi** kullanır. Tüm OpenCV dizi yapıları Numpy dizilerine / dizilerinden dönüştürülür. Bu aynı zamanda SciPy ve Matplotlib gibi Numpy kullanan diğer kütüphanelerle entegrasyonu kolaylaştırır.

OpenCV-Python Eğiticileri

OpenCV, OpenCV-Python'da bulunan çeşitli işlevler konusunda size rehberlik edecek yeni bir eğitim seti sunar. **Bu kılavuz esas olarak OpenCV 3.x sürümüne odaklanmıştır** (eğiticilerin çoğu OpenCV 2.x ile de çalışacaktır). Bu kılavuzda ele alınmayacakları için Python ve Numpy hakkında önceden bilgi sahibi olmanız önerilir. **OpenCV-Python kullanarak optimize edilmiş kod yazmak için Numpy ile yeterlilik şarttır.**

Bu eğitim ilk olarak *Abid Rahman K.* tarafından *Alexander Mordvintsev* rehberliğinde 2013 Google Kod Yaz 2013 programının bir parçası olarak *başlatılmıştır*.

OpenCV'nin İhtiyacı Var !!!

OpenCV açık kaynaklı bir girişim olduğundan, herkes kütüphaneye, belgelere ve eğiticilere katkıda bulunabilir. Bu eğitimde herhangi bir hata bulursanız (küçük bir yazım hatasından kod veya kavramdaki korkunç bir hataya), [GitHub'da OpenCV'yi](#) klonlayarak ve bir çekme isteği göndererek bunu düzeltmekten çekinmeyin. OpenCV geliştiricileri çekme talebinizi kontrol edecek, size önemli geri bildirimler verecek ve (gözden geçirenin onayını geçtikten sonra) OpenCV ile birleştirilecektir. Daha sonra açık kaynaklı bir katkıda bulunacaksınız :-)

OpenCV-Python'a yeni modüller eklendiğinden, bu öğreticinin genişletilmesi gerekecektir. Belirli bir algoritmaya aşina iseniz ve algoritmanın temel teorisini ve örnek kullanımı gösteren kodu içeren bir öğretici yazabiliyorsanız lütfen bunu yapın.

Unutmayın, **birlikte** bu projeyi büyük bir başarı yapabiliriz !!!

katkıda

Aşağıda OpenCV-Python'a eğitim sunan katılımcıların listesi verilmiştir.

1. Alexander Mordvintsev (GSoC-2013 akıl hocası)
2. Abid Rahman K. (GSoC-2013 stajyeri)

Ek kaynaklar

1. Python için kısa bir kılavuz – [A Byte of Python](#)
2. [Temel Numpy Eğitimleri](#)
3. [Numpy Örnek Listesi](#)
4. [OpenCV Belgeleri](#)
5. [OpenCV Forumu](#)

2.1-) Görsel okuma

Görüntüyü okumak için `cv2.imread()` işlevini kullanın. Görüntü çalışma dizininde olmalı veya tam bir görüntü yolu verilmelidir.

İkinci argüman, görüntünün nasıl okunacağını belirten bir işaretir.

- `cv2.IMREAD_COLOR`: Renkli bir görüntü yükler. Görüntünün saydamlığı ihmal edilir. Varsayılan bayraktır.
- `cv2.IMREAD_GRAYSCALE`: Görüntüyü gri tonlamalı modda yükler

- cv2.IMREAD_UNCHANGED: Alfa kanalı dahil olmak üzere resmi yükler

Not Bu üç bayrak yerine, sırasıyla 1, 0 veya -1 tam sayılarını iletebilirsiniz.

See the code below:

```
import numpy as np
import cv2

# Load an color image in grayscale
img = cv2.imread('messi5.jpg',0)
```

Bir görüntü göster

Bir görüntüyü pencerede görüntülemek için **cv2.imshow()** işlevini kullanın. Pencere otomatik olarak görüntü boyutuna sığar.

İlk argüman bir dize olan bir pencere adıdır. ikinci argüman bizim imajımızdır. Farklı pencere adlarıyla istediğiniz kadar pencere oluşturabilirsiniz.

```
cv2 . imshow ( 'image' , img )
cv2 . waitKey ( 0 )
cv2 . destroyAllWindows ()
```

Pencerenin ekran görüntüsü şu şekilde görünecektir



cv2.waitKey() bir klavye bağlama işlevidir. Argümanı milisaniye cinsinden zamandır. İşlev, herhangi bir klavye olayı için belirtilen milisaniye kadar bekler. Bu süre içinde herhangi bir

tuşa basarsanız, program devam eder. Eğer 0 geçirilir, bir anahtar inme için süresiz bekler. Ayrıca, aşağıda tartışacağımız *a* tuşuna basılırsa vb. Gibi belirli tuş vuruşlarını tespit etmek için de ayarlanabilir .

Not Bağlanan klavye olaylarının yanı sıra, bu işlev diğer birçok GUI olayını da işler, bu nedenle görüntüyü gerçekten görüntülemek için kullanmanız GEREKİR.

cv2.destroyAllWindows (), oluşturduğumuz tüm pencereleri yok eder. Belirli bir pencereyi yok etmek istiyorsanız, tam pencere adını bağımsız değişken olarak **ilettiğiniz cv2.destroyWindow ()** işlevini kullanın .

Not Zaten bir pencere oluşturabileceğiniz ve daha sonra görüntüyü yükleyebileceğiniz özel bir durum vardır. Bu durumda, pencerenin yeniden boyutlandırılabilir olup olmadığını belirtebilirsiniz. **Cv2.namedWindow ()** işlevi ile yapılır . Varsayılan olarak, bayrak **cv2.WINDOW_AUTOSIZE** . Ancak bayrağı **cv2.WINDOW_NORMAL** olarak **belirtirseniz** pencereyi yeniden boyutlandırılabilirsiniz. Görüntü boyut olarak çok büyük olduğunda ve pencerelere izleme çubuğu eklediğinde yardımcı olacaktır.

Aşağıdaki koda bakın:

```
cv2.namedWindow('image', cv2.WINDOW_NORMAL)
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Bir resim yazın

Görüntüyü kaydetmek için **cv2.imwrite ()** işlevini kullanın .

İlk argüman dosya adı, ikinci argüman ise kaydetmek istediğiniz görüntündür.

```
cv2.imwrite('messigray.png',img)
```

Bu işlem görüntüyü PNG formatında çalışma dizinine kaydeder.

Özetle

Programın altındaki bir görüntü gri tonlamalı olarak yüklenir, görüntülenir, 's' tuşuna basarsanız ve çıkışsanız görüntüyü kaydeder veya *ESC* tuşuna basarsanız kaydetmeden çıkar .

```
import numpy as np
import cv2

img = cv2.imread('messi5.jpg',0)
cv2.imshow('image',img)
k = cv2.waitKey(0)
if k == 27:          # wait for ESC key to exit
    cv2.destroyAllWindows()
elif k == ord('s'): # wait for 's' key to save and exit
    cv2.imwrite('messigray.png',img)
    cv2.destroyAllWindows()
```

Uyarı 64 bit makine kullanıyorsanız, `k = cv2.waitKey (0)` satırını aşağıdaki gibi değiştirmeniz gereklidir : `k = cv2.waitKey (0) ve 0xFF`

Matplotlib kullanma

Matplotlib size çok çeşitli çizim yöntemleri sunan Python için bir çizim kütüphanesidir. Onları gelecek makalelerde göreceksiniz. Burada, Matplotlib ile görüntünün nasıl görüntüleneceğini öğreneceksiniz. Matplotlib kullanarak görüntüleri yakınlaştırabilir, kaydedebilirsiniz.

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

img = cv2.imread('messi5.jpg',0)
plt.imshow(img, cmap = 'gray', interpolation = 'bicubic')
plt.xticks([]), plt.yticks([]) # to hide tick values on X and Y axis
plt.show()
```

➔ Açıklama X ve Y ekseni plt üzerindeki tik değerlerini gizlemek için
yticks ([]) #

Uyarı OpenCV tarafından yüklenen renkli görüntü BGR modunda. Ancak Matplotlib RGB modunda görüntülenir. Bu nedenle, görüntü OpenCV ile okunursa renkli görüntüler Matplotlib'de düzgün görüntülenmez. Daha fazla bilgi için lütfen alıştırmalara bakınız.

2.2-)Kameradan Video Çekin

Genellikle, canlı akışı kamera ile yakalamamız gereklidir. OpenCV buna çok basit bir arayüz sağlar. Kameradan bir video çekelim (dizüstü bilgisayarımın yerleşik web kamerasını kullanıyorum), gri tonlamalı videoya dönüştürelim ve görüntüleyelim. Başlamak için basit bir görev.

Video yakalamak için bir **VideoCapture** nesnesi oluşturmanız gereklidir. Argümanı, aygit dizini veya bir video dosyasının adı olabilir. Cihaz dizini yalnızca hangi kamerayı belirleyeceğiniza söylemeyecektir. Normalde bir kamera bağlanır (benim durumumda olduğu gibi). Bu yüzden sadece 0 (veya -1) değerini geçiyorum. İkinci kamerayı 1 ve benzerlerini geçerek seçebilirsiniz. Bundan sonra, kare kare yakalayabilirsiniz. Ama sonunda, yakalamayı bırakmayı unutma.

```
import numpy as np
import cv2

cap = cv2.VideoCapture(0)

while(True):
    # Capture frame-by-frame
    ret, frame = cap.read()

    # Our operations on the frame come here
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Display the resulting frame
    cv2.imshow('frame',gray)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# When everything done, release the capture
cap.release()
cv2.destroyAllWindows()
```

`cap.read()` bir bool (True / False) döndürür. Çerçeve doğru okunursa, Doğru olur. Bu nedenle, bu dönüş değerini kontrol ederek videonun sonunu kontrol edebilirsiniz.

Bazen, `cap` yakalamayı başlatmamış olabilir. Bu durumda, bu kod hatayı gösterir. Başlatılmış başlatılmadığını `cap.isOpened()` yöntemiyle kontrol edebilirsiniz. Doğru ise, tamam. Aksi takdirde `cap.open()` kullanarak açın.

Bu videonun bazı özelliklerine `propId'nin` (Properties id)0 ile 18 arasında bir sayı olduğu `cap.get(propId)` yöntemini kullanarak da erişebilirsiniz. Her sayı videonun bir özelliğini (bu video için geçerliyse) gösterir ve tüm ayrıntılar burada görülebilir: [Özellik Tanımlayıcısı](#). Bu değerlerin bazıları `cap.set(propId, value)` kullanılarak değiştirilebilir. Değer, istediğiniz yeni değerdir.

Örneğin, çerçeve genişliğini ve yüksekliğini `cap.get(3)` ve `cap.get(4)` ile kontrol edebilirim. Bana varsayılan olarak 640x480 veriyor. Ama 320x240 olarak değiştirmek istiyorum. Sadece kullanmak `ret = cap.set(3320)` ve `ret = cap.set(4240)`.

Dosyadan Video Oynatma

Kameradan yakalama ile aynıdır, sadece kamera dizinini video dosyası adıyla değiştirin. Ayrıca çerçeveyi görüntülerken `cv2.waitKey()` için uygun zamanı kullanın. Çok azsa, video çok hızlı olacak ve çok yüksekse video yavaş olacaktır (Evet, videoları ağır çekimde bu şekilde görüntüleyebilirsiniz). Normal durumlarda 25 milisaniye yeterli olacaktır.

```
import numpy as np
import cv2

cap = cv2.VideoCapture('vtest.avi')

while(cap.isOpened()):
    ret, frame = cap.read()

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    cv2.imshow('frame',gray)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Not Ffmpeg veya gstreamer'in uygun sürümlerinin kurulu olduğundan emin olun. Bazen, çoğunlukla ffmpeg / gstreamer'in yanlış kurulumu nedeniyle Video Yakalama ile çalışmak bir baş ağırsıdır.

Video Kaydetme

Bu yüzden bir video çekiyoruz, kare kare işliyoruz ve bu videoyu kaydettmek istiyoruz. Görüntüler için çok basit, sadece `cv2.imwrite()` kullanın. Burada biraz daha çalışma gerekiyor.

Bu sefer bir **VideoWriter** nesnesi oluşturuyoruz. Çıktı dosya adını belirtmeliyiz (örneğin: output.avi). O zaman **FourCC** kodunu belirtmeliyiz (sonraki paragrafta ayrıntılar). Ardından saniyedeki kare sayısı (fps) ve kare boyutu geçirilmelidir. Ve sonucusu **isColor** bayrağı. True ise, enkoder renkli çerçeve bekler, aksi takdirde gri tonlamalı çerçeve ile çalışır.

FourCC, video codec bileşenini belirtmek için kullanılan 4 baylıklı bir koddur. Mevcut kodların listesi fourcc.org adresinde bulunabilir. Platforma bağlıdır. Aşağıdaki codec bileşenleri benim için iyi çalışıyor.

FourCC kodu, MJPG için `cv2.VideoWriter_fourcc ('M', 'J', 'P', 'G')` veya `cv2.VideoWriter_fourcc (* 'MJPG')` olarak geçirilir.

Kameradan kod yakalamanın altında, her kareyi dikey yönde çevirin ve kaydeder.

```

import numpy as np
import cv2

cap = cv2.VideoCapture(0)

# Define the codec and create VideoWriter object
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))

while(cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
        frame = cv2.flip(frame,0)

        # write the flipped frame
        out.write(frame)

        cv2.imshow('frame',frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break

# Release everything if job is finished
cap.release()
out.release()
cv2.destroyAllWindows()

```

2.3-)OpenCV'de Çizim İşlevleri Hedef

- OpenCV ile farklı geometrik şekiller çizmeyi öğrenin
- Bu işlevleri öğreneceksiniz: `cv2.line ()` , `cv2.circle ()` , `cv2.rectangle ()` , `cv2.ellipse ()` , `cv2.putText ()` vb.

kod

Yukarıdaki tüm işlevlerde, aşağıda verilen bazı ortak argümanları göreceksiniz:

- img: Şekil çizmek istediğiniz görüntü
- renk: Şeklin rengi. BGR için bir demet olarak geçirin , örneğin: `(255,0,0)` mavi için. Gri tonlama için sadece skaler değeri iletin.
- kalınlık: Çizgi veya dairenin kalınlığı vb. Daireler gibi kapalı şekiller için `-1` geçirilirse, şekli dolduracaktır. `varsayılan kalınlık = 1`
- lineType: Hat tipi, 8 bağlantılı, kenar yumuşatmalı çizgi vb . `Varsayılan olarak, 8 bağlantılıdır.` `cv2.LINE_AA` eğriler için harika görünen kenar yumuşatma çizgisi verir.

Çizim Hattı

Çizgi çizmek için çizginin başlangıç ve bitiş koordinatlarını geçmeniz gereklidir. Siyah bir görüntü oluşturacağınız ve üzerine sol üstten sağ alt köşelere mavi bir çizgi çizeceğiz.

```
import numpy as np
import cv2

# Create a black image //siyah görüntü oluşturma
img = np.zeros((512,512,3), np.uint8)

# Draw a diagonal blue line with thickness of 5 px 5 pixel kalınlığında çapraz mavi çizgi çizdi
cv2.line(img,(0,0),(511,511),(255,0,0),5)
```

Dikdörtgen Çizimi

Bir dikdörtgen çizmek için, dikdörtgenin sol üst köşesine ve sağ alt köşesine ihtiyacınız vardır. Bu kez görüntünün sağ üst köşesine yeşil bir dikdörtgen çizeceğiz.

```
cv2.rectangle(img,(384,0),(510,128),(0,255,0),3)
```

Çizim Çemberi

Bir daire çizmek için merkez koordinatlarına ve yarıçapına ihtiyacınız vardır. Yukarıda çizilen dikdörtgenin içine bir daire çizeceğiz.

```
cv2.circle(img,(447,63), 63, (0,0,255), -1)
```

Elips Çizim

Elips çizmek için birkaç argüman geçirmemiz gereklidir. Bir argüman merkez konumudur (x, y). Bir sonraki argüman eksen uzunluklarıdır (ana eksen uzunluğu, küçük eksen uzunluğu). açı , elipsin saat yönünün tersine dönme açısıdır. startAngle ve endAngle , ana eksenden saat yönünde ölçülen elips arkının başlangıcını ve sonunu belirtir. yani 0 ve 360 değerleri vermek tam elipsi verir. Daha fazla ayrıntı için **cv2.ellipse()** belgelerine bakın . Aşağıdaki örnek görüntünün ortasına yarımlı elips çizer.

Çokgen Çizimi

Bir çokgen çizmek için, önce köşelerin koordinatlarına ihtiyacınız vardır. Bu noktaları ROWSx1x2 şeklinde bir dizi haline getirin ; burada ROWS köşe sayısıdır ve int32 türünde olmalıdır . Burada sarı renkli dört köşeli küçük bir çokgen çiziyoruz.

```
pts = np.array([[10,5],[20,30],[70,20],[50,10]], np.int32)
pts = pts.reshape((-1,1,2))
cv2.polylines(img,[pts],True,(0,255,255))
```

Not Üçüncü argüman False ise , kapalı bir şekil değil, tüm noktalara katılan bir çok hat alırsınız.

Not `cv2.polyline()` birden çok çizgi çizmek için kullanılabilir. Sadece çizmek istediğiniz tüm satırların bir listesini oluşturun ve fonksiyona iletin. Tüm çizgiler ayrı ayrı çizilecektir. Bir satır çizmek, her satır için `cv2.line()` ögesini çağrımdan çok daha iyi ve hızlı bir yoldur.

Görüntülere Metin Ekleme:[¶]

Resimlere metin koymak için aşağıdaki şeyleri belirtmeniz gereklidir.

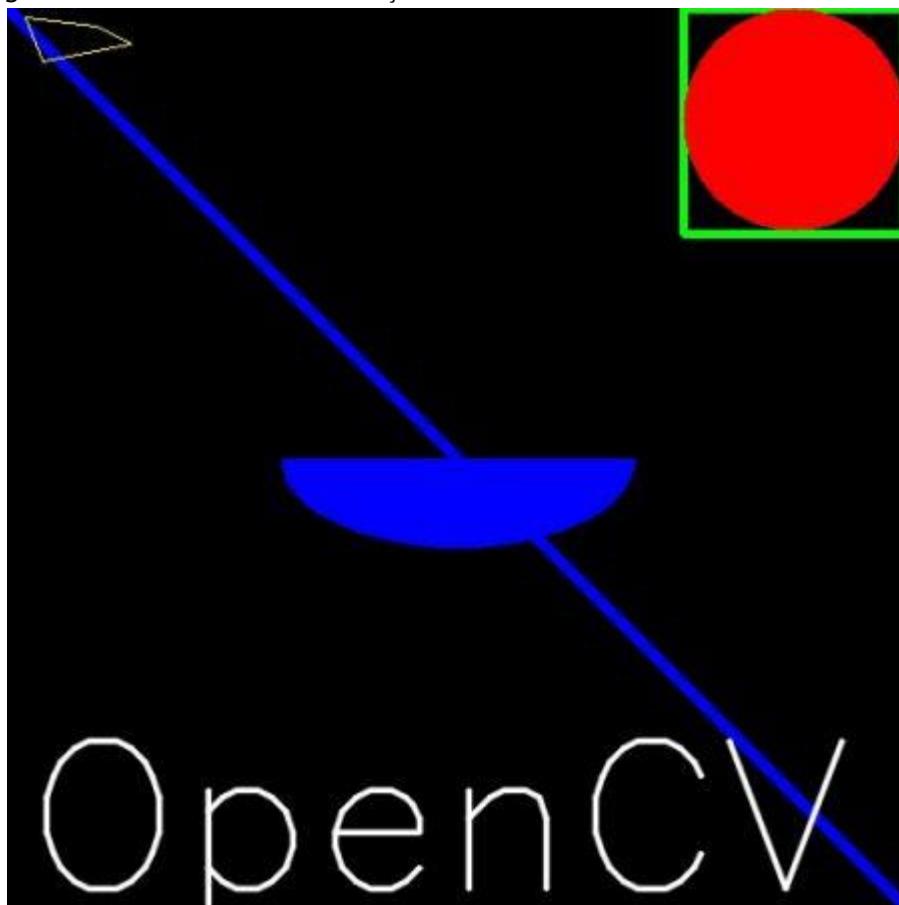
- Yazmak istediğiniz metin verileri
- Koymak istediğiniz yerin koordinatlarını konumlandırın (örn. Verinin başladığı sol alt köşe).
- Yazı tipi (Desteklenen yazı tipleri için `cv2.putText()` belgelerini kontrol edin)
- Yazı Tipi Ölçeği (yazı tipinin boyutunu belirtir)
- Daha iyi görünüm için renk, kalınlık, çizgi tipi vb gibi düzenli şeyler `Linetype = cv2.LINE_AA` önerilir.

Resmimize beyaz renkte OpenCV yazacağız .

```
font = cv2.FONT_HERSHEY_SIMPLEX  
cv2.putText(img, 'OpenCV',(10,500), font, 4,(255,255,255),2,cv2.LINE_AA)
```

Sonuç

Yani çizimizin son sonucunu görmenin zamanı geldi. Önceki makalelerde çalışığınız gibi, görmek için resmi görüntüleyin.



2.4-) Bir Boya Fırçası Olarak Fare

Hedef

- OpenCV'de fare olaylarını yönetmeyi öğrenin
- Bu işlevleri öğreneceksiniz: **cv2.setMouseCallback()**

Basit Demo

Burada, görüntüyü çift tıkladığımız her yere daire çizen basit bir uygulama oluşturuyoruz.

İlk olarak, bir fare olayı gerçekleştiğinde yürütülen bir fare geri arama işlevi yaratırız. Fare olayı, fare ile ilgili sol düğme aşağı, sol düğme yukarı, sol düğme çift tıklama vb. Gibi herhangi bir şey olabilir. Her fare olayı için koordinatları (x, y) verir. Bu etkinlik ve konumla istediğimizi yapabiliriz. Mevcut tüm olayları listelemek için Python terminalinde aşağıdaki kodu çalıştırın:

```
>>> import cv2  
>>> events = [i for i in dir(cv2) if 'EVENT' in i]  
>>> print events
```

Fare geri arama işlevi oluşturmanın her yerde aynı olan belirli bir biçimini vardır. Sadece işlevin ne yaptığından farklıdır. Fare geri arama fonksiyonumuz bir şey yapar, çift tıkladığımız bir daire çizer. Bu yüzden aşağıdaki koda bakın. Kod yorumlardan kendini açıklamaktadır:

```
import cv2  
import numpy as np  
  
# mouse callback function // fare geri çağrıma işlevi  
def draw_circle(event,x,y,flags,param):  
    if event == cv2.EVENT_LBUTTONDOWN:  
        cv2.circle(img,(x,y),100,(255,0,0),-1)  
  
# Create a black image, a window and bind the function to window  
Siyah bir görüntü, bir pencere oluşturun ve işlevi pencereye bağlayın  
img = np.zeros((512,512,3), np.uint8)  
cv2.namedWindow('image')  
cv2.setMouseCallback('image',draw_circle)  
  
while(1):  
    cv2.imshow('image',img)  
    if cv2.waitKey(20) & 0xFF == 27:  
        break  
cv2.destroyAllWindows()
```

Daha Gelişmiş Demo

Şimdi çok daha iyi bir uygulamaya geçiyoruz. Burada, Paint uygulamasında yaptığımız gibi fareyi sürükleyerek dikdörtgenler veya daireler çiziyoruz (seçtiğimiz moda bağlı olarak). Bu nedenle, fare geri arama işlevimizin biri dikdörtgen çizmek, diğer de daireler çizmek için iki bölümü vardır. Bu özel örnek, nesne izleme, görüntü segmentasyonu vb. Gibi bazı etkileşimli uygulamalar oluşturma ve anlamada yardımcı olacaktır.

```
import cv2
import numpy as np

drawing = False # true if mouse is pressed
mode = True # if True, draw rectangle. Press 'm' to toggle to curve
ix,iy = -1,-1

# mouse callback function
def draw_circle(event,x,y,flags,param):
    global ix,iy,drawing,mode

    if event == cv2.EVENT_LBUTTONDOWN:
        drawing = True
        ix,iy = x,y

    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing == True:
            if mode == True:
                cv2.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
            else:
                cv2.circle(img,(x,y),5,(0,0,255),-1)

    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        if mode == True:
            cv2.rectangle(img,(ix,iy),(x,y),(0,255,0),-1)
        else:
            cv2.circle(img,(x,y),5,(0,0,255),-1)
```

Sonra bu fare geri arama işlevini OpenCV penceresine bağlamak zorundayız. Ana döngüde, dikdörtgen ve daire arasında geçiş yapmak için 'm' tuşunun klavye bağlamasını ayarlamalıyız.

```
img = np.zeros((512,512,3), np.uint8)
cv2.namedWindow('image')
cv2.setMouseCallback('image',draw_circle)

while(1):
    cv2.imshow('image',img)
    k = cv2.waitKey(1) & 0xFF
    if k == ord('m'):
        mode = not mode
    elif k == 27:
```

```
break
```

```
cv2.destroyAllWindows()
```

2.5-)Renk Paleti Olarak İzleme Çubuğu Hedef

- İzleme çubuğunu OpenCV pencerelerine bağlamayı öğrenin
- Bu işlevleri öğreneceksiniz: `cv2.getTrackbarPos()` , `cv2.createTrackbar()` vb.

Kod Demosu

Burada belirttiğiniz rengi gösteren basit bir uygulama oluşturacağız. B, G, R renklerinin her birini belirtmek için rengi ve üç izleme çubuğunu gösteren bir pencereniz var. İzleme çubuğunu kaydırırsınız ve buna bağlı olarak pencere rengi değişir. Varsayılan olarak, başlangıç rengi Siyah olarak ayarlanır.

`Cv2.getTrackbarPos()` işlevi için, ilk argüman izleme çubuğu adı, ikincisi eklendiği pencere adı, üçüncü argüman varsayılan değer, dördüncü maksimum değer ve beşinci yürüttülen geri arama işlevidir her zaman izleme çubuğu değeri değişir. Geri arama işlevi her zaman izleme çubuğu konumu olan varsayılan bir bağımsız değişkene sahiptir. Bizim durumumuzda, işlev hiçbir şey yapmaz, bu yüzden basitçe geçeriz.

İzleme çubuğunun bir başka önemli uygulaması, onu bir düğme veya anahtar olarak kullanmaktır. OpenCV, varsayılan olarak düğme işlevine sahip değildir. Böylece bu işlevi almak için izleme çubuğunu kullanabilirsiniz. Uygulamamızda, uygulamanın yalnızca anahtar AÇIK konumdayken çalıştığı bir anahtar oluşturduk, aksi takdirde ekran her zaman siyah.

```
import cv2
import numpy as np

def nothing(x):
    pass

# Create a black image, a window
img = np.zeros((300,512,3), np.uint8)
cv2.namedWindow('image')

# create trackbars for color change
cv2.createTrackbar('R','image',0,255,nothing)
cv2.createTrackbar('G','image',0,255,nothing)
cv2.createTrackbar('B','image',0,255,nothing)

# create switch for ON/OFF functionality
switch = '0 : OFF \n1 : ON'
cv2.createTrackbar(switch, 'image',0,1,nothing)

while(1):
    cv2.imshow('image',img)
    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break
```

```

# get current positions of four trackbars
r = cv2.getTrackbarPos('R','image')
g = cv2.getTrackbarPos('G','image')
b = cv2.getTrackbarPos('B','image')
s = cv2.getTrackbarPos(switch,'image')

if s == 0:
    img[:] = 0
else:
    img[:] = [b,g,r]

cv2.destroyAllWindows()

```

Uygulamanın ekran görüntüsü aşağıdaki gibi görünür:



3.1-) Görüntü Özelliklerine Erişim

Görüntü özellikleri satır, sütun ve kanal sayısını, görüntü verisi türünü, piksel sayısını vb. içerir.

Görüntünün şekline `img.shape` ile erişilir . Bir dizi satır, sütun ve kanal döndürür (görüntü renkli ise):

```

>>> print img.shape
(342, 548, 3)

```

NotGörüntü gri tonlamalısa, döndürülen demet yalnızca sayıda satır ve sütun içerir. Bu nedenle, yüklenen görüntünün gri tonlamalı mı yoksa renkli görüntü mü olduğunu kontrol etmek iyi bir yöntemdir.

Toplam piksel sayısına `img.size` ile erişilir :

```
>>> print img.size  
562248
```

Görüntü veri türü `img.dtype` ile elde edilir :

```
>>> print img.dtype  
uint8
```

Not `img.dtype` hata ayıklama sırasında çok önemlidir, çünkü OpenCV-Python kodundaki çok sayıda hata geçersiz veri tipinden kaynaklanır.

Resim YG'si

Bazen, görüntülerin belirli bölgeleriyle oynamanız gerekebilir. Görüntülerde göz algılama için, ilk yüz algılama görüntünün her yerinde yapılır ve yüz elde edildiğinde, yalnızca yüz bölgesini seçer ve tüm görüntüyü aramak yerine içindeki gözleri ararız. Doğruluğu artırır (çünkü gözler her zaman yüzlerde bulunur: D) ve performansı (çünkü küçük bir alan ararız)

ROI yine Numpy indeksleme kullanılarak elde edilir. Burada topu seçiyorum ve görüntüdeki başka bir bölgeye kopyalıyorum:

```
>>> top = img [ 280 : 340 , 330 : 390 ]  
>>> img [ 273 : 333 , 100 : 160 ] = top
```

Aşağıdaki sonuçları kontrol edin:



Görüntü Kanallarını Bölme ve Birleştirme

Bazen B, G, R görüntü kanalları üzerinde ayrı ayrı çalışmanız gereklidir. O zaman BGR görüntülerini tek düzlemlere bölmelisiniz. Ya da başka bir zaman, bu ayrı kanallara BGR görüntüsüne katılmanız gerekebilir. Bunu basitçe şu şekilde yapabilirsiniz:

```
>>> b,g,r = cv2.split(img)
>>> img = cv2.merge((b,g,r))
```

Veya

```
>>> b = img[:, :, 0]
```

Tüm kırmızı pikselleri sıfır getirmek istediğiniz varsayıyalım, bu şekilde bölünmeniz ve sıfır eşitlemeniz gerekmez. Numpy indekslemesini kullanabilirsiniz ve bu daha hızlıdır.

```
>>> img[:, :, 2] = 0
```

Uyarı `cv2.split()` maliyetli bir işlevdir (zaman açısından). Bu yüzden sadece ihtiyacınız varsa yapın. Aksi takdirde Numpy dizinine gidin.

Görüntüler İçin Sınır Oluşturma (Dolgu)

Resmin etrafında, fotoğraf çerçevesi gibi bir kenarlık oluşturmak istiyorsanız, `cv2.copyMakeBorder()` işlevini kullanabilirsiniz. Ancak, evrişim işlemi, sıfır dolgu vb. için daha fazla uygulaması vardır. Bu işlev aşağıdaki argümanları alır:

- **src** – giriş resmi
- **üst , alt , sol , sağ** – karşılık gelen yönlerdeki piksel sayısı olarak kenarlık genişliği
- **borderType** – Ne tür bir kenarlık ekleneceğini tanımlayan bayrak. Aşağıdaki tipler olabilir:
 - **cv2.BORDER_CONSTANT** – Sabit renkli bir kenarlık ekler. Değer bir sonraki argüman olarak verilmelidir.
 - **cv2.BORDER_REFLECT** – Kenarlık, kenarlık öğelerinin ayna yansımıası olacaktır, örneğin: *fedcba* / *abcdefgh* / *hgfedcb*
 - **cv2.BORDER_REFLECT_101** veya **cv2.BORDER_DEFAULT** – Yukarıdaki ile aynı, ancak bunun gibi küçük bir değişiklikle: *gfedcb* / *abcdefgh* / *gfedcba*
 - **cv2.BORDER_REPLICATE** – Son öğe şu şekilde çoğaltılar: *aaaaaaaa* / *abcdefgh* / *hhhhhhh*
 - **cv2.BORDER_WRAP** – Açıklayamıyorum, şöyle görünecek: *cdefgh* / *abcdefgh* / *abcdefg*
- **value** – Kenarlık türü `cv2` ise kenarlığın rengi. `BORDER_CONSTANT`

Aşağıda, daha iyi anlaşılmasında tüm bu kenarlık türlerini gösteren bir örnek kod verilmiştir:

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

BLUE = [255,0,0]

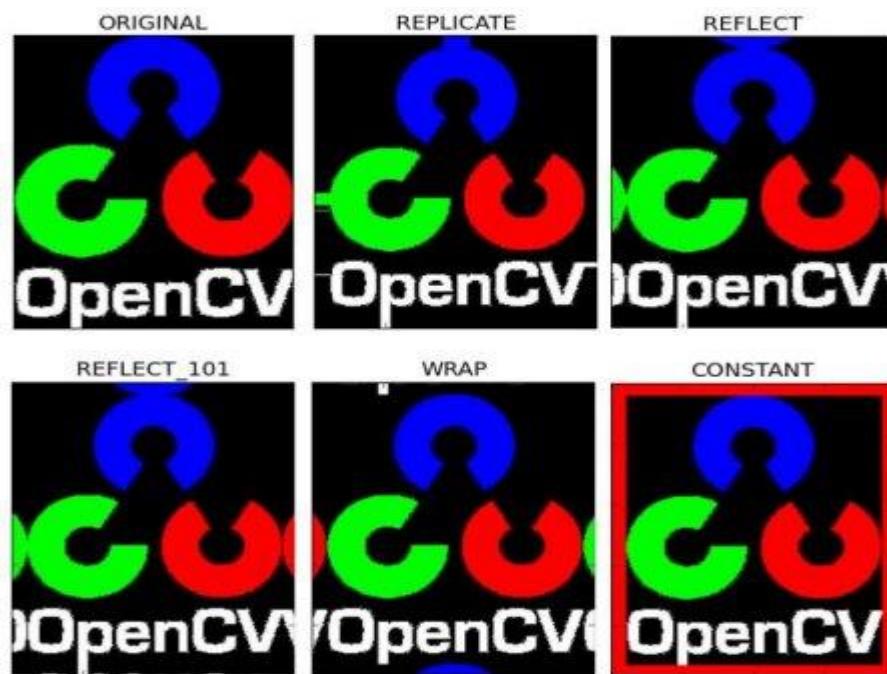
img1 = cv2.imread('opencv_logo.png')

replicate = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REPLICATE)
reflect = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT)
reflect101 = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT_101)
wrap = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_WRAP)
constant= cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_CONSTANT,value=BLUE)

plt.subplot(231),plt.imshow(img1,'gray'),plt.title('ORIGINAL')
plt.subplot(232),plt.imshow(replicate,'gray'),plt.title('REPLICATE')
plt.subplot(233),plt.imshow(reflect,'gray'),plt.title('REFLECT')
plt.subplot(234),plt.imshow(reflect101,'gray'),plt.title('REFLECT_101')
plt.subplot(235),plt.imshow(wrap,'gray'),plt.title('WRAP')
plt.subplot(236),plt.imshow(constant,'gray'),plt.title('CONSTANT')

plt.show()
```

Aşağıdaki sonuca bakın. (Görüntü matplotlib ile görüntülenir. Dolayısıyla KIRMIZI ve MAVİ düzlemler birbirinin yerine kullanılır):



3.2-) Görüntülerde Aritmetik İşlemler Hedef

- Toplama, çıkarma, bitsel işlemler vb. Görüntülerdeki çeşitli aritmetik işlemleri öğrenin.
- Bu işlevleri öğreneceksiniz: `cv2.add()`, `cv2.addWeighted()` vb.

Görüntü Ekleme

OpenCV işlevi, `cv2.add()` veya basitçe numpy işlemi, `res = img1 + img2` ile iki görüntü ekleyebilirsiniz. Her iki görüntü de aynı derinlikte ve türde olmalı veya ikinci görüntü sadece skaler bir değer olabilir.

Not OpenCV ilavesiyle Numpy ilacı arasında bir fark vardır. OpenCV ekleme doymuş bir işlemken Numpy ekleme bir modulo işlemidir.

Örneğin, aşağıdaki örneği göz önünde bulundurun:

```
>>> x = np.uint8([250])
>>> y = np.uint8([10])

>>> print cv2.add(x,y) # opencv 250+10 = 260 => 255
[[255]]

>>> print x+y          #numpy 250+10 = 260 % 256 = 4
[4]
```

İki resim eklediğinizde daha görünür olur. OpenCV işlevi daha iyi bir sonuç sağlayacaktır. Bu yüzden her zaman OpenCV işlevlerine daha iyi yapışır.

Görüntü Karıştırma

Bu aynı zamanda görüntü eklemesidir, ancak görüntülere farklı ağırlıklar verilir, böylece bir karıştırma veya şeffaflık hissi verir. Görüntüler aşağıdaki denklemeye göre eklenir:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

Değişken α olarak $0 \rightarrow 1$, bir görüntüden diğerine serin bir geçiş yapabilirsiniz.

Burada onları bir araya getirmek için iki resim çektim. İlk görüntüye 0.7 ağırlık, ikinci görüntüye 0.3 ağırlık verilmiştir. `cv2.addWeighted()`, görüntüdeki aşağıdaki denklemi uygular.

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

Burada γ sıfır olarak alınır.

```
img1 = cv2.imread('ml.png')
img2 = cv2.imread('opencv_logo.jpg')
```

```
dst = cv2.addWeighted(img1,0.7,img2,0.3,0)

cv2.imshow('dst',dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Aşağıdaki sonucu kontrol edin:



Bitsel İşlemler

Buna bit, AND, OR, NOT ve XOR işlemleri dahildir. Görüntünün herhangi bir parçasını çıkarırken (gelecek bölümlerde göreceğimiz gibi), dikdörtgen olmayan YG'yi tanımlarken ve bunlarla çalışırken çok faydalı olacaklar. Aşağıda, görüntünün belirli bir bölgesini nasıl değiştireceğine dair bir örnek göreceğiz.

OpenCV logosunu görüntünün üstüne koymak istiyorum. İki resim eklersem renk değişecektir. Karıştırırsam şeffaf bir etki elde ederim. Ama opak olmasını istiyorum. Eğer dikdörtgen bir bölge olsaydı, son bölümde yaptığımız gibi yatırım getirisini kullanabilirdim. Ancak OpenCV logosu dikdörtgen bir şekil değildir. Böylece aşağıdaki gibi bitsel işlemlerle yapabilirsiniz:

```
# Load two images
img1 = cv2.imread('messi5.jpg')
img2 = cv2.imread('opencv_logo.png')

# I want to put logo on top-left corner, So I create a ROI
rows,cols,channels = img2.shape
roi = img1[0:rows, 0:cols ]

# Now create a mask of Logo and create its inverse mask also
img2gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(img2gray, 10, 255, cv2.THRESH_BINARY)
mask_inv = cv2.bitwise_not(mask)

# Now black-out the area of logo in ROI
img1_bg = cv2.bitwise_and(roi,roi,mask = mask_inv)

# Take only region of logo from logo image.
img2_fg = cv2.bitwise_and(img2,img2,mask = mask)
```

```
# Put Logo in ROI and modify the main image
dst = cv2.add(img1_bg,img2_fg)
img1[0:rows, 0:cols ] = dst

cv2.imshow('res',img1)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Aşağıdaki sonuca bakın. Soldaki görüntü oluşturduğumuz maskeyi gösterir. Sağdaki görüntü son sonucu gösterir. Daha fazla bilgi için, yukarıdaki imgede bulunan tüm ara görüntüleri, özellikle img1_bg ve img2_fg'yi görüntüleyin



3.3-)Performans Ölçme ve Geliştirme Teknikleri

Hedef

Görüntü işlemede, saniyede çok sayıda işlemle uğraştığınız için, kodunuzun sadece doğru çözümü sağlamakla kalmayıp aynı zamanda en hızlı şekilde olması zorunludur. Yani bu bölümde, öğreneceksiniz

- Kodunuzun performansını ölçmek için.
- Kodunuzun performansını artırmak için bazı ipuçları.
- Şu işlevleri göreceksiniz: `cv2.getTickCount` , `cv2.getTickFrequency` vb.

OpenCV dışında, Python yürütme zamanını ölçümede yardımcı olan bir modül **zamanı** da sağlar . Başka bir modül **profil** , koddaki her bir işlevin ne kadar sürdüğü, işlevin kaç kez çağrıldığı vb. Gibi kod hakkında ayrıntılı rapor alınmasına yardımcı olur. Ancak, IPython kullanıyorsanız, tüm bu özellikler kullanıcı dostu bir şekilde entegre edilmiştir tavır. Bazı önemli olanları göreceğiz ve daha fazla ayrıntı için **Ek Kaynaklar** bölümündeki bağlantıları kontrol edeceğiz .

OpenCV ile Performans Ölçme

cv2.getTickCount işlevi, bir başvuru olayından (makine açıldığı an gibi) sonraki saat döngü sayısını bu işlevin **çağrıldığı ana kadar** döndürür. Dolayısıyla, işlev yürütmeden önce ve sonra çağrırsanız, bir işlevi yürütmek için kullanılan saat döngüsü sayısını alırsınız.

cv2.getTickFrequency işlevi saat döngü sıklığını veya saniye başına saat döngü sayısını döndürür. Böylece, yürütme süresini saniye cinsinden bulmak için aşağıdakileri yapabilirsiniz:

```
e1 = cv2.getTickCount()  
# your code execution  
e2 = cv2.getTickCount()  
time = (e2 - e1)/ cv2.getTickFrequency()
```

Aşağıdaki örnekle göstereceğiz. Aşağıdaki örnek, 5 ile 49 arasında değişen tek boyutlu bir çekirdeğe sahip medyan filtreleme uygulayın. (Sonuç nasıl görüneceğinden endişe etmeyin, hedefimiz bu değil):

```
img1 = cv2.imread('messi5.jpg')  
  
e1 = cv2.getTickCount()  
for i in xrange(5,49,2):  
    img1 = cv2.medianBlur(img1,i)  
e2 = cv2.getTickCount()  
t = (e2 - e1)/cv2.getTickFrequency()  
print t  
  
# Result I got is 0.521107655 seconds
```

Not Aynı şeyi zaman modülüyle de yapabilirsiniz. Cv2.getTickCount yerine time.time() işlevini kullanın. Sonra iki kez farkı alın.

OpenCV'de Varsayılan Optimizasyon

OpenCV işlevlerinin çoğu SSE2, AVX vb. Kullanılarak optimize edilmiştir. Dolayısıyla, sistemimiz bu özellikleri destekliyorsa, bunlardan faydalamalıyız (neredeyse tüm modern işlemciler bunları destekliyor). Derleme sırasında varsayılan olarak etkindir. Bu nedenle, OpenCV etkinleştirilirse optimize edilmiş kodu çalıştırır, aksi takdirde optimize edilmemiş kodu çalıştırır. Sen kullanabilirsiniz **cv2.useOptimized()** devre dışı ve / etkin olup olmadığını kontrol etmek için **cv2.setUseOptimized()** o etkin / devre dışı etmek. Basit bir örnek görelim.

```
# check if optimization is enabled  
optimizasyonun etkin olup olmadığını kontrol edin  
  
In [5]: cv2.useOptimized()  
Out[5]: True
```

```
In [6]: %timeit res = cv2.medianBlur(img,49)
10 loops, best of 3: 34.9 ms per loop

# Disable it Devre Dışı Bırak
In [7]: cv2.setUseOptimized(False)

In [8]: cv2.useOptimized()
Out[8]: False

In [9]: %timeit res = cv2.medianBlur(img,49)
10 loops, best of 3: 64.1 ms per loop
```

Optimize edilmiş medyan filtreleme, optimize edilmemiş versiyondan ~ 2 kat daha hızlıdır. Kaynağını kontrol ederseniz, medyan filtrelemenin SIMD optimize edildiğini görebilirsiniz. Böylece, kodunuzun üst kısmında optimizasyonu etkinleştirmek için bunu kullanabilirsiniz (varsayılan olarak etkin olduğunu unutmayın).

IPython'da Performansı Ölçme

Bazen iki benzer işlemin performansını karşılaştırmanız gerekebilir. IPython bunu yapmak için size sihirli bir komut `%timeit` verir. Daha doğru sonuçlar almak için kodu birkaç kez çalıştırır. Bir kez daha, tek satır kodlarını ölçmek için uygundurlar.

Örneğin, aşağıdaki toplama işlemlerinden hangisinin daha iyi olduğunu biliyor musunuz, `x = 5; y = x ** 2`, `x = 5; y = x * x`, `x = np.uint8([5]); y = x * x` veya `y = np.square(x)`? IPython kabuğuunda `%timeit` ile bulacağız.

```
In [10]: x = 5

In [11]: %timeit y=x**2
10000000 loops, best of 3: 73 ns per loop //döngü 3 ün en iyisi

In [12]: %timeit y=x*x
10000000 loops, best of 3: 58.3 ns per loop

In [15]: z = np.uint8([5])

In [17]: %timeit y=z*z
1000000 loops, best of 3: 1.25 us per loop

In [19]: %timeit y=np.square(z)
1000000 loops, best of 3: 1.16 us per loop
```

Bunu görebilirsiniz, `x = 5 ; y = x * x` en hızlıdır ve Numpy'ye kıyasla yaklaşık 20 kat daha hızlıdır. Dizi oluşturmayı da düşünürseniz, 100 kata kadar daha hızlı ulaşılabilir. Harika, değil mi? (*Numpy devs bu konuda çalışıyor*)

Not Python skaler işlemleri Numpy skaler işlemlerinden daha hızlıdır. Bu nedenle, bir veya iki eleman içeren işlemler için Python skaler Numpy dizilerinden daha iyidir. Dizinin boyutu biraz daha büyük olduğunda Numpy avantaj sağlar.

Bir örnek daha deneyeceğiz. Bu kez, aynı görüntü için `cv2.countNonZero()` ve `np.count_nonzero()` performansını karşılaştıracağız .

```
In [35]: %timeit z = cv2.countNonZero(img)
100000 loops, best of 3: 15.8 us per loop
```

```
In [36]: %timeit z = np.count_nonzero(img)
1000 loops, best of 3: 370 us per loop
```

OpenCV işlevi Numpy işlevinden yaklaşık 25 kat daha hızlıdır.

Not Normalde, OpenCV işlevleri Numpy işlevlerinden daha hızlıdır. Dolayısıyla aynı işlem için OpenCV fonksiyonları tercih edilir. Ancak, özellikle Numpy kopyalar yerine görüntülerle çalıştığında istisnalar olabilir.

Performans Optimizasyonu Teknikleri

Python ve Numpy'nin maksimum performansından yararlanmak için çeşitli teknikler ve kodlama yöntemleri vardır. Burada sadece ilgili olanlar not edilir ve önemli kaynaklara bağlantıları verilir. Burada dikkat edilmesi gereken ana şey, önce algoritmayı basit bir şekilde uygulamaya çalışın. Çalıştıktan sonra profil oluşturun, darboğazları bulun ve optimize edin.

1. Python'da döngüler, özellikle çift / üçlü döngüler vb. Kullanmaktan kaçının. Doğası gereği yavaşırlar.
2. Numpy ve OpenCV vektör işlemleri için optimize edildiğinden algoritmayı / kodu mümkün olan en üst düzeyde vektörleştirin.
3. Önbellek tutarlığını kullanın.
4. Gerekli olmadıkça dizinin kopyalarını asla oluşturmayın. Bunun yerine görüntüler kullanmayı deneyin. Dizi kopyalama maliyetli bir işlemidir.

Tüm bu işlemleri yaptıktan sonra bile, kodunuz hala yavaşsa veya büyük döngüler kullanmak kaçınılmazsa, daha hızlı hale getirmek için Cython gibi ek kitaplıklar kullanın.

Ek kaynaklar

1. [Python Optimizasyon Teknikleri](#)
2. Scipy Ders Notları – [İleri Numpy](#)
3. [IPython'da Zamanlama ve Profil Oluşturma](#)

4.1-)Renk Alanlarını Değiştirme Hedef

- Bu öğreticide, görüntülerin BGR \leftrightarrow Gray, BGR \leftrightarrow HSV vb. gibi bir renk uzayından diğerine nasıl dönüştüreceğinizi öğreneceksiniz .
- Buna ek olarak, bir videoda renkli bir nesneyi ayıklayan bir uygulama oluşturacağız
- Aşağıdaki işlevleri öğreneceksiniz: `cv2.cvtColor()` , `cv2.inRange()` vb.

Renk Boşluğunu Değiştirme

OpenCV'de 150'den fazla renk alanı dönüştürme yöntemi bulunmaktadır. Ancak en yaygın kullanılan iki taneye bakacağınız, BGR \leftrightarrow Gray ve BGR \leftrightarrow HSV.

Renk dönüşümü sağlamak için, işlevini kullanın `cv2.cvtColor()` (`input_image`, bayrak) bayrağı dönüşüm tipini tespit eder.

BGR \rightarrow Gray dönüşümü için `cv2.COLOR_BGR2GRAY` bayraklarını kullanıyoruz . Benzer şekilde BGR \rightarrow HSV için `cv2.COLOR_BGR2HSV` bayrağını kullanıyoruz . Diğer bayrakları almak için Python terminalinizde aşağıdaki komutları çalıştırın:

```
>>> import cv2
>>> flags = [i for i in dir(cv2) if i.startswith('COLOR_')]
>>> print flags
```

Not HSV için Ton aralığı [0,179], Doygunluk aralığı [0,255] ve Değer aralığı [0,255] 'dir. Farklı yazılımlar farklı ölçekler kullanır. Bu nedenle, OpenCV değerlerini onlarla karşılaştırıyorsanız, bu aralıkları normalleştirmeniz gereklidir.

Nesne Takibi

Şimdi BGR görüntüsünü HSV'ye nasıl dönüştüreceğimizi biliyoruz, bunu renkli bir nesneyi çıkarmak için kullanabiliriz. HSV'de, bir rengi temsil etmek RGB renk uzayından daha kolaydır. Bizim uygulamada, mavi renkli bir nesne çıkarmaya çalışacağız. İşte yöntem:

- Videonun her karesini alın
- Dönüştürmek BGR a HSV renk-uzay
- HSV görüntüsünü bir dizi mavi renk için eşleştiriyoruz
- Şimdi mavi nesneyi tek başına çıkarın, istediğimiz görüntüde ne yapabiliriz.

Aşağıda ayrıntılı olarak yorumlanan kod verilmiştir:

```
import cv2
import numpy as np

cap = cv2.VideoCapture(0)

while(1):
    # Take each frame her çerçeveyi
    _, frame = cap.read()
```

```

# Convert BGR to HSV
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

# define range of blue color in HSV
lower_blue = np.array([110,50,50])
upper_blue = np.array([130,255,255])

# Threshold the HSV image to get only blue colors
mask = cv2.inRange(hsv, lower_blue, upper_blue)

# Bitwise-AND mask and original image
res = cv2.bitwise_and(frame,frame, mask= mask)

cv2.imshow('frame',frame)
cv2.imshow('mask',mask)
cv2.imshow('res',res)
k = cv2.waitKey(5) & 0xFF
if k == 27:
    break

cv2.destroyAllWindows()

```

Aşağıdaki görüntü mavi nesnenin izlenmesini göstermektedir:



Not Görüntüde bazı sesler var. Daha sonraki bölümlerde bunları nasıl kaldıracağımızı göreceğiz.

Not Bu, nesne izlemedeki en basit yöntemdir. Konturların işlevlerini öğrendikten sonra, bu nesnenin centroidini bulmak ve nesneyi izlemek, sadece elinizi kamerasın önünde hareket ettirmek ve daha birçok komik şey yapmak için diyagramlar çizmek gibi birçok şey yapabilirsiniz.

Izlenecek HSV değerleri nasıl bulunur?

Bu, [stackoverlow.com'da](http://stackoverflow.com) bulunan yaygın bir sorudur. Çok basit ve aynı işlevi kullanabilirsiniz, `cv2.cvtColor ()`. Bir görüntü iletmek yerine, sadece istediğiniz BGR değerlerini iletirsiniz. Örneğin, Green'in HSV değerini bulmak için Python terminalinde aşağıdaki komutları deneyin:

```
>>> green = np.uint8([[0,255,0]])
>>> hsv_green = cv2.cvtColor(green, cv2.COLOR_BGR2HSV)
>>> print hsv_green
[[[ 60 255 255]]]
```

Şimdi sırasıyla $[H-10, 100, 100]$ ve $[H + 10, 255, 255]$ alt sınır ve üst sınır olarak kabul edilir. Bu yöntemin dışında, bu değerleri bulmak için GIMP veya çevrimiçi dönüştürücüler gibi görüntü düzenleme araçlarını kullanabilirsiniz, ancak HSV aralıklarını ayarlamayı unutmayın.

4.2-) Görüntülerin Geometrik Dönüşümleri Hedefler

- Çeviri, döndürme, afin dönüşüm vb. Görüntülere farklı geometrik dönüşümler uygulamayı öğrenin.
- Şu işlevleri göreceksiniz: `cv2.getPerspectiveTransform`

Dönüşümler

OpenCV, `cv2.warpAffine` ve `cv2.warpPerspective` olmak üzere her türlü dönüştürme işlemeye sahip olabileceğiniz iki dönüştürme işlevi sağlar. `cv2.warpAffine` 2×3 dönüşüm matrisini alırken `cv2.warpPerspective` girdi olarak 3×3 dönüşüm matrisini alır.

Ölçekleme

Ölçekleme yalnızca görüntünün yeniden boyutlandırılmasıdır. OpenCV bu amaçla `cv2.resize` işleviyle birlikte gelir. Görüntünün boyutu manuel olarak belirlenebilir veya ölçeklendirme faktörünü belirleyebilirsiniz. Farklı interpolasyon yöntemleri kullanılır. Tercih edilen interpolasyon yöntemleri, büzülme için `cv2.INTER_AREA` ve yakınlaştırma için `cv2.INTER_CUBIC` (yavaş) ve `cv2.INTER_LINEAR`'dır. Varsayılan olarak, kullanılan yeniden değerlendirme yöntemi tüm yeniden boyutlandırma amaçları için `cv2.INTER_LINEAR` şeklidir. Bir giriş görüntüsünü aşağıdaki yöntemlerden biriyle yeniden boyutlandıabilirsiniz:

```
import cv2
import numpy as np

img = cv2.imread('messi5.jpg')

res = cv2.resize(img, None, fx=2, fy=2, interpolation = cv2.INTER_CUBIC)

#OR

height, width = img.shape[:2]
res = cv2.resize(img, (2*width, 2*height), interpolation = cv2.INTER_CUBIC)
```

Tercüme

Çeviri, nesnenin konumunun kaydırılmasıdır. (X, Y) yönünde kaymayı biliyorsanız, bırakın (t_x, t_y) , dönüşüm matrisini Maşağidakı gibi oluşturabilirsiniz :

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

Sen tip bir Numpy dizisi haline getirerek alabilir `np.float32` ve içine geçmek `cv2.warpAffine` () fonksiyonu. (100,50) değerinin kaydırılması için aşağıdaki örneğe bakın:

```
import cv2
import numpy as np

img = cv2.imread('messi5.jpg',0)
rows,cols = img.shape

M = np.float32([[1,0,100],[0,1,50]])
dst = cv2.warpAffine(img,M,(cols,rows))

cv2.imshow('img',dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Uyarı `Cv2.warpAffine` () işlevinin üçüncü argümanı , çıktı görüntüsünün boyutudur (**genişlik, yükseklik**) . Genişlik = sütun sayısı ve yükseklik = satır sayısı olduğunu unutmayın.

Aşağıdaki sonuca bakın:



Rotasyon

Görüntünün bir açı için döndürülmesi θ , formun dönüşüm matrisi ile elde edilir

$$M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Ancak OpenCV, ayarlanabilir dönüş merkezi ile ölçekli rotasyon sağlar, böylece istediğiniz herhangi bir yerde dönebilirsiniz. Değiştirilmiş dönüşüm matrisi,

$$\begin{bmatrix} \alpha & \beta & (1 - \alpha) \cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1 - \alpha) \cdot center.y \end{bmatrix}$$

nerede:

$$\alpha = \text{scale} \cdot \cos \theta,$$

$$\beta = \text{scale} \cdot \sin \theta$$

Bu dönüşüm matrisini bulmak için OpenCV, **cv2.getRotationMatrix2D** adlı bir işlev sağlar. Görüntüyü ölçeklendirme olmadan merkeze göre 90 derece döndüren aşağıdaki örneği kontrol edin.

```
img = cv2.imread('messi5.jpg',0)
rows,cols = img.shape

M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)
dst = cv2.warpAffine(img,M,(cols,rows))
```

See the result:



Afin Dönüşüm

Afin dönüşümde, orijinal görüntüdeki tüm paralel çizgiler çıktı görüntüsünde yine paralel olacaktır. Dönüşüm matrisini bulmak için, giriş görüntüsünden ve çıkış görüntüsündeki karşılık gelen konumlarından üç noktaya ihtiyacımız var. Daha sonra **cv2.getAffineTransform** geçirilecek olan bir 2x3 matris oluşturur **cv2.warpAffine**.

Aşağıdaki örneği kontrol edin ve seçtiğim noktalara da bakın (bunlar Yeşil renkle işaretlenmiştir):

```
img = cv2.imread('drawing.png')
rows,cols,ch = img.shape

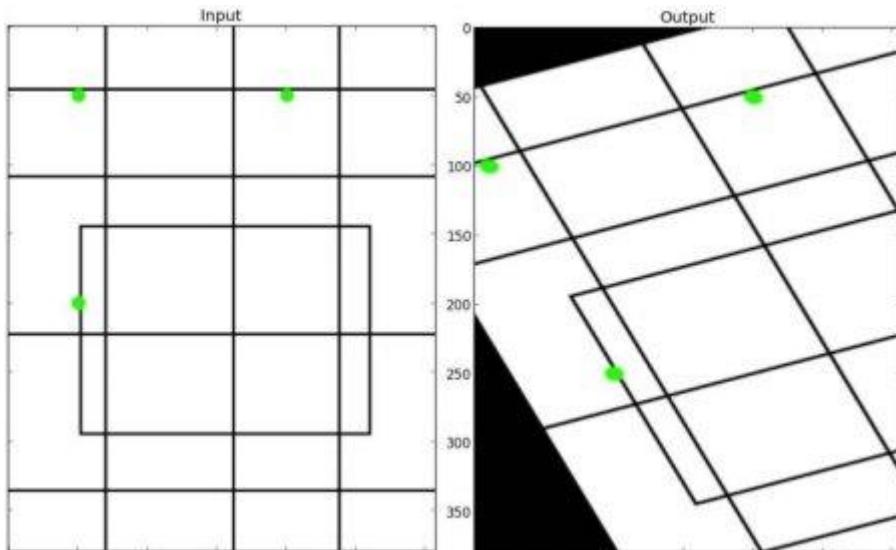
pts1 = np.float32([[50,50],[200,50],[50,200]])
pts2 = np.float32([[10,100],[200,50],[100,250]])

M = cv2.getAffineTransform(pts1,pts2)

dst = cv2.warpAffine(img,M,(cols,rows))

plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

See the result:



Perspektif Dönüşümü [1](#)

Perspektif dönüşümü için 3×3 dönüşüm matrisine ihtiyacınız vardır. Düz çizgiler dönüşümden sonra bile düz kalacaktır. Bu dönüşüm matrisini bulmak için, giriş görüntüsünde 4 noktaya ve çıkış görüntüsünde karşılık gelen noktalara ihtiyacınız vardır. Bu 4 puan arasında 3 tanesi eş zamanlı olmamalıdır. Daha sonra dönüşüm matrisi `cv2.getPerspectiveTransform` işlevi tarafından bulunabilir. Sonra bu 3×3 dönüşüm matrisi ile `cv2.warpPerspective` uygulayın.

Aşağıdaki koda bakın:

```
img = cv2.imread('sudokusmall.png')
rows,cols,ch = img.shape

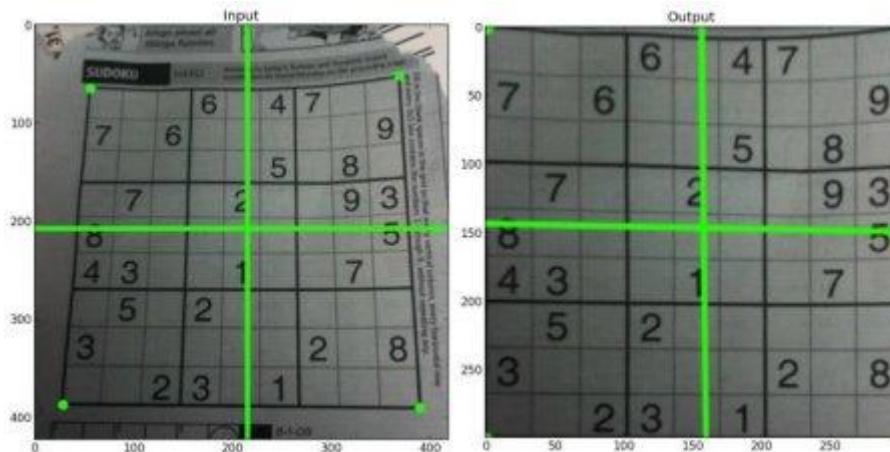
pts1 = np.float32([[56,65],[368,52],[28,387],[389,390]])
pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])

M = cv2.getPerspectiveTransform(pts1,pts2)

dst = cv2.warpPerspective(img,M,(300,300))

plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

Result:



4.3-) Görüntü Eşiği Hedef

- Bu öğreticide Basit eşik, Uyarlanabilir eşik, Otsu eşiklerini vb. Öğreneceksiniz.
- Bu işlevleri öğreneceksiniz: `cv2.threshold` , `cv2.adaptiveThreshold` vs.

Basit Eşikleme

Burada mesele düzgündür. Piksel değeri bir eşik değerden büyükse, bir değere (beyaz olabilir), başka bir değere (siyah olabilir) atanır. Kullanılan işlev `cv2.threshold`'dur . İlk argüman, **gri tonlamalı bir görüntü olması gereken** kaynak görüntüsündür . İkinci argüman, piksel değerlerini sınıflandırmak için kullanılan eşik değeridir. Üçüncü argüman, piksel değeri eşik değerden daha yüksekse (bazen daha küçükse) verilecek değeri temsil eden `maxVal` değeridir. OpenCV, farklı eşikleme stilleri sağlar ve işlevin dördüncü parametresi ile karar verilir. Farklı türler:

- `cv2.THRESH_BINARY`
- `cv2.THRESH_BINARY_INV`
- `cv2.THRESH_TRUNC`
- `cv2.THRESH_TOZERO`
- `cv2.THRESH_TOZERO_INV`

Belgeler, her türün ne anlamına geldiğini açıkça açıklar. Lütfen belgelere bakın.

İki çıkış elde edilir. Birincisi, daha sonra açıklanacak bir `retval` . İkinci çıktı **eşikli imajımızdır** .

Kod:

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

img = cv2.imread('gradient.png',0)
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
```

```

ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]

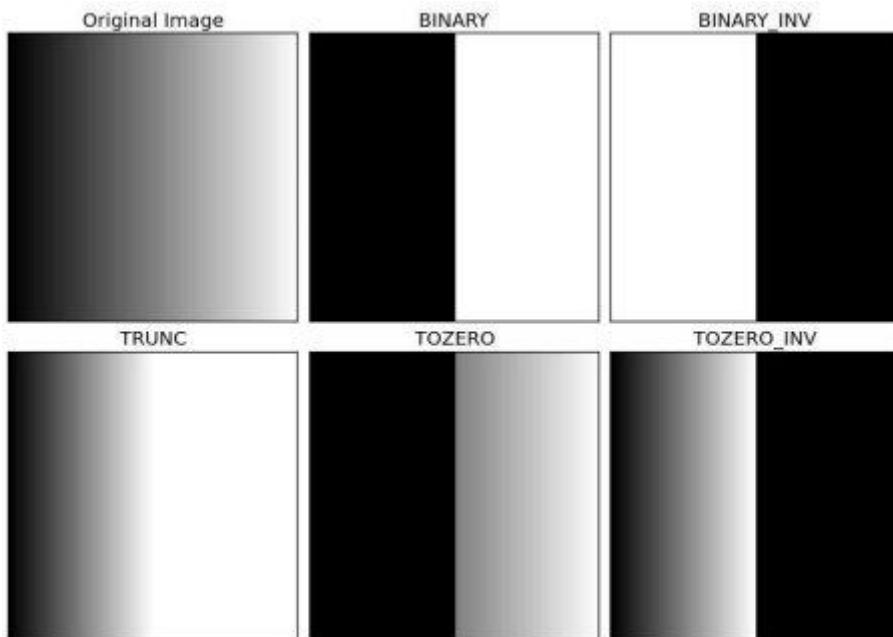
for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])

plt.show()

```

Not Birden çok görüntüyü çizmek için `plt.subplot()` işlevini kullandık. Daha fazla bilgi için lütfen Matplotlib dokümanlarına göz atın.

Sonuç aşağıda verilmiştir:



Uyarlanabilir Eşik

Önceki bölümde eşik değeri olarak global bir değer kullandık. Ancak görüntünün farklı alanlarda farklı aydınlatma koşullarına sahip olduğu tüm koşullarda iyi olmayabilir. Bu durumda, uyarlamalı eşiklemeye gidiyoruz. Burada, algoritma görüntünün küçük bir bölgesi için eşiği hesaplar. Bu nedenle, aynı görüntünün farklı bölgeleri için farklı eşikler elde ederiz ve bu, bize farklı aydınlatmaya sahip görüntüler için daha iyi sonuçlar verir.

Üç 'özel' giriş parametresi ve yalnızca bir çıktı argümanı vardır.

Uyarlanabilir Yöntem – Eşik değerinin nasıl hesaplanacağına karar verir.

- `cv2.ADAPTIVE_THRESH_MEAN_C`: eşik değeri komşuluk alanının ortalamasıdır.

- cv2.ADAPTIVE_THRESH_GAUSSIAN_C: eşik değeri, ağırlıkların bir gauss penceresi olduğu mahalle değerlerinin ağırlıklı toplamıdır.

Blok Boyutu – Mahalle(komşuluk) alanının büyüklüğüne karar verir.

C – Sadece hesaplanan ortalama veya ağırlıklı ortalamadan çıkarılan bir sabittir.

Aşağıdaki kod parçası, değişken aydınlatmalı bir görüntü için global eşik ve uyarlanabilir eşik değerlerini karşılaştırır:

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

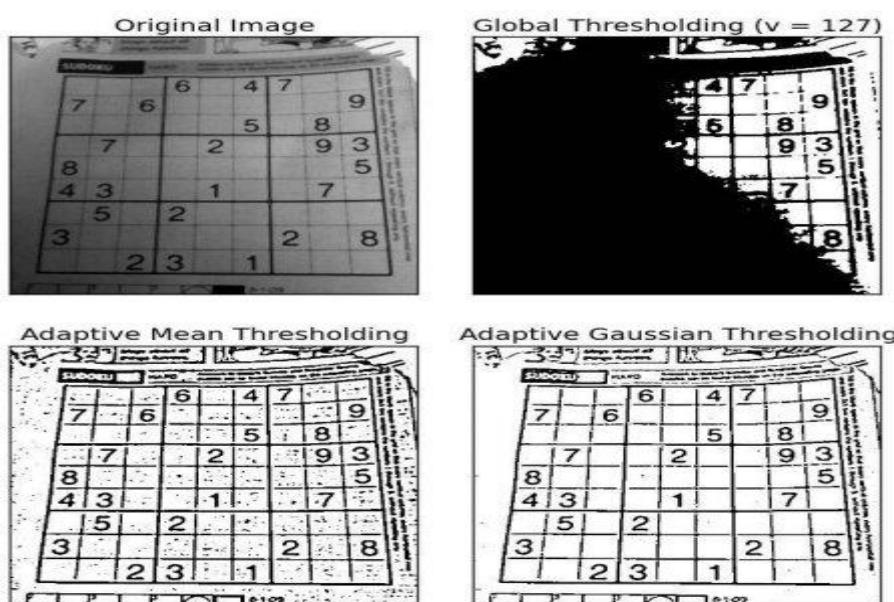
img = cv2.imread('dave.jpg',0)
img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255, cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255, cv2.ADAPTIVE_THRESH_MEAN_C,\n
                           cv2.THRESH_BINARY,11,2)
th3 = cv2.adaptiveThreshold(img,255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\n
                           cv2.THRESH_BINARY,11,2)

titles = ['Original Image', 'Global Thresholding (v = 127)',\n
          'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]

for i in xrange(4):
    plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

Result :



Otsu'nun İkilileşmesi

İlk bölümde, ikinci bir `retval` parametresi olduğunu söylediğim . Otsu'nun Binarizasyonuna gittiğimizde kullanımı geliyor. Öyleyse nedir?

Küresel eşiklemede, eşik değeri için rastgele bir değer kullandık, değil mi? Peki, seçtiğimiz bir değerin iyi olup olmadığını nasıl bilebiliriz? Cevap, deneme yanılma yöntemidir. Ancak bir **bimodal görüntü** düşünün (*Basit bir deyişle, bimodal görüntü histogramında iki tepe noktası olan bir görüntüdür*). Bu görüntü için, bu piklerin ortasında yaklaşık olarak bir değer eşik değeri alabiliriz, değil mi? Otsu ikileminin yaptığı şey budur. Basit bir deyişle, bimodal bir görüntü için görüntü histogramından bir eşik değerini otomatik olarak hesaplar. (İki modlu olmayan görüntüler için, binarizasyon doğru olmaz.)

Bunun için `cv2.threshold()` işlevimiz kullanılır, ancak `cv2.THRESH_OTSU` adlı ekstra bir bayrak geçirin . **Eşik değeri için sıfırı iletmeyen yeterlidir** . Ardından algoritma en uygun eşik değerini bulur ve sizi ikinci çıktı `retval` olarak döndürür . Otsu eşiği kullanılmazsa, `retval` kullandığınız eşik değeri ile aynıdır.

Aşağıdaki örneği inceleyin. Giriş görüntüsü gürültülü bir görüntüdür. İlk durumda, 127 eşik değeri için küresel eşik değeri uyguladım. İkinci durumda, doğrudan Otsu eşliğini uyguladım. Üçüncü durumda, gürültüyü gidermek için görüntüyü 5x5 gauss çekirdeği ile filtreledim, daha sonra Otsu eşiği uyguladım. Gürültü filtrelemenin sonucu nasıl iyileştirdiğini görün.

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

img = cv2.imread('noisy2.png',0)

# global thresholding
ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)

# Otsu's thresholding
ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# Otsu's thresholding after Gaussian filtering
blur = cv2.GaussianBlur(img,(5,5),0)
ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# plot all the images and their histograms
images = [img, th1,
          img, th2,
          blur, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
          'Original Noisy Image','Histogram',"Otsu's Thresholding",
          'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]

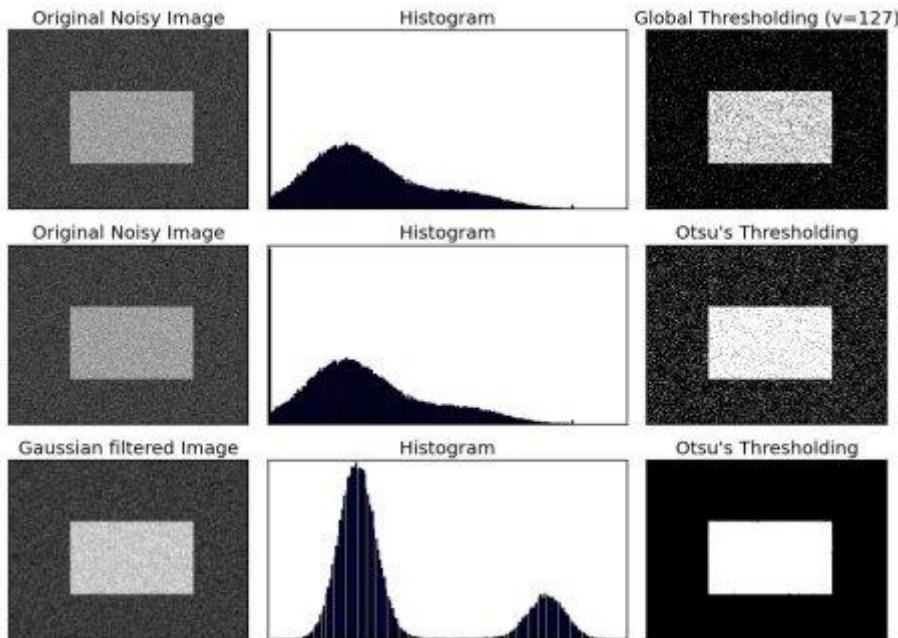
for i in xrange(3):
    plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
    plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
    plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
```

```

plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2], 'gray')
plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()

```

Result :



Otsu'hun Binarizasyonu Nasıl Çalışır?

Bu bölüm, aslında nasıl çalıştığını göstermek için Otsu'nun ikili hale getirilmesinin bir Python uygulamasını gösterir. İlgiilenmiyorsanız, bunu atlayabilirsiniz.

İki modlu görüntülerle çalıştığımız için, Otsu algoritması ilişkinin verdiği **sınıf içi ağırlıklı varyansı** en azı indiren bir eşik değeri (t) bulmaya çalışır :

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

nerede

$$\begin{aligned}
 q_1(t) &= \sum_{i=1}^t P(i) & q_2(t) &= \sum_{i=t+1}^I P(i) \\
 \mu_1(t) &= \sum_{i=1}^t \frac{iP(i)}{q_1(t)} & \mu_2(t) &= \sum_{i=t+1}^I \frac{iP(i)}{q_2(t)} \\
 \sigma_1^2(t) &= \sum_{i=1}^t [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} & \sigma_2^2(t) &= \sum_{i=t+1}^I [i - \mu_1(t)]^2 \frac{P(i)}{q_2(t)}
 \end{aligned}$$

Aslında her iki sınıfı sapma minimum olacak şekilde iki tepe arasında yer alan t değerini bulur. Basitçe aşağıdaki gibi Python'da uygulanabilir:

```

img = cv2.imread('noisy2.png',0)
blur = cv2.GaussianBlur(img,(5,5),0)

# find normalized histogram, and its cumulative distribution function
hist = cv2.calcHist([blur],[0],None,[256],[0,256])
hist_norm = hist.ravel()/hist.max()
Q = hist_norm.cumsum()

bins = np.arange(256)

fn_min = np.inf
thresh = -1

for i in xrange(1,256):
    p1,p2 = np.hsplit(hist_norm,[i]) # probabilities
    q1,q2 = Q[i],Q[255]-Q[i] # cum sum of classes
    b1,b2 = np.hsplit(bins,[i]) # weights

    # finding means and variances
    m1,m2 = np.sum(p1*b1)/q1, np.sum(p2*b2)/q2
    v1,v2 = np.sum(((b1-m1)**2)*p1)/q1,np.sum(((b2-m2)**2)*p2)/q2

    # calculates the minimization function
    fn = v1*q1 + v2*q2
    if fn < fn_min:
        fn_min = fn
        thresh = i

# find otsu's threshold value with OpenCV function
ret, otsu = cv2.threshold(blur,0,255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)
print thresh,ret

```

4.4-) Görüntüler Düzeltme Hedefler

Öğrenmek:

- Çeşitli düşük geçiş filtreleriyle görüntüler bulanıklaştırın
- Görüntülere özel yapım filtreler uygulama (2D evrişim)

2D Evrişim (Görüntü Filtreleme)

Tek boyutlu sinyallerde olduğu gibi, görüntüler çeşitli düşük geçişli filtreler (LPF), yüksek geçişli filtreler (HPF) vb. ile de filtrelenebilir. LPF, gürültüleri gidermeye, görüntüler bulanıklaştırmaya vb. Yardımcı olur. HPF filtreleri, Görüntüler.

OpenCV, bir çekirdeği görüntüyle birleştirmek için `cv2.filter2D()` işlevini sağlar. Örnek olarak, bir görüntü üzerinde bir ortalama滤resi deneyeceğiz. 5x5 ortalama filtre çekirdeği aşağıdaki gibi görünecektir:

$$K = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

İşlem şu şekildedir: bu çekirdeği bir pikselin üzerinde tutun, 25 çekirdeğin tümünü bu çekirdeğin altına ekleyin, ortalamasını alın ve merkezi pikseli yeni ortalama değerle değiştirin. Görüntüdeki tüm pikseller için bu işleme devam eder. Bu kodu deneyin ve sonucu kontrol edin:

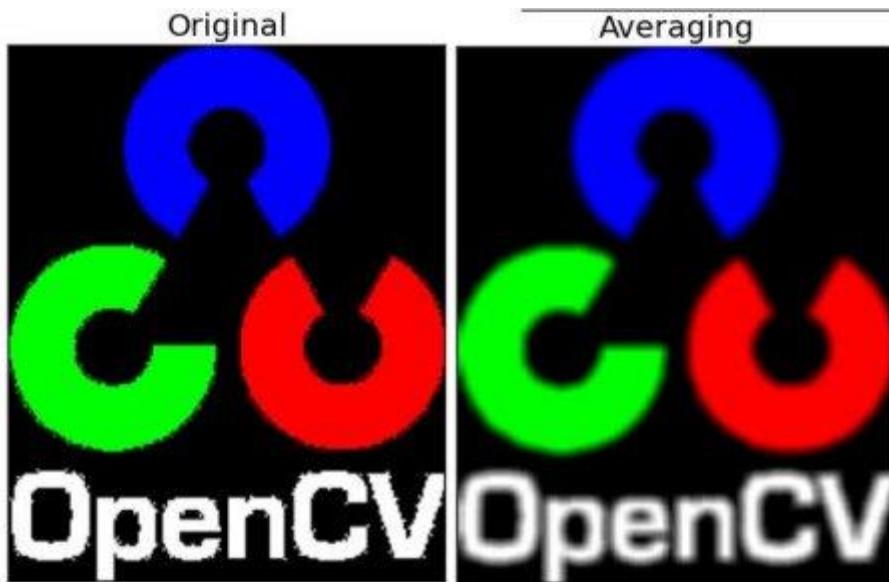
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

kernel = np.ones((5,5),np.float32)/25
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([]), plt.yticks([])
plt.show()
```

Result:



Görüntü Bulanıklaştırma (Görüntü Düzeltme)

Görüntü bulanıklığı, görüntüyü alçak geçiren bir filtre çekirdeğiyle kıvrılarak elde edilir. Gürültü gidermek için kullanışlıdır. Aslında yüksek frekanslı içeriği (örn. Parazit, kenarlar) görüntüsünden kaldırır. Yani bu işlemde kenarlar biraz bulanık. (Kenarları da bulanıklaştırmayan bulanıklaştırma teknikleri vardır). OpenCV temel olarak dört tip bulanıklaştırma tekniği sağlar.

1. Ortalama

Bu, görüntüyü normalleştirilmiş bir kutufiltresi ile döndürerek yapılır. Sadece çekirdek alanının altındaki tüm piksellerin ortalamasını alır ve merkezi öğeyi değiştirir. Bu, `cv2.blur()` veya `cv2.boxFilter()` işlevi tarafından yapılır. Çekirdek hakkında daha fazla bilgi için dokümanlara bakın. Çekirdeğin genişliğini ve yüksekliğini belirtmeliyiz. 3×3 normalleştirilmiş bir kutufiltresi aşağıdaki gibi görünecektir:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Not Normalleştirilmiş kutu filtresi kullanmak istemiyorsanız, `cv2.boxFilter()` kullanın. İşlevde `normalize = False` argümanını iletin.

5×5 boyutunda bir çekirdeğe sahip aşağıdaki örnek bir demosu kontrol edin:

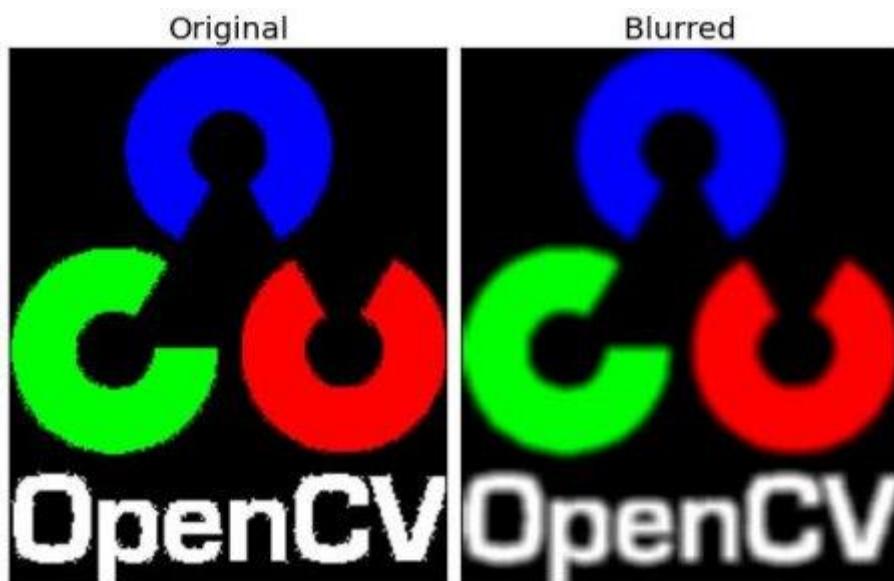
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('opencv_logo.png')

blur = cv2.blur(img,(5,5))

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur),plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.show()
```

Result:



Burada, kutu filtresi yerine, gauss çekirdeği kullanılır. **Cv2.GaussianBlur ()** işlevi ile yapılır. Pozitif ve garip olması gereken çekirdeğin genişliğini ve yüksekliğini belirtmeliyiz. Ayrıca sırasıyla X ve Y yönünde, sigmaX ve sigmaY'de standart sapmayı belirtmeliyiz. Yalnızca sigmaX belirtilirse, sigmaY sigmaX ile aynı alınır. Her ikisi de sıfır olarak verilirse, çekirdek boyutundan hesaplanır. Gauss bulanıklığı, görüntüdeki gauss gürültüsünün giderilmesinde oldukça etkilidir.

İsterseniz, **cv2.getGaussianKernel ()** işleviyle bir Gauss çekirdeği oluşturabilirsiniz .

Yukarıdaki kod Gauss bulanıklığı için değiştirilebilir:

```
blur = cv2.GaussianBlur(img,(5,5),0)
```

Sonuç:



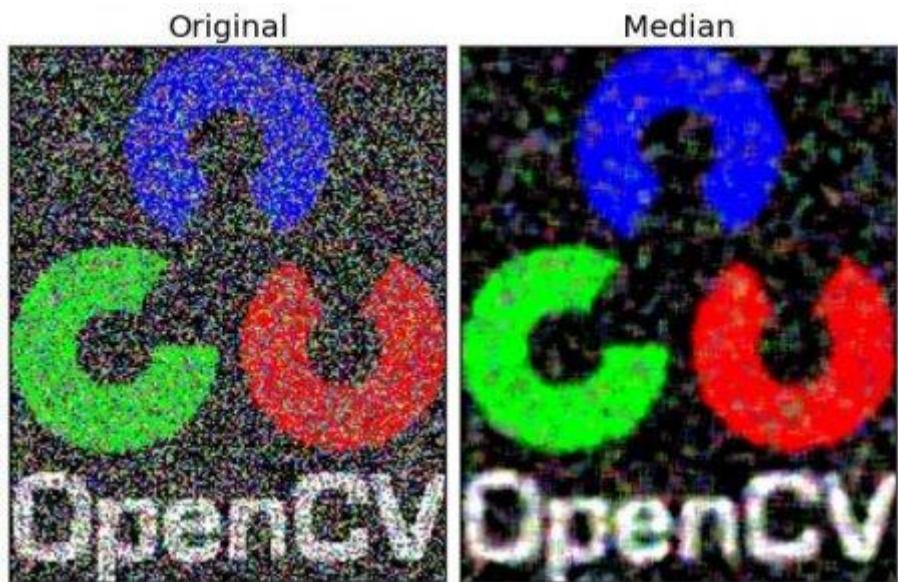
3. Medyan Bulanıklaştırma

Burada, **cv2.medianBlur ()** işlevi çekirdek alanı altındaki tüm piksellerin medyanını alır ve merkezi eleman bu medyan değerle değiştirilir. Bu, görüntülerdeki tuz ve karabiber gürültüsüne karşı oldukça etkilidir. İlginç olan şey, yukarıdaki filtrelerde, merkezi elemanın görüntüde bir piksel değeri veya yeni bir değer olabilecek yeni hesaplanmış bir değer olmasıdır. Ancak ortalama bulanıklaşmadı, merkezi öğe her zaman görüntüdeki bir piksel değeri ile değiştirilir. Gürültüyü etkili bir şekilde azaltır. Çekirdek boyutu pozitif tek bir tam sayı olmalıdır.

Bu demoda orijinal resmimize% 50 gürültü ekledim ve medyan bulanıklık uyguladım. Sonucu kontrol edin:

```
medyan = cv2 . medianBlur ( img , 5 )
```

Sonuç:



4. İkili Filtreleme

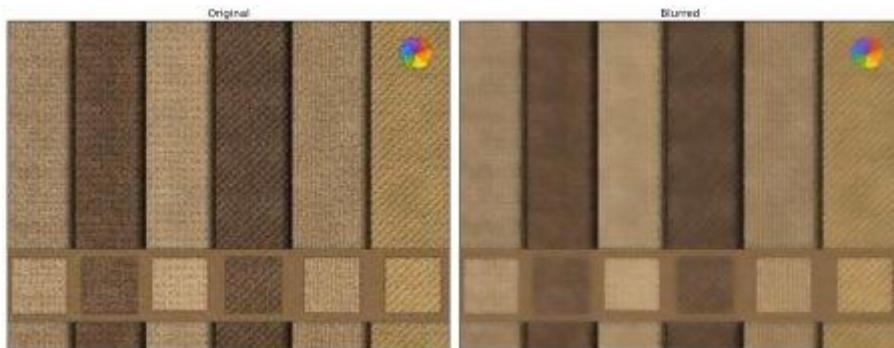
`cv2.bilateralFilter` () kenarları keskin tutarken gürültünün giderilmesinde oldukça etkilidir. Ancak işlem diğer filtrelere göre daha yavaştır. Gauss filtresinin pikselin etrafındaki bir mahalleyi aldığı ve gauss ağırlıklı ortalamasını bulduğunu zaten gördük. Bu gauss滤resi yalnızca alanın bir işlevidir, yani filtreleme sırasında yakındaki pikseller dikkate alınır. Piksellerin neredeyse aynı yoğunluğa sahip olup olmadığını düşünmez. Pikselin kenar piksel olup olmadığını dikkate almaz. Bu yüzden yapmak istemediğimiz kenarları da bulanıklaştırır.

İki taraflı filtre de uzayda bir gauss filtresi alır, ancak piksel farkının bir fonksiyonu olan bir başka gauss filtresi. Gauss fonksiyonu, yalnızca yakındaki piksellerin bulanıklaştırma için dikkate alınmasını sağlarken, gaussian yoğunluk farkı fonksiyonu sadece merkezi piksele benzer yoğunluğa sahip piksellerin bulanıklaştırma için dikkate alınmasını sağlar. Böylece kenarlardaki pikseller büyük yoğunluk varyasyonuna sahip olacağından kenarları korur.

Aşağıdaki örneklerde bilateral filtre kullanılmıştır (Bağımsız değişkenlerle ilgili ayrıntılar için dokümanları ziyaret edin).

```
bulanıklık = cv2 . bilateralFilter ( img , 9 , 75 , 75 )
```

Sonuç:



Bakın, yüzeydeki doku gitti, ancak kenarlar hala korunuyor.

4.5-)Morfolojik Dönüşümler

Hedef

Bu bölümde,

- Erozyon, Dilatasyon, Açıılış, Kapanış vb.Gibi farklı morfolojik işlemleri öğreneceğiz.
- Farklı fonksiyonlar göreceğiz: `cv2.erode ()` , `cv2.dilate ()` , `cv2.morphologyEx ()` vb.

teori

Morfolojik dönüşümler görüntü şekline dayanan bazı basit işlemlerdir. Normalde ikili görüntülerde gerçekleştirilir. İki girişe ihtiyacı vardır, biri orijinal imgemiz, ikincisi ise **yapının** doğasına karar veren **yapıllandırma elemanı** veya **çekirdek** olarak adlandırılır . İki temel morfolojik operatör Erozyon ve Dilatasyon'dur. Ardından Açıılış, Kapanış, Gradyan vb. Varyant formları da devreye girer. Aşağıdaki görüntü yardımıyla bunları tek tek göreceğiz:



1. Erozyon

Erozyonun temel fikri sadece toprak erozyonu gibidir, ön plan nesnesinin sınırlarını aşındırır (Her zaman ön planı beyaz tutmaya çalışın). Peki ne yapıyor? Çekirdek görüntünün içinde kayar (2B evrişimde olduğu gibi). Orijinal görüntüdeki bir piksel (1 veya 0) yalnızca çekirdeğin altındaki tüm pikseller 1 ise, aksi takdirde aşınır (sıfıra yapılır) 1 olarak kabul edilir.

Böylece olan şey, çekirdeğin büyüklüğüne bağlı olarak sınır yakınındaki tüm piksellerin atılacağıdır. Böylece ön plan nesnesinin kalınlığı veya boyutu azalır veya görüntüde

sadece beyaz bölge azalır. Küçük beyaz gürültüleri (renk boşluğu bölümünde gördüğümüz gibi) kaldırırmak, bağlı iki nesneyi çıkarmak vb. İçin yararlıdır.

Burada, bir örnek olarak, 5x5 çekirdek ile dolu bir çekirdek kullanacağım. Nasıl çalıştığını görelim:

```
import cv2
import numpy as np

img = cv2.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```

Result:



2. Dilatasyon

Erozyonun tam tersidir. Burada, çekirdeğin altındaki en az bir piksel '1' ise bir piksel ögesi '1'dir. Böylece görüntüdeki beyaz bölge artar veya ön plan nesnesinin boyutu artar. Normalde, gürültü giderme gibi durumlarda, erozyonu dilatasyon takip eder. Çünkü, erozyon beyaz sesleri çıkarır, ama aynı zamanda nesnemizi de küçültür. Yani onu genişletiyoruz. Gürültü gittiği için geri gelmeyecekler, ancak nesne alanımız artıyor. Ayrıca bir nesnenin kırık kısımlarının birleştirilmesinde de yararlıdır.

```
dilation = cv2.dilate(img,kernel,iterations = 1)
```

Result:



3. Açılış

Açılma, erozyonun ardından dilatasyonun başka bir adıdır. Yukarıda açıkladığımız gibi gürültüyü gidermede yararlıdır. Burada `cv2.morphologyEx()` fonksiyonunu kullanıyoruz

```
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```

Result:



4. Kapanış

Kapanış, Açılmaya, Dilatasyon ve ardından Erozyon'un tersidir . Ön plandaki nesnelerin içindeki küçük deliklerin veya nesnenin üzerindeki küçük siyah noktaların kapatılmasında faydalıdır.

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

Result:



5. Morfolojik Gradyan

Bir görüntünün genişlemesi ve erozyonu arasındaki farktır.

Sonuç, nesnenin anahattı gibi görünecektir.

```
gradient = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```

Result:



6. Silindir Şapka

Giriş görüntüsü ile görüntünün açılması arasındaki farktır. Aşağıdaki örnek 9x9 çekirdek için yapılmıştır.

```
tophat = cv2.morphologyEx(img, cv2.MORPH_TOPHAT, kernel)
```

Result:



7. Siyah Şapka

Giriş görüntüsünün kapanması ile giriş görüntüsünün farkı arasındaki farktır.

```
blackhat = cv2.morphologyEx(img, cv2.MORPH_BLACKHAT, kernel)
```

Result:



Yapılandırma Elemanı

Numpy'nin yardımıyla önceki örneklerde manuel olarak yapılandırma ögesi oluşturduk. Dikdörtgen şeklidir. Ancak bazı durumlarda eliptik / dairesel şekilli çekirdekler ihtiyacınız olabilir. Bu nedenle, **OpenCV'nin cv2.getStructuringElement ()** işlevi vardır . Sadece çekirdeğin şeklini ve boyutunu geçersiniz, istenen çekirdeği alırsınız.

```
# Rectangular Kernel
>>> cv2.getStructuringElement(cv2.MORPH_RECT,(5,5))
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)

# Elliptical Kernel
>>> cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)

# Cross-shaped Kernel
>>> cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))
array([[0, 0, 1, 0, 0],
```

```
[0, 0, 1, 0, 0],  
[1, 1, 1, 1, 1],  
[0, 0, 1, 0, 0],  
[0, 0, 1, 0, 0]], dtype=uint8)
```

4.6-) Görüntü Degradeleri Hedef

Bu bölümde şunları öğreneceğiz:

- Görüntü gradyanlarını, kenarlarını vb. Bulun
- Aşağıdaki işlevleri göreceğiz: `cv2.Sobel()`, `cv2.Scharr()`, `cv2.Laplacian()` vb.

teori

OpenCV, üç tür gradyan filtresi veya Yüksek geçirgen filtre sağlar, Sobel, Scharr ve Laplacian. Her birini göreceğiz.

1. Sobel ve Scharr Türevleri

Sobel operatörleri ortak bir Gaussian yumuşatma artı farklılaştırma işlemidir, bu nedenle gürültüye daha dayanıklıdır. Alınacak türevlerin yönünü dikey veya yatay olarak `belirtebilirsiniz` (sırasıyla argümanlar, `yorder` ve `xorder` ile). Çekirdek boyutunu da `ksize` bağımsız değişkeniyle `belirtebilirsiniz`. `Ksize = -1` ise, `3x3` Sobel filtreden daha iyi sonuç veren bir `3x3` Scharrfiltre kullanılır. Kullanılan çekirdekler için dokümanlara bakınız.

2. Laplacian Türevleri

$\Delta_{src} = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2}$ Her bir türevin Sobel türevleri kullanılarak bulunduğu ilişkinin verdiği görüntünün Laplacianını hesaplar. Eğer `ksize = 1`, aşağıdaki çekirdek filtreden edilmesi için kullanılır:

$$\text{kernel} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

kod

Aşağıdaki kod tüm operatörleri tek bir şemada göstermektedir. Tüm çekirdekler `5x5` boyutundadır. Sonuç `np.uint8` türünde sonuç elde etmek için çıktı görüntüsünün derinliği `-1` geçirilir.

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

```

img = cv2.imread('dave.jpg',0)

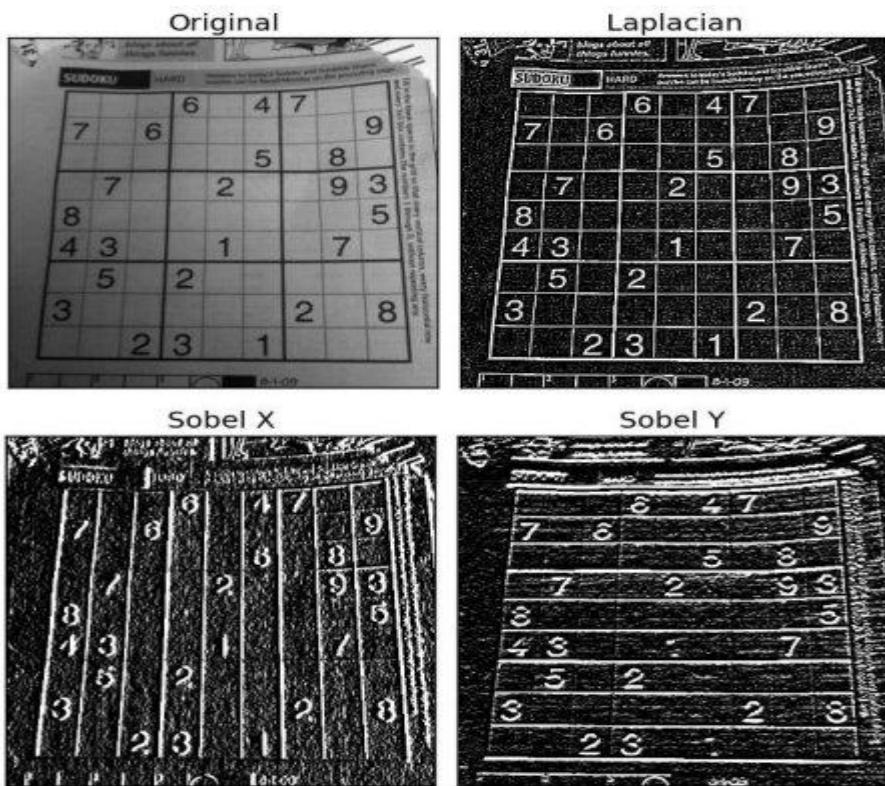
laplacian = cv2.Laplacian(img,cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])

plt.show()

```

Result:



Önemli Bir Konu!

Son örneğimizde, çıktı veri türü `cv2.CV_8U` veya `np.uint8`'dir. Ancak bununla ilgili küçük bir sorun var. Siyahanın Beyaza geçiş Pozitif eğim (pozitif değere sahiptir), Beyazdan Siyaha geçiş Negatif eğim olarak alınır (Negatif değere sahiptir). Bu nedenle, verileri `np.uint8`'e dönüştürdiğinizde, tüm negatif eğimler sıfırlanır. Basit bir ifadeyle, o kenarı kaçırmışsınızdır.

Her iki kenarı da tespit etmek istiyorsanız, daha iyi bir seçenek çıktı veri türünü cv2.CV_16S, cv2.CV_64F vb. Aşağıdaki kod, yatay bir Sobel filtresi ve sonuçlardaki fark için bu prosedürü göstermektedir.

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

img = cv2.imread('box.png',0)

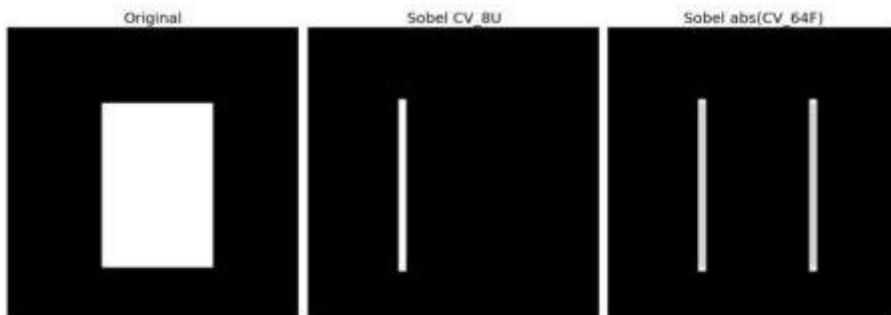
# Output dtype = cv2.CV_8U
sobelx8u = cv2.Sobel(img,cv2.CV_8U,1,0,ksize=5)

# Output dtype = cv2.CV_64F. Then take its absolute and convert to cv2.CV_8U
sobelx64f = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
abs_sobel64f = np.absolute(sobelx64f)
sobel_8u = np.uint8(abs_sobel64f)

plt.subplot(1,3,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(1,3,2),plt.imshow(sobelx8u,cmap = 'gray')
plt.title('Sobel CV_8U'), plt.xticks([]), plt.yticks([])
plt.subplot(1,3,3),plt.imshow(sobel_8u,cmap = 'gray')
plt.title('Sobel abs(CV_64F)'), plt.xticks([]), plt.yticks([])

plt.show()
```

Check the result below:



4.7-)Canny Kenar Algılama Hedef

Bu bölümde,

- Canny kenar algılama kavramı
- Bunun için OpenCV işlevleri: **cv2.Canny ()**

teori

Canny Edge Detection popüler bir kenar algılama algoritmasıdır. 1986 yılında John F. Canny tarafından geliştirildi. Çok aşamalı bir algoritma ve her aşamadan geçeceğiz.

1. Görültü Azaltma

Kenar algılama görüntüdeki parazite duyarlı olduğundan, ilk adım görüntüdeki paraziti 5×5 Gaussfiltresi ile kaldırmaktadır. Bunu daha önceki bölümlerde görmüştük.

2. Görüntünün Yoğunluk Degradesini Bulma

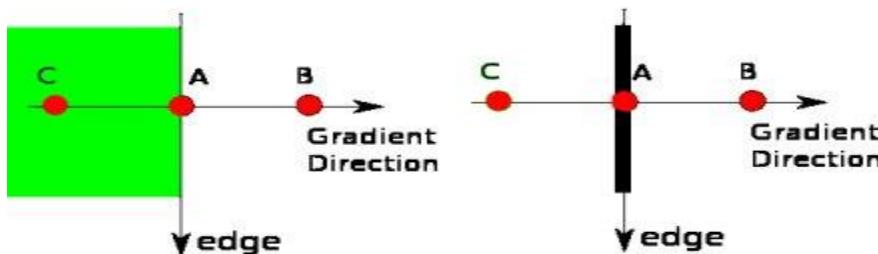
Daha sonra düzgünleştirilmiş görüntü yatay yönde (G_x) ve dikey yönde (G_y) ilk türevi elde etmek için yatay ve dikey yönde bir Sobel çekirdeği ile filtrelenir. Bu iki görüntüden, her piksel için kenar gradyanını ve yönünü aşağıdaki gibi bulabiliriz:

$$\text{Edge-Gradient } (G) = \sqrt{G_x^2 + G_y^2}$$
$$\text{Angle } (\theta) = \tan^{-1} \left(\frac{G_y}{G_x} \right)$$

Degradde yönü her zaman kenarlara diktir. Dikey, yatay ve iki çapraz yönü temsil eden dört açıdan birine yuvarlanır.

3. Maksimum Olmayan Bastırma

Degradde büyülüük ve yön elde edildikten sonra, kenarı teşkil etmeyecek istenmeyen pikselleri kaldırmak için tam bir görüntü taraması yapılır. Bunun için, her pikselde, piksel mahallesinde gradyan yönünde yerel bir maksimum olup olmadığı kontrol edilir. Aşağıdaki resmi kontrol edin:



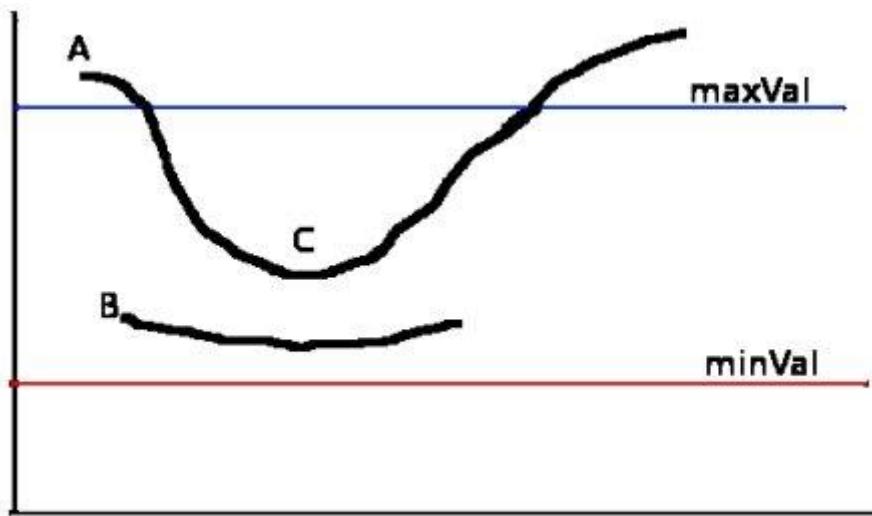
A noktası kenardadır (dikey yönde). Gradyan yönü kenara normaldir. B ve C noktaları gradyan yönündedir. Bu nedenle A noktası, yerel bir maksimum oluşturup oluşturmadığını görmek için B ve C noktaları ile kontrol edilir. Eğer öyleyse, bir sonraki aşama için düşünülür, aksi takdirde bastırılır (sıfıra konur).

Kısacası, elde ettiğiniz sonuç “ince kenarları” olan ikili bir görüntündür.

4. Histerezis Eşiği

Bu aşama, tüm kenarların hangilerinin gerçekten kenar olduğunu ve hangilerinin olmadığına karar verir. Bunun için iki eşik değerine ihtiyacımız var, $minVal$ ve $maxVal$. Yoğunluğu gradyanı $maxVal$ değerinden fazla olan kenarların kenar olduğundan ve $minVal$ değerinin altındaki kenarların kenar olmadığından emin olun, böylece atılır. Bu iki eşik

arasında kalanlar, bağlantılarına göre kenarlar veya kenar olmayanlar olarak sınıflandırılır. Eğer "kesin kenarlı" piksellere bağlanırlarsa, kenarların bir parçası olarak kabul edilirler. Aksi takdirde, bunlar da atılır. Aşağıdaki resme bakın:



A kenarı, "kesin kenar" olarak kabul edilen *maxVal* değerinin üzerindeyidir. C kenarı *maxVal* değerinin altındamasına rağmen, A kenarına bağlanır, böylece geçerli kenar olarak da kabul edilir ve tam eğriyi elde ederiz. Ancak B kenarı, *minVal*'in üstünde olmasını ve C kenarı ile aynı bölgede olmasına rağmen, herhangi bir "kesin kenara" bağlı değildir, bu nedenle atılır. Bu nedenle, doğru sonucu elde etmek için *minVal* ve *maxVal*'ı seçmemiz çok önemlidir.

Bu aşama ayrıca kenarların uzun çizgiler olduğu varsayımdaki küçük piksel seslerini de ortadan kaldırır.

Sonunda elde ettiğimiz görüntüdeki güçlü kenarlar.

OpenCV'de Canny Edge Algılama

OpenCV, yukarıdakilerin tümünü `cv2.Canny()` gibi tek bir işlev koyar. Nasıl kullanılacağını göreceğiz. İlk argüman girdi imajımızdır. İkinci ve üçüncü argümanlar sırasıyla *minVal* ve *maxVal*'ımızdır. Üçüncü argüman *aperture_size*. Görüntü gradyanlarını bulmak için kullanılan Sobel çekirdeğinin boyutudur. Varsayılan olarak 3'tür. Son bağımsız değişken, degrade büyütüğünü bulmak için denklemi belirten *L2gradient*'dır. Eğer öyleyse Doğru, aksi durumda bu işlevini kullanır, daha doğru olduğu yukarıda belirtilen denklemi kullanır: $\text{Edge Gradient } (G) = |G_x| + |G_y|$. Varsayılan olarak Yanlış .

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

img = cv2.imread('messi5.jpg',0)
edges = cv2.Canny(img,100,200)
```

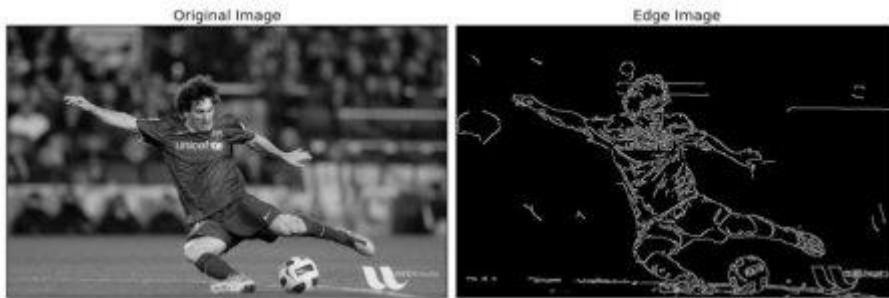
```

plt.subplot(121),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(edges,cmap = 'gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])

plt.show()

```

See the result below:



4.8-) Görüntü Piramitleri Hedef

Bu bölümde,

- Görüntü Piramitleri hakkında bilgi edineceğiz
- Yeni bir meyve olan “Orapple” yaratmak için Görüntü piramitlerini kullanacağız
- Şu işlevleri göreceğiz: `cv2.pyrUp()` , `cv2.pyrDown()`

teori

Normalde, sabit boyutlu bir görüntüyle çalışırdık. Ancak bazı durumlarda, aynı görüntünün farklı çözünürlükteki görüntülerle çalışmamız gereklidir. Örneğin, bir görüntüde yüz gibi bir şey ararken, nesnenin görüntüde hangi boyutta bulunacağından emin değiliz. Bu durumda, farklı çözünürlüğe sahip bir dizi görüntü oluşturmamız ve tüm görüntülerde nesne aramamız gerekecektir. Farklı çözünürlükteki bu görüntü kümelerine Görüntü Piramitleri denir (çünkü en küçük görüntüye ve üstte en büyük görüntüye sahip bir yığın halinde tutulduğunda piramit gibi görünürler).

İki tür Görüntü Piramidi vardır. 1) Gauss Piramidi ve 2) Laplacian Piramidi

Gauss Piramidi'nde daha yüksek seviye (Düşük çözünürlük), Alt seviye (daha yüksek çözünürlük) görüntüde ardışık satırlar ve sütunlar kaldırılarak oluşturulur. Daha sonra, daha yüksek seviyedeki her piksel, alt düzeydeki 5 pikselden gauss ağırlıklarına yapılan katkı ile oluşturulur. Böylece $M \times N$ görüntü imaj haline gelir $M/2 \times N/2$. Böylece alan orijinal alanın dörtte birine düşer. Buna Oktav denir. Piramitte yukarı çıktıktan sonra desen devam eder (yani çözünürlük azalır). Benzer şekilde genişlerken, alan her seviyede 4 kez olur. Gauss piramitlerini `cv2.pyrDown()` ve `cv2.pyrUp()` işlevlerini kullanarak bulabiliriz .

```
img = cv2.imread('messi5.jpg')
```

```
lower_reso = cv2.pyrDown(higher_reso)
```

Below is the 4 levels in an image pyramid.



Artık **cv2.pyrUp()** işleviyle görüntü piramidine **inebilirsiniz**.

```
higher_reso2 = cv2.pyrUp(lower_reso)
```

Unutmayın **higher_reso2** için eşit değildir **higher_reso** çözünürlüğü azaltmak kez, bilgileri gevşek çünkü. Aşağıdaki görüntü, önceki durumda en küçük görüntülerden oluşturulan piramidin 3 kat aşağısındaadır. Orijinal görüntü ile karşılaştırın:

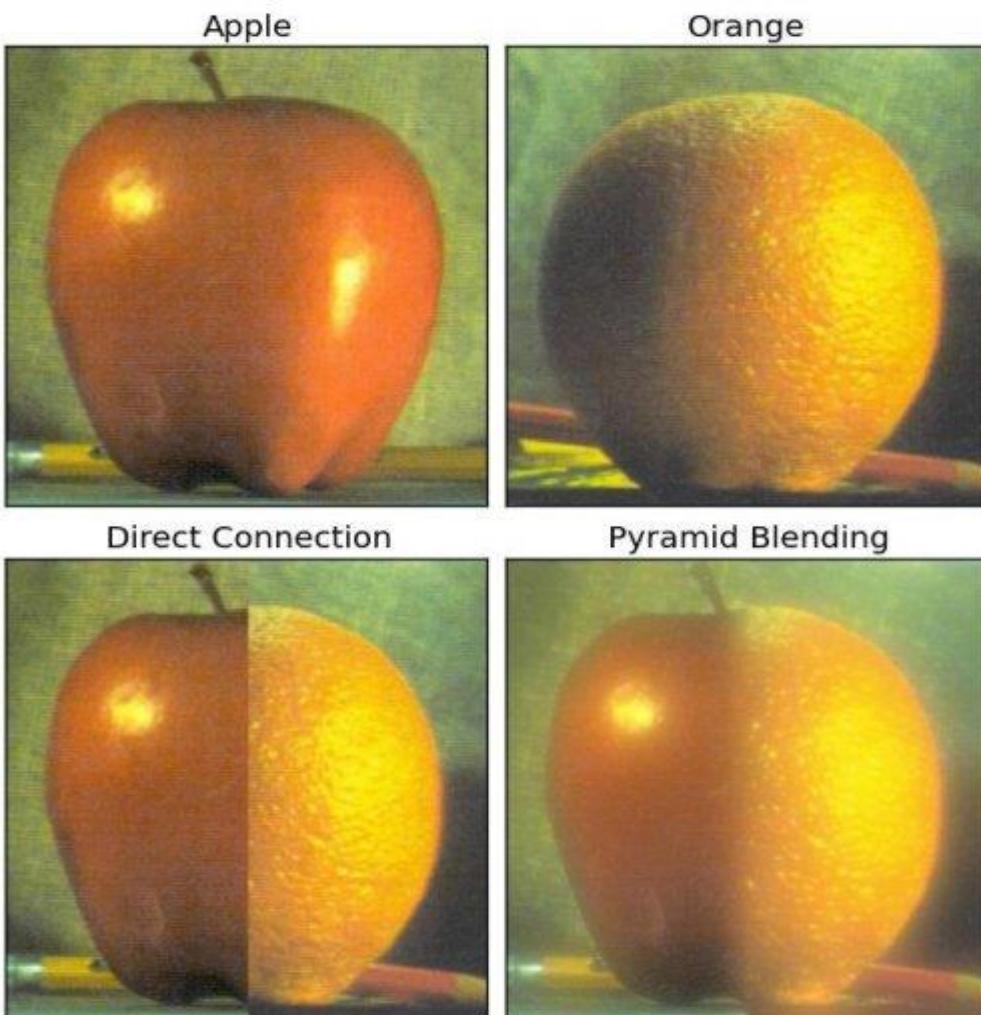


Laplacian Piramitleri Gauss Piramitlerinden oluşur. Bunun için özel bir işlev yoktur. Laplacian piramit görüntüleri sadece kenar görüntüler gibidir. Elemanlarının çoğu sıfırdır. Görüntü sıkıştırma kullanılırlar. Laplacian Piramidi'nde bir seviye, Gauss Piramidi'ndeki o seviye ile Gauss Piramidi'ndeki üst seviyesinin genişletilmiş versiyonu arasındaki farktan oluşur. Laplacian seviyesinin üç seviyesi aşağıdaki gibi görünecektir (kontrast içeriği geliştirmek için ayarlanır):



Piramitler Kullanarak Görüntü Karıştırma

Piramitlerin bir uygulaması Görüntü Karıştırmadır. Örneğin, görüntü dikişinde iki görüntüyü birlikte istiflemeniz gereklidir, ancak görüntüler arasındaki süreksızlıklar nedeniyle iyi görünmeyecektir. Bu durumda, Piramitlerle görüntü karıştırma, görüntülerde fazla veri bırakmadan kesintisiz karıştırma sağlar. Bunun klasik bir örneği, portakal ve elma olmak üzere iki meyvenin harmanlanmasıdır. Ne dediğimi anlamak için sonucu şimdi görün:



Lütfen ek kaynaklardaki ilk referansı kontrol edin, görüntü harmanlama, Laplacian Piramitleri vb. ile ilgili tam şematik ayrıntılara sahiptir.

1. Elma ve portakalın iki görüntüsünü yükleyin
2. Elma ve portakal için Gauss Piramitlerini bulun (bu özel örnekte seviye sayısı 6'dır)
3. Gauss Piramitlerinden Lapla Piramitlerini bulun
4. Laplacian Piramitlerinin her seviyesinde elmanın sol yarısına ve portakalın sağ yarısına katılın
5. Son olarak bu ortak görüntü piramitlerinden orijinal görüntüyü yeniden yapılandırın.

Tam kod aşağıdadır. (Basitlik amacıyla, her adım ayrı ayrı yapılır, bu da daha fazla bellek alabilir. İsterseniz optimize edebilirsiniz).

```
import cv2
import numpy as np,sys

A = cv2.imread('apple.jpg')
B = cv2.imread('orange.jpg')

# generate Gaussian pyramid for A
G = A.copy()
gpA = [G]
for i in xrange(6):
    G = cv2.pyrDown(G)
    gpA.append(G)

# generate Gaussian pyramid for B
G = B.copy()
gpB = [G]
for i in xrange(6):
    G = cv2.pyrDown(G)
    gpB.append(G)

# generate Laplacian Pyramid for A
lpA = [gpA[5]]
for i in xrange(5,0,-1):
    GE = cv2.pyrUp(gpA[i])
    L = cv2.subtract(gpA[i-1],GE)
    lpA.append(L)

# generate Laplacian Pyramid for B
lpB = [gpB[5]]
for i in xrange(5,0,-1):
    GE = cv2.pyrUp(gpB[i])
    L = cv2.subtract(gpB[i-1],GE)
    lpB.append(L)

# Now add left and right halves of images in each level
LS = []
for la,lb in zip(lpA,lpB):
    rows,cols,dpt = la.shape
    ls = np.hstack((la[:,0:cols/2], lb[:,cols/2:]))
    LS.append(ls)

# now reconstruct
ls_ = LS[0]
for i in xrange(1,6):
    ls_ = cv2.pyrUp(ls_)
    ls_ = cv2.add(ls_, LS[i])

# image with direct connecting each half
real = np.hstack((A[:,0:cols/2],B[:,cols/2:]))

cv2.imwrite('Pyramid_blending2.jpg',ls_)
cv2.imwrite('Direct_blending.jpg',real)
```

4.9.1-)Kontürler: Başlarken Hedef

- Kontürlerin ne olduğunu anlayın.
- Kontür bulmayı, kontur çizmeyi vb.
- Şu işlevleri göreceksiniz: `cv2.findContours()` , `cv2.drawContours()`

Kontür nedir?

Kontürler, aynı renk veya yoğunluğa sahip tüm sürekli noktaları (sınır boyunca) birleştiren bir eğri olarak açıklanabilir. Konturlar şekil analizi ve nesne algılama ve tanıma için kullanışlı bir araçtır.

- Daha iyi doğruluk için ikili görüntüler kullanın. Bu nedenle, konturları bulmadan önce eşik veya keskin kenar algılama uygulayın.
- `findContours` işlevi kaynak görüntüyü değiştirir. Bu nedenle, konturları bulduktan sonra bile kaynak görüntü istiyorsanız, zaten diğer bazı değişkenlerde saklayın.
- OpenCV'de kontur bulmak, siyah arka plandan beyaz nesne bulmak gibidir. Unutmayın, bulunacak nesne beyaz ve arka plan siyah olmalıdır.

İkili bir görüntünün konturlarını nasıl bulacağımızı görelim:

```
import numpy as np
import cv2

im = cv2.imread('test.jpg')
imggray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(imggray, 127, 255, 0)
contours, hierarchy =
cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Bkz. `Cv2.findContours()` fonksiyonunda üç argüman vardır, birincisi kaynak görüntündür, ikincisi kontur alma modudur, üçüncüsü kontur yaklaşım yöntemidir. Ve konturları ve hiyerarşiyi çıktılar. konturlar görüntüdeki tüm konturların bir Python listesiidir. Her bir kontur, nesnenin sınır noktalarının Numpy (x, y) koordinat dizisidir.

Not İkinci ve üçüncü argümanları ve hiyerarşiyi daha sonra ayrıntılı olarak tartışacağız. O zamana kadar, kod örneğinde kendilerine verilen değerler tüm görüntüler için iyi çalışır.

Kontür nasıl çizilir?

Konturları çizmek için `cv2.drawContours` işlevi kullanılır. Sınır noktaları olması koşuluyla herhangi bir şekil çizmek için de kullanılabilir. İlk argüman kaynak görüntündür, ikinci argüman bir Python listesi olarak geçirilmesi gereken konturlardır, üçüncü argüman kontur indeksidir (tek tek kontur çizerken faydalıdır. Tüm konturları çizmek için -1 geçisi) ve kalan argümanlar renk, kalınlıkta vb.

Bir görüntündeki tüm konturları çizmek için:

```
cv2.drawContours(img, contours, -1, (0,255,0), 3)
```

Tek bir kontur çizmek için 4. kontur deyin:

```
cv2.drawContours(img, contours, 3, (0,255,0), 3)
```

Ancak çoğu zaman, aşağıdaki yöntem yararlı olacaktır:

```
cnt = contours[4]
cv2.drawContours(img, [cnt], 0, (0,255,0), 3)
```

Not Son iki yöntem aynıdır, ancak ileri gittiğinizde sonuncunun daha kullanışlı olduğunu göreceksiniz.

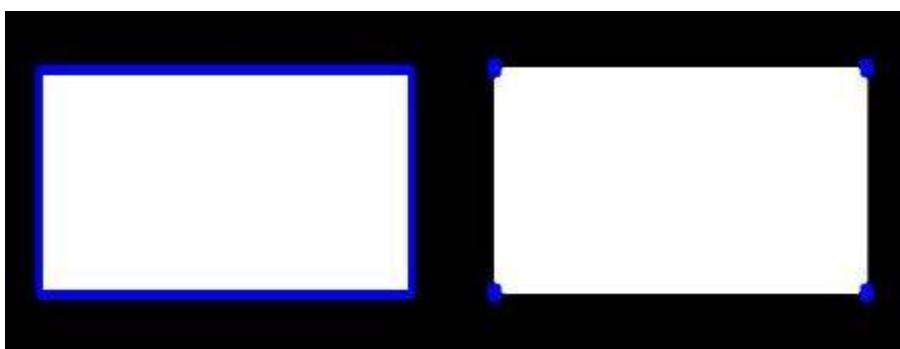
Kontur Yaklaşırma Yöntemi

Bu cv2.findContours işlevindeki üçüncü bağımsız değişkendir . Aslında neyi ifade ediyor?

Yukarıda, konturların aynı yoğunlukta bir şeklin sınırları olduğunu söyledik. Bir şeklin sınırının (x, y) koordinatlarını saklar. Fakat tüm koordinatları saklıyor mu? Bu, bu kontur yaklaşma yöntemi ile belirtilir.

Eğer başarılı olursa cv2.CHAIN_APPROX_NONE , tüm sınır noktalarının saklanır. Ama aslında tüm noktalara ihtiyacımız var mı? Örneğin, düz bir çizginin konturunu bulduk. Bu çizgiyi temsil etmek için çizgideki tüm noktalara mı ihtiyacınız var? Hayır, bu çizginin sadece iki uç noktasına ihtiyacımız var. Bu ne cv2.CHAIN_APPROX_SIMPLE yapar. Tüm gereksiz noktaları kaldırır ve konturu sıkıştırır, böylece bellek tasarrufu sağlar.

Aşağıdaki bir dikdörtgenin görüntüsü bu teknigi göstermektedir. Kontur dizisindeki tüm koordinatlara bir daire çizin (mavi renkte çizilmiş). İlk görüntü cv2.CHAIN_APPROX_NONE (734 puan) ile alındığım noktaları, ikinci görüntü cv2.CHAIN_APPROX_SIMPLE (sadece 4 puan) ile alındığım noktaları gösteriyor. Bakın, ne kadar hafıza kazandırır !!!



4.9.2-)Kontur Özellikleri Hedef

Bu yazında öğreneceğiz

- Alan, çevre, centroid, sınırlayıcı kutu vb.Gibi konturların farklı özelliklerini bulmak için

- Konturlarla ilgili birçok işlev göreceksiniz.

1. Anlar

Görüntü anları, nesnenin kütle merkezi, nesnenin alanı vb. Gibi bazı özellikleri hesaplamana yardımcı olur. [Görüntü Anları'ndaki wikipedia sayfasını](#) inceleyin

Cv2.moments () fonksiyonu hesaplanan tüm moment değerlerinin bir sözlüğünü verir. Aşağıya bakınız:

```
import cv2
import numpy as np

img = cv2.imread('star.jpg',0)
ret,thresh = cv2.threshold(img,127,255,0)
contours,hierarchy = cv2.findContours(thresh, 1, 2)

cnt = contours[0]
M = cv2.moments(cnt)
print M
```

Bu anlardan alan, centroid vb. Gibi yararlı verileri elde edebilirsiniz. Centroid, ilişkiler tarafından verilir $C_x = \frac{M_{10}}{M_{00}}$ $C_y = \frac{M_{01}}{M_{00}}$. Bu şöyle yapılabilir:

```
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])
```

2. Kontur Alanı

Kontur alanı **cv2.contourArea ()** fonksiyonu tarafından veya **M ["m00"]** anlarından verilir .

```
area = cv2.contourArea(cnt)
```

3. Kontur Çevresi

Ark uzunluğu olarak da adlandırılır. **Cv2.arcLength ()** işlevi kullanılarak bulunabilir . İkinci argüman, şeklin kapalı bir kontur (Doğru iletilirse) mi yoksa sadece bir eğri mi olduğunu belirtir .

```
perimeter = cv2.arcLength(cnt,True)
```

4. Kontur Yaklaşımı

Kontur şeklini, belirttiğimiz hassasiyete bağlı olarak daha az sayıda köşe ile başka bir şeke yaklaşır. [Douglas-Peucker algoritmasının](#) bir uygulamasıdır . Algoritma ve tanıtım için wikipedia sayfasını kontrol edin.

Bunu anlamak için, bir görüntüde bir kare bulmaya çalıştığını varsayıp, ancak görüntüdeki bazı sorunlar nedeniyle mükemmel bir kare elde etmediğinizi, ancak "kötü bir şekil" (aşağıdaki ilk resimde gösterildiği gibi). Şimdi bu işlevi şekele yakınlaştırmak için kullanabilirsiniz. Burada, ikinci argümana konturdan yaklaşık konturuna maksimum mesafe olan `epsilon` denir. Bir doğruluk parametresidir. Doğru çıktıyi elde etmek için akıllıca bir `epsilon` seçimi gereklidir.

```
epsilon = 0.1*cv2.arcLength(cnt,True)
approx = cv2.approxPolyDP(cnt,epsilon,True)
```

Aşağıda, ikinci görüntüde, yeşil hat gösterir yakınsanan eğri `epsilon = % 10` arasında yay uzunluğu. Üçüncü resim Şekil aynı `epsilon = % 1` arasında yay uzunluğu. Üçüncü argüman eğrinin kapalı olup olmadığını belirtir.



5. Dışbükey Gövde

Dışbükey Gövde, kontur yaklaşımına benzeyecektir, ancak değildir (Her ikisi de bazı durumlarda aynı sonuçları sağlayabilir). Burada `cv2.convexHull()` işlevi, bir eğriyi konveksite kusurları açısından kontrol eder ve düzeltir. Genel olarak, dışbükey eğriler her zaman şişkin veya en azından düz olan eğrilerdir. İçine şişmişse, dışbükeylik kusurları denir. Örneğin, aşağıdaki el görüntüsünü kontrol edin. Kırmızı çizgi elin dışbükey gövdesini gösterir. Çift taraflı ok işaretleri, gövdenin konturlardan yerel maksimum sapmaları olan dışbükeylik kusurlarını gösterir.



Onun sözdizimi hakkında tartışmak için biraz şey var:

```
hull = cv2.convexHull(points[, hull[, clockwise[, returnPoints]]])
```

Bağımsız değişken ayrıntıları:

- **puanlar** geçtiğimiz konturlardır.
- **gövde** çıktı, normalde biz kaçının.
- **saat yönünde**: Yönlendirme bayrağı. Eğer öyleyse Doğru , çıkış dışbükey saat yönünde yönlendirilmiştir. Aksi takdirde, saat yönünün tersine yönlendirilir.
- **returnPoints** : Varsayılan olarak `True`. Sonra gövde noktalarının koordinatlarını döndürür. Eğer yanlış , bu gövde puanına karşılık gelen kontur noktalarının endekslerini döndürür.

Yukarıdaki görüntüdeki gibi bir dışbükey gövde elde etmek için aşağıdakiler yeterlidir:

```
hull = cv2.convexHull(cnt)
```

Ancak dışbükeylik hataları bulmak istiyorsanız, `returnPoints = False` değerini işaretmeniz gereklidir. Bunu anlamak için yukarıdaki dikdörtgen görüntüsünü alacağız. İlk önce konturunu `cnt` olarak bulduk . Şimdi konveks gövdesini `returnPoints = True` ile bulduk : Aşağıdaki değerleri aldım: `[[[234 202]], [[51 202]], [[51 79]], [[234 79]]]` dört köşe dikdörtgenin noktaları. Şimdi aynısını `returnPoints = False` ile yaparsam , şu sonucu alıyorum: `[[129], [67], [0], [142]]`. Bunlar konturlarda karşılık gelen noktaların endeksleridir. Örneğin, ilk değeri kontrol edin: `cnt[129] = [[234, 202]]` ilk sonuçla aynıdır (ve diğerleri için böyle devam eder).

Konveksite kusurları hakkında konuşduğumuzda bunu tekrar göreceksiniz.

6. Konveksliğin Kontrolü

Bir eğrinin dışbükey olup olmadığını kontrol etmek için bir işlev vardır, `cv2.isContourConvex()` . Doğru ya da yanlış olsun. Önemli bir şey değil.

```
k = cv2.isContourConvex(cnt)
```

7. Sınırlayıcı Dikdörtgen

İki tür sınırlayıcı dikdörtgen vardır.

7.a. Düz Sınırlayıcı Dikdörtgen

Düz bir dikdörtgendir, nesnenin dönüşünü dikkate almaz. Dolayısıyla sınırlayıcı dikdörtgenin alanı minimum olmayacağıdır. `Cv2.boundingRect()` işlevi tarafından bulunur .

(X, y), dikdörtgenin sol üst koordinatı ve (w, h) genişliği ve yüksekliği olsun.

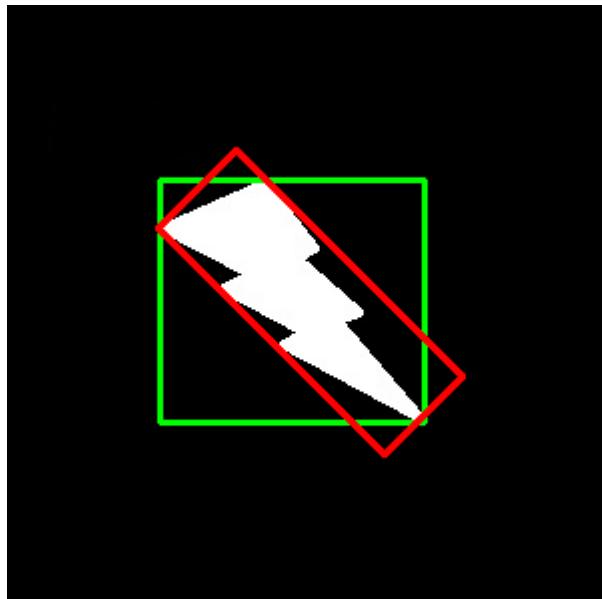
```
x,y,w,h = cv2.boundingRect(cnt)
cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
```

7.b. Döndürülmüş Dikdörtgen

Burada, sınırlayıcı dikdörtgen minimum alanla çizilir, bu nedenle dönüşü de dikkate alır. Kullanılan işlev **cv2.minAreaRect()** işlevidir . Aşağıdaki ayrıntıları içeren bir Box2D yapısı döndürür – (merkez (x, y), (genişlik, yükseklik), dönüş açısı). Ancak bu dikdörtgeni çizmek için dikdörtgenin 4 köşesine ihtiyacımız var. **Cv2.boxPoints()** işlevi ile elde edilir.

```
rect = cv2.minAreaRect(cnt)
box = cv2.boxPoints(rect)
box = np.int0(box)
cv2.drawContours(img,[box],0,(0,0,255),2)
```

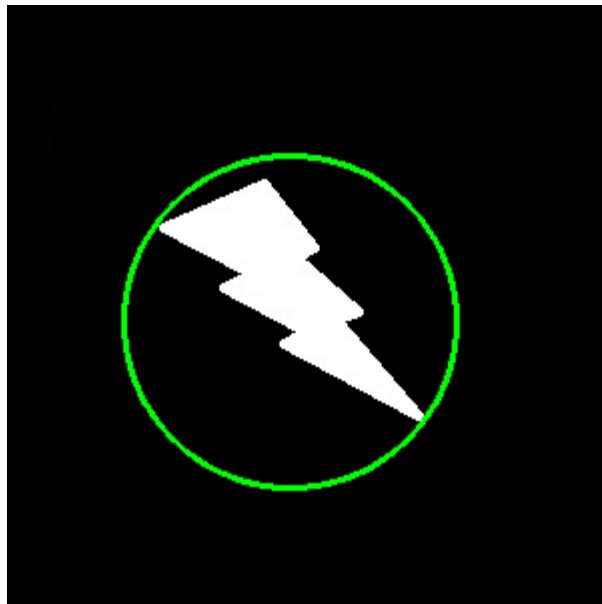
Her iki dikdörtgen de tek bir görüntüde gösterilir. Yeşil dikdörtgen normal sınırlayıcı doğrultuyu gösterir. Kırmızı dikdörtgen döndürülmüş doğrultudur.



8. Minimum Kapalı Çember

Sonra **cv2.minEnclosingCircle()** işlevini kullanarak bir nesnenin çemberini buluruz . Nesneyi minimum alanla tamamen kaplayan bir daire.

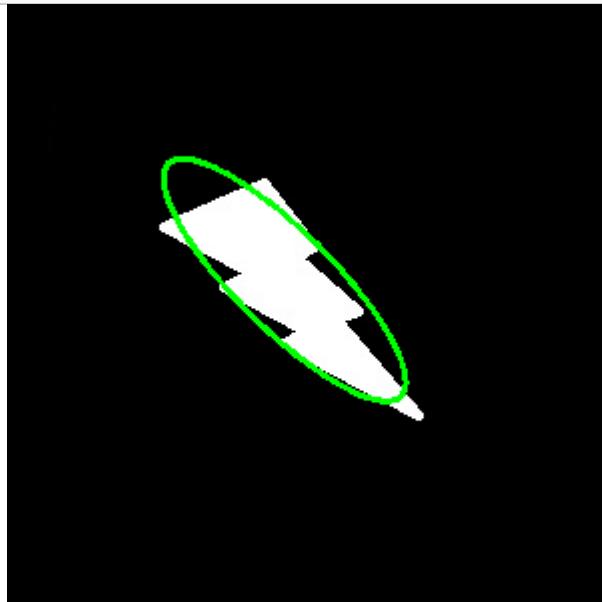
```
(x,y),radius = cv2.minEnclosingCircle(cnt)
center = (int(x),int(y))
radius = int(radius)
cv2.circle(img,center,radius,(0,255,0),2)
```



9. Elips Takma

Bir sonraki, bir elips bir nesneye sığdırılmaktır. Elipsin yazıldığı döndürülmüş dikdörtgeni döndürür.

```
ellipse = cv2.fitEllipse(cnt)
cv2.ellipse(img,ellipse,(0,255,0),2)
```

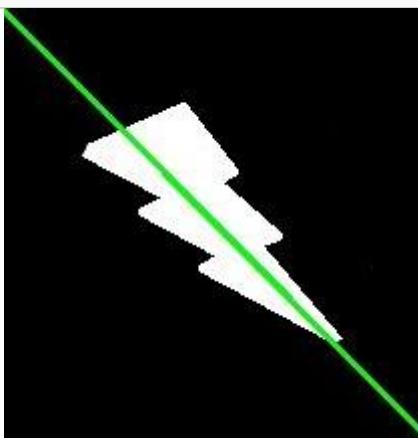


10. Hat Takma

Benzer şekilde, bir çizгиyi bir dizi noktaya sığdırabiliriz. Aşağıdaki resimde bir dizi beyaz nokta bulunmaktadır. Düz bir çizgiye yaklaşabiliris.

```
rows,cols = img.shape[:2]
[vx,vy,x,y] = cv2.fitLine(cnt, cv2.DIST_L2,0,0.01,0.01)
lefty = int((-x*vy/vx) + y)
```

```
righty = int(((cols-x)*vy/vx)+y)
cv2.line(img,(cols-1,righty),(0,lefty),(0,255,0),2)
```



4.9.3-)Kontur Özellikleri

Burada, Katılık, Eşdeğer Çap, Maske görüntüsü, Ortalama Yoğunluk [vb.Gibi](#) nesnelerin sık kullanılan bazı özelliklerini çıkarmayı öğreneceğiz .

(Not: Centroid, Alan, Çevre vb. De bu kategoriye aittir, ancak son bölümde gördük)

1. En Boy Oranı

Nesnenin sınırlayıcı doğrultusunun genişliğinin yüksekliğine oranıdır.

$$\text{Aspect Ratio} = \frac{\text{Width}}{\text{Height}}$$

```
x,y,w,h = cv2.boundingRect(cnt)
aspect_ratio = float(w)/h
```

2. Kapsam

Genişlik, kontur alanının sınırlayıcı dikdörtgen alanına oranıdır.

$$\text{Extent} = \frac{\text{Object Area}}{\text{Bounding Rectangle Area}}$$

```
area = cv2.contourArea(cnt)
x,y,w,h = cv2.boundingRect(cnt)
rect_area = w*h
extent = float(area)/rect_area
```

3. Sağlamlık

Sağlamlık, kontur alanının dışbükey gövde alanına oranıdır.

$$\text{Solidity} = \frac{\text{Contour Area}}{\text{Convex Hull Area}}$$

```
area = cv2.contourArea(cnt)
hull = cv2.convexHull(cnt)
hull_area = cv2.contourArea(hull)
solidity = float(area)/hull_area
```

4. Eşdeğer Çap

Eşdeğer Çap, alanı kontur alanıyla aynı olan dairenin çapıdır.

$$\text{Equivalent Diameter} = \sqrt{\frac{4 \times \text{Contour Area}}{\pi}}$$

```
area = cv2.contourArea(cnt)
equi_diameter = np.sqrt(4*area/np.pi)
```

5. Yönlendirme

Yön, nesnenin yönlendirildiği açıdır. Aşağıdaki yöntem ayrıca Ana Eksen ve Küçük Eksen uzunluklarını verir.

```
(x,y),(MA,ma),angle = cv2.fitEllipse(cnt)
```

6. Maske ve Piksel Noktaları

Bazı durumlarda, bu nesneyi içeren tüm noktalara ihtiyacımız olabilir. Aşağıdaki gibi yapılabilir:

```
mask = np.zeros(imggray.shape,np.uint8)
cv2.drawContours(mask,[cnt],0,255,-1)
pixelpoints = np.transpose(np.nonzero(mask))
#pixelpoints = cv2.findNonZero(mask)
```

Burada, biri Numpy işlevlerini kullanan, diğeri OpenCV işlevini (son yorumlanan satır) kullanan iki yöntem de aynı şekilde yapılır. Sonuçlar da aynı, ancak küçük bir farkla. Numpy koordinatları (**satır, sütun**) biçiminde verirken, OpenCV koordinatları (**x, y**) biçiminde verir. Yani temel olarak cevaplar birbirinin yerine geçecek. **Satır = x** ve **sütun = y** olduğuna dikkat edin .

7. Maksimum Değer, Minimum Değer ve konumları

Bu parametreleri bir maske görüntüsü kullanarak bulabiliriz.

```
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(imggray,mask = mask)
```

8. Ortalama Renk veya Ortalama Yoğunluk

Burada, bir nesnenin ortalama rengini bulabiliyoruz. Veya gri tonlamalı modda nesnenin ortalama yoğunluğu olabilir. Bunu yapmak için yine aynı maskeyi kullanıyoruz.

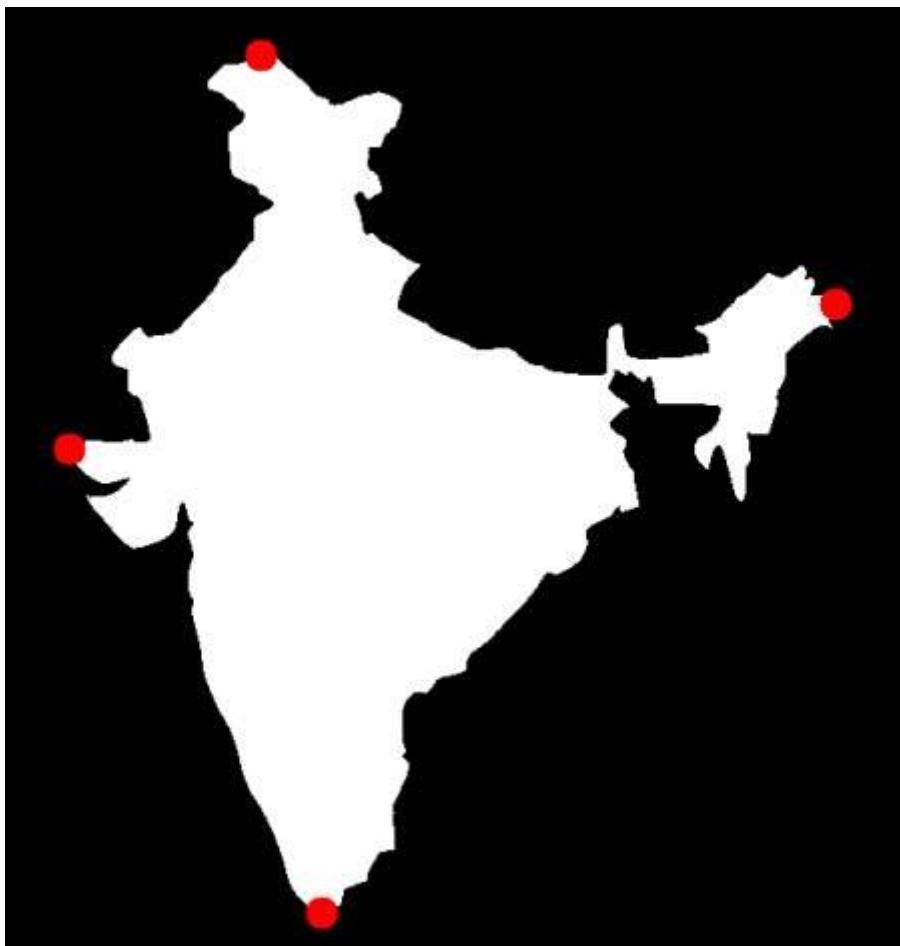
```
mean_val = cv2.mean(im,mask = mask)
```

9. Uç Noktalar

Uç Noktalar, nesnenin en üst, en alt, en sağ ve en soldaki noktaları anlamına gelir.

```
leftmost = tuple(cnt[cnt[:, :, 0].argmin()][0])
rightmost = tuple(cnt[cnt[:, :, 0].argmax()][0])
topmost = tuple(cnt[cnt[:, :, 1].argmin()][0])
bottommost = tuple(cnt[cnt[:, :, 1].argmax()][0])
```

Örneğin, Hindistan haritasına uygularsam, aşağıdaki sonucu elde ederim:



4.9.4-)Kontürler: Daha Fazla İşlev Hedef

Bu bölümde,

- Konveksite kusurları ve bunların bulunması.

- Bir noktadan çokgene en kısa mesafeyi bulma
- Farklı şekilleri eşleme

Teori ve Kod

1. Konveksite Hataları

Konturlar hakkında ikinci bölümde dışbükey gövde ne olduğunu gördük. Nesnenin bu gövdeden herhangi bir sapması, dışbükeylik hatası olarak kabul edilebilir.

OpenCV, bunu bulmak için hazır bir işlevle birlikte gelir, `cv2.convexityDefects()`. Temel bir işlev çağrıları aşağıdaki gibi görünecektir:

```
hull = cv2.convexHull(cnt,returnPoints = False)
defects = cv2.convexityDefects(cnt,hull)
```

Not Biz geçmek zorunda hatırlı `returnPoints = False` konveksite kusurlarını bulmak için, dışbükey gövde bulurken.

Her satırın şu değerleri içерdiği bir dizi döndürür - **[başlangıç noktası, bitiş noktası, en uzak nokta, en uzak noktaya yaklaşık mesafe]**. Bir görüntü kullanarak görselleştirebiliriz. Başlangıç noktasını ve bitiş noktasını birleştiren bir çizgi çiziyoruz, sonra en uzak noktada bir daire çiziyoruz. Döndürülen ilk üç değerin `cnt` indeksleri olduğunu unutmayın . Bu yüzden bu değerleri `cnt`'den getirmeliyiz .

```
import cv2
import numpy as np

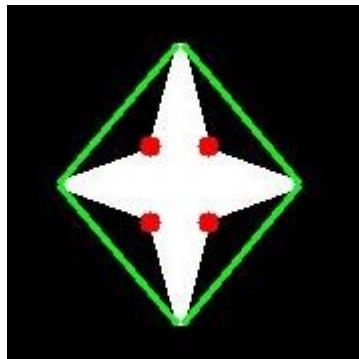
img = cv2.imread('star.jpg')
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(img_gray, 127, 255,0)
contours,hierarchy = cv2.findContours(thresh,2,1)
cnt = contours[0]

hull = cv2.convexHull(cnt,returnPoints = False)
defects = cv2.convexityDefects(cnt,hull)

for i in range(defects.shape[0]):
    s,e,f,d = defects[i,0]
    start = tuple(cnt[s][0])
    end = tuple(cnt[e][0])
    far = tuple(cnt[f][0])
    cv2.line(img,start,end,[0,255,0],2)
    cv2.circle(img,far,5,[0,0,255],-1)

cv2.imshow('img',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Ve sonuca bakın:



2. Nokta Çokgen Testi

Bu işlev görüntüdeki bir nokta ile bir kontur arasındaki en kısa mesafeyi bulur. Nokta kontur dışında olduğunda negatif, nokta içinde olduğunda pozitif, nokta konturda ise sıfır olan mesafeyi döndürür.

Örneğin, noktayı (50,50) aşağıdaki gibi kontrol edebiliriz:

```
dist = cv2.pointPolygonTest(cnt,(50,50),True)
```

İşlevde üçüncü argüman MeasDist'tır. Eğer öyleyse Doğru , bu imzalı mesafeyi bulur. Eğer yanlış , bu nokta içinde veya dışında veya kontur üzerinde olup olmadığına bakar (sırasıyla +1, -1, 0 döndürür).

Not Mesafeyi bulmak istemiyorsanız, üçüncü argümanın Yanlış olduğundan emin olun , çünkü bu zaman alıcı bir işlemdir. Yani, False yapmak yaklaşık 2-3X hız verir.

3. Maç Şekilleri

OpenCV, iki şekli veya iki konturu karşılaştırmamızı sağlayan ve benzerliği gösteren bir metrik döndüren bir **cv2.matchShapes ()** işleviyle birlikte gelir . Sonuç ne kadar düşük olursa, o kadar iyi eşleşir. Hu-moment değerlerine göre hesaplanır. Dokümanlarda farklı ölçüm yöntemleri açıklanmaktadır.

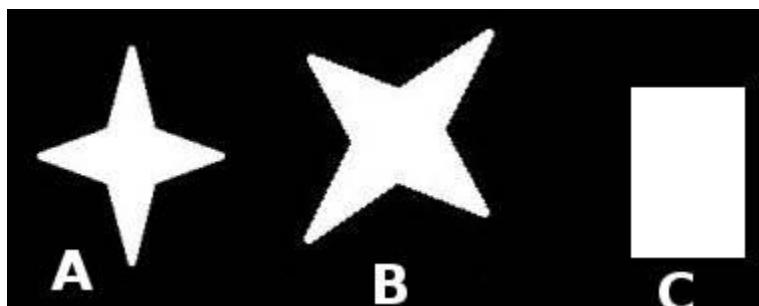
```
import cv2
import numpy as np

img1 = cv2.imread('star.jpg',0)
img2 = cv2.imread('star2.jpg',0)

ret, thresh = cv2.threshold(img1, 127, 255,0)
ret, thresh2 = cv2.threshold(img2, 127, 255,0)
contours,hierarchy = cv2.findContours(thresh,2,1)
cnt1 = contours[0]
contours,hierarchy = cv2.findContours(thresh2,2,1)
cnt2 = contours[0]

ret = cv2.matchShapes(cnt1,cnt2,1,0.0)
print ret
```

Aşağıda verilen farklı şekillerle şekilleri eşleştirmemeyi denedim:



Aşağıdaki sonuçları aldım:

- Resim A'yı kendisiyle eşleştirme = 0.0
- Resim A'yı Resim B ile Eşleştirme = 0.001946
- Resim A'yı Resim C ile Eşleştirme = 0.326911

Bakın, görüntü döndürme bile bu karşılaştırmayı pek etkilemez.

Ayrıca bakınız[Hu-Moments](#), çeviri, rotasyon ve ölçüye değişmeyen yedi andır. Yedinci çarpık değişmezdir. Bu değerler **cv2.HuMoments()** işlevi kullanılarak **bulunabilir**.

4.10.1-) Histogramlar – 1: Bul, Çiz, Analiz !!! Hedef

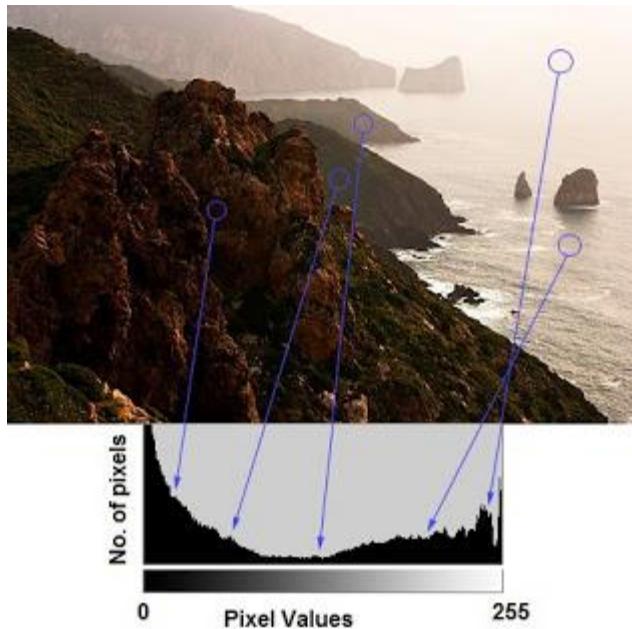
Öğrenmek

- Hem OpenCV hem de Numpy işlevlerini kullanarak histogramları bulun
- OpenCV ve Matplotlib işlevlerini kullanarak histogramları çizme
- Şu işlevleri göreceksiniz: **cv2.calcHist()** , **np.histogram()** vb.

teori

Peki histogram nedir? Histogramı, bir görüntünün yoğunluk dağılımı hakkında genel bir fikir veren bir grafik veya çizim olarak düşünebilirsiniz. X ekseninde piksel değerlerine (her zaman değil 0 ila 255 arasında değişir) ve Y eksenindeki görüntüdeki karşılık gelen piksel sayısına sahip bir çizimdir.

Görüntüyü anlamanın başka bir yoludur. Bir görüntünün histogramına bakarak, o görüntünün kontrasti, parlaklığı, yoğunluk dağılımı vb. Hakkında sezgiye sahip olursunuz. Bugün neredeyse tüm görüntü işleme araçları histogram üzerinde özellikler sunuyor. Aşağıda [Cambridge in Color web sitesinden](#) bir resim bulunmaktadır ve daha fazla ayrıntı için siteyi ziyaret etmenizi öneririm.



Görüntüyü ve histogramını görebilirsiniz. (Unutmayın, bu histogram renkli görüntü için değil gri tonlamalı görüntü için çizilir). Histogramın sol bölgesi görüntüdeki daha koyu piksellerin sayısını gösterirken, sağ bölge daha parlak piksellerin miktarını gösterir. Histogramdan, karanlık bölgenin daha parlak bir bölgeden daha fazla olduğunu ve orta tonların (orta aralıktaki piksel değerleri, yaklaşık 127 gibi) çok daha az olduğunu görebilirsiniz.

Histogram Bulun

Şimdi histogramın ne olduğu hakkında bir fikrimiz var, bunu nasıl bulacağımızı inceleyelim. Hem OpenCV hem de Numpy bunun için yerleşik fonksiyonla birlikte gelir. Bu işlevleri kullanmadan önce, histogramlarla ilgili bazı terminolojileri anlamamız gereklidir.

BINS : Yukarıdaki histogram her piksel değeri için piksel sayısını, yani 0 – 255 arasında gösterir. Yani yukarıdaki histogramı göstermek için 256 değere ihtiyacınız vardır. Ancak, tüm piksel değerleri için piksel sayısını ayrı olarak değil, bir piksel değerleri aralığındaki piksel sayısını bulmanız gerekiyorsa ne düşünün? örneğin, 0 ila 15, sonra 16 ila 31, ..., 240 ila 255 arasında yatan piksel sayısını bulmanız gereklidir. Histogramı temsil etmek için yalnızca 16 değere ihtiyacınız olacaktır. [Histogramlardaki OpenCV Derslerinde](#) verilen örnekte gösterilen budur .

Böylece, tüm histogramı 16 alt bölüme ayırmamız yeterlidir ve her alt bölümün değeri, içindeki tüm piksel sayısının toplamıdır. Bu her alt kısma “BIN” denir. İlk durumda, 256 (her piksel için bir tane) iken, yalnızca 16 olduğu kutu **sayısıdır** . **BINS** , OpenCV dokümanlarında **histSize** terimi ile temsil edilir .

DIMS : Verileri topladığımız parametre sayısıdır. Bu durumda, sadece bir şey, yoğunluk değeri ile ilgili veri toplarız. İşte burada 1.

RANGE : Ölçmek istediğiniz yoğunluk değerleri aralığıdır. Normal olarak, [0,256], yani tüm yoğunluk değerleri.

1. OpenCV'de Histogram Hesaplaması

Şimdi histogramı bulmak için **cv2.calcHist** () fonksiyonunu kullanıyoruz. Fonksiyonu ve parametrelerini tanıyalım:

cv2.calcHist (resimler, kanallar, maske, histSize, aralıklar [, hist [, birikir]])

1. **images**: uint8 veya float32 türünün kaynak görüntüsüdür. köşeli parantez içinde verilmelidir, yani “[img]”.
2. **kanallar**: köşeli parantez içinde de verilir. Histogramı hesapladığımız kanalın dizinidir. Örneğin, girdi gri tonlamalı bir görüntü ise değeri [0] olur. Renkli görüntü için, sırasıyla mavi, yeşil veya kırmızı kanalın histogramını hesaplamak için [0], [1] veya [2] 'yi geçirebilirsiniz.
3. **maske**: maske görüntüsü. Tam görüntünün histogramını bulmak için “Yok” olarak verilir. Ancak, görüntünün belirli bir bölgesinin histogramını bulmak istiyorsanız, bunun için bir maske görüntüsü oluşturmanız ve bunu maske olarak vermeniz gereklidir. (Daha sonra bir örnek göstereceğim.)
4. **histSize**: Bu bizim BIN sayımızı temsil eder. Köşeli parantez içinde verilmesi gereklidir. Tam ölçek için geçiyoruz [256].
5. **aralıkları**: bu bizim ARALIĞIMIZ. Normalde [0,256] 'dır.

Şimdi örnek bir görüntü ile başlayalım. Bir görüntüyü gri tonlamalı moda yükleyin ve tam histogramını bulun.

```
img = cv2.imread('home.jpg',0)
hist = cv2.calcHist([img],[0],None,[256],[0,256])
```

hist 256x1 dizisidir, her değer o görüntünün piksel sayısına karşılık gelen piksel sayısına karşılık gelir.

2. Numpy'de Histogram Hesaplaması

Numpy ayrıca **np.histogram** () işlevini de sağlar . Yani calcHist () işlevi yerine aşağıdaki satırı deneyebilirsiniz:

```
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

hist daha önce hesapladığımızla aynı. Ancak kutular 257 elemente sahip olacaktır, çünkü Numpy kutuları 0-0.99, 1-1.99, 2-2.99 vb. Olarak hesaplar. Böylece son aralık 255-

255.99 olacaktır. Bunu temsil etmek için, kutuların sonuna 256 eklerler. Ama 256'ya ihtiyacımız yok. 255'e kadar yeterli.

Ayrıca **bakınız** Numpy'nin başka bir işlevi vardır, **np.bincount ()** (yaklaşık 10X) **np.histogram ()**'dan çok daha hızlıdır. Yani tek boyutlu histogramlar için bunu daha iyi deneyebilirsiniz. **Np.bincount**'ta **minlength = 256** ayarlamayı unutmayın. Örneğin, **hist = np.bincount (img.ravel (), minlength = 256)**

Not OpenCV işlevi **np.histogram**ından () daha hızlıdır (yaklaşık 40X). Yani OpenCV fonksiyonu ile sopa.

Şimdi histogramları çizmeliyiz, ama nasıl?

Histogramları Çizme

Bunun iki yolu var,

1. Kısa Yol: Matplotlib çizim işlevlerini kullanın
2. Uzun Yol: OpenCV çizim işlevlerini kullanın

1. Matplotlib Kullanımı

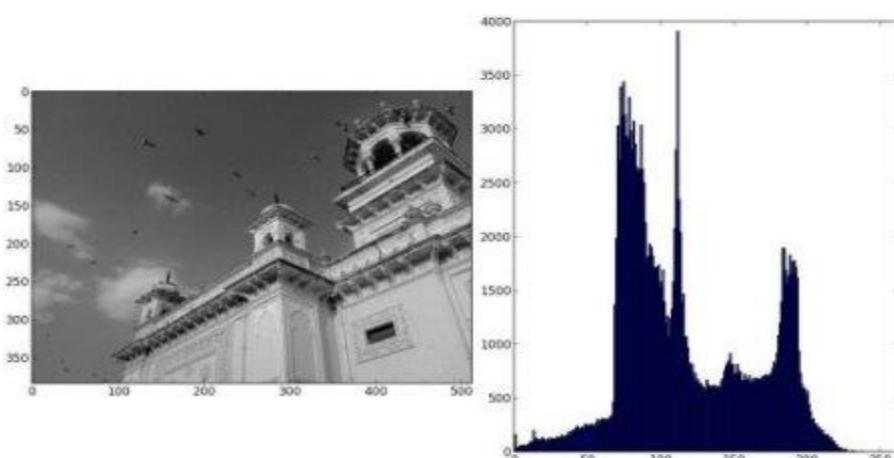
Matplotlib bir histogram çizme işlevi ile birlikte gelir: **matplotlib.pyplot.hist ()**

Doğrudan histogramı bulur ve çizer. Histogramı bulmak için **calcHist ()** veya **np.histogram ()** işlevini kullanmanız gerekmektedir. Aşağıdaki koda bakın:

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

img = cv2.imread('home.jpg',0)
plt.hist(img.ravel(),256,[0,256]); plt.show()
```

Aşağıdaki gibi bir arsa alacaksınız:

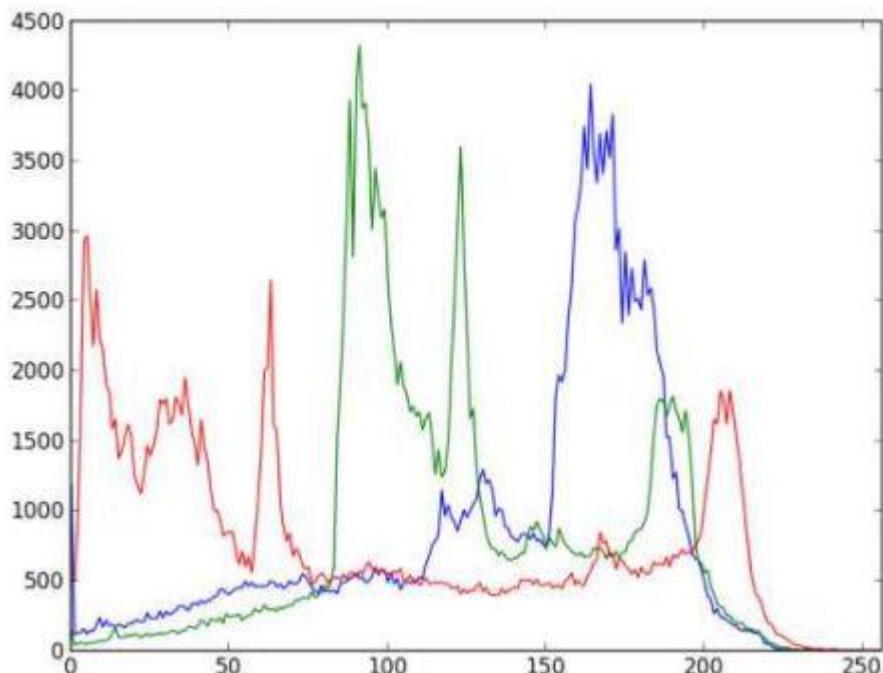


Veya normal matplotlib grafiğini kullanabilirsiniz, ki bu da BGR grafiğine iyi gelir. Bunun için önce histogram verilerini bulmanız gereklidir. Aşağıdaki kodu deneyin:

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

img = cv2.imread('home.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
    histr = cv2.calcHist([img],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

Sonuç:



Yukarıdaki grafikten, mavinin görüntüde bazı yüksek değerli alanlara sahip olduğu sonucuna varabilirsiniz (açıkçası gökyüzüne bağlı olması gereklidir)

2. OpenCV Kullanımı

Burada histogramların değerlerini, bin değerleri ile birlikte x, y koordinatlarına benzeyebileceğiniz şekilde ayarlayabilirsiniz, böylece yukarıdaki gibi aynı görüntüyü oluşturmak için `cv2.line()` veya `cv2.polyline()` işlevini kullanarak çizebilirsiniz. Bu, OpenCV-Python2 resmi örnekleri ile zaten mevcuttur. [Kodu Kontrol Edin](#)

Maske Uygulaması

Tam görüntünün histogramunu bulmak için cv2.calcHist () yöntemini kullandık. Bir görüntünün bazı bölgelerinin histogramlarını bulmak isterseniz ne olur? Histogramı bulmak istediğiniz bölgede beyaz renkte ve aksi takdirde siyah olan bir maske görüntüsü oluşturun. Sonra bunu maske olarak geçirin.

```
img = cv2.imread('home.jpg',0)

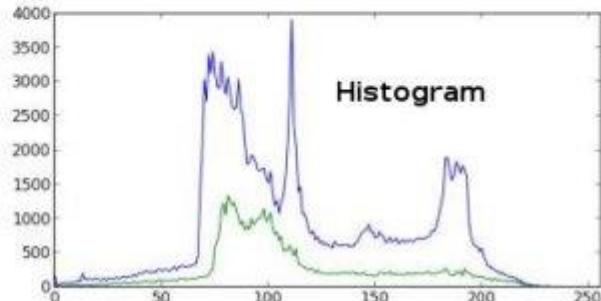
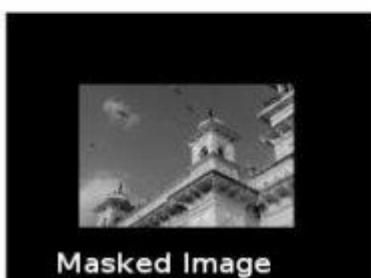
# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
masked_img = cv2.bitwise_and(img,img,mask = mask)

# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv2.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv2.calcHist([img],[0],mask,[256],[0,256])

plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask,'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
plt.xlim([0,256])

plt.show()
```

Sonuca bakın. Histogram grafiğinde mavi çizgi tam görüntünün histogramını gösterirken yeşil çizgi maskelenmiş bölgenin histogramını gösterir.



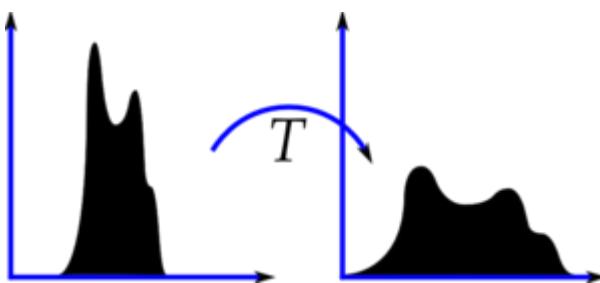
4.10.2-)Histogramlar – 2: Histogram Eşitlemesi Hedef

Bu bölümde,

- Histogram eşitleme kavramlarını öğrenecek ve görüntülerimizin kontrastını iyileştirmek için kullanacağız.

teori

Piksel değerleri yalnızca belirli değer aralıklarıyla sınırlı bir görüntü düşünün. Örneğin, daha parlak görüntü tüm pikselleri yüksek değerlerle sınırlar. Ancak iyi bir görüntü, resmin tüm bölgelerinden piksellere sahip olacaktır. Bu nedenle, bu histogramı her iki uca da uzatmanız gereklidir (aşağıdaki resimde, wikipedia'dan verildiği gibi) ve Histogram Eşitlemesinin yaptığı şey budur (basit kelimelerle). Bu normalde görüntünün kontrastını iyileştirir.



Bununla ilgili daha fazla bilgi için [Histogram Eşitleme'deki](#) wikipedia sayfasını okumanızı tavsiye ederim. Çalışılan örneklerle çok iyi bir açıklaması var, böylece bunu okuduktan sonra neredeyse her şeyi anlayacaksınız. Bunun yerine, burada Numpy uygulamasını göreceğiz. Bundan sonra OpenCV fonksiyonunu göreceğiz.

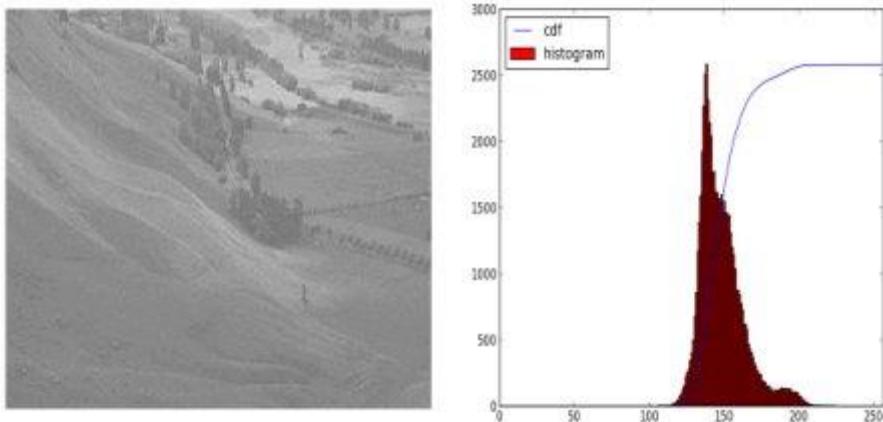
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('wiki.jpg',0)

hist,bins = np.histogram(img.flatten(),256,[0,256])

cdf = hist.cumsum()
cdf_normalized = cdf * hist.max()/ cdf.max()

plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(['cdf','histogram'], loc = 'upper left')
plt.show()
```



Histogramın daha parlak bir bölgede olduğunu görebilirsiniz. Tam spektruma ihtiyacımız var. Bunun için, daha parlak bölgedeki giriş piksellerini tam bölgedeki piksellerin çıkışıyla eşleyen bir dönüştürme fonksiyonuna ihtiyacımız var. Histogram eşitlemesinin yaptığı budur.

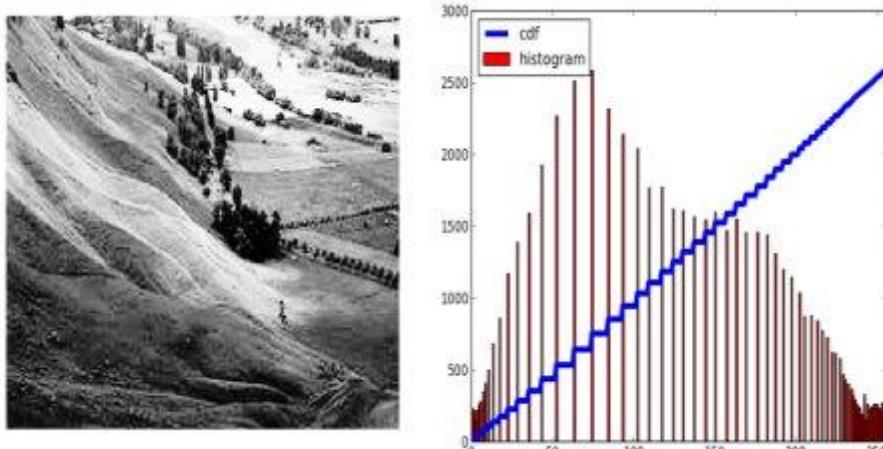
Şimdi minimum histogram değerini buluyoruz (0 hariç) ve histogram eşitleme denklemini wiki sayfasında verildiği gibi uyguluyoruz. Ama burada, Numpy'nin maskeli dizi konsepti dizisini kullandım. Maskelenmiş dizi için, tüm işlemler maskelenmemiş öğeler üzerinde gerçekleştirilir. Maskelenmiş dizilerdeki Numpy belgelerinden daha fazla bilgi edinebilirsiniz.

```
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())
cdf = np.ma.filled(cdf_m,0).astype('uint8')
```

Şimdi, her girdi pikseli değeri için çıkış pikseli değeri hakkında bilgi veren arama tablosuna sahibiz. Yani dönüşümü uyguluyoruz.

```
img2 = cdf[img]
```

Şimdi histogramını ve cdf'sini daha önce olduğu gibi (yapıyoruz) hesaplıyoruz ve sonuç aşağıdaki gibi görünüyor:



Bir başka önemli özellik, görüntü daha koyu bir görüntü olsa bile (kullandığımız daha parlak bir görüntü yerine), eşitlemeden sonra neredeyse aynı görüntüyü elde edeceğimizdir. Sonuç olarak, bu, aynı aydınlatma koşullarına sahip tüm görüntüleri yapmak için bir "referans aracı" olarak kullanılır. Bu birçok durumda yararlıdır. Örneğin, yüz tanımda, yüz verilerini eğitmeden önce, yüzlerin görüntüleri histogram eşitlenir ve hepsi aynı aydınlatma koşullarında olur.

OpenCV'de Histogramların Dengelenmesi

OpenCV'nin bunu yapacak bir işlevi vardır, `cv2.equalizeHist()`. Girdi sadece gri tonlamalı görüntü ve çıktı histogram eşitli görüntündür.

Aşağıda, kullandığımız aynı resim için kullanımını gösteren basit bir kod snippet'i bulunmaktadır:

```
img = cv2.imread('wiki.jpg',0)
equ = cv2.equalizeHist(img)
res = np.hstack((img, equ)) #stacking images side-by-side
cv2.imwrite('res.png',res)
```

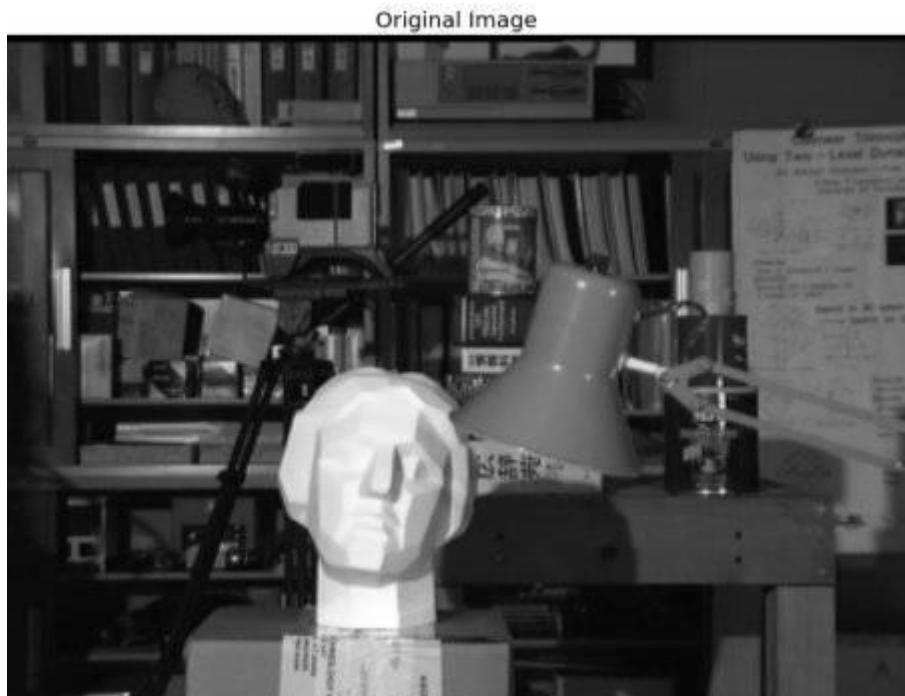


Böylece farklı ışık koşullarında farklı görüntüler çekebilir, dengeleyebilir ve sonuçları kontrol edebilirsiniz.

Görüntünün histogramı belirli bir bölgeyle sınırlı olduğunda histogram eşitlemesi iyidir. Histogramın geniş bir bölgeyi kapsadığı, yani hem parlak hem de karanlık piksellerin bulunduğu büyük yoğunluk varyasyonlarının olduğu yerlerde iyi çalışmaz. Lütfen Ek Kaynaklar'daki SOF bağlantılarını kontrol edin.

CLAHE (Kontrast Sınırlı Uyarlanabilir Histogram Eşitlemesi)

Az önce gördüğümüz ilk histogram eşitleme, görüntünün küresel kontrasını dikkate alır. Çoğu durumda, bu iyi bir fikir değildir. Örneğin, aşağıdaki görüntü bir giriş görüntüsünü ve global histogram eşitlemesinden sonraki sonucunu göstermektedir.



Histogram eşitlemesinden sonra arka plan kontrastının iyileştiği doğrudur. Ancak her iki görüntüde de heykelin yüzünü karşılaşırız. Aşırı parlaklık nedeniyle oradaki bilgilerin çoğunu kaybettik. Bunun nedeni, önceki durumlarda gördüğümüz gibi histogramının belirli bir bölgeyle sınırlı olmamasıdır (Giriş görüntüsünün histogramını çizmeye çalışın, daha fazla sezgi alacaksınız).

Bu sorunu çözmek için **uyarlanabilir histogram eşitleme** kullanılır. Burada, görüntü “fayans” adı verilen küçük bloklara bölünür (OpenCV'de tileSize varsayılan olarak 8x8'dir). Daha sonra bu blokların her biri her zamanki gibi histogram eşitlenir. Yani küçük bir alanda, histogram

küçük bir bölgeyle sınırlı olacaktır (gürültü yoksa). Gürültü varsa, güçlendirilecektir. Bundan kaçınmak için **kontrast sınırlama** uygulanır. Herhangi bir histogram bölmesi belirtilen kontrast sınırının üzerindeyse (varsayılan olarak OpenCV'de 40), bu pikseller kırılır ve histogram eşitlemesi uygulanmadan önce diğer bölmelere eşit olarak dağıtılr. Eşitleme işleminden sonra, karo kenarlarındaki artefaktları gidermek için çift doğrusal interpolasyon uygulanır.

Aşağıdaki kod snippet'i, CLAHE'nin OpenCV'de nasıl uygulanacağını gösterir:

```
import numpy as np
import cv2

img = cv2.imread('tsukuba_1.png',0)

# create a CLAHE object (Arguments are optional).
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)

cv2.imwrite('clahe_2.jpg',cl1)
```

Aşağıdaki sonuca bakın ve yukarıdaki sonuçlarla, özellikle de heykel bölgesiyle karşılaştırın:



4.10.3-)Histogramlar – 3: 2D Histogramlar Hedef

Bu bölümde, 2B histogramları bulmayı ve çizmeyi öğreneceğiz. Gelecek bölümlerde yardımcı olacaktır.

Giriş

İlk makalede, tek boyutlu histogramı hesapladık ve çizdik. Tek boyutlu olarak adlandırılıyor çünkü dikkate aldığımız tek bir özelliği alıyoruz, yani pikselin gri tonlama yoğunluğu değeri. Ancak iki boyutlu histogramlarda iki özelliği dikkate alırsınız. Normalde, iki özelliğin her pikselin Ton ve Doygunluk değerleri olduğu renk histogramlarını bulmak için kullanılır.

[Resmi örneklerde](#) renk histogramlarını bulmak için zaten bir [python örneği var](#). Böyle bir renk histogramının nasıl oluşturulacağını anlamaya çalışacağınız ve Histogram Geri Projeksiyon gibi diğer konuları anlamada yararlı olacaktır.

OpenCV'de 2D Histogram

Oldukça basittir ve aynı işlev kullanılarak hesaplanır, `cv2.calcHist()`. Renk histogramları için görüntüyü BGR'den HSV'ye dönüştürmemiz gereklidir. (1D histogram için BGR'den Gri Tonlamaya dönüştürüldüğümüzü unutmayın). 2D histogramlar için parametreleri aşağıdaki gibi değiştirecektir:

- **Kanallar = [0,1]** çünkü hem H hem de S düzlemini işlememiz gerekiyor.
- **kutular = [180,256]** H düzlemi için 180 ve S düzlemi için 256.
- **range = [0,180,0,256]** Ton değeri 0 ile 180 arasında ve Doygunluk 0 ile 256 arasındadır.

Şimdi aşağıdaki kodu kontrol edin:

```
import cv2
import numpy as np

img = cv2.imread('home.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hist = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])
```

Bu kadar.

Numpy'de 2D Histogram

Numpy ayrıca bunun için belirli bir işlev sağlar: `np.histogram2d()`. (1D histogram için `np.histogram()` kullandığımızı **unutmayın**).

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

img = cv2.imread('home.jpg')
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

hist, xbins, ybins =
np.histogram2d(h.ravel(), s.ravel(), [180, 256], [[0, 180], [0, 256]])
```

İlk argüman H düzlemini, ikincisi S düzlemini, üçüncüsü her biri için kutu sayısı ve dördüncüsü aralıklarıdır.

Şimdi bu renk histogramının nasıl çizileceğini kontrol edebiliriz.

2D Histogramları Çizme

Yöntem – 1: cv2.imshow () kullanımı

Elde ettiğimiz sonuç, 180x256 boyutunda iki boyutlu bir dizidir. Böylece normalde yaptığımız gibi cv2.imshow () fonksiyonunu kullanarak gösterebiliriz. Gri tonlamalı bir görüntü olacak ve farklı renklerin Ton değerlerini bilmiyorsanız, hangi renklerin orada olduğu hakkında pek bir fikir vermeyecektir.

Yöntem – 2: Matplotlib Kullanımı

2D histogramı farklı renk haritalarıyla çizmek için `matplotlib.pyplot.imshow()` işlevini kullanabiliriz. Bu bize farklı piksel yoğunluğu hakkında daha iyi bir fikir verir. Ancak bu, farklı renklerin Hue değerlerini bilmediğiniz sürece, ilk bakışta hangi rengin olduğu hakkında bize fikir vermez. Yine de bu yöntemi tercih ediyorum. Basit ve daha iyi.

Not Bu işlevi kullanırken, daha iyi sonuçlar için interpolasyon bayrağının en yakın olması gerektiğini unutmayın .

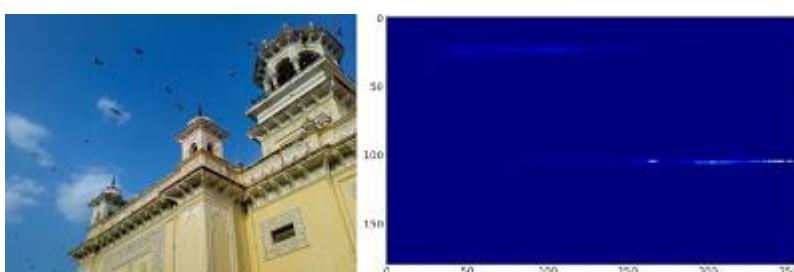
Kodu düşünün:

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

img = cv2.imread('home.jpg')
HSV = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
hist = cv2.calcHist([HSV], [0, 1], None, [180, 256], [0, 180, 0, 256] )

plt.imshow(hist, interpolation = 'nearest')
plt.show()
```

Aşağıda giriş görüntüsü ve renk histogramı grafiği verilmiştir. X ekseni S değerlerini ve Y ekseni Tonu gösterir.



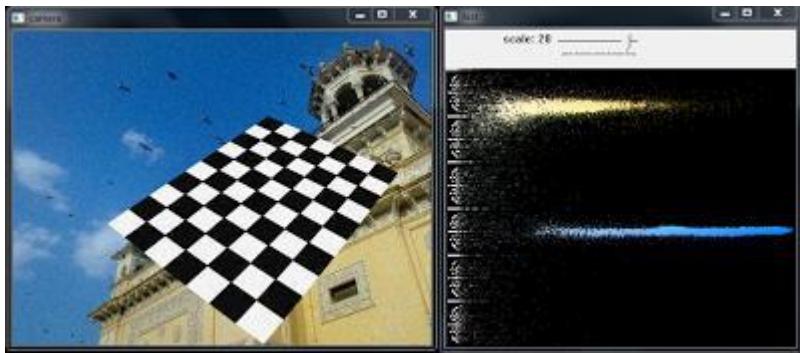
Histogramda, $H = 100$ ve $S = 200$ yakınında bazı yüksek değerler görebilirsiniz. Gökyüzünün mavisine karşılık gelir. Benzer şekilde $H = 25$ ve $S = 100$ yakınında başka bir tepe görülebilir. Sarayın sarısına karşılık gelir. GIMP gibi herhangi bir görüntü düzenleme aracıyla doğrulayabilirsiniz.

Yöntem 3: OpenCV örnek stili !!

[OpenCV-Python2 örneklerinde renk histogramı için bir örnek kod](#) vardır. Kodu çalıştırırsanız, histogramın ilgili rengi de gösterdiğini görebilirsiniz. Ya da sadece renk kodlu bir histogram çıkarır. Sonuç çok iyi (her ne kadar ekstra satır eklemeniz gerekiyorsa da).

Bu kodda, yazar HSV'de bir renk haritası oluşturdu. Daha sonra BGR'ye dönüştürdü. Ortaya çıkan histogram görüntüsü bu renk haritası ile çarpılır. Ayrıca, küçük izole pikselleri kaldırmak için bazı önişleme adımlarını kullanır ve bu da iyi bir histogramla sonuçlanır.

Ben kodu çalıştmak, analiz etmek ve kendi kesmek etrafında etrafında okuyuculara bırakın. Aşağıda, yukarıdaki ile aynı görüntü için bu kodun çıktısı verilmiştir:



Histogramda hangi renklerin mevcut olduğunu, mavi var, sarı var ve satranç tahtası nedeniyle biraz beyaz olduğunu açıkça görebilirsiniz. Güzel !!!

4.10.4-)Histogram – 4: Histogram Geri Projeksiyonu Hedef

Bu bölümde histogram geri projeksiyonu hakkında bilgi edineceğiz.

teori

Michael J. Swain, Dana H. Ballard tarafından renk histogramları ile İndeksleme makalesinde önerilmiştir .

Aslında basit kelimelerle nedir? Görüntü segmentasyonu veya bir görüntüde ilgilenilen nesneleri bulmak için kullanılır. Basit bir ifadeyle, giriş görüntümüzle aynı boyutta (ancak tek kanallı) bir görüntü oluşturur; burada her piksel, nesnenin ait olduğu pikselin olasılığına karşılık gelir. Daha basit dünyalarda, çıktı görüntüsünde ilgi alanımız kalan parçaya kıyasla daha beyaz olacaktır. Bu sezgisel bir açıklama. (Daha basit yapamam). Histogram Backprojection, camshift algoritması vb. ile kullanılır.

Bunu nasıl yaparız ? İlgilendiğimiz nesneyi içeren bir imgenin histogramını yaratıyoruz (bizim durumumuzda zemin, oyuncudan ayrılma ve diğer şeyler). Nesne, daha iyi sonuçlar için görüntüyü mümkün olduğunda doldurmalıdır. Gri tonlamalı histogram yerine bir renk histogramı tercih edilir, çünkü nesnenin rengi, nesneyi gri tonlama yoğunluğundan daha iyi tanımlamanın bir yoludur. Daha sonra bu histogramı nesneyi bulmamız gereken test görüntümüze “geri yansıtıyoruz”, yani başka bir deyişle, yere ait her pikselin olasılığını hesaplıyor ve gösteriyoruz. Uygun eşikleme sonucu ortaya çıkan çıktı bize sadece zemin verir.

Numpy'de Algoritma

1. Öncelikle hem bulmamız gereken nesnenin ('M' olsun) renk histogramını ve arayacağımız görüntünün ('I' olsun) renk histogramını hesaplamamız gereklidir.

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

#roi is the object or region of object we need to find
roi = cv2.imread('rose_red.png')
hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

#target is the image we search in
target = cv2.imread('rose.png')
hsvt = cv2.cvtColor(target, cv2.COLOR_BGR2HSV)

# Find the histograms using calcHist. Can be done with np.histogram2d also
M = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )
I = cv2.calcHist([hsvt], [0, 1], None, [180, 256], [0, 180, 0, 256] )
```

2. Oranı bulun $R = \frac{M}{I}$. Ardından, R'yi geri projeyeleyin, yani R'yi palet olarak kullanın ve her piksel ile hedef olma olasılığı olarak yeni bir görüntü oluşturun. yani $B(x, y) = R[h(x, y), s(x, y)]$ burada h renk tonu ve s, pikselin (x, y) doygunluğudur. Bundan sonra koşulu uygulayın $B(x, y) = \min[B(x, y), 1]$.

```
h,s,v = cv2.split(hsv)
B = R[h.ravel(),s.ravel()]
B = np.minimum(B,1)
B = B.reshape(hsvt.shape[:2])
```

3. Şimdi dairesel bir diskle, $B = D * B_D$ 'nin disk çekirdeği olduğu bir kıvrım uygulayın .

```
disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
cv2.filter2D(B,-1,disc,B)
B = np.uint8(B)
cv2.normalize(B,B,0,255,cv2.NORM_MINMAX)
```

4. Şimdi maksimum yoğunluğun yeri bize nesnenin yerini verir. Resimde bir bölge bekliyoruz, uygun bir değer için eşik olması hoş bir sonuç verir.

```
ret,thresh = cv2.threshold(B,50,255,0)
```

Bu kadar !!

OpenCV'de geri projeksiyon

OpenCV dahili bir **cv2.calcBackProject** () işlevi sağlar . Parametreleri **cv2.calcHist** () işleviyle neredeyse aynıdır . Parametresinden biri, nesnenin histogramı olan histogramdır ve onu bulmamız gereklidir. Ayrıca, nesne projektörü, geri proje işlevine geçmeden önce normalleştirilmelidir. Olasılık görüntüsünü döndürür. Sonra görüntüyü bir disk çekirdeği ile birleştirir ve eşik uygularız. Kodum ve çıktıım aşağıdadır:

```
import cv2
import numpy as np

roi = cv2.imread('rose_red.png')
hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

target = cv2.imread('rose.png')
hsvt = cv2.cvtColor(target, cv2.COLOR_BGR2HSV)

# calculating object histogram
roihist = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )

# normalize histogram and apply backprojection
cv2.normalize(roihist,roihist,0,255,cv2.NORM_MINMAX)
dst = cv2.calcBackProject([hsvt],[0,1],roihist,[0,180,0,256],1)

# Now convolute with circular disc
disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
cv2.filter2D(dst,-1,disc,dst)

# threshold and binary AND
ret,thresh = cv2.threshold(dst,50,255,0)
thresh = cv2.merge((thresh,thresh,thresh))
res = cv2.bitwise_and(target,thresh)

res = np.vstack((target,thresh,res))
cv2.imwrite('res.jpg',res)
```

Aşağıda üzerinde çalıştığım bir örnek var. Mavi dikdörtgenin içindeki bölgeyi örnek nesne olarak kullandım ve tüm alanı çıkarmak istedim.



4.11-)Fourier dönüşümü Hedef

Bu bölümde öğreneceğiz

- OpenCV kullanarak görüntülerin Fourier Dönüşümü'nü bulmak için
- Numpy'de bulunan FFT işlevlerini kullanmak için
- Fourier Dönüşümünün bazı uygulamaları
- Aşağıdaki işlevleri göreceğiz: `cv2.dft()`, `cv2.idft()` vb.

teori

Fourier Dönüşümü, çeşitli filtrelerin frekans karakteristiklerini analiz etmek için kullanılır. Görüntüler için, frekans alanını bulmak için **2D Ayrik Fourier Dönüşümü (DFT)** kullanılır. DFT'nin hesaplanması için **Fast Fourier Dönüşümü (FFT)** adı verilen hızlı bir algoritma kullanılır. Bunlarla ilgili ayrıntılar herhangi bir görüntü işleme veya sinyal işleme ders kitabında bulunabilir. Lütfen [Ek Kaynaklar](#) bölümüne bakın.

Sinüzoidal bir sinyal için, sinyalin frekansı $x(t) = A \sin(2\pi ft)$ diyebiliriz f ve frekans alanı alınırsa, bir artış görebiliriz f . Sinyal ayrik bir sinyal oluşturmak için örneklenirse, aynı frekans alanını elde ederiz, ancak aralıkta $[-\pi, \pi]$ veya $[0, 2\pi]$ (veya $[0, N]$ noktası DFT için) periyodiktir. Bir görüntüyü iki yönde örneklenmiş bir sinyal olarak düşünübilirsiniz. Böylece hem X hem de Y yönlerinde fourier dönüşümü almak görüntünün frekans gösterimini sağlar.

Daha sezgisel olarak, sinüzoidal sinyal için, genlik kısa sürede çok hızlı değişirse, bunun yüksek frekanslı bir sinyal olduğunu söyleyebilirsiniz. Yavaşça değişiyorsa, düşük frekanslı bir sinyaldir. Aynı fikri görüntülere de genişletebilirsiniz. Görüntülerde genlik nerede büyük ölçüde değişir? Kenar noktalarında veya gürültülerde. Yani diyebiliriz ki, kenarlar ve sesler görüntüdeki yüksek frekans içerikleridir. Genlikte çok fazla değişiklik yoksa, düşük frekanslı bir bileşendir. (Frekans dönüşümünü örneklerle sezgisel olarak açıklayan [Ek Kaynaklara](#) bazı bağlantılar eklenmiştir).

Şimdi Fourier Dönüşümü'nü nasıl bulacağımızı göreceğiz.

Numier'de Fourier Dönüşümü

İlk önce Numpy kullanarak Fourier Dönüşümü'nü nasıl bulacağımızı göreceğiz. Numpy'nin bunu yapmak için bir FFT paketi var. `np.fft.fft2()` bize karmaşık bir dizi olacak frekans dönüşümünü sağlar. İlk argümanı gri tonlamalı girdi görüntüsüdür. Çıktı dizisinin boyutuna karar veren ikinci argüman ise sağlıdır. Giriş görüntüsünün boyutundan büyükse, giriş görüntüsü FFT'nin hesaplanmasından önce sıfırlarla doldurulur. Giriş görüntüsünden küçükse, giriş görüntüsü kırılır. Herhangi bir argüman iletilmezse, çıkış dizisi boyutu girişle aynı olur.

Şimdi sonucu aldıktan sonra, sıfır frekans bileşeni (DC bileşeni) sol üst köşede olacaktır. Merkeze getirmek istiyorsanız, sonucu $\frac{N}{2}$ her iki yönde de kaydırmanız

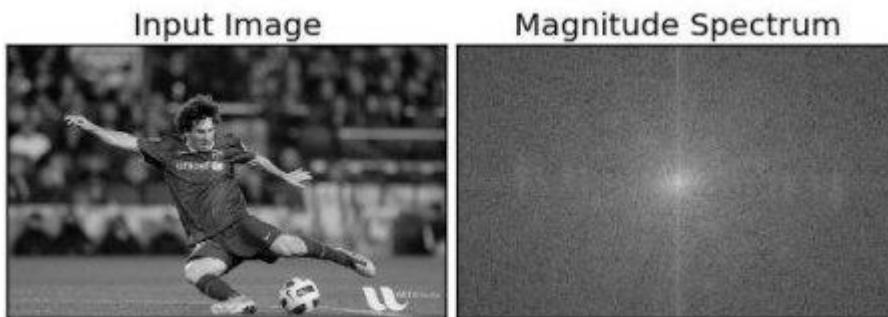
gerekir . Bu, **np.fft.fftshift** () işlevi tarafından yapılır . (Analiz etmek daha kolaydır). Frekans dönüşümünü bulduğunuzda, büyülü spektrumunu bulabilirsiniz.

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

img = cv2.imread('messi5.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitude_spectrum = 20*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

Sonuç aşağıdaki gibi görünür:



Merkezde, düşük frekanslı içeriğin daha fazla olduğunu gösteren daha beyaz bir bölge görebilirsiniz.

Böylece frekans dönüşümünü bulduk. Yüksek frekanslı filtreleme ve görüntüyü yeniden yapılandırma gibi ters etki alanında bazı işlemleri yapabilirsiniz. Bunun için düşük frekansları 60x60 boyutunda dikdörtgen bir pencereyle maskeleyerek kaldırabilirsiniz. Sonra **np.fft.ifftshift** () kullanarak ters kaydırma uygulayın, böylece DC bileşeni tekrar sol üst köşeye gelir. Sonra **np.ifft2** () işlevini kullanarak ters FFT'yi bulun . Sonuç yine karmaşık bir sayı olacaktır. Mutlak değerini alabilirsiniz.

```
rows, cols = img.shape
crow,ccol = rows/2 , cols/2
fshift[crow-30:crow+30, ccol-30:ccol+30] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)

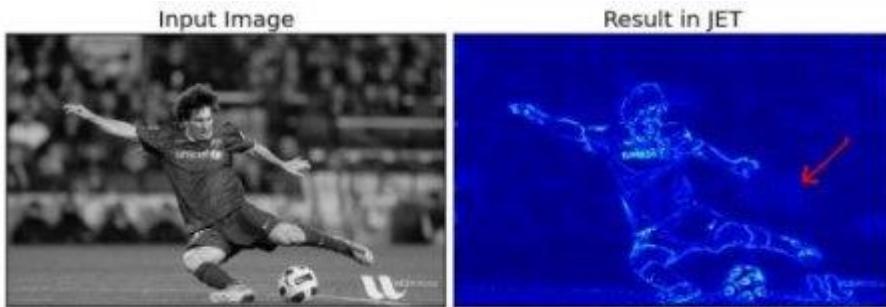
plt.subplot(131),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(img_back, cmap = 'gray')
plt.title('Image after HPF'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img_back)
```

```

plt.title('Result in JET'), plt.xticks([]), plt.yticks([])
plt.show()

```

Sonuç aşağıdaki gibi görünür:



Sonuç, Yüksek Geçişli Filtrelemenin bir kenar algılama işlemi olduğunu gösterir. Görüntü Degradeleri bölümünde gördüğümüz budur. Bu aynı zamanda görüntü verilerinin çoğunun spektrumun Düşük frekans bölgesinde bulunduğunu gösterir. Her neyse, Numpy'de DFT, IDFT vb. Şimdi OpenCV'de nasıl yapılacağını görelim.

Sonucu, özellikle de JET rengindeki son görüntüyü yakından izlerseniz, bazı eserler görebilirsiniz (Bir örnek kırmızı okla işaretledim). Orada bazı dalgalanma benzeri yapılar gösterir ve buna **zil efektleri** denir. Maskeleme için kullandığımız dikdörtgen pencereden kaynaklanır. Bu maske, bu soruna neden olan samimi şekilde dönüştürülür. Dolayısıyla dikdörtgen pencereler filtreleme için kullanılmaz. Daha iyi bir seçenek Gaussian Windows.

OpenCV'de Fourier Dönüşümü

OpenCV bunun için **cv2.dft()** ve **cv2.idft()** işlevlerini sağlar. Önceki ile aynı sonucu döndürür, ancak iki kanalla döner. Birinci kanal sonucun gerçek kısmına, ikinci kanal ise sonucun hayali kısmına sahip olacaktır. Giriş görüntüsü önce np.float32 biçimine dönüştürülmelidir. Bunu nasıl yapacağımızı göreceğiz.

```

import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg',0)

dft = cv2.dft(np.float32(img),flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

magnitude_spectrum =
20*np.log(cv2.magnitude(dft_shift[:, :, 0],dft_shift[:, :, 1]))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])

```

```
plt.show()
```

Not Tek bir çekimde hem büyüklüğü hem de fazı döndüren `cv2.cartToPolar()` yöntemini de kullanabilirsiniz

Şimdi, ters DFT yapmak zorundayız. Önceki oturumda bir HPF oluşturduk, bu sefer görüntüdeki yüksek frekanslı içeriklerin nasıl kaldırılacağını göreceğiz, yani görüntüye LPF uyguluyoruz. Aslında görüntüyü bulanıklaştırır. Bunun için önce düşük frekanslarda yüksek değerli (1) bir maske oluştururuz, yani LF içeriğini ve HF bölgesinde 0 geçiririz.

```
rows, cols = img.shape
crow,ccol = rows/2 , cols/2

# create a mask first, center square is 1, remaining all zeros
mask = np.zeros((rows,cols,2),np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1

# apply mask and inverse DFT
fshift = dft_shift*mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv2.idft(f_ishift)
img_back = cv2.magnitude(img_back[:, :, 0],img_back[:, :, 1])

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img_back, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

Sonuca bakın:



Not Her zaman olduğu gibi, `cv2.dft()` ve `cv2.idft()` OpenCV işlevleri Numpy meslektaşlarından daha hızlıdır. Ancak Numpy işlevleri daha kullanıcı dostudur. Performans sorunları hakkında daha fazla bilgi için aşağıdaki bölüme bakın.

DFT Performans Optimizasyonu

DFT hesaplamasının performansı bazı dizi boyutları için daha iyidir. Dizi boyutu ikinin gücü olduğunda en hızlıdır. Boyutu 2, 3 ve 5'in bir ürünü olan diziler de oldukça verimli bir şekilde işlenir. Bu nedenle, kodunuzun performansı konusunda endişeleriniz varsa,

DFT'yi bulmadan önce dizinin boyutunu herhangi bir en uygun boyuta (sıfırlama yaparak) değiştirebilirsiniz. OpenCV için sıfırları manuel olarak doldurmanız gereklidir. Ancak Numpy için, yeni FFT hesaplaması boyutunu belirlersiniz ve sizin için otomatik olarak sıfırlar doldurur.

Peki bu optimum boyutu nasıl buluruz? OpenCV bunun için bir `cv2.getOptimalDFTSize()` işlevi sağlar. Hem için geçerlidir `cv2.dft()` ve `np.fft.fft2()`. IPython sihirli komutu `%timeit` kullanarak performanslarını kontrol edelim.

```
In [16]: img = cv2.imread('messi5.jpg',0)
In [17]: rows,cols = img.shape
In [18]: print rows,cols
342 548

In [19]: nrows = cv2.getOptimalDFTSize(rows)
In [20]: ncols = cv2.getOptimalDFTSize(cols)
In [21]: print nrows, ncols
360 576
```

Bkz. (342,548) boyutu (360, 576) olarak değiştirilmiştir. Şimdi sıfırlarla dolduralım (OpenCV için) ve DFT hesaplama performanslarını bulalım. Yeni bir büyük sıfır dizisi oluşturarak ve bu diziye veri kopyalayabilir ya da `cv2.copyMakeBorder()` kullanabilirsiniz.

```
nimg = np.zeros((nrows,ncols))
nimg[:rows,:cols] = img
```

VEYA:

```
right = ncols - cols
bottom = nrows - rows
bordertype = cv2.BORDER_CONSTANT #just to avoid line breakup in PDF file
nimg = cv2.copyMakeBorder(img,0,bottom,0,right,bordertype, value = 0)
```

Şimdi Numpy fonksiyonunun DFT performans karşılaştırmasını hesaplıyoruz:

```
In [22]: %timeit fft1 = np.fft.fft2(img)
10 loops, best of 3: 40.9 ms per loop
In [23]: %timeit fft2 = np.fft.fft2(img,[nrows,ncols])
100 loops, best of 3: 10.4 ms per loop
```

4x hızlanma gösterir. Şimdi ayınısını OpenCV fonksiyonları ile deneyeceğiz.

```
In [24]: %timeit dft1= cv2.dft(np.float32(img),flags=cv2.DFT_COMPLEX_OUTPUT)
100 loops, best of 3: 13.5 ms per loop
In [27]: %timeit dft2= cv2.dft(np.float32(nimg),flags=cv2.DFT_COMPLEX_OUTPUT)
100 loops, best of 3: 3.11 ms per loop
```

Ayrıca 4x hızlanma gösterir. OpenCV işlevlerinin Numpy işlevlerinden yaklaşık 3 kat daha hızlı olduğunu da görebilirsiniz. Bu ters FFT için de test edilebilir ve bu sizin için bir egzersiz olarak bırakılır.

Laplacian Neden Yüksek Geçişli Filtredir?

Bir forumda benzer bir soru soruldu. Soru şu: Laplacian neden yüksek geçirgen bir filtre? Sobel neden bir HPF? Ve ona verilen ilk cevap Fourier Dönüşümü açısından oldu. Biraz daha yüksek FFT boyutu için Laplacian'ın fourier dönüşümünü almanız yeterlidir. Analiz et:

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

# simple averaging filter without scaling parameter
mean_filter = np.ones((3,3))

# creating a guassian filter
x = cv2.getGaussianKernel(5,10)
gaussian = x*x.T

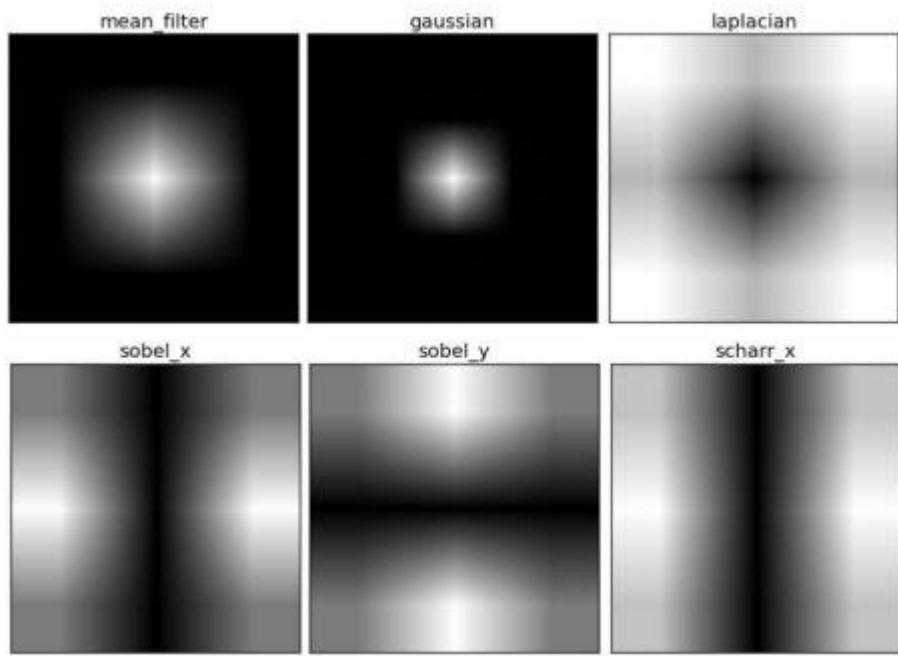
# different edge detecting filters
# scharr in x-direction
scharr = np.array([[-3, 0, 3],
                  [-10,0,10],
                  [-3, 0, 3]])
# sobel in x direction
sobel_x= np.array([[-1, 0, 1],
                  [-2, 0, 2],
                  [-1, 0, 1]])
# sobel in y direction
sobel_y= np.array([[-1,-2,-1],
                  [0, 0, 0],
                  [1, 2, 1]])
# Laplacian
laplacian=np.array([[0, 1, 0],
                    [1,-4, 1],
                    [0, 1, 0]])

filters = [mean_filter, gaussian, laplacian, sobel_x, sobel_y, scharr]
filter_name = ['mean_filter', 'gaussian','laplacian', 'sobel_x', \
               'sobel_y', 'scharr_x']
fft_filters = [np.fft.fft2(x) for x in filters]
fft_shift = [np.fft.fftshift(y) for y in fft_filters]
mag_spectrum = [np.log(np.abs(z)+1) for z in fft_shift]

for i in xrange(6):
    plt.subplot(2,3,i+1),plt.imshow(mag_spectrum[i],cmap = 'gray')
    plt.title(filter_name[i]), plt.xticks([]), plt.yticks([])

plt.show()
```

Sonuca bakın:



Görüntüden, her bir çekirdeğin hangi frekans bölgesini engellediğini ve hangi bölgeden geçtiğini görebilirsiniz. Bu bilgilerden, her bir çekirdeğin neden HPF veya LPF olduğunu söyleyebiliriz

4.12-)Şablon eşleme Hedefler

Bu bölümde öğreneceksiniz

- Şablon Eşleme'yi kullanarak görüntüdeki nesneleri bulmak için
- Şu işlevleri göreceksiniz: `cv2.matchTemplate()` , `cv2.minMaxLoc()`

teori

Şablon Eşleme, daha büyük bir görüntüde şablon görüntüsünün yerini arama ve bulma yöntemidir. OpenCV bu amaçla `cv2.matchTemplate()` işleviyle birlikte gelir. Şablon görüntüsünü giriş görüntüsünün üzerine kaydırır (2D evrişimde olduğu gibi) ve şablon görüntüsünün altındaki şablon ve yamayı karşılaştırır. OpenCV'de çeşitli karşılaştırma yöntemleri uygulanmaktadır. (Daha fazla ayrıntı için dokümanları kontrol edebilirsiniz). Her pikselin o pikselin mahallesinin şablonla ne kadar eşleştiğini ifade ettiği gri tonlamalı bir görüntü döndürür.

Giriş resim boyutunun ise (GxY) ve şablon resim boyutu olan (wxh) , çıkış görüntü arasında bir büyülükle sahip olur ($1 + , 'H-h + 1 W w-$) . Sonucu aldıktan sonra , maksimum / minimum değerin nerede olduğunu bulmak için `cv2.minMaxLoc()` işlevini kullanabilirsiniz. Dikdörtgenin sol üst köşesi olarak alın ve dikdörtgenin genişliği ve yüksekliği olarak (w, h) alın . Bu dikdörtgen sizin şablon bölgeinizdir.

Not Karşılaştırma yöntemi olarak `cv2.TM_SQDIFF` kullanıyorsanız , minimum değer en iyi eşleşmeyi sağlar.

OpenCV'de Şablon Eşleme

Burada, bir örnek olarak, fotoğrafında Messi'nin yüzünü arayacağız. Bu yüzden aşağıdaki gibi bir şablon oluşturdum:



Sonuçlarının nasıl göründüğünü görebilmemiz için tüm karşılaştırma yöntemlerini deneyeceğiz:

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

img = cv2.imread('messi5.jpg',0)
img2 = img.copy()
template = cv2.imread('template.jpg',0)
w, h = template.shape[::-1]

# All the 6 methods for comparison in a list
methods = ['cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED', 'cv2.TM_CCORR',
            'cv2.TM_CCORR_NORMED', 'cv2.TM_SQDIFF', 'cv2.TM_SQDIFF_NORMED']

for meth in methods:
    img = img2.copy()
    method = eval(meth)

    # Apply template Matching
    res = cv2.matchTemplate(img,template,method)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

    # If the method is TM_SQDIFF or TM_SQDIFF_NORMED, take minimum
    if method in [cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED]:
        top_left = min_loc
    else:
        top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)

    cv2.rectangle(img,top_left, bottom_right, 255, 2)

    plt.subplot(121),plt.imshow(res,cmap = 'gray')
    plt.title('Matching Result'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(img,cmap = 'gray')
    plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
    plt.suptitle(meth)

plt.show()
```

Aşağıdaki sonuçlara bakın:

- cv2.TM_CCOEFF

Matching Result



Detected Point



- cv2.TM_CCOEFF_NORMED
- Matching Result



Detected Point



- cv2.TM_CCORR
- Matching Result



Detected Point



- cv2.TM_CCORR_NORMED
- Matching Result



Detected Point



- cv2.TM_SQDIFF



- cv2.TM_SQDIFF_NORMED



Cv2.TM_CCORR kullanan sonucun beklediğimiz gibi iyi olmadığını görebilirsiniz .

Birden Çok Nesneyle Eşleşen Şablon

Bir önceki bölümde, görüntüde sadece bir kez meydana gelen Messi'nin yüzüne ait resmi araştırdık. Birden fazla **yinelemeye** sahip bir nesne aradığınızı varsayıyalım, **cv2.minMaxLoc()** size tüm konumları vermeyecektir. Bu durumda eşiklemeyi kullanacağız. Bu örnekte, ünlü **Mario** oyunu ekran görüntüsünü kullanacağız ve içindeki paraları bulacağız.

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

img_rgb = cv2.imread('mario.png')
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
template = cv2.imread('mario_coin.png',0)
w, h = template.shape[::-1]

res = cv2.matchTemplate(img_gray,template,cv2.TM_CCOEFF_NORMED)
threshold = 0.8
loc = np.where( res >= threshold)
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)

cv2.imwrite('res.png',img_rgb)
```

Sonuç:



4.13 hough line dönüşüm koymadım :D

4.14 Hough Circle Dönüşümü

Hedef

Bu bölümde,

- Bir görüntüdeki çevreleri bulmak için Hough Transform'u kullanmayı öğreneceğiz.
- Bu işlevleri göreceğiz: `cv2.HoughCircles()`

teori

Bir daire, matematiksel olarak temsil edilir $(x - x_{center})^2 + (y - y_{center})^2 = r^2$ burada (x_{center}, y_{center}) dairenin merkezi ve r dairenin yarıçapıdır. Denklemden, 3 parametremiz olduğunu görebiliriz, bu yüzden hough dönüşümü için oldukça etkisiz olacak bir 3D akümülatöre ihtiyacımız var. Bu nedenle OpenCV, kenarların gradyan bilgilerini kullanan daha sert bir yöntem olan **Hough Gradient Method**'u kullanır .

Burada kullandığımız işlev `cv2.HoughCircles()` işlevidir. Belgede iyi açıklanmış birçok argüman var. Bu yüzden doğrudan koda gidiyoruz.

```
import cv2
import numpy as np

img = cv2.imread('opencv_logo.png',0)
img = cv2.medianBlur(img,5)
cimg = cv2.cvtColor(img,cv2.COLOR_GRAY2BGR)

circles = cv2.HoughCircles(img,cv2.HOUGH_GRADIENT,1,20,
                           param1=50,param2=30,minRadius=0,maxRadius=0)

circles = np.uint16(np.around(circles))
for i in circles[0,:]:
    # draw the outer circle
    cv2.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
    # draw the center of the circle
```

```
cv2.circle(cimg,(i[0],i[1]),2,(0,0,255),3)  
cv2.imshow('detected circles',cimg)  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

Sonuç aşağıda gösterilmiştir:



4.15-)Havza Algoritması ile Görüntü Bölütleme Hedef

Bu bölümde,

- Havza algoritmasını kullanarak işaretçi tabanlı görüntü segmentasyonu kullanmayı öğreneceğiz
- Göreceğiz: `cv2.watershed()`

teori

Herhangi bir gri tonlamalı görüntü, yüksek yoğunluğun tepe ve tepeleri, düşük yoğunluklu vadileri ifade ettiği bir topografik yüzey olarak görülebilir. Her izole vadiyi (yerel minima) farklı renkli suyla (etiketler) doldurmaya başlarsınız. Su yükseldikçe, yakınlardaki zirvelere (gradyanlara) bağlı olarak, farklı vadilerden gelen su, belli ki farklı renklerle birleşmeye başlayacaktır. Bundan kaçınmak için, suyun birleştiği yerlerde bariyerler inşa edersiniz. Tüm pikler su altında olana kadar su doldurma ve bariyer oluşturma çalışmalarına devam ediyorsunuz. Ardından oluşturduğunuz engeller size segmentasyon sonucunu verir. Bu havzanın arkasındaki "felsefe" dir. Bazı animasyonların yardımıyla anlamak için [havza üzerindeki CMM web sayfasını](#) ziyaret edebilirsiniz .

Ancak bu yaklaşım, parazit veya görüntüdeki diğer düzensizlikler nedeniyle size aşırı bir sonuç verir. Bu nedenle OpenCV, hangi noktalarının birleştirileceğini ve hangilerinin birleştirilmeyeceğini belirlediğiniz işaretçi tabanlı bir havza algoritması uyguladı. Etkileşimli bir görüntü segmentasyonu. Yaptığımız, bildiğimiz nesnemiz için farklı etiketler vermektedir. Ön plan veya nesne olduğumuzdan emin olduğumuz bölgeyi bir renkle (veya yoğunlukla) etiketleyin, arka plan veya nesne olmadığından emin olduğumuz bölgeyi başka bir renkle etiketleyin ve son olarak hiçbir şeyden emin olmadığımız bölgeyi, 0 ile etiketleyin. Bu bizim markamız. Ardından havza algoritmasını

uygulayın. Daha sonra işaretleyicimiz verdığımız etiketlerle güncellenecek ve nesnelerin sınırları -1 olacak.

kod

Aşağıda, karşılıklı dokunan nesneleri segmentlere ayırmak için Mesafe Dönüşümü ile birlikte havza nasıl kullanılacağına dair bir örnek göreceğiz.

Aşağıdaki paralar görüntüsünü düşünün, paralar birbirine dokunuyor. Eşleştirseniz bile, birbirine degecektir.

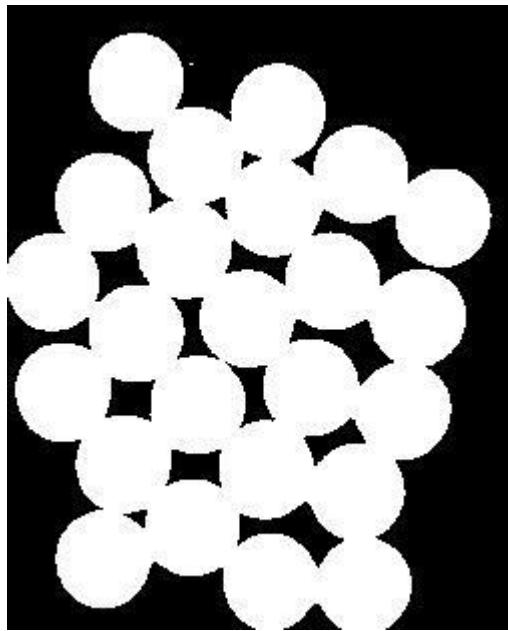


Madeni paraların yaklaşık bir tahminini bulmakla başlıyoruz. Bunun için Otsu'nun ikilemini kullanabiliriz.

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

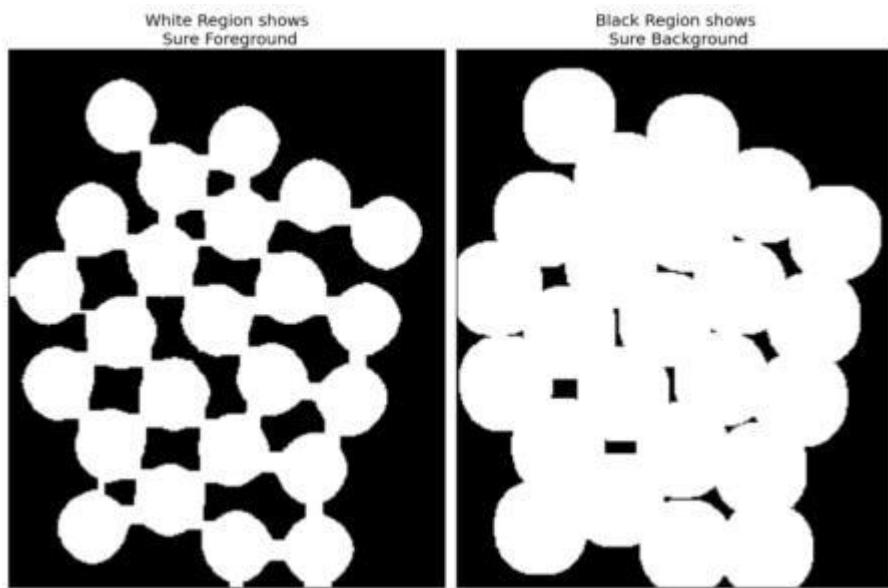
img = cv2.imread('coins.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

Sonuç:



Şimdi görüntüdeki küçük beyaz sesleri çıkarmamız gerekiyor. Bunun için morfolojik açıklığı kullanabiliriz. Nesnedeki küçük delikleri kaldırmak için morfolojik kapamayı kullanabiliriz. Artık, nesnelerin merkezine yakın olan bölgenin ön plan olduğundan ve nesneden çok uzakta olan bölgenin arka plan olduğundan emin olduk. Emin olmadığımız tek bölge madeni paraların sınır bölgesidir.

Bu yüzden para olduklarından emin olduğumuz alanı çıkarmamız gerekiyor. Erozyon sınır piksellerini kaldırır. Yani kalan ne olursa olsun, bozuk para olduğundan emin olabiliriz. Nesneler birbirine değmezse bu işe yarar. Ancak birbirlerine dokundukları için, iyi bir seçenek de mesafe dönüşümünü bulmak ve uygun bir eşik uygulamak olacaktır. Daha sonra para olmadılarından emin olduğumuz alanı bulmamız gerekiyor. Bunun için sonucu genişletiyoruz. Genişletme, nesne sınırını arka plana artırır. Bu şekilde, sonuçta arka plandaki herhangi bir bölgenin gerçekten bir arka plan olduğundan emin olabiliriz, çünkü sınır bölgesi kaldırılır. Aşağıdaki resme bakın.



Geri kalan bölgeler, madeni para veya arka plan olsun, hiçbir fikrimiz olmayan bölgelerdir. Havza algoritması bunu bulmalıdır. Bu alanlar normalde ön plan ve arka planın birleştiği (veya iki farklı madeni para bile birleşir) sikkelerin sınırları etrafındadır. Biz buna sınır diyoruz. Sure_fg alanının sure_bg alanından çıkarılmasıyla elde edilebilir.

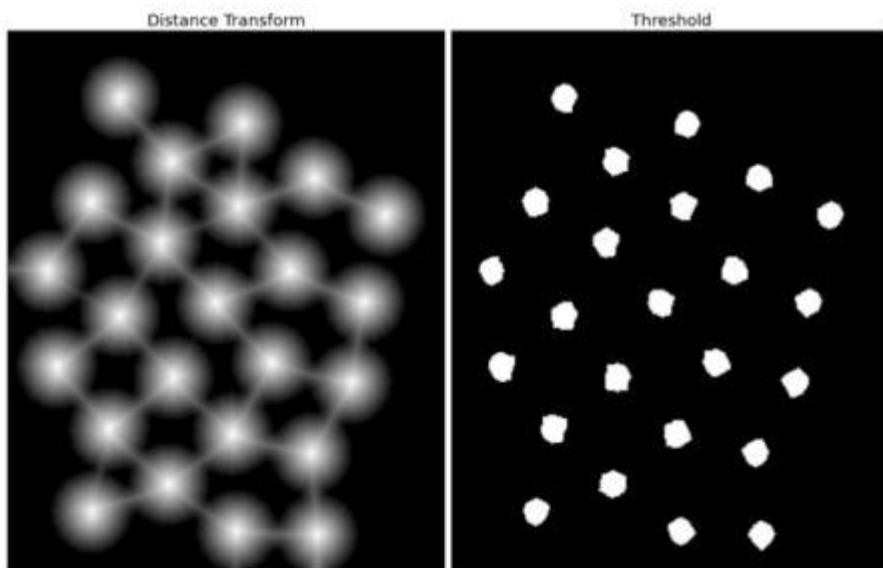
```
# noise removal
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN,kernel, iterations = 2)

# sure background area
sure_bg = cv2.dilate(opening,kernel,iterations=3)

# Finding sure foreground area
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
ret, sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)

# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
```

Sonuca bakın. Eşikli görüntüde, madeni paralardan emin olduğumuz bazı madeni para bölgelerini alıyoruz ve şimdi ayrılmışlar. (Bazı durumlarda, karşılıklı dokunan nesneleri ayırmakla değil, sadece ön plan segmentasyonu ile ilgilenebilirsiniz. Bu durumda, mesafe dönüşümü kullanmanıza gerek yoktur, sadece erozyon yeterlidir. Erozyon, ön plan alanını çıkarmak için başka bir yöntemdir, bu herşey.)



Şimdi kesin olarak hangi sikkelerin bölgesi olduğunu, hangilerinin arka plan olduğunu ve hepsini biliyoruz. Bu yüzden işaretleyici oluşturuyoruz (orijinal görüntü ile aynı boyutta, ancak int32 veri tipinde bir dizi) ve içindeki bölgeleri etiketliyoruz. Kesin olarak bildiğimiz bölgeler (ön plan veya arka plan olsun) herhangi bir pozitif tamsayı ile etiketlenir, ancak farklı tamsayılarla etiketlenir ve kesin olarak bilmediğimiz alan sıfır

olarak kalır. Bunun için `cv2.connectedComponents()` kullanıyoruz. Görüntünün arka planını 0 ile etiketler, ardından diğer nesneler 1'den başlayarak tamsayılarla etiketlenir.

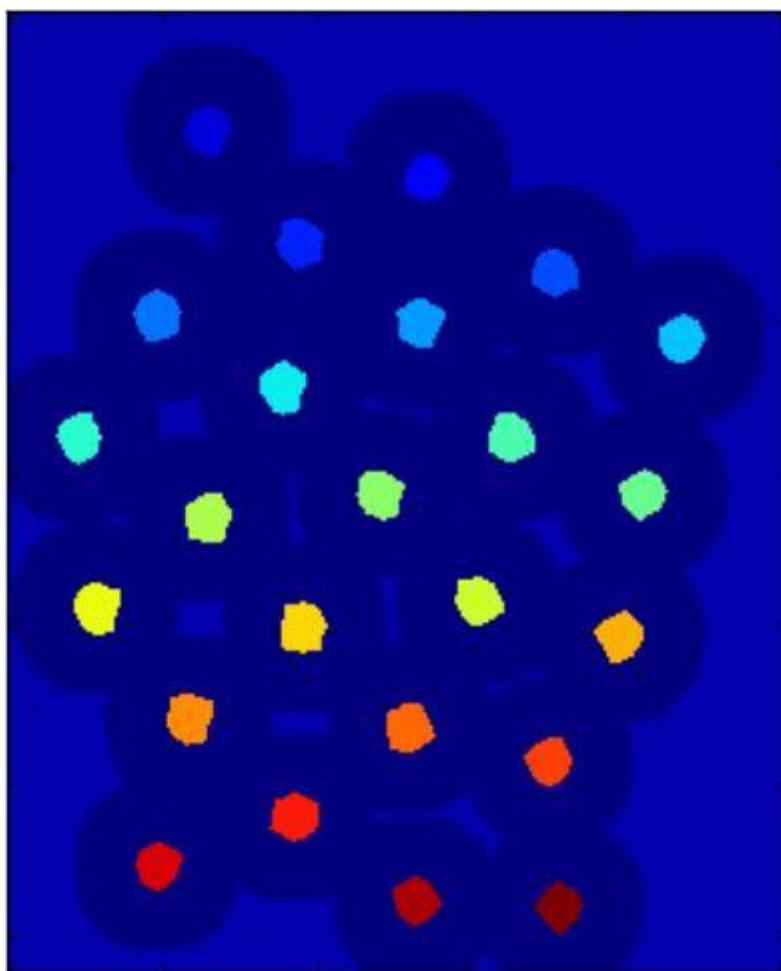
Ancak, arka plan 0 ile işaretlenmiş havzanın bilinmeyen alan olarak kabul edeceğini biliyoruz. Bu yüzden onu farklı bir tamsayı ile işaretlemek istiyoruz. Bunun yerine, tarafından tanımlanan bilinmeyen bölgeyi işaretler bilinmeyen 0 ile.

```
# Marker labelling
ret, markers = cv2.connectedComponents(sure_fg)

# Add one to all labels so that sure background is not 0, but 1
markers = markers+1

# Now, mark the region of unknown with zero
markers[unknown==255] = 0
```

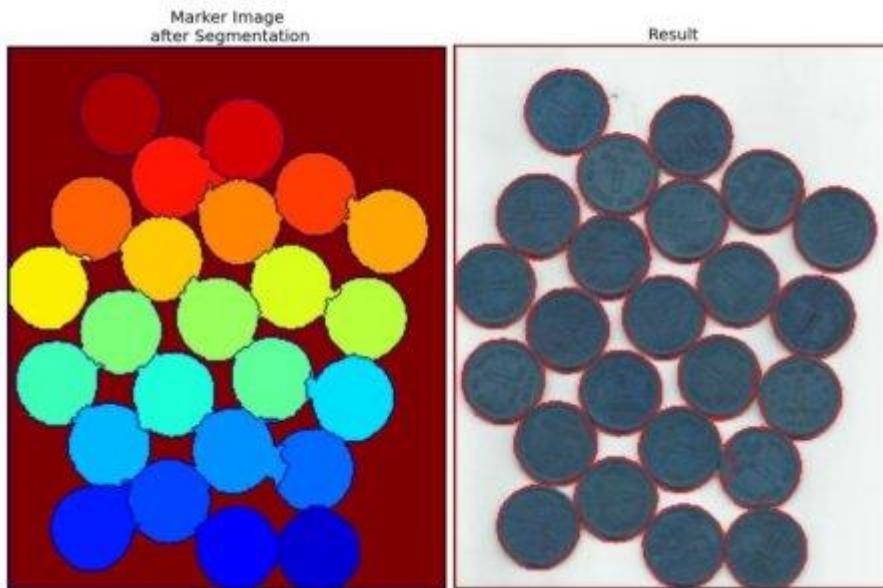
JET renk haritasında gösterilen sonuca bakın. Koyu mavi bölge bilinmeyen bir bölge gösterir. Tabii paralar farklı değerlerle renklendirilir. Arka planın olduğundan emin olan kalan alan bilinmeyen bölgeye kıyasla daha açık mavi gösterilir.



Şimdi markamız hazır. Son adımın zamanı, havza uygulayın. Ardından işaretçi görüntüsü değiştirilecektir. Sınır bölgesi -1 ile işaretlenecektir.

```
markers = cv2.watershed(img,markers)
img[markers == -1] = [255,0,0]
```

Aşağıdaki sonuca bakın. Bazı madeni paralar için, dokundukları bölge düzgün bir şekilde böülümlere ayrılmıştır ve bazıları için değildir.



4.16-)GrabCut Algoritmasını Kullanarak Etkileşimli Ön Plan Çıkarma

Hedef

Bu bölümde

- Görüntülerde ön plan çıkarmak için GrabCut algoritmasını göreceğiz
- Bunun için interaktif bir uygulama oluşturacağız.

teori

GrabCut algoritması, Microsoft Research Cambridge, İngiltere'den Carsten Rother, Vladimir Kolmogorov ve Andrew Blake tarafından tasarlandı. Onların kağıt, içinde [“GrabCut”: interaktif ön plan çalışma iterated grafik kesme kullanarak](#). Minimum kullanıcı etkileşimi ile ön plana çalışma için bir algoritma ihtiyaç duyuldu ve sonuç GrabCut oldu.

Kullanıcı açısından nasıl çalışır? Başlangıçta kullanıcı ön plan bölgesi etrafında bir dikdörtgen çizer (ön plan bölgesi tamamen dikdörtgenin içinde olmalıdır). Ardından algoritma en iyi sonucu almak için yinelemeli olarak böülümlere ayırır. Bitti. Ancak bazı durumlarda, bölümleme iyi olmayacağından, örneğin, bazı ön plan bölgelerini arka plan olarak işaretlemiş olabilir veya tam tersi. Bu durumda, kullanıcının hassas rötuşlar yapması gereklidir. Sadece bazı hatalı sonuçların olduğu görüntülerde bazı vuruşlar yapılabilir. Vuruşlar temelde “Hey, bu bölge ön plan olmalı, arka planı işaretlediniz, bir sonraki yinelemede düzeltin” veya arka planın tersi. Sonra bir sonraki yinelemede daha iyi sonuçlar elde edersiniz.

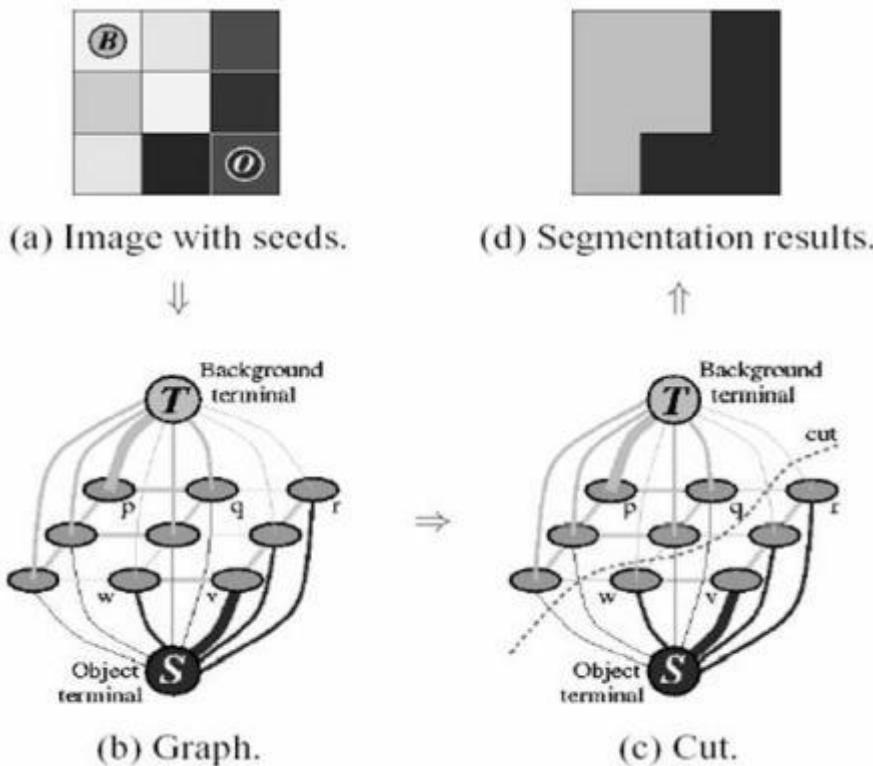
Aşağıdaki resme bakın. İlk oyuncu ve futbol mavi bir dikdörtgenin içinde. Daha sonra beyaz konturlarla (ön plan ifade eden) ve siyah konturlarla (arka plan belirten) bazı son rötuşlar yapılır. Ve güzel bir sonuç elde ediyoruz.



Peki arka planda ne olur?

- Kullanıcı dikdörtgeni girer. Bu dikdörtgenin dışındaki her şey arka plan olarak alınır (Dikdörtgenin tüm nesneleri içermesi için bundan önce bahsedilmesinin nedeni budur). Dikdörtgenin içindeki her şey bilinmiyor. Benzer şekilde, ön plan ve arka planı belirten herhangi bir kullanıcı girişi, sabit etiketleme olarak kabul edilir, bu da işlemde değişmeyecekleri anlamına gelir.
- Bilgisayar, verdiğimiz verilere dayanan bir ilk etiketleme yapar. Ön plan ve arka plan piksellerini etiketler (veya sabit etiketleri)
- Şimdi bir Gauss Karışım Modeli (GMM) ön planı ve arka planı modellemek için kullanılıyor.
- Verdiğimiz verilere bağlı olarak GMM yeni piksel dağılımı öğrenir ve oluşturur. Yani, bilinmeyen pikseller renk istatistikleri açısından diğer sabit etiketli piksellerle ilişkisine bağlı olarak olası ön plan veya olası arka plan olarak etiketlenir (Aynı kümelenme gibidir).
- Bu piksel dağılımından bir grafik oluşturulur. Grafiklerdeki düğümler pikseldir. Ek iki düğüm eklenir: **Kaynak düğümü** ve **Lavabo düğümü**. Her ön plan pikseli Kaynak düğümüne ve her arka plan pikseli Lavabo düğümüne bağlıdır.
- Pikselleri kaynak düğüme / uç düğüme bağlayan kenarların ağırlıkları, bir pikselin ön plan / arka plan olasılığı ile tanımlanır. Pikseller arasındaki ağırlıklar kenar bilgisi veya piksel benzerliği ile tanımlanır. Piksel renginde büyük bir fark varsa, aralarındaki kenar düşük bir ağırlık alır.
- Daha sonra grafiği bölmek için bir mincut algoritması kullanılır. Grafiği, minimum maliyet fonksiyonu ile iki ayırcı kaynak düğümü ve lavabo düğümü olarak keser. Maliyet fonksiyonu, kesilen kenarların tüm ağırlıklarının toplamıdır. Kesimden sonra, Kaynak düğümüne bağlı olan tüm pikseller ön plana ve Lavabo düğümüne bağlı olan pikseller arka plan haline gelir.
- Sınıflandırma yakınılaşana kadar işleme devam edilir.

Aşağıdaki resimde gösterilmiştir (Resim Nezaket: <http://www.cs.ru.ac.za/research/g02m1682/>)



gösteri

Şimdi OpenCV ile grabcut algoritmasına geçiyoruz. OpenCV bunun için **cv2.grabCut()** işlevine sahiptir . Önce argümanlarını göreceğiz:

- *img* – Giriş resmi
- *maske* – Biz arka plan, ön plan veya arka plan muhimmel / ön plan vb aşağıdaki bayrakları, yapılr hangi alanlarda belirttiğiniz bir maske görüntündür **cv2.GC_BGD**, **cv2.GC_FGD**, **cv2.GC_PR_BGD**, **cv2.GC_PR_FGD** , ya da sadece pas Görüntüye 0,1,2,3.
- *rect* – Ön plan nesnesini (x, y, w, h) biçiminde içeren bir dikdörtgenin koordinatlarıdır
- *bdgModel* , *fgdModel* – Bunlar algoritma tarafından dahili olarak kullanılan dizilerdir. Sadece iki adet np.float64 tip sıfır dizisi (1,65) oluşturursunuz.
- *iterCount* – Algoritmanın çalışması gereken yineleme sayısı.
- *mode* – **cv2.GC_INIT_WITH_RECT** veya **cv2.GC_INIT_WITH_MASK** veya birleşik olmalı, bu da dikdörtgen veya son rötuş konturları çizip çizmememize karar verir.

İlk önce dikdörtgen mod ile görelim. Görüntüyü yükleriz, benzer bir maske görüntüsü oluştururuz. Biz yaratmak *fgdModel* ve *bgdModel* . Dikdörtgen parametrelerini veriyoruz. Her şey basit. Algoritmayı 5 yineleme için çalıştırın. Dikdörtgen kullandığımız için *mod* **cv2.GC_INIT_WITH_RECT** olmalıdır . Sonra kavrayışı

çalıştırın. Maske görüntüsünü değiştirir. Yeni maske görüntüsünde pikseller, yukarıda belirtildiği gibi arka plan / ön planı belirten dört bayrakla işaretlenir. Bu yüzden maskeyi, tüm 0 pikselleri ve 2 pikselleri 0 (yani arka plan) ve tüm 1 pikselleri ve 3 pikselleri 1 (yani ön plan pikselleri) olacak şekilde değiştiririz. Şimdi son maskemiz hazır. Bölümlenmiş görüntüyü elde etmek için giriş görüntüsü ile çarpmanız yeterlidir.

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

img = cv2.imread('messi5.jpg')
mask = np.zeros(img.shape[:2],np.uint8)

bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)

rect = (50,50,450,290)
cv2.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_RECT)

mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask2[:, :, np.newaxis]

plt.imshow(img),plt.colorbar(),plt.show()
```

Aşağıdaki sonuçlara bakın:



Hata! Messi'nin saçları gitti. *Messi saçları olmadan kim sever?* Onu geri getirmeliyiz. Bu yüzden orada 1 piksel (kesinlikle ön plan) ile ince bir rötuş yapacağız. Aynı zamanda, toprağın bir kısmı istemediğimiz resme ve ayrıca bir logoya geldi. Onları kaldırmamız lazımdır. Orada bazı 0 piksel rötuşları veriyoruz (kesinlikle arka plan). Bu yüzden, sonuçta gösterdiğimiz maskeyi daha önce de söyledığımız gibi değiştirelim.

Aslında yaptığım şey, boyalı uygulamasında giriş görüntüsünü açtım ve görüntüye başka bir katman ekledim. Boya fırça aracını kullanarak, bu yeni katmanda siyah ile beyaz ve istenmeyen arka plan (logo, zemin vb.) ile kaçırılan ön planı (saç, ayakkabı, top vb)

işaretledim. Sonra kalan arka planı gri ile doldurun. Daha sonra bu maske görüntüsünü yeni eklenen maske görüntüsünde karşılık gelen değerlerle edindiğimiz düzenlenmiş orijinal maske görüntüsünü OpenCV'ye yükledik. Aşağıdaki kodu kontrol edin:

```
# newmask is the mask image I manually labelled
newmask = cv2.imread('newmask.png',0)

# wherever it is marked white (sure foreground), change mask=1
# wherever it is marked black (sure background), change mask=0
mask[newmask == 0] = 0
mask[newmask == 255] = 1

mask, bgdModel, fgdModel =
cv2.grabCut(img,mask,None,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_MASK)

mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask[:, :, np.newaxis]
plt.imshow(img),plt.colorbar(),plt.show()
```

Aşağıdaki sonuca bakın:



İşte bu kadar. Burada rect modunda başlatmak yerine doğrudan maske moduna geçebilirsiniz. Maske görüntüsünde dikdörtgen alanı 2 piksel veya 3 piksel (olası arka plan / ön plan) ile işaretlemeniz yeterlidir. Ardından sure_foreground alanımızı ikinci örnekte yaptığımız gibi 1 piksel ile işaretleyin. Ardından grabCut işlevini maske modıyla doğrudan uygulayın.

5.1-)Özellikleri Anlama Hedef

Bu bölümde, özelliklerin ne olduğunu, neden önemli olduğunu, köşelerin neden önemli olduğunu anlamaya çalışacağız.

açıklama

Çoğunuz bilmecenin oyunlar oynadı olacak. Büyük bir gerçek görüntü oluşturmak için bunları doğru bir şekilde monte etmeniz gereken bir sürü küçük görüntü parçası alırsınız. **Soru şu, nasıl yapıyorsunuz?** Bilgisayarın yapboz bulmacaları oynayabilmesi için aynı teoriyi bir bilgisayar programına yansıtmayla ne dersiniz? Bilgisayar yapboz bulmacaları oynayabilirse, neden iyi bir doğal manzaraya sahip çok sayıda gerçek yaşam görüntüsünü bilgisayara veremeyiz ve tüm bu görüntüleri büyük bir tek görüntüye dikmesini söyleyemeyiz? Bilgisayar birkaç doğal görüntüyü bir araya getirebiliyorsa, bir binanın veya herhangi bir yapının çok sayıda resmini vermeye ve bilgisayara bir 3D model oluşturmmasını söylemeye ne dersiniz?

Sorular ve hayaller devam ediyor. Ama her şey en temel soruya bağlı mı? Yapbozları nasıl oynuyorsun? Çok sayıda karıştırılmış görüntü parçasını tek bir büyük görüntüde nasıl düzenlersiniz? Bir çok doğal görüntüyü tek bir görüntüye nasıl dikebilirsınız?

Cevap, benzersiz olan, kolayca takip edilebilen, kolayca karşılaştırılabilen belirli desenler veya spesifik özellikler arıyoruz. Böyle bir özelliğin tanımına gidersek, onu kelimelerle ifade etmeyi zor bulabiliyoruz, ancak bunların ne olduğunu biliyoruz. Bazıları birkaç görüntü arasında karşılaştırılabilecek iyi bir özelliği belirtmenizi isterse, bir tanesini işaret edebilirsiniz. Bu nedenle, küçük çocuklar bile bu oyunları oynayabilir. Bu özellikleri bir görüntüde ararız, buluruz, aynı özellikleri diğer görüntülerde buluruz, hizalarız. Bu kadar. (Yapbozda, farklı görüntülerin sürekliliğine daha çok bakıyoruz). Tüm bu yetenekler içimizde doğal olarak mevcuttur.

Yani temel sorumuz daha fazla sayıda genişliyor, ancak daha belirgin hale geliyor. **Bu özellikler nelerdir? . (Yanıt bir bilgisayar için de anlaşılabilir olmalıdır.)**

İnsanların bu özellikleri nasıl bulduğunu söylemek zor. Zaten beynimizde programlanmıştır. Ancak bazı resimlere derinlemesine bakarsak ve farklı desenler ararsak ilginç bir şey buluruz. Örneğin, aşağıdaki görüntüyü alın:

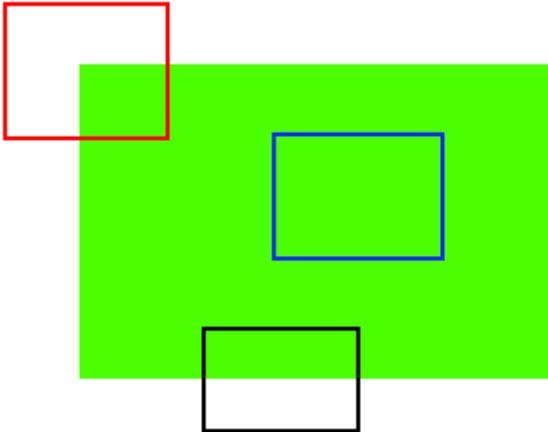


Görüntü çok basit. Resmin üst kısmında altı küçük resim yaması verilir. Sizin için soru bu yamaların orijinal görüntüde tam yerini bulmaktır. Kaç tane doğru sonuç bulabilirsiniz?

A ve B düz yüzeylerdir ve birçok alana yayılırlar. Bu yamaların tam yerini bulmak zordur.

C ve D çok daha basittir. Bunlar binanın kenarlarıdır. Yaklaşık bir konum bulabilirsiniz, ancak kesin konum hala zor. Çünkü kenar boyunca her yerde aynı. Kenardan normal, farklı. Yani kenar düz alana göre çok daha iyi bir özelliktir, ancak yeterince iyi değildir (kenarların sürekliliğini karşılaştırmak için bilmecede iyidir).

Son olarak, E ve F binanın bazı köşeleridir. Ve kolayca bulunabilirler. Çünkü köşelerde, bu yamayı hareket ettirdiğiniz her yerde, farklı görünecektir. Böylece iyi bir özellik olarak kabul edilebilirler. Bu yüzden şimdi daha iyi anlamak için daha basit (ve yaygın olarak kullanılan görüntüye) geçiyoruz.



Yukarıdaki gibi mavi yama düz bir alandır ve bulunması zordur. Mavi yamayı nereye götürürseniz götürün, aynı görünüyor. Siyah yama için bir kenardır. Dikey yönde hareket ettirirseniz (örneğin, gradyan boyunca) değişir. Kenar boyunca koyun (kenara paralel), aynı görünüyor. Ve kırmızı yama için bir köşe. Yamayı nereye götürürseniz götürün, farklı görünüyor, benzersiz olduğu anlamına geliyor. Temel olarak, köşeler bir görüntüdeki iyi özellikler olarak kabul edilir. (Sadece köşeler değil, bazı durumlarda lekeler iyi özellikler olarak kabul edilir).

Şimdi soruyu cevapladık, "bu özellikler nedir?". Ama bir sonraki soru ortaya çıkıyor. Onları nasıl buluruz? Ya da köşeleri nasıl buluruz? Aynı zamanda sezgisel bir şekilde yanıtladık, yani, etrafındaki tüm bölgelerde taşındığında (az miktarda) maksimum varyasyona sahip bölgeleri arayın. Bu, gelecek bölümlerde bilgisayar diline yansıtılacaktır. Bu görüntü özelliklerini bulmaya **Özellik Algılama** denir.

Bu yüzden görüntüdeki özellikleri bulduk (varsayılm). Bir kez bulduğunuzda, diğer resimlerde de aynı şeyi bulmalısınız. Ne yapıyoruz? Özelliğin etrafında bir bölge alıyoruz, "üst kısım mavi gökyüzü, alt kısım inşaat bölgesi, o binada bazı gözlükler var" gibi kendi sözlerimizle açıklıyoruz ve diğer görüntülerde aynı alanı araştırıyoruz. Temel olarak, özelliği açıklıyorsunuz. Benzer şekilde, bilgisayar özelliğin etrafındaki bölgeyi diğer görüntülerde bulabilmesi için tanımlamalıdır. Sözde açıklama **Özellik Açıklaması** olarak adlandırılır. Özelliklere ve açıklamasına sahip olduktan sonra, tüm görüntülerde aynı özellikleri bulabilir ve onları hizalayabilir, birleştirilebilir veya istedığınızı yapabilirsiniz.

Bu modülde, özellikleri bulmak, tanımlamak, eşleştirmek vb. İçin OpenCV'de farklı algoritmalar arıyoruz.

5.2-)Harris Köşe Algılama Hedef

Bu bölümde,

- Harris Corner Detection'ın arkasındaki kavramları anlayacağız.
- İşlevleri göreceğiz: `cv2.cornerHarris()` , `cv2.cornerSubPix()`

teori

Son bölümde, köşelerin görüntüdeki tüm yönlerde yoğunlukta büyük değişiklik gösteren bölgeler olduğunu gördük. Bu köşe bulmak için bir erken girişim tarafından yapıldı **Chris Harris & Mike Stephens** kendi yazısında **A Kombine Köşe ve Kenar Dedektör** yani şimdiki Harris Köşe Dedektör denir, 1988 yılında. Bu basit fikri matematiksel bir forma soktu. Temelde (u, v) , her yönde yer değiştirmeye için yoğunluk farkını bulur. Bu, aşağıdaki gibi ifade edilir:

$$E(u, v) = \sum_{x,y} \underbrace{w(x, y)}_{\text{window function}} \frac{\underbrace{[I(x + u, y + v) - I(x, y)]^2}_{\text{shifted intensity}}}{\underbrace{I(x, y)}_{\text{intensity}}}$$

Pencere işlevi, dikdörtgen bir pencere veya altındaki piksele ağırlık veren gauss penceresidir.

$E(u, v)$ Köşe algılama için bu işlevi en üst düzeye çıkarmalıyız. Bu, ikinci terimi en üst düzeye çıkarmak zorunda olduğumuz anlamına geliyor. Taylor denklemini yukarıdaki denkleme uygulamak ve bazı matematiksel adımları kullanmak (lütfen tam türev için istediğiniz standart ders kitaplarına bakın), son denklemi şu şekilde elde ederiz:

$$E(u, v) \approx [u \ v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

nerede

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Burada, I_x ve I_y sırasıyla x ve y yönlerinde görüntü türevleridir. (**Cv2.Sobel()** kullanılarak kolayca bulunabilir).

Sonra ana kısım geliyor. Bundan sonra, bir pencerenin köşe içerip içermediğini belirleyecek bir puan, temel olarak bir denklem oluşturduklar.

$$R = \det(M) - k(\text{trace}(M))^2$$

nerede

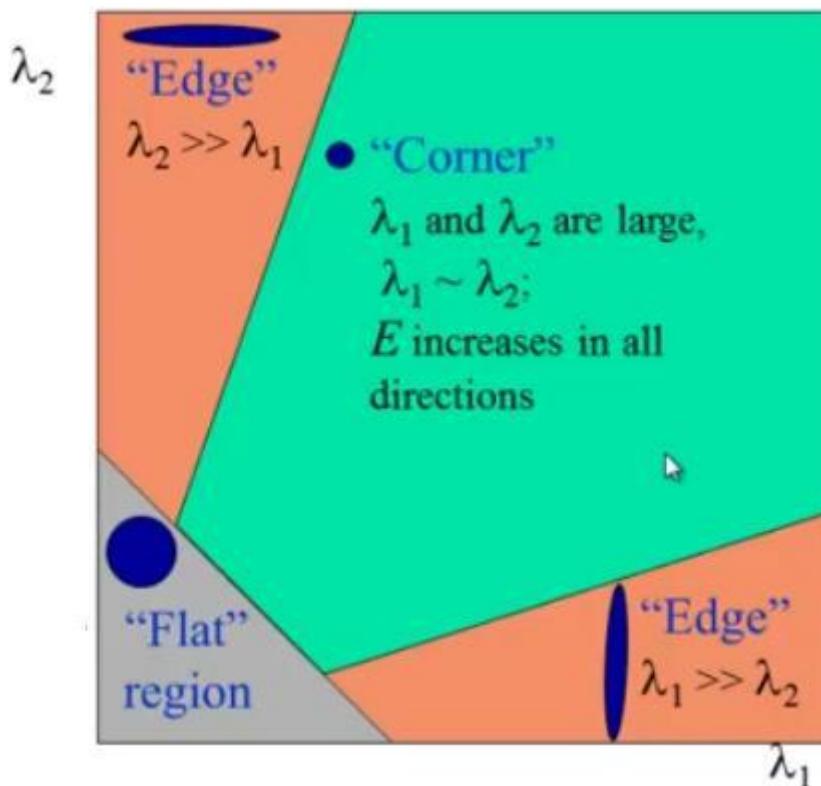
- $\det(M) = \lambda_1 \lambda_2$
- $\text{trace}(M) = \lambda_1 + \lambda_2$
- λ_1 ve λ_2 M'nin öz değerleri

Dolayısıyla bu öz değerlerin değerleri bir bölgenin köşe, kenar veya düz olmasına karar verir.

- Ne zaman $|R|$ küçükse, ne zaman λ_1 ve ne zaman küçük olursa λ_2 , bölge düzdür.
- Ne zaman $R < 0$, bu ne zaman olur, $\lambda_1 >> \lambda_2$ ya da tam tersi, bölge kenardır.

- Ne zaman R büyük, ne zaman λ_1 ve λ_2 büyükse olur ve $\lambda_1 \sim \lambda_2$ bölge bir köşedir.

Aşağıdaki gibi güzel bir resimde temsil edilebilir:



Yani Harris Köşe Algılama'nın sonucu, bu puanlarla gri tonlamalı bir görüntüdür. Uygun bir eşik, görüntüdeki köşeleri verir. Basit bir görüntü ile yapacağımız.

OpenCV'de Harris Köşe Dedektörü

OpenCV bu amaçla **cv2.cornerHarris()** işlevine sahiptir . Argümanları:

- **img** – Giriş resmi, gri tonlamalı ve float32 türünde olmalıdır.
- **blockSize** – Köşe algılama için dikkate alınan mahalle büyüğünü
- **ksize** – Kullanılan Sobel türevinin açıklık parametresi.
- **k** – Denklemde Harris detektörsüz parametresi.

Aşağıdaki örneğe bakın:

```
import cv2
import numpy as np

filename = 'chessboard.jpg'
img = cv2.imread(filename)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

gray = np.float32(gray)
dst = cv2.cornerHarris(gray, 2, 3, 0.04)
```

```

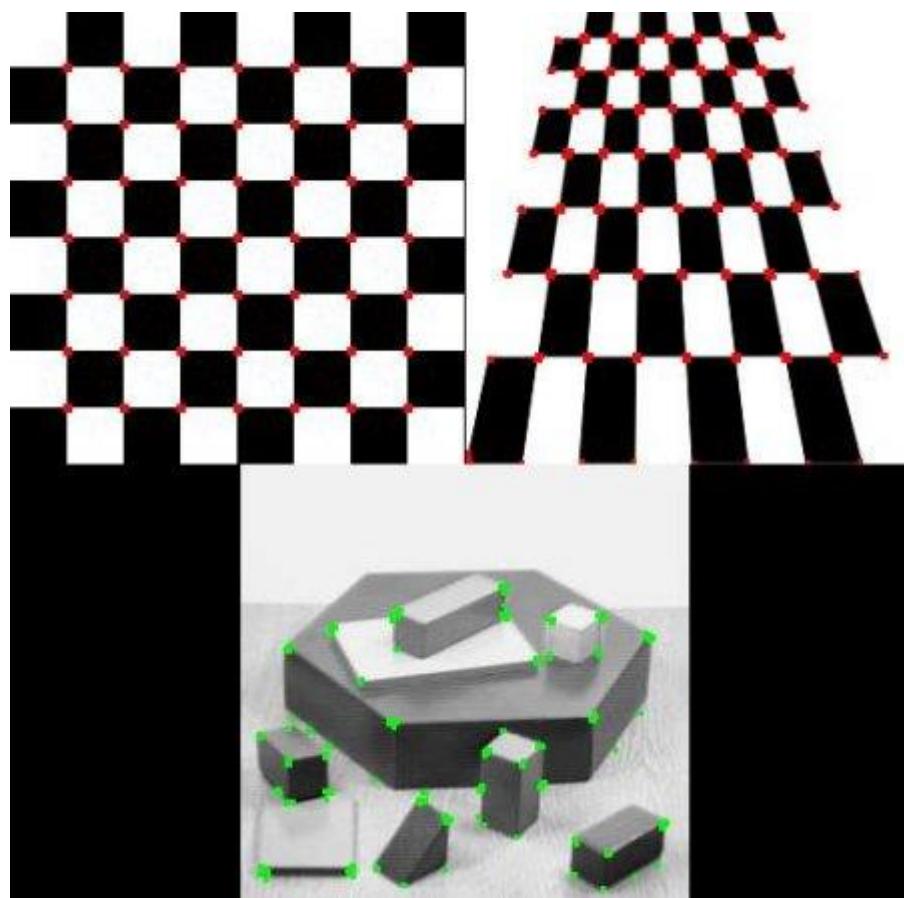
#result is dilated for marking the corners, not important
dst = cv2.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]

cv2.imshow('dst',img)
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()

```

Üç sonuç aşağıdadır:



SubPixel Doğruluklu Köşe

Bazen, köşeleri maksimum doğrulukla bulmanız gerekebilir. OpenCV, alt piksel doğruluğu ile algılanan köşeleri daha da hassaslaşdırın `cv2.cornerSubPix()` işleviyle birlikte gelir. Aşağıda bir örnek verilmiştir. Her zamanki gibi, önce harris köşelerini bulmamız gerekiyor. Sonra bu köşelerin centroidlerini geçiyoruz (Bir köşede bir grup piksel olabilir, sentroidlerini alırız). Harris köşeleri kırmızı piksellerle, rafine köşeler yeşil piksellerle işaretlenmiştir. Bu işlev için, yinelemenin ne zaman durdurulacağını ölçmek zorundayız. Hangisi önce olursa olsun, belirli sayıda yineleme veya belirli bir doğruluk elde edildikten sonra durdururuz. Köşeleri arayacağı mahallenin boyutunu da tanımlamamız gereklidir.

```

import cv2
import numpy as np

filename = 'chessboard2.jpg'
img = cv2.imread(filename)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find Harris corners
gray = np.float32(gray)
dst = cv2.cornerHarris(gray, 2, 3, 0.04)
dst = cv2.dilate(dst, None)
ret, dst = cv2.threshold(dst, 0.01*dst.max(), 255, 0)
dst = np.uint8(dst)

# find centroids
ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)

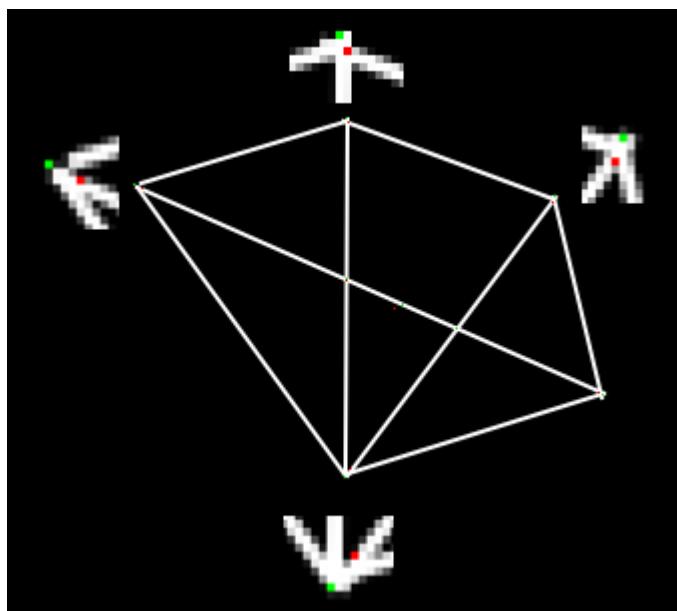
# define the criteria to stop and refine the corners
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.001)
corners = cv2.cornerSubPix(gray, np.float32(centroids), (5, 5), (-1, -1), criteria)

# Now draw them
res = np.hstack((centroids, corners))
res = np.int0(res)
img[res[:, 1], res[:, 0]] = [0, 0, 255]
img[res[:, 3], res[:, 2]] = [0, 255, 0]

cv2.imwrite('subpixel5.png', img)

```

Aşağıda, bazı önemli konumların görselleştirmek için yakınlaştırılmış pencerede gösterildiği sonuç aşağıda verilmiştir:



5.3-)Shi-Tomasi Köşe Dedektörü ve İzlenecek İyi Özellikler Hedef

Bu bölümde,

- Başka bir köşe dedektörü hakkında bilgi edeceğiz: Shi-Tomasi Köşe Dedektörü
- İşlevi göreceğiz: `cv2.goodFeaturesToTrack()`

teori

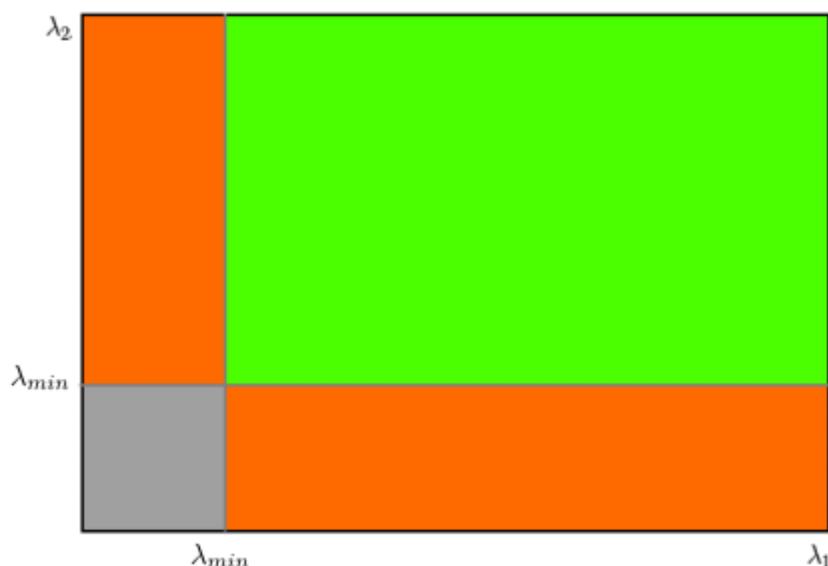
Son bölümde Harris Corner Detector'u gördük. Daha sonra 1994'te J. Shi ve C. Tomasi, Harris Corner Detector'a kıyasla daha iyi sonuçlar gösteren **Good Features to Track** makalelerinde küçük bir değişiklik yaptılar . Harris Corner Detector'daki puanlama fonksiyonu:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Bunun yerine Shi-Tomasi şunları önerdi:

$$R = \min(\lambda_1, \lambda_2)$$

Bir eşik değerden büyükse, köşe olarak kabul edilir. Biz bunu çizmek Eğer $\lambda_1 > \lambda_2$ biz Harris Köşe Dedektörü olduğu gibi uzayda, biz aşağıdaki gibi bir görüntü elde edersiniz:



Şekilde, sadece minimum değerin üstünde λ_1 ve λ_2 üzerinde olduğunda , λ_{\min} bir köşe (yeşil bölge) olarak kabul edildiğini görebilirsiniz.

kod

OpenCV'nin `cv2.goodFeaturesToTrack()` işlevi vardır . Shi-Tomasi yöntemiyle (veya belirtirseniz Harris Köşe Algılama) görüntüdeki N en güçlü köşeleri bulur. Her zamanki gibi, görüntü gri tonlamalı bir görüntü olmalıdır. Ardından bulmak istediğiniz köşelerin sayısını belirtirsiniz. Daha sonra, herkesin reddedildiği minimum köşe kalitesini gösteren 0-1 arasında bir değer olan kalite seviyesini belirtirsiniz. Ardından tespit edilen köşeler arasındaki minimum öklid mesafesini sağlıyoruz.

Tüm bu bilgilerle, işlev görüntüdeki köşeleri bulur. Kalite seviyesinin altındaki tüm köşeler reddedilir. Ardından kalan köşeleri kaliteye göre azalan düzende sıralar. Daha sonra

fonksiyon ilk en güçlü köşeyi alır, yakındaki tüm köşeleri minimum mesafe aralığında atar ve N en güçlü köşeleri döndürür.

Aşağıdaki örnekte, en iyi 25 köşeyi bulmaya çalışacağız:

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

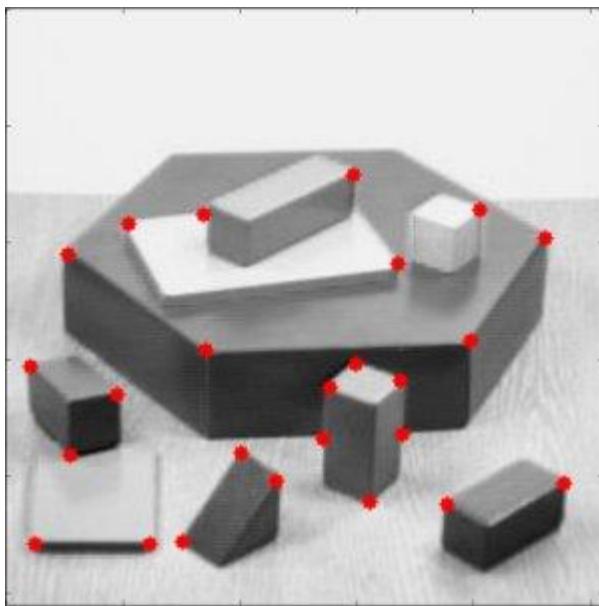
img = cv2.imread('simple.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray, 25, 0.01, 10)
corners = np.int0(corners)

for i in corners:
    x,y = i.ravel()
    cv2.circle(img,(x,y),3,255,-1)

plt.imshow(img),plt.show()
```

Aşağıdaki sonuca bakın:



Bu işlev izleme için daha uygundur. Zamanı geldiğinde göreceğiz.

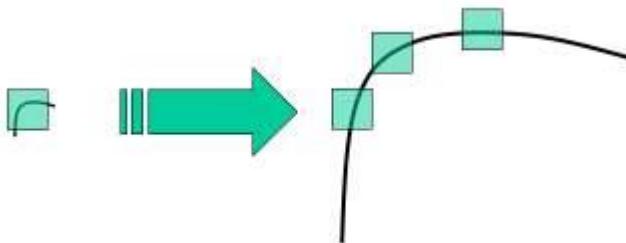
5.4-)SIFT'e Giriş (Ölçek Değişmez Özellik Dönüşümü) Hedef

Bu bölümde,

- SIFT algoritması kavramlarını öğreneceğiz
- SIFT Anahtar Noktalarını ve Tanımlayıcılarını bulmayı öğreneceğiz.

teori

Son birkaç bölümde Harris gibi bazı köşe dedektörleri gördük. Bunlar dönme-değışmez, yani görüntü döndürülmüş olsa bile aynı köşeleri bulabiliyoruz. Açıktır, çünkü köşeler döndürülmüş görüntüde de köşeler olarak kalır. Peki ya ölçeklendirme? Görüntü ölçeklendirilirse köşe köşe olmayabilir. Örneğin, aşağıdaki basit bir resmi kontrol edin. Küçük pencerede küçük bir görüntüdeki bir köşe, aynı pencerede yakınlaştırıldığında düzdir. Yani Harris köşesi ölçek değişmez değil.



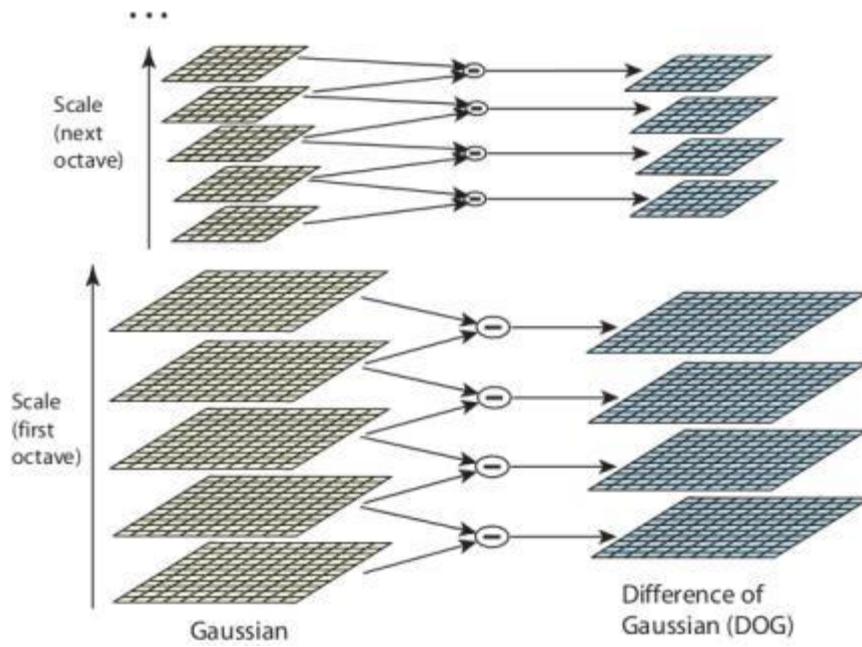
Böylece, 2004 yılında, British Columbia Üniversitesi D.Lowe, makalesinde yeni bir algoritma, Ölçek Değışmez Özellik Dönüşümü (SIFT), temel noktaları çıkarıp tanımlayıcılarını hesaplayan Ölçek Değışmez Anahtar Noktalarından Ayırt Edici Görüntü Özellikleri ile geldi. (Bu makalenin anlaşılması kolaydır ve SIFT'de mevcut olan en iyi malzeme olarak kabul edilmektedir. Bu nedenle bu açıklama bu makalenin kısa bir özetiidir).

SIFT algoritmasında temel olarak dört adım vardır. Onları tek tek göreceğiz.

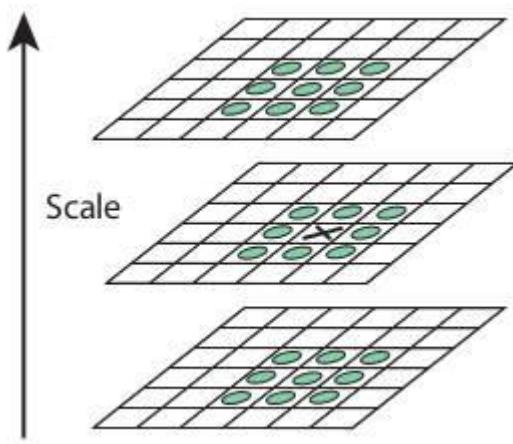
1. Ölçek-Uzay Ekstrema Tespiti

Yukarıdaki görüntünden, farklı ölçüye sahip tuş noktalarını tespit etmek için aynı pencereyi kullanamayacağımız açıktır. Küçük köşe ile sorun yok. Ancak daha büyük köşeleri tespit etmek için daha büyük pencerelere ihtiyacımız var. Bunun için ölçek alanı filtrelemesi kullanılır. İçinde, çeşitli σ değerlere sahip görüntü için Gaussian Laplacian bulunur. LoG, değişikliğe bağlı olarak çeşitli boyutlardaki lekeleri tespit eden bir damla dedektörü olarak işlev görür σ . Kısacası, σ ölçekleme parametresi olarak işlev görür. Örneğin, yukarıdaki görüntüde, düşük gauss çekirdeği σ küçük köşe için yüksek değer verirken, yüksek gaussian çekirdek σ daha büyük köşe için uygundur. Bu nedenle, ölçek ve boşluk boyunca yerel maksimumları bulabiliyoruz, bu da bize (x, y, σ) değerlerin bir listesini verir, bu da (x, y) σ ölçüğinde potansiyel bir kilit nokta olduğu anlamına gelir.

Ancak bu LoG biraz masraflıdır, bu nedenle SIFT algoritması LoG'nin bir tahmini olan Gauss Farkını kullanır. Gauss farkı, bir görüntünün Gauss bulanıklaştırılmasının iki farklı ile farklı olarak elde edilir σ , olsun σ ve olsun $k\sigma$. Bu işlem Gauss Piramidi'ndeki görüntünün farklı oktavlari için yapılır. Aşağıdaki resimde temsil edilmektedir:



Bu DoG bulunduğuunda, görüntüler ölçek ve boşluk üzerinden lokal ekstrema için aranır. Örneğin, bir görüntüdeki bir piksel 8 komşusuyla ve bir sonraki ölçekte 9 piksel ve önceki ölçeklerde 9 piksel ile karşılaştırılır. Yerel bir ekstrema ise, potansiyel bir anahtar noktadır. Temel olarak, kilit noktanın en iyi şekilde bu ölçekte temsil edildiği anlamına gelir. Aşağıdaki resimde gösterilmiştir:



Farklı parametreler ile ilgili olarak, kağıt oktav = 4 sayısı, derece seviyelerinde = 5, ilk sayısı şu şekilde özetlenebilir bazı deneySEL veriler verir $\sigma = 1.6$, $k = \sqrt{2}$ uygun değerler olarak vb.

2. Anahtar Noktası Yerelleştirmesi

Potansiyel kilit nokta konumları bulunduğunda, daha doğru sonuçlar elde etmek için bunların iyileştirilmesi gerekir. Ekstremin daha doğru yerini elde etmek için Taylor serisinin ölçek alanının genişlemesini kullandılar ve bu ekstremadaki yoğunluk bir eşik

değerden (kağıt başına 0.03) daha azsa reddedilir. Bu eşiğe OpenCV'de **kontrast Eşiği** denir

DoG kenarlar için daha yüksek tepkiye sahiptir, bu nedenle kenarların da çıkarılması gereklidir. Bunun için Harris köşe dedektörüne benzer bir konsept kullanılır. İki uçlu eğriliği hesaplamak için 2×2 Hessian matrisi (H) kullanılır. Harris köşe detektöründen kenarlar için bir öz değerinin diğerinden daha büyük olduğunu biliyoruz. Burada basit bir işlev kullandılar,

Bu oran **OpenCV'de edgeThreshold** olarak adlandırılan bir eşik değerden büyükse, o anahtar nokta atılır. Kağıtta 10 olarak verilmiştir.

Böylece, düşük kontrastlı tuş noktalarını ve kenar tuş noktalarını ortadan kaldırır ve geriye kalan güçlü ilgi noktalarıdır.

3. Oryantasyon Ataması

Şimdi, görüntü döndürmeye değişmezlik sağlamak için her bir kilit noktaya bir yönlendirme atanmıştır. Ölçeğe bağlı olarak kilit nokta konumu etrafında bir komşuluk alınır ve bu bölgede gradyan büyülüğu ve yönü hesaplanır. 360 derece kapsayan 36 bölmeli bir yönlendirme histogramı oluşturulur. (Degrade büyülük ve gauss ağırlıklı ağırlıklı pencere ile ağırlık σ noktası ölçüğünün 1,5 katına eşittir. Histogramdaki en yüksek tepe alınır ve% 80'in üzerindeki herhangi bir tepe de yönlendirmeyi hesaplar. aynı konuma ve ölçüye, ancak farklı yönlere sahiptir.

4. Anahtar Nokta Tanımlayıcısı

Şimdi kilit nokta tanımlayıcısı oluşturuldu. Kilit noktanın etrafında 16×16 mahalle alınır. 4×4 boyutunda 16 alt bloğa ayrılmıştır. Her alt blok için 8 bölmeli yön histogramı oluşturulur. Böylece toplam 128 bin değer mevcuttur. Anahtar nokta tanımlayıcısı oluşturmak için bir vektör olarak temsil edilir. Buna ek olarak, aydınlatma değişikliklerine, rotasyona vs. karşı sağlamlık elde etmek için çeşitli önlemler alınır.

5. Keypoint Eşleme

İki görüntü arasındaki ana noktalar, en yakın komşularını belirleyerek eşleştirilir. Ancak bazı durumlarda, en yakın ikinci maçı birinciye çok yakın olabilir. Gürültü veya başka nedenlerden dolayı olabilir. Bu durumda, en yakın mesafenin ikinci en yakın mesafeye oranı alınır. 0.8'den büyükse reddedilir. Yanlış eşleşmelerin yaklaşık% 90'ını ortadan kaldırırken, kağıda göre sadece% 5'lik doğru eşleşmeleri atar.

Bu SIFT algoritmasının bir özetidir. Daha fazla ayrıntı ve anlayış için orijinal makalenin okunması önemle tavsiye edilir. Bir şeyi hatırlayın, bu algoritma patentlidir. Bu nedenle bu algoritma OpenCV'deki Serbest Olmayan modülüne dahil edilmiştir.

OpenCV'de SIFT

Şimdi OpenCV'de bulunan SIFT işlevlerini görelim. Tuş noktası tespiti ile başlayalım ve çizelim. İlk önce bir SIFT nesnesi inşa etmeliyiz. Ona istege bağlı olan ve dokümanlarda iyi açıklanan farklı parametreler iletebiliriz.

```
import cv2
import numpy as np

img = cv2.imread('home.jpg')
gray= cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

sift = cv2.SIFT()
kp = sift.detect(gray,None)

img=cv2.drawKeypoints(gray,kp)

cv2.imwrite('sift_keypoints.jpg',img)
```

sift.detect () işlevi görüntülerdeki **kilit noktası** bulur. Görüntünün yalnızca bir bölümünü aramak istiyorsanız bir maske iletebilirsiniz. Her bir kilit nokta, (x, y) koordinatları, anlamlı mahallenin büyülüğu, yönünü belirten açı, kilit noktaların gücünü belirten yanıt vb. Birçok özelliği sahip özel bir yapıdır.

OpenCV ayrıca, anahtar noktalarının konumlarına küçük daireler çizen **cv2.drawKeyPoints ()** işlevi de sağlar . Eğer ona bir bayrak **gönderirseniz** , **cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS** ona bir anahtar noktası büyüğünde bir daire çizecek ve hatta yönünü bile gösterecektir. Aşağıdaki örneğe bakın.

```
img=cv2.drawKeypoints(gray,kp,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imwrite('sift_keypoints.jpg',img)
```

Aşağıdaki iki sonuca bakın:



Şimdi tanımlayıcıyı hesaplamak için OpenCV iki yöntem sunmaktadır.

1. Zaten anahtar noktaları bulduğunuzdan , tanımlayıcıları bulduğumuz anahtar noktalardan hesaplayan `sift.compute ()` yöntemini çağırabilirsiniz . Örnek: `kp, des = sift.compute (gri, kp)`
2. Anahtar noktaları bulamadıysanız, `sift.detectAndCompute ()` işleviyle tek bir adımda doğrudan anahtar noktalarını ve tanımlayıcıları bulun .

İkinci yöntemi göreceğiz:

```
sift = cv2.SIFT()  
kp, des = sift.detectAndCompute(gray, None)
```

Burada `kp`, anahtar noktaların bir listesi olacak ve `des`, numpy bir şekil dizisidir **Number_of_Keypoints × 128**.

Böylece anahtar noktaları, tanımlayıcıları vb. Var. Şimdi farklı görüntülerdeki anahtar noktaları nasıl eşleştireceğimizi görmek istiyoruz. Önümüzdeki bölümlerde öğreneceğimiz.

5.5-)SURF'a Giriş (Hızlandırılmış Sağlam Özellikler) Hedef

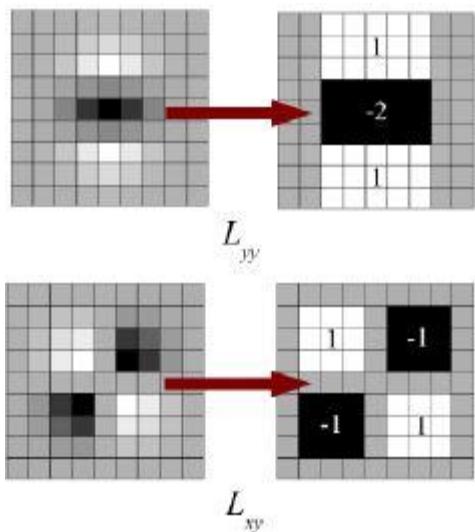
Bu bölümde,

- SURF'un temellerini göreceğiz
- OpenCV'de SURF işlevlerini göreceğiz

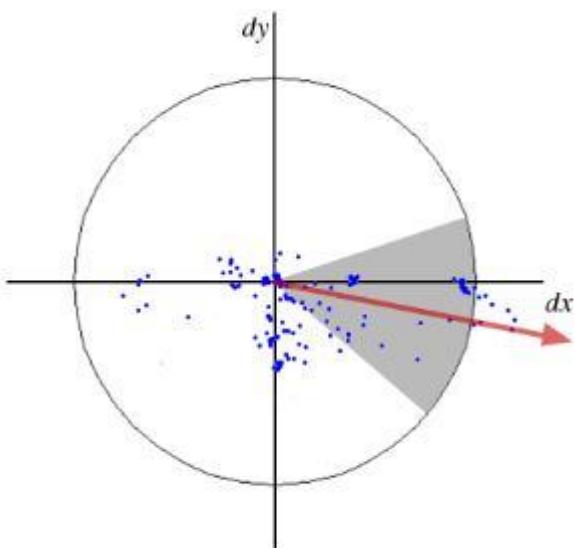
teori

Son bölümde, anahtar nokta tespiti ve açıklaması için SIFT'i gördük. Ancak nispeten yavaştı ve insanların daha hızlı bir sürüme ihtiyacı vardı. 2006 yılında, Bay, H., Tuytelaars, T. ve Van Gool, L adlı üç kişi, SURF adında yeni bir algoritma getiren "SURF: Hızlandırılmış Sağlam Özellikleri" adlı başka bir makale yayınladı. Adından da anlaşılacağı gibi, SIFT'nin hızlandırılmış bir sürümüdür.

SIFT'de Lowe, ölçek uzayı bulmak için Gaussian Farkı ile Gaussian Laplacian'a yaklaştı. SURF biraz daha ileri gider ve Kutu Filtresi ile LoG'ye yaklaşır. Aşağıdaki görüntü böyle bir yaklaşımın bir örneğini göstermektedir. Bu yaklaşımın büyük bir avantajı, kutu filtreli konvolüsyonun entegre görüntüler yardımıyla kolayca hesaplanabilmesidir. Ve farklı ölçekler için paralel olarak kullanılabilir. Ayrıca SURF hem ölçek hem de konum için Hessen matrisinin determinantına güvenmektedir.



Yönlendirme ataması için SURF, 6s büyüğünde bir mahalle için dalgacık tepkilerini yatay ve dikey yönde kullanır. Buna yeterli gaussian ağırlıkları da uygulanır. Daha sonra aşağıdaki resimde gösterildiği gibi bir alana çizilirler. Baskın yönelim, 60 derecelik bir kayma yönelimi penceresindeki tüm cevapların toplamının hesaplanmasıyla tahmin edilir. İlginç olan şey, dalgacık yanıtının herhangi bir ölçekte integral görüntüler kullanılarak kolayca bulunabilmesidir. Birçok uygulama için rotasyon değişmezliği gerekli değildir, bu nedenle süreci hızlandıran bu yönlendirmeyi bulmaya gerek yoktur. SURF, Upright-SURF veya U-SURF adı verilen bir işlevsellik sağlar. Hızı artırır ve dayanıklıdır $\pm 15^\circ$. OpenCV bayrağı bağlı olarak, destekler dik. 0 ise, yönlendirme hesaplanır. 1 ise yön hesaplanmaz ve daha hızlıdır.

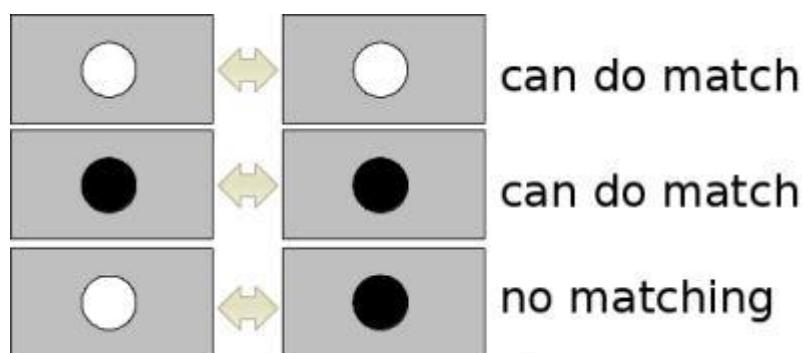


Özellik açıklaması için SURF, Dalgacık yanıtlarını yatay ve dikey yönde kullanır (yne, entegre görüntülerin kullanılması işleri kolaylaştırır). 20sX20s büyüğünde bir mahal, s'nin büyüğünün olduğu kilit noktanın etrafına alınır. 4x4 alt bölgeye ayrılmıştır. Her alt bölge için yatay ve dikey dalgacık tepkileri alınır ve böyle bir vektör oluşturulur $v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$. Bu, bir vektör olarak temsil edildiğinde toplam 64 boyutlu SURF özellik tanımlayıcısı verir. Boyutu düşürün,

hesaplama ve eşleştirme hızını artırın, ancak özelliklerin daha iyi ayırt edilmesini sağlayın.

Daha belirgin olmak için, SURF özellik tanımlayıcısının genişletilmiş 128 boyutlu bir sürümü vardır. Toplamları d_x ve $|d_x|$ ayrı ayrı hesaplanır $d_y < 0$ ve $d_y \geq 0$. Benzer şekilde, toplamı d_y ve $|d_y|$ işaretine göre bölünür d_x , böylece özellik sayısını ikiye katlar. Çok fazla hesaplama karmaşıklığı eklemez. OpenCV, 64-dim ve 128-dim için sırasıyla 0 ve 1 ile **genişletilen** bayrağın değerini ayarlayarak destekler (varsayılan 128-dim)

Bir diğer önemli gelişme, altta yatan faiz noktası için Laplacian (Hessian Matrix'in izi) işaretinin kullanılmasıdır. Algılama sırasında zaten hesaplandığından hesaplama maliyeti eklemez. Laplacian'ın işaretini koyu arka planlardaki parlak lekeleri ters durumdan ayırır. Eşleştirme aşamasında, özellikleri yalnızca aynı kontrast türüne sahipse karşılaşırız (aşağıdaki resimde gösterildiği gibi). Bu minimum bilgi, tanımlayıcının performansını düşürmeden daha hızlı eşleştirmeye olanak tanır.



Kısaltısı, SURF her adımda hızı artırmak için birçok özellik ekler. Analiz, performans SIFT ile karşılaştırılabilirken SIFT'den 3 kat daha hızlı olduğunu göstermektedir. SURF, bulanıklaştırma ve döndürmeyle görüntüleri işlemede iyidir, ancak bakış açısı değişikliği ve aydınlatma değişikliğini işlemede iyi değildir.

OpenCV'de SURF

OpenCV, tıpkı SIFT gibi SURF işlevleri sağlar. SURF nesnesini 64/128-dim tanımlayıcılar, Dik / Normal SURF vb. Gibi bazı isteğe bağlı koşullarla başlatırsınız. Tüm ayrıntılar dokümanlarda iyi açıklanmıştır. Daha sonra SIFT'de yaptığımız gibi, anahtar noktaları ve tanımlayıcıları bulmak için SURF.detect (), SURF.compute () vb.

İlk olarak, SURF anahtar noktalarını ve tanımlayıcılarını nasıl bulacağınız ve çizeceğiniz hakkında basit bir demo göreceğiz. Sadece SIFT ile aynı olduğundan tüm örnekler Python terminalinde gösterilmiştir.

```
>>> img = cv2.imread('fly.png',0)

# Create SURF object. You can specify params here or later.
# Here I set Hessian Threshold to 400
```

```

>>> surf = cv2.SURF(400)

# Find keypoints and descriptors directly
>>> kp, des = surf.detectAndCompute(img,None)

>>> len(kp)
699

```

1199 tuş noktası bir resimde gösterilemeyecek kadar fazla. Bir görüntüye çizmek için 50'ye indiriyoruz. Eşleştirirken, tüm bu özelliklere ihtiyacımız olabilir, ancak şimdi değil. Hessian Eşliğini artırıyoruz.

```

# Check present Hessian threshold
>>> print surf.hessianThreshold
400.0

# We set it to some 50000. Remember, it is just for representing in picture.
# In actual cases, it is better to have a value 300-500
>>> surf.hessianThreshold = 50000

# Again compute keypoints and check its number.
>>> kp, des = surf.detectAndCompute(img,None)

>>> print len(kp)
47

```

50'den az. Görüntünün üzerine çizelim.

```

>>> img2 = cv2.drawKeypoints(img,kp,None,(255,0,0),4)

>>> plt.imshow(img2),plt.show()

```

Aşağıdaki sonuca bakın. SURF'un daha çok bir damla dedektörü gibi olduğunu görebilirsiniz. Kelebek kanatlarındaki beyaz lekeleri algılar. Diğer görüntülerle test edebilirsiniz.



Şimdi U-SURF uygulamak istiyorum, böylece yönünü bulamaz.

```

# Check upright flag, if it False, set it to True
>>> print surf.upright
False

>>> surf.upright = True

# Recompute the feature points and draw it
>>> kp = surf.detect(img,None)
>>> img2 = cv2.drawKeypoints(img,kp,None,(255,0,0),4)

>>> plt.imshow(img2),plt.show()

```

Aşağıdaki sonuçlara bakın. Tüm yönler aynı yönde gösterilir. Bir öncekinden daha hızlı. Yönlendirmenin sorun olmadığı durumlar (panorama dikiş gibi) vb. Üzerinde çalışıyorsanız, bu daha iyidir.



Son olarak, tanımlayıcı boyutunu kontrol ediyoruz ve yalnızca 64-dim ise 128 olarak değiştiriyoruz.

```

# Find size of descriptor
>>> print surf.descriptorSize()
64

# That means flag, "extended" is False.
>>> surf.extended
False

# So we make it to True to get 128-dim descriptors.
>>> surf.extended = True
>>> kp, des = surf.detectAndCompute(img,None)
>>> print surf.descriptorSize()
128
>>> print des.shape
(47, 128)

```

Kalan kısım, başka bir bölümde yapacağımız eşleşmedir.

5.6-) Köşe Algılama için FAST Algoritması Hedef

Bu bölümde,

- FAST algoritmasının temellerini anlayacağız
- FAST algoritması için OpenCV işlevlerini kullanarak köşeleri bulacağız.

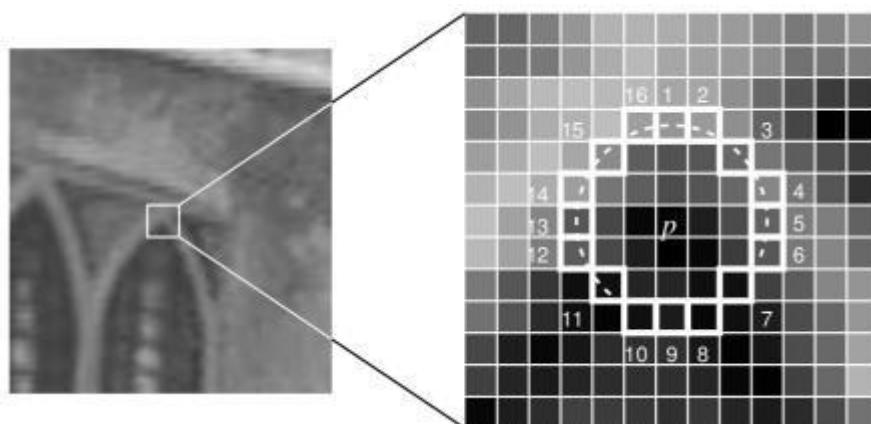
teori

Birkaç özellik dedektörü gördük ve birçoğu gerçekten iyi. Ancak gerçek zamanlı bir uygulama açısından bakıldığında, yeterince hızlı değildirler. Bunun en iyi örneklerinden biri, sınırlı hesaplama kaynakları olan SLAM (Eşzamanlı Yerelleştirme ve Haritalama) mobil robotudur.

Buna bir çözüm olarak, 2006 yılında Edward Rosten ve Tom Drummond tarafından "Yüksek hızlı köşe algılaması için makine öğrenmesi" başlıklı makalesinde FAST (Hızlandırılmış Segment Testinden Özellikler) algoritması önerilmiştir (Daha sonra 2010'da revize edilmiştir). Algoritmanın temel bir özeti aşağıda sunulmuştur. Daha fazla ayrıntı için orijinal kağıda başvurun (Tüm görüntüler orijinal kağıttan alınmıştır).

FAST kullanarak Özellik Algılama

1. Görüntüde bir ilgi noktası olarak tanımlanacak ya da tanımlanmayacak bir piksel seçin. Yoğunluğu olsun I_p .
2. Uygun eşik değerini seçin t .
3. Test edilen pikselin etrafında 16 piksellik bir daire düşünün. (Aşağıdaki resme bakın)



4. Şimdi, daire içinde tümü daha parlak veya daha koyu olan I_p bir dizi nbitişik piksel varsa (16 piksel) piksel bir köşedir. (Yukarıdaki resimde beyaz kesik çizgilerle gösterilmiştir). 12 olarak seçildi. $I_p + tI_p - t_n$
5. Çok sayıda köşeyi dışarıda bırakmak için **yüksek hızlı bir test** önerilmiştir. Bu test 1, 9, 5 ve 13'teki yalnızca dört pikseli inceler (ilk 1 ve 9, çok daha parlak veya

daha koyu olup olmadıklarını test eder. Varsa, 5 ve 13'ü kontrol eder). Eğer P bir köşe, sonra bunlardan en az üç hepsinden daha parlak olmalıdır $I_p + t$ veya daha koyu $I_p - t$. Bunlardan hiçbirini böyle P değilse, köşe olamaz. Tam segment test kriteri, daire içindeki tüm pikseller incelenerek geçen adaylara uygulanabilir. Bu dedektör kendi başına yüksek performans gösterir, ancak birkaç zayıflık vardır:

- $N < 12$ için çok fazla adayı reddetmez.
- Piksel seçimi optimum değildir, çünkü etkinliği soruların sırasına ve köşe görünümülerinin dağılımına bağlıdır.
- Yüksek hızlı testlerin sonuçları atılır.
- Birbirine bitişik birden fazla özellik algılanır.

İlk 3 nokta bir makine öğrenimi yaklaşımı ile ele alınmaktadır. Sonuncusu, maksimal olmayan bastırma kullanılarak ele alınır.

Makine Bir Köşe Dedektörü Öğrenme

1. Eğitim için bir dizi görüntü seçin (tercihen hedef uygulama alanından)
2. Özellik noktalarını bulmak için her görüntüde FAST algoritmasını çalıştırın.
3. Her özellik noktası için, etrafındaki 16 pikseli vektör olarak saklayın. Tüm görüntülerin özellik vektörü olmasını sağlayın P .
4. \forall Bu 16 pikseldeki her piksel (say) aşağıdaki üç durumdan birine sahip olabilir:

$$S_{p \rightarrow x} = \begin{cases} d, & I_{p \rightarrow x} \leq I_p - t \quad (\text{darker}) \\ s, & I_p - t < I_{p \rightarrow x} < I_p + t \quad (\text{similar}) \\ b, & I_p + t \leq I_{p \rightarrow x} \quad (\text{brighter}) \end{cases}$$

5. Bu devletlerin bağlı olarak, özellik vektörü P_3 alt kümeleri bölünmüştür, P_d, P_s, P_b .
6. Yeni bir boole değişkeni tanımlayın K_p ; bu P bir köşe ise doğrudur, aksi takdirde false olur.
7. K_p Gerçek sınıf hakkında bilgi için değişkeni kullanarak her alt kümeyi sorgulamak için ID3 algoritmasını (karar aacı sınıflandırıcısı) kullanın. \forall Aday pikselinin entropisi ile ölçülen bir köşe olup olmadığı hakkında en fazla bilgiyi veren öğeyi seçer K_p .
8. Bu, entropisi sıfır oluncaya kadar tüm alt kümelere yinelemeli olarak uygulanır.
9. Bu şekilde oluşturulan karar aacı diğer görüntülerde hızlı tespit için kullanılır.

Maksimal Olmayan Bastırma

Bitişik konumlarda birden fazla ilgi noktasını tespit etmek başka bir sorundur. Maksimum Olmayan Bastırma kullanılarak çözülür.

1. ∇ Algılanan tüm özellik noktaları için bir puan işlevi hesaplayın . ve çevresindeki 16 piksel değeri ∇ arasındaki mutlak farkın toplamıdır P .
2. Bitişik iki anahtar noktayı düşünün ve ∇ değerlerini hesaplayın .
3. Daha düşük ∇ değerli olanı atın .

Özet

Mevcut diğer köşe dedektörlerinden birkaç kat daha hızlıdır.

Ancak yüksek gürültü seviyelerine dayanıklı değildir. Bir eşik değere bağlıdır.

OpenCV'de HIZLI Özellik Dedektörü

OpenCV'de başka bir özellik dedektörü olarak adlandırılır. İsterseniz maksimum eşik değerinin uygulanıp uygulanmayacağı, kullanılacak mahalleyi vb. Eşiği belirleyebilirsiniz.

Semt için üç bayrak tanımlanır, cv2.FAST_FEATURE_DETECTOR_TYPE_5_8 , cv2.FAST_FEATURE_DETECTOR_TYPE_7_12 ve cv2.FAST_FEATURE_DETECTOR_TYPE_9_16 . Aşağıda HIZLI özellik noktalarının nasıl algılanacağı ve çizileceği ile ilgili basit bir kod bulunmaktadır.

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

img = cv2.imread('simple.jpg',0)

# Initiate FAST object with default values
fast = cv2.FastFeatureDetector()

# find and draw the keypoints
kp = fast.detect(img,None)
img2 = cv2.drawKeypoints(img, kp, color=(255,0,0))

# Print all default params
print "Threshold: ", fast.getInt('threshold')
print "nonmaxSuppression: ", fast.getBool('nonmaxSuppression')
print "neighborhood: ", fast.getInt('type')
print "Total Keypoints with nonmaxSuppression: ", len(kp)

cv2.imwrite('fast_true.png',img2)

# Disable nonmaxSuppression
fast.setBool('nonmaxSuppression',0)
kp = fast.detect(img,None)

print "Total Keypoints without nonmaxSuppression: ", len(kp)
```

```
img3 = cv2.drawKeypoints(img, kp, color=(255,0,0))  
cv2.imwrite('fast_false.png',img3)
```

Sonuçları görün. İlk görüntü, maxmaxSuppression ile FAST ve ikinci maxmaxSuppression olmadan gösterir.



5.7-)BRIEF (kısa)(ikili Sağlam Bağımsız Temel Özellikler) Hedef

Bu bölümde

- BRIEF algoritmasının temellerini göreceğiz

teori

SIFT'in tanımlayıcılar için 128-dim vektör kullandığını biliyoruz. Kayan nokta sayıları kullandığından, temel olarak 512 bayt alır. Benzer şekilde SURF da en az 256 bayt alır (64-dim için). Binlerce özellik için böyle bir vektör oluşturmak, özellikle gömülü sistemler için kaynak kısıtlama uygulamaları için mümkün olmayan çok fazla bellek gerektirir. Daha büyük bellek, eşleştirme için daha uzun süre gereklidir.

Ancak gerçek eşleme için tüm bu boyutlara gerek olmamalıdır. PCA, LDA gibi çeşitli yöntemler kullanarak sıkıştırabiliriz. LSH (Locality Sensitive Hashing) kullanarak karma yapma gibi diğer yöntemler bile, bu SIFT tanımlayıcılarını kayan nokta sayılarında ikili dizelere dönüştürmek için kullanılır. Bu ikili dizeler Hamming mesafesini kullanarak özellikleri eşleştirmek için kullanılır. Bu, daha iyi bir hızlanma sağlar çünkü çekiçleme mesafesini bulmak sadece SSE talimatları ile modern CPU'larda çok hızlı olan XOR ve bit sayısını uygular. Ama burada, önce tanımlayıcıları bulmalıyız, o zaman sadece hafızadaki ilk sorunumuzu çözmeyen hash uygulayabiliriz.

KISA şu anda resme giriyor. Tanımlayıcıları bulmadan doğrudan ikili dizeleri bulmak için bir kısayol sağlar. Düzgünleştirilmiş görüntü yamasını alır ve bir dizi $n_d(x, y)$ konum çiftini benzersiz bir şekilde seçer (kağıtta açıklanmıştır). Daha sonra bu konum çiftlerinde bazı piksel yoğunluğu karşılaştırması yapılır. Örneğin, ilk konum çiftleri P ve olsun Q . Eğer $I(P) < I(Q)$ sonuç 1 ise, başka 0 ise. Bu boyutsal bir bit dizgesi n_d elde etmek için tüm konum çiftleri için uygulanır n_d .

Bu n_d 128, 256 veya 512 olabilir. OpenCV bunların tümünü destekler, ancak varsayılan olarak 256 olur (OpenCV bunu bayt cinsinden gösterir. Dolayısıyla değerler 16, 32 ve 64 olacaktır). Bunu elde ettiğinizde, bu tanımlayıcıları eşleştirmek için Hamming Mesafesini kullanabilirsiniz.

Önemli bir nokta, BRIEF'in bir özellik tanımlayıcısı olması, özellikleri bulmak için herhangi bir yöntem sunmamasıdır. Bu nedenle, SIFT, SURF vb. Gibi diğer özellik dedektörlerini kullanmanız gerekecektir. Kağıt, hızlı bir dedektör olan CenSurE kullanılmasını önermektedir ve BRIEF, CenSurE noktaları için SURF noktalarından daha iyi çalışır.

Kısacası, BRIEF daha hızlı bir yöntem özelliği tanımlayıcı hesaplaması ve eşleştirmesidir. Aynı zamanda, büyük düzlem içi dönüş olmadığı sürece yüksek tanıma oranı sağlar.

OpenCV'de KISA

Aşağıdaki kod, CenSurE dedektörü yardımıyla BRIEF tanımlayıcılarının hesaplanması göstermektedir. (CenSurE dedektörüne OpenCV'de STAR dedektörü denir)

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

img = cv2.imread('simple.jpg',0)

# Initiate STAR detector
star = cv2.FeatureDetector_create("STAR")

# Initiate BRIEF extractor
brief = cv2.DescriptorExtractor_create("BRIEF")

# find the keypoints with STAR
kp = star.detect(img,None)

# compute the descriptors with BRIEF
kp, des = brief.compute(img, kp)

print brief.getInt('bytes')
print des.shape
```

`Brief.getInt ('bytes')` işlevi, baytlarda n_d kullanılan boyutu verir. Varsayılan olarak 32'dir. Bir sonraki eşleştirme, başka bir bölümde yapılacaktır.

5.8-)ORB (HIZLI ve Döndürülmüş KISA) Hedef

Bu bölümde,

- ORB'in temellerini göreceğiz

teori

OpenCV meraklısı olarak ORB ile ilgili en önemli şey "OpenCV Labs" den gelmesidir. Bu algoritma, **ORB: 2011'de SIFT veya SURF'a etkili bir alternatif olan Ethan Rublee, Vincent Rabaud, Kurt Konolige ve Gary R. Bradski tarafından gündeme getirildi . maliyet, uygun performans ve esas olarak patentler.** Evet, SIFT ve SURF patentlidir ve kullanıcıları için ödeme yapmanız gereklidir. Ama ORB değil !!!

ORB temel olarak, FAST anahtar noktası dedektörü ve BRIEF tanımlayıcısının performansı artırmak için birçok modifikasyonla birleşimdir. Önce kilit noktaları bulmak için HIZLI kullanın, sonra aralarında en iyi N noktalarını bulmak için Harris köşe ölçüsünü uygulayın. Ayrıca çok ölçekli özellikler üretmek için piramit kullanır. Ama bir problem şu ki, FAST yönelimi hesaplamıyor. Peki rotasyon değişmezliği ne olacak? Yazarlar aşağıdaki değişiklik ile geldi.

Merkezde bulunan köşe ile yamanın yoğunluk ağırlıklı sentroidini hesaplar. Vektörün bu köşe noktasından centroide yönü yönelimi verir. Dönme değişmezlik geliştirmek için, anlar yarıçaplı dairesel bir alanı olmalıdır x ve y ile hesaplanır T , T ama boyutudur.

Şimdi tanımlayıcılar için ORB, BRIEF tanımlayıcılarını kullanmaktadır. Ancak daha önce BRIEF'in rotasyon ile iyi performans göstermediğini gördük. ORB'nin yaptığı şey, kısa noktaları anahtar noktaların yönüne göre "yönlendirmek" tir. Konumda herhangi bir n ikili test özellik kümesi için, bu piksellerin koordinatlarını içeren (x_i, y_i) bir $2 \times n$ matris tanımlayın S . Daha sonra yamanın yönünü kullanarak, Θ dönme matrisi bulunur ve S yonlendirilmiş (döndürülmüş) versiyonu almak için döner S_θ .

ORB, açayı $2\pi/30$ (12 derecelik) artıslara ayırır ve önceden hesaplanmış BRIEF modellerinin bir arama tablosunu oluşturur. Keypoint yönlendirmesi Θ görünümler arasında tutarlı olduğu sürece S_θ , tanımlayıcısını hesaplamak için doğru nokta kümesi kullanılacaktır.

BRIEF, her bir bit özelliğinin büyük bir varyansa ve ortalama 0.5'e yakın olması önemli bir özelliğe sahiptir. Ancak kilit nokta yönünde yönlendirildikten sonra, bu özelliği kaybeder ve daha fazla dağıtır. Yüksek varyans özelliği girişlere farklı tepki verdiği için bir özelliği daha ayırmacı hale getirir. Bir başka arzu edilen özellik, testlerin ilişkisiz hale getirilmesidir, çünkü o zaman her test sonuca katkıda bulunacaktır. Tüm bunları çözmek için ORB, yüksek varyans ve 0.5'e yakın olan ve ilişkisiz olanları bulmak için tüm olası ikili testler arasında açgözlü bir arama yapar. Sonuç **rBRIEF** olarak adlandırılır .

Tanımlayıcı eşleştirme için, geleneksel LSH üzerinde gelişen çok prolu LSH kullanılır. Makalede ORB'in SURF'tan çok daha hızlı olduğu ve SIFT ve ORB tanımlayıcısının SURF'den daha iyi çalıştığı belirtiliyor. ORB, panorama dikişi vb. İçin düşük güçlü cihazlarda iyi bir seçimdir.

OpenCV'de ORB

Her zamanki gibi, `cv2.ORB()` işleviyle veya `feature2d` ortak arabirimini kullanarak bir ORB nesnesi oluşturmamız gereklidir. Çok sayıda isteğe bağlı parametre vardır. En yararlı olanlar **nözelikler** (varsayılan 500) tarafından muhafaza edilmesi için bir çok özellik sayısını belirtmektedir. `scoreType` Harris skoru olup hızlı vb başka parametre (varsayılan Harris skoru ile) özellikleri sıralamak için skor temsil eder, `WTA_K` nokta sayısını belirler. **yönlendirilmiş BRIEF tanımlayıcısının** her bir öğesini üreten Varsayılan olarak iki, yani bir seferde iki nokta seçer. Bu durumda, eşleştirme için `NORM_HAMMING` mesafesi kullanılır. `WTA_K`, BRIEF tanımlayıcı üretmek için 3 veya 4 puan alan 3 veya 4 ise, eşleşen mesafe şu şekilde tanımlanır: `NORM_HAMMING2`.

Aşağıda ORB kullanımını gösteren basit bir kod bulunmaktadır.

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

img = cv2.imread('simple.jpg',0)

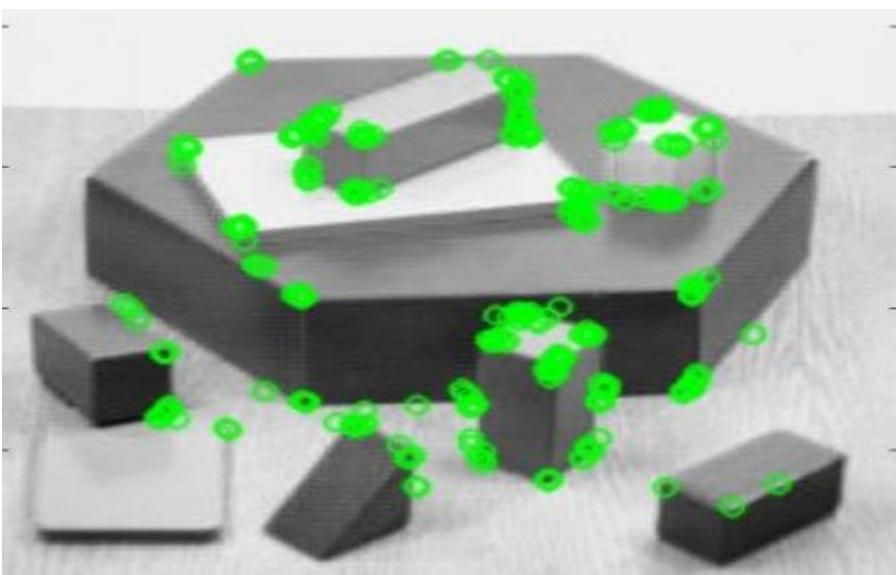
# Initiate STAR detector
orb = cv2.ORB()

# find the keypoints with ORB
kp = orb.detect(img,None)

# compute the descriptors with ORB
kp, des = orb.compute(img, kp)

# draw only keypoints location,not size and orientation
img2 = cv2.drawKeypoints(img,kp,color=(0,255,0), flags=0)
plt.imshow(img2),plt.show()
```

Aşağıdaki sonuca bakın:



ORB özellik eşleştirme, başka bir bölümde yapacağız.

5.9-)Özellik Eşleme Hedef

Bu bölümde

- Bir görüntüdeki özelliklerin diğerleriyle nasıl eşleştirileceğini göreceğiz.
- OpenCV'de Brute-Force eşleştiricisini ve FLANN Eşleştiricisini kullanacağız

Brute-Force Matcher'ın Temelleri

Brute-Force eşleştiricisi basittir. İlk kümedeki bir özelliğin tanımlayıcısını alır ve bazı mesafe hesaplamaları kullanılarak ikinci kümedeki diğer tüm özelliklerle eşleştirilir. Ve en yakın olanı geri döner.

BF eşleştiricisi için, önce `cv2.BFMatcher()` kullanarak **BFMatcher** nesnesini oluşturmanız gereklidir. İki isteğe bağlı parametre alır. Birincisi `normType`. Kullanılacak mesafe ölçümünü belirtir. Varsayılan olarak, `cv2.NORM_L2`'dır. SIFT, SURF vb. İçin iyidir (`cv2.NORM_L1` de oradadır). ORB, BRIEF, BRISK vb. Gibi ikili dize tabanlı tanımlayıcılar için Hamming mesafesini kullanır. `cv2.NORM_HAMMING` kullanılmalıdır. ORB `WTA_K == 3` veya `4` kullanıyorsa, `cv2.NORM_HAMMING2` kullanılmalıdır.

İkinci parametre, varsayılan olarak yanlış olan `crossCheck` olan boole değişkenidir. Bu doğruysa, Matcher yalnızca (i, j) değerine sahip eşleşmeleri A kümesindeki i-th tanımlayıcısının B kümesinde en iyi eşleşme olarak j-th tanımlayıcısına sahip olacağı şekilde döndürür. Yani, her iki setteki iki özellik birbirile eşleşmelidir. Tutarlı bir sonuç sağlar ve D.Lowe tarafından SIFT belgesinde önerilen oran testine iyi bir alternatiftir.

Oluşturulduktan sonra iki önemli yöntem `BFMatcher.match()` ve `BFMatcher.knnMatch()` yöntemidir. Birincisi en iyi eşleşmeyi döndürür. İkinci yöntem, k 'nın kullanıcı tarafından belirtildiği yerlerde k en iyi eşleşmelerini döndürür. Bununla ilgili ek çalışmalar yapmamız gereğinde yararlı olabilir.

Anahtar noktaları çizmek için `cv2.drawKeypoints()` kullandığımız gibi, `cv2.drawMatches()` da eşleşmeleri **çizmemize** yardımcı olur. İki görüntüyü yatay olarak istifler ve en iyi eşleşmeleri gösteren ilk görüntüden ikinci görüntüye çizgiler çizer. Ayrıca tüm k en iyi eşleşmeleri çizen `cv2.drawMatchesKnn` vardır. $K = 2$ ise, her bir kilit nokta için iki eşleme çizgisi çizer. Bu yüzden, seçmeli olarak çizmek istiyorsak bir maskeyi geçmeliyiz.

SURF ve ORB'nin her biri için bir örnek görelim (Her ikisi de farklı mesafe ölçümüleri kullanıyor).

ORB Tanımlayıcıları ile Kaba Kuvvet Eşleştirme

Burada, iki resim arasındaki özelliklerin nasıl eşleştirileceğine dair basit bir örnek göreceğiz. Bu durumda, bir `queryImage` ve bir `trainImage` var. Özellik eşleşmesini

kullanarak `trainImage` sorgusunu bulmaya çalışacağız. (Resimler `/samples/c/box.png` ve `/samples/c/box_in_scene.png`'dir)

Özellikleri eşleştirmek için SIFT tanımlayıcılarını kullanıyoruz. Şimdi görüntüleri yüklemek, tanımlayıcıları bulmak vb. ile başlayalım.

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

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
orb = cv2.ORB()

# find the keypoints and descriptors with SIFT
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)
```

Daha sonra mesafe ölçümü `cv2.NORM_HAMMING` (ORB kullandığımızdan beri) ile bir BFMatcher nesnesi oluşturuyoruz ve daha iyi sonuçlar için `crossCheck` açık. Sonra iki görüntüde en iyi eşleşmeleri elde etmek için `Matcher.match()` yöntemini kullanıyoruz. Onları mesafelerinin artan sırasına göre sıralarız, böylece en iyi eşleşmeler (düşük mesafeli) öne çıkar. Sonra sadece ilk 10 maçı çiziyoruz (Sadece görünürlük uğruna. İstediğiniz gibi artırabilirsiniz)

```
# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors.
matches = bf.match(des1,des2)

# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x:x.distance)

# Draw first 10 matches.
img3 = cv2.drawMatches(img1,kp1,img2,kp2,matches[:10], flags=2)

plt.imshow(img3),plt.show()
```

Aldığım sonuç aşağıda:



Bu Eşleştirme Nesnesi nedir?

Sonucu ile eşleşir = `bf.match (des1, des2)` hattı DMatch ait bir liste nesneleri olan. Bu DMatch nesnesi aşağıdaki özelliklere sahiptir:

- `DMatch.distance` – Tanımlayıcılar arasındaki mesafe. Ne kadar düşükse o kadar iyidir.
- `DMatch.trainIdx` – Tren tanımlayıcılarında tanımlayıcının dizini
- `DMatch.queryIdx` – Soru tanımlayıcılarındaki tanımlayıcının dizini
- `DMatch.imgIdx` – Tren görüntüsünün dizini.

SIFT Tanımlayıcıları ile Kaba Kuvvet Eşleştirme ve Oran Testi

Bu sefer en iyi maçları almak için `BFMatcher.knnMatch ()` yöntemini kullanacağız . Bu örnekte, makalesinde D.Lowe tarafından açıklanan oran testini uygulayabilmemiz için `k = 2` alacağız.

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

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)
```

```

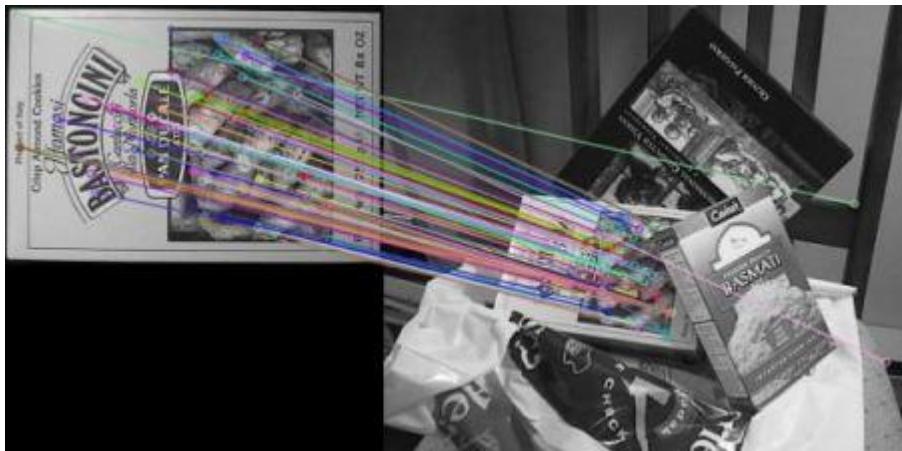
# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.75*n.distance:
        good.append([m])

# cv2.drawMatchesKnn expects list of lists as matches.
img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,good,flags=2)

plt.imshow(img3),plt.show()

```

Aşağıdaki sonuca bakın:



FLANN bazlı Eşleştirici

FLANN, En Yakın Komşular için Hızlı Kütüphane anlamına gelir. Büyük veri kümelerinde en yakın komşu arama ve yüksek boyutlu özellikler için optimize edilmiş bir algoritma koleksiyonu içerir. Büyük veri kümeleri için BFMatcher'dan daha hızlı çalışır. FLANN tabanlı eşleştiriciyle ikinci örneği göreceğiz.

FLANN tabanlı eşleştirici için, kullanılacak algoritmayı, ilgili parametreleri vb. Belirten iki sözlük geçmemiz gereklidir. Birincisi IndexParams. Çeşitli algoritmalar için aktarılacak bilgiler FLANN belgelerinde açıklanmaktadır. Özet olarak, SIFT, SURF vb algoritmalar için aşağıdakileri iletebilirsiniz:

```
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
```

ORB kullanırken aşağıdakileri iletebilirsiniz. Yorumlanan değerler dokümanlara göre önerilir, ancak bazı durumlarda gerekli sonuçları vermedi. Diğer değerler iyi çalıştı ..

```
index_params= dict(algorithm = FLANN_INDEX_LSH,
                    table_number = 6, # 12
                    key_size = 12,     # 20
                    multi_probe_level = 1) #2
```

İkinci sözlük SearchParams. Dizindeki ağaçların kaç kez yinelenmesi gerektiğini belirtir. Daha yüksek değerler daha iyi hassasiyet sağlar, ancak daha fazla zaman alır. Değeri değiştirmek istiyorsanız, search_params = dict (checks = 100) iletin .

Bu bilgilerle gitmeye hazırız.

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

img1 = cv2.imread('box.png',0)          # queryImage
img2 = cv2.imread('box_in_scene.png',0) # trainImage

# Initiate SIFT detector
sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN parameters
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)    # or pass empty dictionary

flann = cv2.FlannBasedMatcher(index_params,search_params)

matches = flann.knnMatch(des1,des2,k=2)

# Need to draw only good matches, so create a mask
matchesMask = [[0,0] for i in xrange(len(matches))]

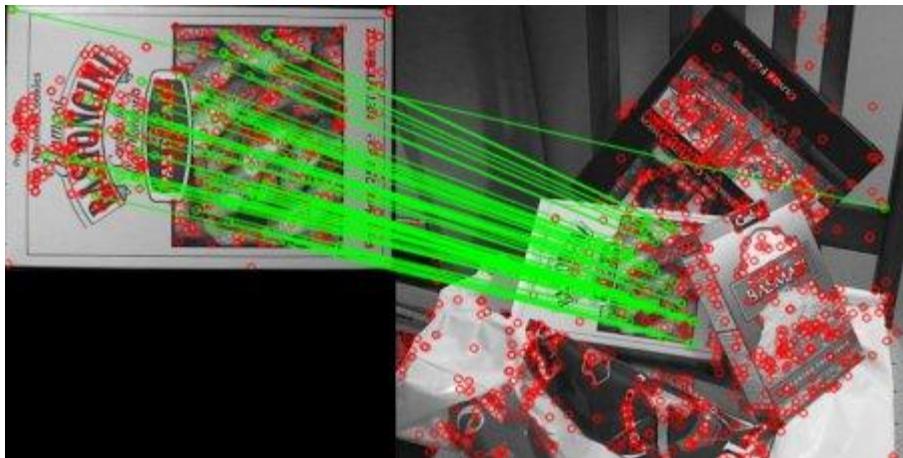
# ratio test as per Lowe's paper
for i,(m,n) in enumerate(matches):
    if m.distance < 0.7*n.distance:
        matchesMask[i]=[1,0]

draw_params = dict(matchColor = (0,255,0),
                   singlePointColor = (255,0,0),
                   matchesMask = matchesMask,
                   flags = 0)

img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,matches,None,**draw_params)

plt.imshow(img3,),plt.show()
```

Aşağıdaki sonuca bakın:



5.10-) Nesneleri bulmak için Eşleştirme + Homografi Özelliği Hedef

Bu bölümde,

- Karmaşık bir görüntüde bilinen nesneleri bulmak için calib3d modülünden özellik eşleştirme ve findHomography'yi karıştıracağız.

temeller

Son oturumda ne yaptık? Sorgu kullandık, içinde bazı özellik noktaları bulduk, başka bir tren aldıkResim, bu görüntüdeki özellikleri de bulduk ve aralarındaki en iyi eşleşmeleri bulduk. Kısacası, bir nesnenin bazı bölümlerinin başka bir dağınık görüntüdeki yerlerini bulduk. Bu bilgi, nesneyi tam olarak trainImage üzerinde bulmak için yeterlidir.

Bunun için calib3d modülünden bir işlev kullanabiliriz, yani **cv2.findHomography()**. Her iki görüntünün de nokta kümesini iletersek, o nesnenin perspektif dönüşümünü bulur. Sonra nesneyi bulmak için **cv2.perspectiveTransform()** yöntemini kullanabiliriz. Dönüşümü bulmak için en az dört doğru noktaya ihtiyacı vardır.

Eşleştirme sırasında sonucu etkileyebilecek bazı olası hatalar olabileceğini gördük. Bu sorunu çözmek için algoritma RANSAC veya LEAST_MEDIAN (bayraklar tarafından kararlaştırılabilir) kullanır. Dolayısıyla, doğru tahmin sağlayan iyi eşleşmelere aykırı değer, geriye kalan ise aykırı değer denir. **cv2.findHomography()**, iç ve dış noktaları belirten bir maske döndürür.

Öyleyse hadi yapalım !!!

kod

İlk olarak, her zamanki gibi, görüntülerde SIFT özelliklerini bulalım ve en iyi eşleşmeleri bulmak için oran testini uygulayalım.

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

MIN_MATCH_COUNT = 10
```

```



```

Şimdi, nesneyi bulmak için en az 10 eşleşmenin (MIN_MATCH_COUNT ile tanımlanan) olması için bir koşul belirledik. Aksi takdirde, yeterli eşleşme olmadığını belirten bir mesaj gösterin.

Yeterli eşleşme bulunursa, her iki görüntüdeki eşleşen anahtar noktalarının konumlarını çıkarırız. Perspektif dönüşümü bulmak için geçerler. Bu 3×3 dönüşüm matrisini elde ettiğimizde, queryImage'in köşelerini trainImage'daki karşılık gelen noktalara dönüştürmek için kullanırız. Sonra çiziyoruz.

```

if len(good)>MIN_MATCH_COUNT:
    src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
    dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)

    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
    matchesMask = mask.ravel().tolist()

    h,w = img1.shape
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv2.perspectiveTransform(pts,M)

    img2 = cv2.polylines(img2,[np.int32(dst)],True,255,3, cv2.LINE_AA)

else:
    print "Not enough matches are found - %d/%d" % (len(good),MIN_MATCH_COUNT)
    matchesMask = None

```

Son olarak, uçbirimlerimizi (nesneyi başarılı bir şekilde bulduysa) veya eşleşen tuş noktalarını (başarısız olursa) çizeriz.

```

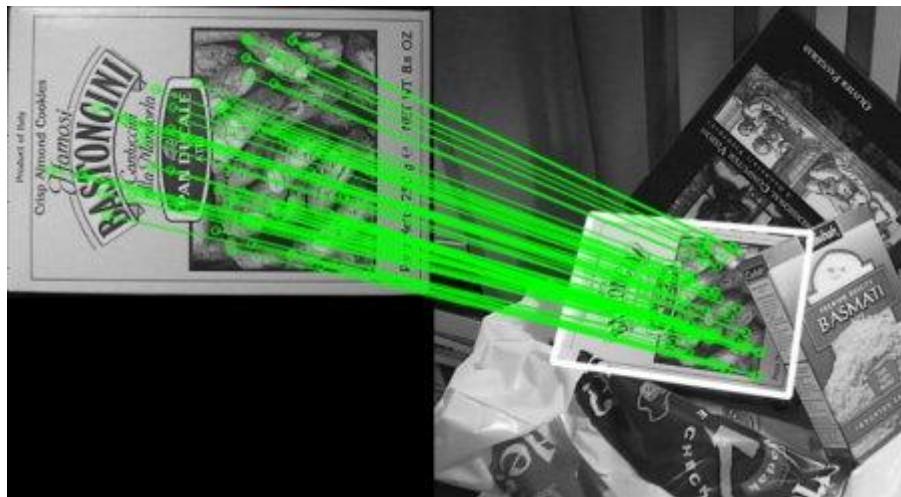
draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)

img3 = cv2.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)

plt.imshow(img3, 'gray'),plt.show()

```

Aşağıdaki sonuca bakın. Nesne dağınık görüntüde beyaz renkle işaretlenir:



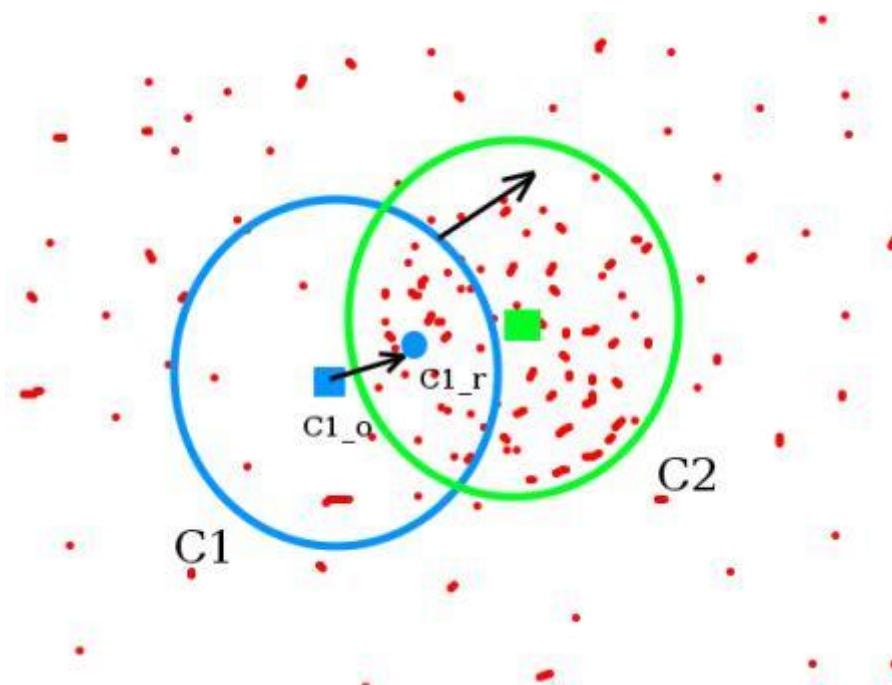
6.1-) Ortalama ve Camshift Hedef

Bu bölümde,

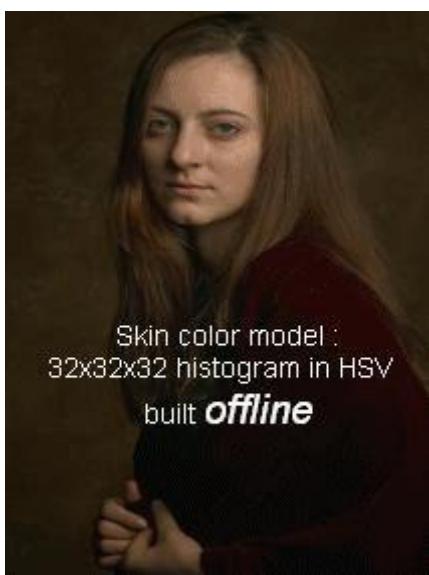
- Videolardaki nesneleri bulmak ve izlemek için Meanshift ve Camshift algoritmaları hakkında bilgi edineceğiz.

Meanshift

Ortalama değişiminin ardından sezgi basittir. Bir dizi noktanız olduğunu düşünün. (Histogram geri projeksiyonu gibi bir piksel dağılımı olabilir). Size küçük bir pencere verilir (daire olabilir) ve bu pencereyi maksimum piksel yoğunluğu (veya maksimum nokta sayısı) alanına taşımanız gereklidir. Aşağıda verilen basit resimde gösterilmiştir:



İlk pencere "C1" adıyla mavi daire şeklinde gösterilir. Orijinal merkezi "C1_o" adlı mavi dikdörtgenle işaretlenmiştir. Ancak bu pencerenin içindeki noktaların sentroidini bulursanız, pencerenin gerçek sentroidi olan "C1_r" (küçük mavi daire şeklinde) noktasını alırsınız. Elbette uyuşmuyorlar. Pencerenizi yeni pencerenin çemberi önceki centroid ile eşleşecek şekilde hareket ettirin. Yine yeni centroidi bul. Büyük olasılıkla eşleşmeyecektir. Bu yüzden tekrar hareket ettirin ve pencerenin merkezi ve sentroidi aynı konuma (veya istenen küçük bir hatayla) düşecek şekilde yinelemelere devam edin. Nihayet elde ettiğiniz maksimum piksel dağılımına sahip bir pencere. "C2" olarak adlandırılan yeşil bir daire ile işaretlenmiştir. Resimde görebileceğiniz gibi, maksimum nokta sayısı vardır. Tüm süreç aşağıdaki statik bir görüntüde gösterilmiştir:



Bu yüzden normalde histogramı geri yansıtılan görüntüyü ve başlangıç hedef konumunu geçiririz. Nesne hareket ettiğinde, hareket açıkça histogram geri yansıtılan görüntüye

yansıtılır. Sonuç olarak, araç kaydırma algoritması penceremizi maksimum yoğunlukta yeni konuma taşır.

OpenCV'de Ortalama Kaydırma

OpenCV'de araç kaydırma işlevini kullanmak için, önce hedefi ayarlamamız, histogramını bulmamız gereklidir, böylece araç kaydırmanın hesaplanması için her karedeki hedefi geri yansıtabiliriz. Pencerenin başlangıç konumunu da sağlamalıyız. Histogram için burada sadece Hue dikkate alınır. Ayrıca, düşük ışık nedeniyle yanlış değerlerden kaçınmak için, düşük ışık değerleri `cv2.inRange()` işlevi kullanılarak atılır.

```
import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# take first frame of the video
ret,frame = cap.read()

# setup initial location of window
r,h,c,w = 250,90,400,125 # simply hardcoded the values
track_window = (c,r,w,h)

# set up the ROI for tracking
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)),
np.array((180., 255., 255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# Setup the termination criteria, either 10 iteration or move by atleast 1 pt
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret ,frame = cap.read()

    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # apply meanshift to get the new location
        ret, track_window = cv2.meanShift(dst, track_window, term_crit)

        # Draw it on image
        x,y,w,h = track_window
        img2 = cv2.rectangle(frame, (x,y), (x+w,y+h), 255,2)
        cv2.imshow('img2',img2)

        k = cv2.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv2.imwrite(chr(k)+".jpg",img2)

    else:
```

```
break
```

```
cv2.destroyAllWindows()  
cap.release()
```

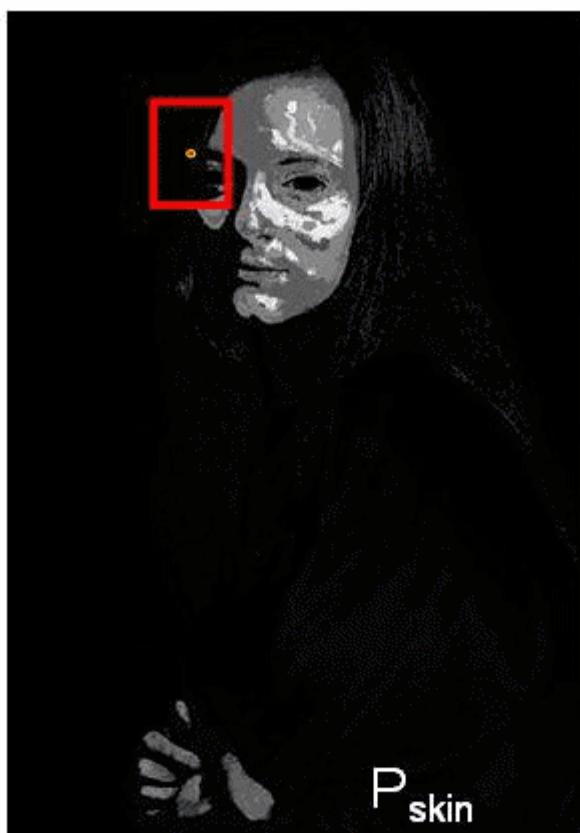
Three frames in a video I used is given below:



Camshift

Son sonucu yakından izledin mi? Bir sorun var. Araç uzaktayken ve kameraya çok yakın olduğunda penceremiz her zaman aynı boyuta sahiptir. Bu iyi değil. Pencere boyutunu hedefin boyutuna ve dönüşüne göre uyarlamamız gereklidir. Bir kez daha, çözüm “OpenCV Labs” den geldi ve buna 1988'de Gary Bradsky tarafından “Algısal Kullanıcı Arayüzünde Kullanım için Bilgisayarla Görme Yüz Takibi” adlı makalesinde yayınlanan CAMshift (Sürekli Adaptive MeanShift) deniyor.

Önce araç kaydırma uygular. MeanShift yakınsandığında, pencerenin boyutunu, olarak günceller $s = 2 \times \sqrt{\frac{M_{00}}{256}}$. Aynı zamanda ona en uygun elips yönünü de hesaplar. Yine, yeni kaydırılmış arama penceresi ve önceki pencere konumu ile ortalama kaydırmayı uygular. Gerekli hassasiyet karşılanması kadar işleme devam edilir.



Mean shift window
initialization

OpenCV'de Camshift

Ortalama kaydırma ile neredeyse aynıdır, ancak döndürülmüş bir dikdörtgen (sonucumuzdur) ve kutu parametrelerini (sonraki yinelemede arama penceresi olarak geçirilir) döndürür. Aşağıdaki koda bakın:

```
import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# take first frame of the video
ret,frame = cap.read()

# setup initial location of window
r,h,c,w = 250,90,400,125 # simply hardcoded the values
track_window = (c,r,w,h)

# set up the ROI for tracking
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)),
np.array((180., 255., 255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# Setup the termination criteria, either 10 iteration or move by atleast 1 pt
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
    ret ,frame = cap.read()

    if ret == True:
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

        # apply meanshift to get the new location
        ret, track_window = cv2.CamShift(dst, track_window, term_crit)

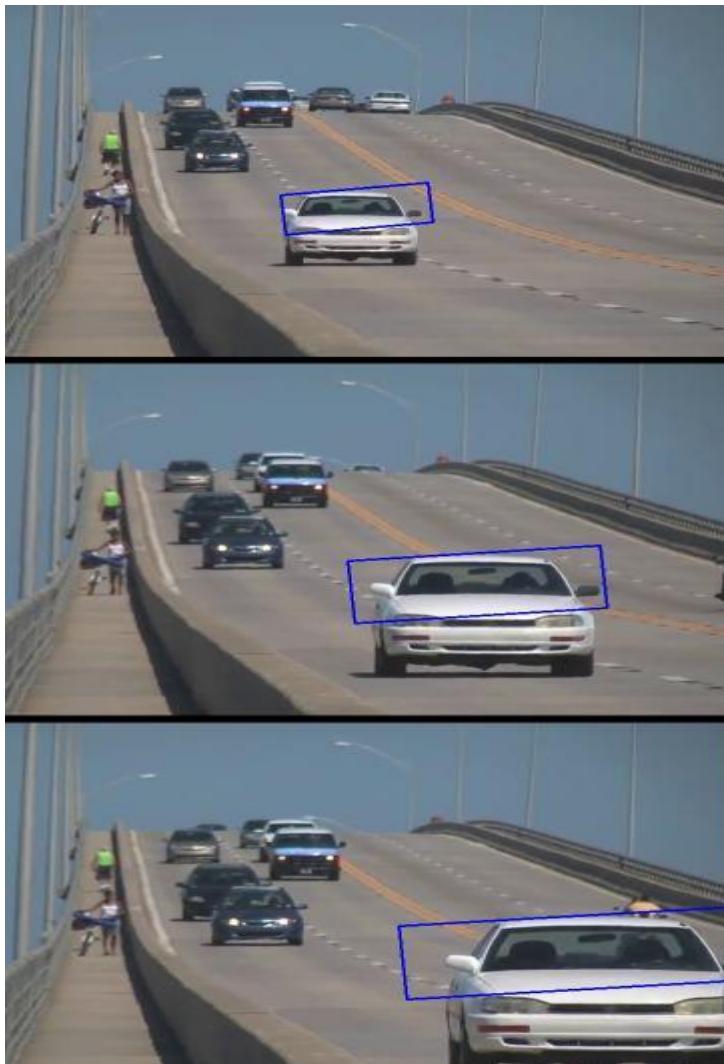
        # Draw it on image
        pts = cv2.boxPoints(ret)
        pts = np.int0(pts)
        img2 = cv2.polylines(frame,[pts],True, 255,2)
        cv2.imshow('img2',img2)

        k = cv2.waitKey(60) & 0xff
        if k == 27:
            break
        else:
            cv2.imwrite(chr(k)+".jpg",img2)

    else:
        break

cv2.destroyAllWindows()
cap.release()
```

Sonucun üç karesi aşağıda gösterilmiştir:



6.2-)Optik Akış

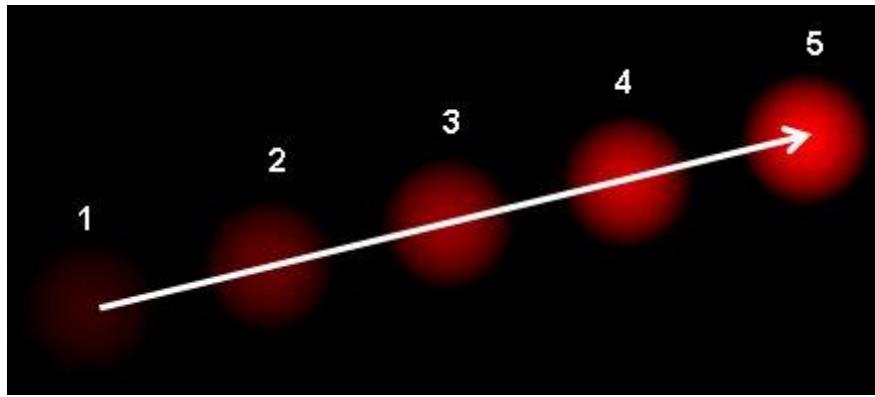
Hedef

Bu bölümde,

- Optik akış kavramlarını ve tahminini Lucas-Kanade yöntemini kullanarak anlayacağız.
- Bir videodaki özellik noktalarını izlemek için `cv2.calcOpticalFlowPyrLK()` gibi işlevleri kullanacağız .

Optik Akış

Optik akış, nesnenin veya kamerasının hareketinden kaynaklanan iki ardışık kare arasındaki görüntü nesnelerinin görünen hareket modelidir. Her vektörün, noktaların birinci çerçeveden ikinciye hareketini gösteren bir yer değiştirme vektörü olduğu 2D vektör alanıdır. Aşağıdaki görüntüyü düşünün (Görüntü Nezaket: [Optik Akış hakkında Wikipedia makalesi](#)).



Birbirini takip eden 5 karede hareket eden bir top gösterir. Ok, yer değiştirme vektörünü gösterir. Optik akış aşağıdaki gibi birçok uygulama alanına sahiptir:

- Hareketten Yapı
- Video sıkıştırma
- Video Sabitleme ...

Optik akış çeşitli varsayımlar üzerinde çalışır:

1. Bir nesnenin piksel yoğunlukları ardışık kareler arasında değişmez.
2. Komşu pikseller benzer harekete sahiptir.

$I(x, y, t)$ ilk karede bir piksel düşünün (Yeni bir boyut kontrol edin, zaman, buraya eklenir. Daha önce sadece görüntülerle çalışıyorduk, bu yüzden zamana gerek yok). Bir süre (dx, dy) sonra çekilen bir sonraki karede mesafeye göre hareket eder dt . Yani bu pikseller aynı olduğundan ve yoğunluk değişmediğinden,

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

Sonra sağ taraftaki Taylor serisi yaklaşımını alın, ortak terimleri kaldırın dt ve aşağıdaki denklemi elde etmek için bölün :

$$f_x u + f_y v + f_t = 0$$

nerede:

$$\begin{aligned} f_x &= \frac{\partial f}{\partial x}; \quad f_y = \frac{\partial f}{\partial y} \\ u &= \frac{dx}{dt}; \quad v = \frac{dy}{dt} \end{aligned}$$

Yukarıdaki denklem Optik Akış denklemi denir. İçinde bulabiliriz f_x ve f_y bunlar görüntü gradyanlarıdır. Benzer şekilde f_t zamanındaki gradyan. Ancak (u, v) bilinmiyor. Bu bir denklemi bilinmeyen iki değişkenle çözemeyiz. Bu sorunu çözmek için birkaç yöntem sağlandı ve bunlardan biri Lucas-Kanade.

Lucas–Kanade yöntemi

Daha önce komşu piksellerin benzer harekete sahip olacağı varsayımini gördük. Lucas–Kanade yöntemi nokta etrafında 3×3 'luk bir yama alır. Yani 9 noktanın hepsi aynı harekete sahip. (f_x, f_y, f_t) Bu 9 puanı bulabiliriz. Böylece şimdi problemimiz, aşırı belirlenmiş iki bilinmeyen değişkenle 9 denklemi çözüyor. En az kare oturtma yöntemi ile daha iyi bir çözüm elde edilir. Aşağıda, iki denklem–iki bilinmeyen problem olan ve çözümü elde etmek için çözülen son çözüm yer almaktadır.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

(Ters matrisin Harris köşe dedektörü ile benzerliğini kontrol edin. Köşelerin izlenecek daha iyi noktalar olduğunu gösterir.)

Yani kullanıcı açısından, fikir basittir, izlenecek bazı noktalar veririz, bu noktaların optik akış vektörlerini alırız. Ama yine bazı sorunlar var. Şimdiye kadar küçük hareketlerle uğraşıyorduk. Bu nedenle, büyük hareket olduğunda başarısız olur. Yine piramitlere gidiyoruz. Piramitte yukarı çıktığımızda, küçük hareketler çıkarılır ve büyük hareketler küçük hareketler haline gelir. O halde Lucas–Kanade'yi uygulayarak, skala ile birlikte optik akış elde ediyoruz.

OpenCV'de Lucas–Kanade Optik Akışı

OpenCV tüm bunları `cv2.calcOpticalFlowPyrLK()` gibi tek bir işlevde sağlar. Burada, bir videodaki bazı noktaları izleyen basit bir uygulama oluşturuyoruz. Noktalara karar vermek için `cv2.goodFeaturesToTrack()` kullanıyoruz. İlk kareyi alıyoruz, içinde bazı Shi-Tomasi köşe noktalarını tespit ediyoruz, sonra Lucas–Kanade optik akışını kullanarak bu noktaları tekrar tekrar izliyoruz. `Cv2.calcOpticalFlowPyrLK()` işlevi için önceki kareyi, önceki noktaları ve sonraki kareyi geçiririz. Bir sonraki nokta bulunursa 1 değerine sahip başka durum numaralarının yanı sıra sıfırdan sonraki noktaları döndürür. Bu sonraki noktaları tekrarlayarak bir sonraki adımda önceki puanlar olarak geçiyoruz. Aşağıdaki koda bakın:

```
import numpy as np
import cv2

cap = cv2.VideoCapture('slow.flv')

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parameters for Lucas kanade optical flow
lk_params = dict( winSize = (15,15),
                  maxLevel = 2,
```

```

        criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
10, 0.03))

# Create some random colors
color = np.random.randint(0,255,(100,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

while(1):
    ret,frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # calculate optical flow
    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)

    # Select good points
    good_new = p1[st==1]
    good_old = p0[st==1]

    # draw the tracks
    for i,(new,old) in enumerate(zip(good_new,good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        mask = cv2.line(mask, (a,b),(c,d), color[i].tolist(), 2)
        frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)
    img = cv2.add(frame,mask)

    cv2.imshow('frame',img)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

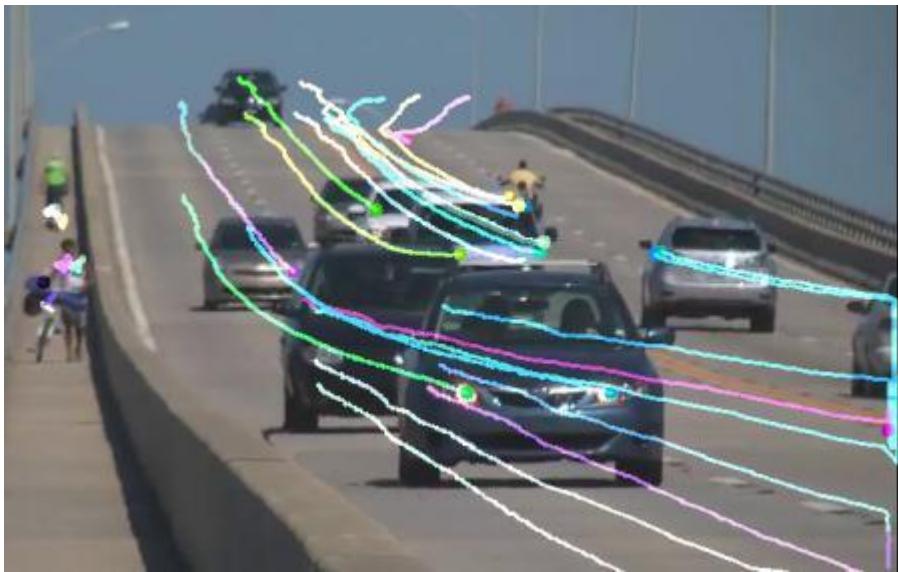
    # Now update the previous frame and previous points
    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)

cv2.destroyAllWindows()
cap.release()

```

(Bu kod, bir sonraki anahtar noktaların ne kadar doğru olduğunu kontrol etmez. Bu nedenle, görüntüde herhangi bir özellik noktası kaybolsa bile, optik akışın, ona yakın görünebilecek bir sonraki noktayı bulma şansı vardır. noktaların belirli aralıklarla tespit edilmesi gereklidir. OpenCV numuneleri, her 5 karede özellik noktalarını bulan bir örnekle gelir. Ayrıca, sadece iyi olanları seçmek zorunda olan optik noktalarının geriye doğru kontrolünü yapar. lk_track.py).

Elde ettiğimiz sonuçları görün:



OpenCV'de Yoğun Optik Akış

Lucas–Kanade yöntemi, seyrek bir özellik kümesi için optik akışı hesaplar (örneğimizde, Shi–Tomasi algoritması kullanılarak tespit edilen köşeler). OpenCV, yoğun optik akışı bulmak için başka bir algoritma sağlar. Çerçevegedeki tüm noktalar için optik akışı hesaplar. Gunner Farneback'in 2003 yılında Gunner Farneback tarafından "Polinom Genleşmesine Dayalı İki Çerçevevi Hareket Tahmini" bölümünde açıklanan algoritmasına dayanmaktadır.

Aşağıdaki örnek, yukarıdaki algoritmayı kullanarak yoğun optik akışın nasıl bulunacağını gösterir. Optik akış vektörleri ile 2 kanallı bir dizi elde ediyoruz (u, v) . Onların büyüklüğünü ve yönünü buluruz. Daha iyi görselleştirme için sonucu renklendiririz. Yön, görüntünün Ton değerine karşılık gelir. Büyüklük, Değer düzlemine karşılık gelir. Aşağıdaki koda bakın:

```
import cv2
import numpy as np
cap = cv2.VideoCapture("vtest.avi")

ret, frame1 = cap.read()
prvs = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame1)
hsv[..., 1] = 255

while(1):
    ret, frame2 = cap.read()
    next = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)

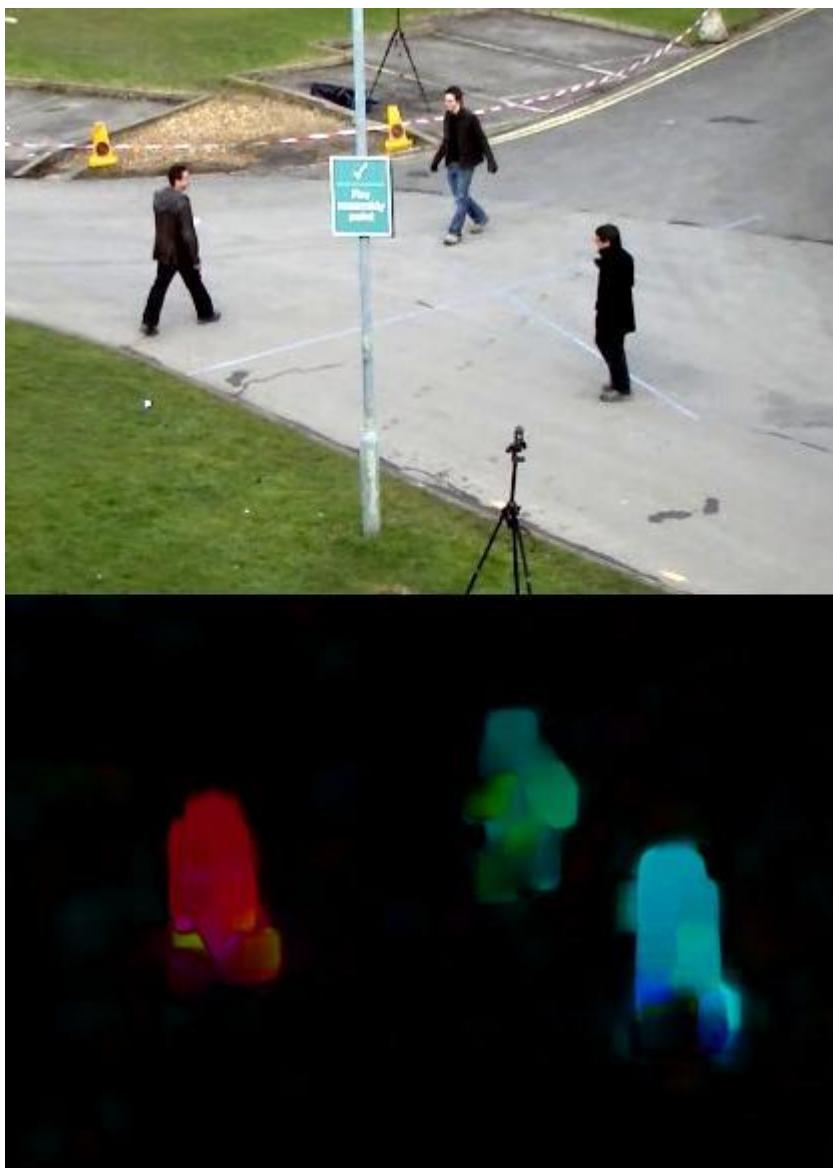
    flow = cv2.calcOpticalFlowFarneback(prvs, next, None, 0.5, 3, 15, 3, 5, 1.2,
0)

    mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])
    hsv[..., 0] = ang*180/np.pi/2
    hsv[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM_MINMAX)
    rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
```

```
cv2.imshow('frame2',rgb)
k = cv2.waitKey(30) & 0xff
if k == 27:
    break
elif k == ord('s'):
    cv2.imwrite('opticalfb.png',frame2)
    cv2.imwrite('opticalhsv.png',rgb)
prvs = next

cap.release()
cv2.destroyAllWindows()
```

Aşağıdaki sonuca bakın:



OpenCV, yoğun optik akışta daha gelişmiş bir örnekle birlikte gelir, lütfen / python2 / opt_flow.py örneklerine bakın .

6.3-)Arka Plan Çıkarma Hedef

Bu bölümde,

- OpenCV'de mevcut arka plan çıkarma yöntemlerini tanıyalacağız.

temeller

Arkaplan çıkartması, birçok vizyon tabanlı uygulamada önemli bir ön işleme aşamasıdır. Örneğin, statik bir kameranın odaya giren veya çıkan ziyaretçi sayısını aldığı ziyaretçi sayacı veya araçlar hakkında bilgi alan bir trafik kamerası vb. Durumları düşünün. Tüm bu durumlarda, önce kişiyi veya araçları tek başına çıkarmanız gereklidir. Teknik olarak, hareketli ön planı statik arka plandan çıkarmanız gereklidir.

Ziyaretçisiz odanın görüntüsü, araçsız yol görüntüsü vb. gibi tek başına bir arka plan görüntünüz varsa, bu kolay bir iştir. Yeni görüntüyü arka plandan çıkarın. Ön plandaki nesneleri tek başına elde edersiniz. Ancak çoğu durumda, böyle bir görüntünüz olmamaktadır, bu nedenle arka planı sahip olduğumuz görüntülerden çıkarmamız gereklidir. Araçların gölgesi olduğunda daha karmaşık hale gelir. Gölge de hareket ettiğinden, basit çıkışma bunu ön plan olarak da işaretler. İşleri zorlaştırmaktadır.

Bu amaçla çeşitli algoritmalar tanıtıldı. OpenCV, kullanımı çok kolay olan bu tür üç algoritma uygulamıştır. Onları tek tek göreceğiz.

BackgroundSubtractorMOG

Gauss Karışımı Tabanlı Arka Plan / Ön Plan Segmentasyon Algoritmasıdır. 2001 yılında P. KadewTraKuPong ve R. Bowden tarafından "Gölge tespiti ile gerçek zamanlı izleme için geliştirilmiş uyarlanabilir bir arka plan karışım modeli" başlıklı makalede tanıtıldı. Her bir arka plan pikselini K Gauss dağılımlarının ($K = 3$ ila 5). Karışımın ağırlıkları, bu renklerin sahnede kaldığı zaman oranlarını temsil eder. Muhtemel arka plan renkleri daha uzun ve daha statik olanlardır.

Kodlama yaparken, `cv2.createBackgroundSubtractorMOG()` işlevini kullanarak bir arka plan nesnesi oluşturmak gereklidir. Tarihin uzunluğu, gauss karışımlarının sayısı, eşik vb. gibi isteğe bağlı bazı parametreleri vardır. Hepsi bazı varsayılan değerlere ayarlanmıştır. Ardından video döngüsünün içinde, ön plan maskesini almak için `backgroundubtractor.apply()` yöntemini kullanın.

Aşağıdaki basit bir örneğe bakın:

```
import numpy as np
import cv2

cap = cv2.VideoCapture('vtest.avi')

fgbg = cv2.createBackgroundSubtractorMOG()

while(1):
```

```

ret, frame = cap.read()

fgmask = fgbg.apply(frame)

cv2.imshow('frame',fgmask)
k = cv2.waitKey(30) & 0xff
if k == 27:
    break

cap.release()
cv2.destroyAllWindows()

```

(Tüm sonuçlar karşılaştırma için sonunda gösterilir).

BackgroundSubtractorMOG

Aynı zamanda bir Gauss Karışımı Tabanlı Arka Plan / Ön Plan Segmentasyon Algoritmasıdır. 2004 yılında Z.Zivkovic'in "Arkaplan çıkartması için geliştirilmiş uyarlanabilir Gausian karışım modeli" ve 2006'da "Arkaplan Çıkarma Görevi için Görüntü Piksel Başına Verimli Uyarlama Yoğunluk Tahmini" adlı iki makaleye dayanmaktadır. Bu algoritmanın önemli bir özelliği her piksel için uygun sayıda gauss dağılımını seçer. (Son durumda, algoritma boyunca bir K gauss dağılımını aldı). Aydınlatma değişiklikleri vb. Nedeniyle değişen sahnelerde daha iyi uyum sağlar.

Önceki durumda olduğu gibi, bir arka plan çıkarıcı nesnesi oluşturmalıyız. Burada, gölgenin algılanıp algılanmayacağını seçme seçeneğiniz vardır. Eğer detectShadows = Doğru (varsayılan olarak böyledir), algıladığı ve işaretleri gölgeler, ancak hızını azaltır. Gölgeler gri renkle işaretlenir.

```

import numpy as np
import cv2

cap = cv2.VideoCapture('vtest.avi')

fgbg = cv2.createBackgroundSubtractorMOG2()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)

    cv2.imshow('frame',fgmask)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv2.destroyAllWindows()

```

(Sonunda verilen sonuçlar)

BackgroundSubtractorGMG

Bu algoritma, istatistiksel arka plan görüntüsü tahmini ile piksel başına Bayes bölümlendirmesini birleştirir. Andrew B. Godbehere, Akihiro Matsukawa, Ken Goldberg tarafından 2012'de "Duyarlı Bir Ses Sanatı Kurulumu için Değişken Aydınlatma Koşullarında İnsan Ziyaretçilerin Görsel Takibi" başlıklı makalesinde tanıtıldı. Makaleye göre sistem başarılı bir interaktif ses yayınladı "Henüz Orada Mıyız?" adlı sanat enstalasyonu 31 Mart – 31 Temmuz 2011 tarihleri arasında Kaliforniya, San Francisco'daki Çağdaş Yahudi Müzesi'nde.

Arka plan modelleme için ilk birkaç (varsayılan olarak 120) kare kullanır. Bayesian çıkarım kullanarak olası ön plan nesnelerini tanımlayan olasılıksal ön plan segmentasyon algoritması kullanır. Tahminler uyarlanabilir; daha yeni gözlemler, değişken aydınlatmaya uyum sağlamak için eski gözlemlerden daha ağırdır. İstenmeyen gürültüyü gidermek için kapatma ve açma gibi çeşitli morfolojik filtreleme işlemleri yapılır. İlk birkaç kare boyunca siyah bir pencere alacaksınız.

Gürültüyü gidermek için sonuca morfolojik açıklık uygulamak daha iyi olacaktır.

```
import numpy as np
import cv2

cap = cv2.VideoCapture('vtest.avi')

kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
fgbg = cv2.createBackgroundSubtractorGMG()

while(1):
    ret, frame = cap.read()

    fgmask = fgbg.apply(frame)
    fgmask = cv2.morphologyEx(fgmask, cv2.MORPH_OPEN, kernel)

    cv2.imshow('frame',fgmask)
    k = cv2.waitKey(30) & 0xff
    if k == 27:
        break

cap.release()
cv2.destroyAllWindows()
```

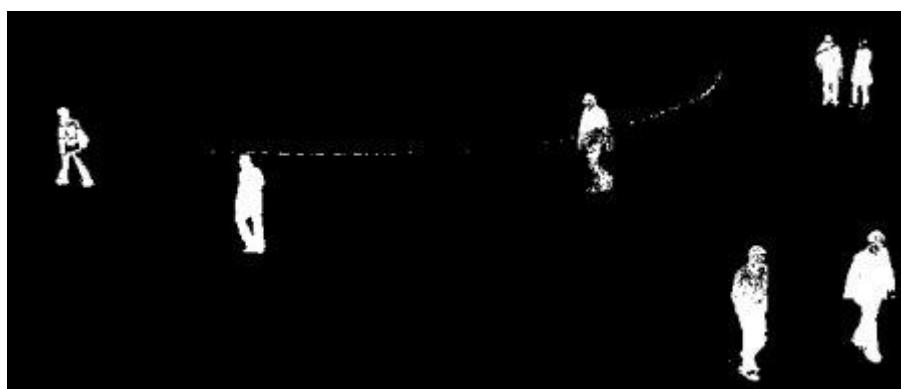
Sonuçlar

Orijinal Çerçeve

Aşağıdaki resim bir videonun 200. karesini göstermektedir



Arkaplan Sonucu



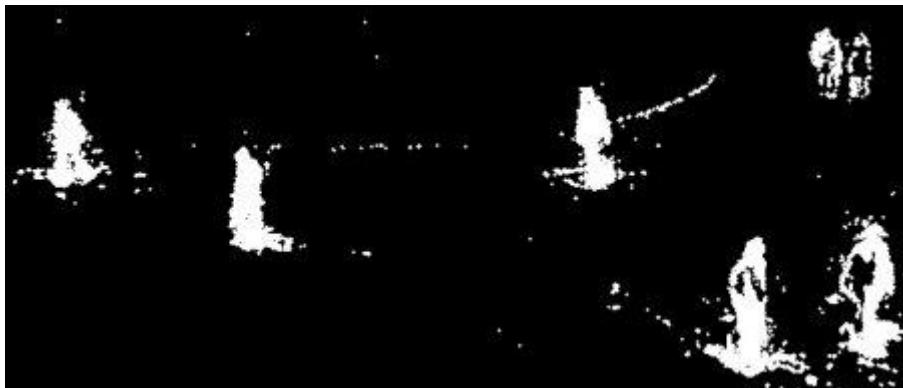
Arkaplan SonucuSubtractorMOG2

Gri renk bölgesi gölge bölgesini gösterir.



BackgroundSubtractorGMG Sonucu

Gürültü morfolojik açıklık ile giderilir.



7.1-)Kamera Kalibrasyonu Hedef

Bu bölümde,

- Kameradaki bozulmalar, kameranın iç ve dış parametreleri vb. Hakkında bilgi edineceğiz.
- Bu parametreleri, bozulmamış görüntüleri vb. Bulmayı öğreneceğiz.

temeller

Bugünün ucuz iğne deliği kameraları görüntülere çok fazla bozulma getiriyor. İki büyük bozulma radyal bozulma ve teğetsel bozulmadır.

Radyal bozulma nedeniyle düz çizgiler kavisli görünür. Görüntünün merkezinden uzaklaşıkça etkisi daha fazladır. Örneğin, bir satranç tahtasının iki kenarının kırmızı çizgilerle işaretlendiği bir görüntü aşağıda gösterilmiştir. Ancak kenarlığın düz bir çizgi olmadığını ve kırmızı çizgiyle eşleşmediğini görebilirsiniz. Beklenen tüm düz çizgiler şitti. Daha fazla bilgi için [Bozulma \(optik\) sayfasını](#) ziyaret edin .



Bu bozulma aşağıdaki gibi çözülmüştür:

$$x_{\text{corrected}} = x(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

$$y_{\text{corrected}} = y(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

Benzer şekilde, başka bir bozulma, görüntü alma merceği görüntüleme düzlemine mükemmel bir şekilde paralel hizalanmadığı için oluşan tegetsel bozulmadır. Bu nedenle görüntüdeki bazı alanlar beklenenden daha yakın görünebilir. Aşağıdaki gibi çözüldü:

$$x_{\text{corrected}} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

Kısaltısı, şu şekilde verilen bozulma katsayıları olarak bilinen beş parametre bulmamız gereklidir:

$$\text{Distortion coefficients} = (k_1 \ k_2 \ p_1 \ p_2 \ k_3)$$

Buna ek olarak, bir kameranın içsel ve dışsal parametreleri gibi birkaç bilgi daha bulmamız gerekiyor. Gerçek parametreler bir kameraya özgüdür. Odak uzaklığı (f_x, f_y), optik merkezler (c_x, c_y) vb. Bilgileri içerir. Kamera matrisi olarak da adlandırılır. Sadece kameraya bağlıdır, bu nedenle hesaplandıktan sonra gelecekteki amaçlar için saklanabilir. 3×3 'lük bir matris olarak ifade edilir:

$$\text{camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Dışsal parametreler, 3B bir noktanın koordinatlarını bir koordinat sisteme çeviren dönüş ve çeviri vektörlerine karşılık gelir.

Stereo uygulamalar için, önce bu bozulmaların düzeltilmesi gereklidir. Tüm bu parametreleri bulmak için, yapmamız gereken iyi tanımlanmış bir desenin (örn. Satranç tahtası) bazı örnek görüntülerini sağlamaktır. İçinde bazı belirli noktalar buluyoruz (satranç tahtasında kare köşeler). Gerçek dünya uzayındaki koordinatlarını ve görüntüdeki koordinatlarını biliyoruz. Bu verilerle, bozulma katsayılarını elde etmek için bazı matematiksel problemler arka planda çözülür. Bütün hikayenin özeti budur. Daha iyi sonuçlar için en az 10 test paternine ihtiyacımız var.

kod

Yukarıda belirtildiği gibi, kamera kalibrasyonu için en az 10 test desenine ihtiyacımız var. OpenCV, satranç tahtasının bazı görüntülerini ile birlikte gelir (bkz. [Samples / cpp / left01.jpg - left14.jpg](#)), bu yüzden kullanacağımız. Anlamak için, satranç tahtasının sadece bir görüntüsünü düşünün. Kamera kalibrasyonu için gerekli önemli giriş verileri, bir dizi 3B gerçek dünya noktası ve karşılık gelen 2B görüntü noktalarıdır. 2B görüntü noktaları iyidir, bu da görüntüden kolayca bulabiliriz. (Bu görüntü noktaları, satranç tahtalarında iki siyah karenin birbirine değdiği yerlerdir)

Gerçek dünya uzayından gelen 3B noktalar ne olacak? Bu görüntüler statik bir kameradan alınır ve satranç tahtaları farklı konumlara ve yönlere yerleştirilir. Bu yüzden (X, Y, Z) değerleri bilmemiz gerekiyor. Ancak basitlik için, satranç tahtasının XY düzleminde sabit tutulduğunu söyleyebiliriz (bu yüzden $Z = 0$ her zaman) ve kamera buna göre hareket ettirildi. Bu düşünce sadece X, Y değerlerini bulmamıza yardımcı olur. Şimdi X, Y değerleri için noktaları (0,0), (1,0), (2,0), ... olarak işaretleyebiliriz. Bu durumda, elde ettiğimiz sonuçlar satranç tahtası karesi boyutunda olacaktır. Ama eğer kare boyutunu bilersek, (30 mm diyelim) ve değerleri (0,0), (30,0), (60,0), ... olarak geçebilirsek, sonuçları mm cinsinden alırız. (Bu durumda, bu görüntülerini almadığımız için kare boyutunu bilmiyoruz, bu yüzden kare boyutu açısından geçiyoruz).

3B noktalara **nesne noktaları** ve 2B görüntü noktalarına görüntü noktaları denir.

Kurmak

Yani satranç tahtasında desen bulmak için **cv2.findChessboardCorners()** fonksiyonunu kullanıyoruz. 8x8 ızgara, 5x5 ızgara vb. gibi ne tür bir desen aradığımızı da geçmemiz gerekiyor. Bu örnekte 7x6 ızgara kullanıyoruz. (Normalde bir satranç tahtasında 8x8 kareler ve 7x7 iç köşeler bulunur). Köşe noktalarını döndürür ve desen elde edilirse True olur. Bu köşeler bir sırayla yerleştirilecektir (soldan sağa, yukarıdan aşağıya)

Ayrıca bakınız Bu işlev, tüm görüntülerde gerekli deseni bulamayabilir. Bu nedenle iyi bir seçenek, kodu kamerası başlatacak ve her kareyi gerekli desen için kontrol edecek şekilde yazmaktır. Desen elde edildiğinde, köşeleri bulun ve bir listede saklayın. Ayrıca satranç tahtasını farklı yöne ayarlayabilmemiz için bir sonraki kareyi okumadan önce bir miktar aralık sağlar. Gerekli sayıda iyi desen elde edilene kadar bu işleme devam edin. Burada verilen örnekte bile, verilen 14 görüntüden, kaçının iyi olduğundan emin değiliz. Böylece tüm görüntülerini okuyoruz ve iyi olanları alıyoruz.

Ayrıca bakınız Satranç tahtası yerine, bazı dairesel ızgaraları kullanabiliriz, ancak kalıbı bulmak için **cv2.findCirclesGrid()** işlevini kullanabiliriz. Dairesel ızgara kullanılırken daha az sayıda görüntünün yeterli olduğu söylenir.

Köşeleri bulduğumuzda, **cv2.cornerSubPix()** kullanarak doğruluklarını artırabiliriz. Deseni **cv2.drawChessboardCorners()** kullanarak da çizebiliriz. Tüm bu adımlar aşağıdaki koda dahil edilmiştir:

```
import numpy as np
import cv2
import glob

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)

# Arrays to store object points and image points from all the images.
```

```

objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)

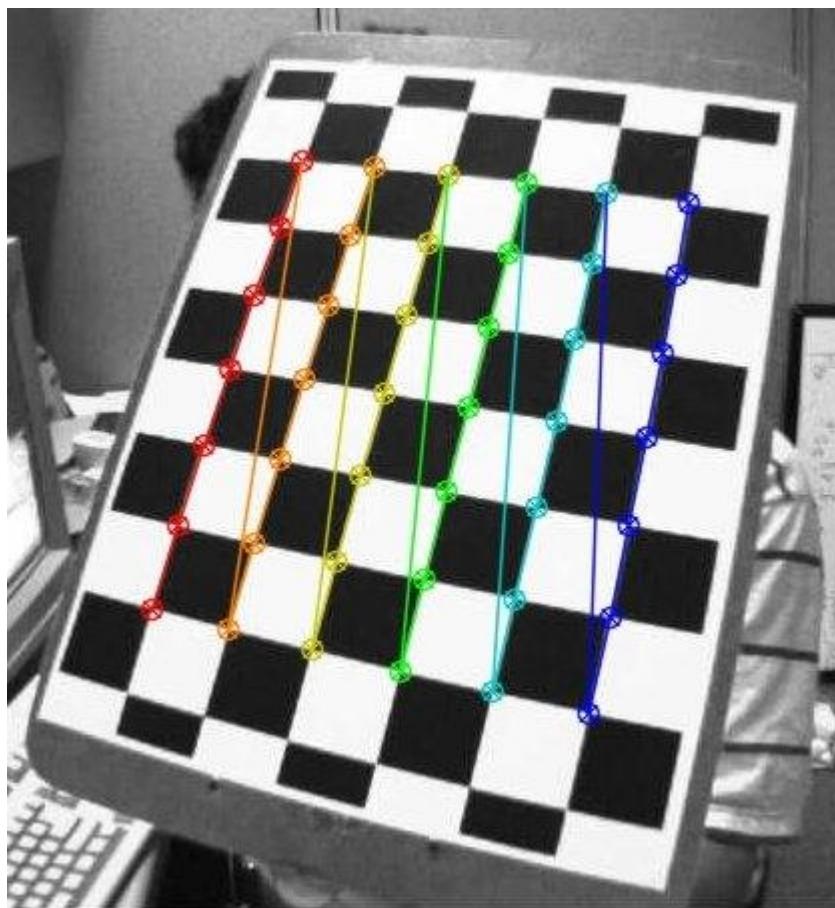
        cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners)

        # Draw and display the corners
        cv2.drawChessboardCorners(img, (7,6), corners2,ret)
        cv2.imshow('img',img)
        cv2.waitKey(500)

cv2.destroyAllWindows()

```

Üzerine desen çizilen bir görüntü aşağıda gösterilmiştir:



ayarlama

Şimdi nesne noktalarımız ve görüntü noktalarımız var, kalibrasyona hazırız. Bunun için **cv2.calibrateCamera()** işlevini kullanıyoruz. Kamera matrisini, bozulma katsayılarını, döndürme ve öteleme vektörlerini vb. Döndürür.

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,  
gray.shape[:: -1], None, None)
```

Undistortion

Denediğimiz şeyi aldık. Şimdi bir görüntü çekip bozabiliriz. OpenCV iki yöntemle geliyor, her ikisini de göreceğiz. Ancak bundan önce, **cv2.getOptimalNewCameraMatrix()** öğesini kullanarak ücretsiz bir ölçeklendirme parametresine dayalı olarak kamera matrisini hassaslaştırabiliriz. Ölçekleme parametresi `alpha = 0` ise, minimum istenmeyen piksellerle bozulmamış görüntüyü döndürür. Bu yüzden görüntü köşelerindeki bazı pikselleri bile kaldırabilir. Eğer `alfa = 1`, tüm pikseller bazı ekstra siyah görüntülerle korunur. Ayrıca sonucu kırmak için kullanılabilen bir görüntü YG'si de döndürür.

Bu yüzden yeni bir görüntü alıyoruz (`left12.jpg` bu durumda. Bu bölümdeki ilk görüntü budur)

```
img = cv2.imread('left12.jpg')  
h, w = img.shape[:2]  
newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))
```

1. **cv2.undistort()** öğesini kullanma

Bu en kısa yoldur. Sadece işlevi çağrıın ve sonucu elde etmek için yukarıda elde edilen ROI'yi kullanın.

```
# undistort  
dst = cv2.undistort(img, mtx, dist, None, newcameramtx)  
  
# crop the image  
x,y,w,h = roi  
dst = dst[y:y+h, x:x+w]  
cv2.imwrite('calibresult.png',dst)
```

2. Yeniden eşlemeyi kullanma

Bu kavisli yoldur. Önce bozuk görüntüden bozulmamış görüntüye bir eşleme işlevi bulun. Ardından yeniden eşleme işlevini kullanın.

```
# undistort  
mapx,mapy = cv2.initUndistortRectifyMap(mtx,dist,None,newcameramtx,(w,h),5)  
dst = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)
```

```
# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
cv2.imwrite('calibresult.png',dst)
```

Her iki yöntem de aynı sonucu verir. Aşağıdaki sonuca bakın:



Sonuçta tüm kenarların düz olduğunu görebilirsiniz.

Artık kamera matrisini ve bozulma katsayılarını ileride kullanmak üzere Numpy'de (np.savez, np.savetxt vb.) Yazma işlevlerini kullanarak saklayabilirsiniz.

Yeniden Yansıtma Hatası

Yeniden yansıtma hatası, bulunan parametrelerin ne kadar kesin olduğuna dair iyi bir tahmin verir. Bu mümkün olduğunda sıfır yakını olmalıdır. İçsel, bozulma, döndürme ve çeviri matrisleri göz önüne alındığında, önce **cv2.projectPoints ()** kullanarak nesne noktasını görüntü noktasına dönüştürüyoruz. Ardından, dönüşümümüzle elde ettiğimiz ve köşe bulma algoritması arasındaki mutlak normu hesaplıyoruz. Ortalama hatayı bulmak için tüm kalibrasyon görüntülerini için hesaplanan hataların aritmetik ortalamasını hesaplıyoruz.

```
mean_error = 0
for i in xrange(len(objpoints)):
    imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx,
    dist)
    error = cv2.norm(imgpoints[i],imgpoints2, cv2.NORM_L2)/len(imgpoints2)
    tot_error += error

print "total error: ", mean_error/len(objpoints)
```

7.2-)Poz Tahmini Hedef

Bu bölümde,

- Görüntülerde bazı 3D efektler oluşturmak için calib3d modülünden yararlanmayı öğreneceğiz.

temeller

Bu küçük bir bölüm olacak. Kamera kalibrasyonu ile ilgili son oturum sırasında kamera matrisini, bozulma katsayılarını vb. Bulduk. Bir kalıp görüntüsü verildiğinde, pozunu veya nesnenin nasıl döndürüldüğünü hesaplamak için yukarıdaki bilgileri kullanabiliriz, Düzlemsel bir nesne için, $Z = 0$ olduğunu varsayıyoruz, öyle ki, sorun şimdi kameranın desen imajımızı görmek için uzaya nasıl yerleştirildiği haline gelir. Dolayısıyla, nesnenin uzayda nasıl yattığını bilersek, 3D efektini simüle etmek için bazı 2D diyagramlar çizebiliriz. Nasıl yapılacağını görelim.

Bizim sorunumuz, 3D koordinat eksenimizi (X, Y, Z eksenleri) satranç tahtasının ilk köşesine çizmek istiyoruz. Mavi renkte X eksen, yeşil renkte Y eksen ve kırmızı renkte Z eksen. Sonuç olarak, Z eksen satranç tahtası uçağımıza dik gibi hissetmelidir.

İlk olarak, önceki kalibrasyon sonucundan kamera matrisini ve bozulma katsayılarını yükleyelim.

```
import cv2
import numpy as np
import glob

# Load previously saved data
with np.load('B.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]
```

Şimdi, bir işlev yapalım **çizmek** (kullanarak elde satranç tahtası köşeleri aldığı **cv2.findChessboardCorners()** ve **puan eksen** 3D eksen çizmek için.

```
def draw(img, corners, imgpts):
    corner = tuple(corners[0].ravel())
    img = cv2.line(img, corner, tuple(imgpts[0].ravel()), (255,0,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[1].ravel()), (0,255,0), 5)
    img = cv2.line(img, corner, tuple(imgpts[2].ravel()), (0,0,255), 5)
    return img
```

Sonra, önceki durumda olduğu gibi, sonlandırma kriterleri, nesne noktaları (satranç tahtasında köşelerin 3D noktaları) ve eksen noktaları yaratırız. Eksen noktaları, eksen çizmek için 3B alandaki noktalardır. Uzunluk 3 eksenini çiziyoruz (bu boyuta göre kalibre edildiğimizden beri birimler satranç kare boyutu cinsinden olacaktır). Yani X eksenimiz (0,0,0)'dan (3,0,0)' e çekilir, yani Y eksen için. Z eksen için (0,0,0) ila (0,0, -3) arasında çizilir. Negatif, kameraya doğru çekildiğini gösterir.

```

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
objp = np.zeros((6*7,3), np.float32)
objp[:, :, 2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)

axis = np.float32([[3, 0, 0], [0, 3, 0], [0, 0, -3]]).reshape(-1, 3)

```

Şimdi, her zamanki gibi, her görüntüyü yükliyoruz. 7×6 izgarayı arayın. Eğer bulunursa, alt köşe pikselleri ile hassaslaştırırız. Daha sonra döndürme ve çeviriyi hesaplamak için `cv2.solvePnP` () işlevini kullanırız. Biz bu dönüşüm matrislerini kullandığımızda, **eksen noktalarımızı** görüntü düzlemine yansıtmak için kullanırız. Basit bir ifadeyle, görüntü düzleminde 3B alanda (3,0,0), (0,3,0), (0,0,3) her birine karşılık gelen noktaları buluyoruz. Onları aldıktan sonra, `draw ()` fonksiyonumuzu kullanarak ilk köşeden bu noktaların her birine çizgiler çizeriz. Yapıldı !!!

```

for fname in glob.glob('left*.jpg'):
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (7, 6), None)

    if ret == True:
        corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)

        # Find the rotation and translation vectors.
        rvecs, tvecs, inliers = cv2.solvePnP(objp, corners2, mtx, dist)

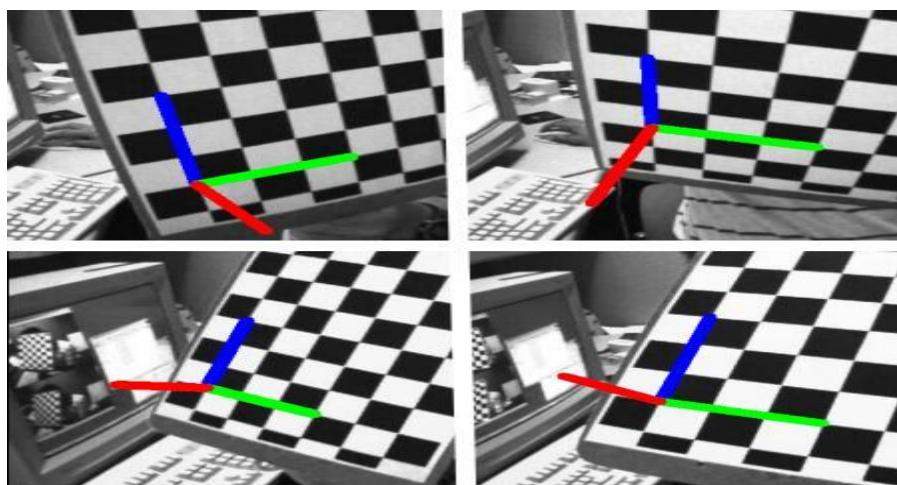
        # project 3D points to image plane
        imgpts, jac = cv2.projectPoints(axis, rvecs, tvecs, mtx, dist)

        img = draw(img, corners2, imgpts)
        cv2.imshow('img', img)
        k = cv2.waitKey(0) & 0xff
        if k == 's':
            cv2.imwrite(fname[:-4] + '.png', img)

cv2.destroyAllWindows()

```

Aşağıdaki bazı sonuçlara bakın. Her eksenin 3 kare uzunluğunda olduğuna dikkat edin.:



Küp Oluştur

Bir küp çizmek istiyorsanız, draw () işlevini ve eksen noktalarını aşağıdaki gibi değiştirin.

Değiştirilmiş draw () işlevi:

```
def draw(img, corners, imgpts):
    imgpts = np.int32(imgpts).reshape(-1,2)

    # draw ground floor in green
    img = cv2.drawContours(img, [imgpts[:4]], -1, (0,255,0), -3)

    # draw pillars in blue color
    for i,j in zip(range(4),range(4,8)):
        img = cv2.line(img, tuple(imgpts[i]), tuple(imgpts[j]), (255), 3)

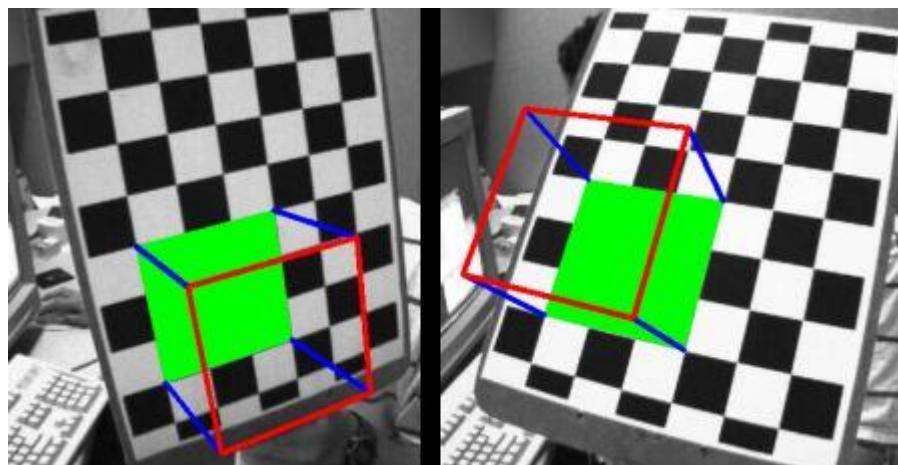
    # draw top layer in red color
    img = cv2.drawContours(img, [imgpts[4:]], -1, (0,0,255), 3)

    return img
```

Değiştirilmiş eksen noktaları. 3D uzayda bir küpün 8 köşesi:

```
axis = np.float32([[0,0,0], [0,3,0], [3,3,0], [3,0,0],
                   [0,0,-3],[0,3,-3],[3,3,-3],[3,0,-3] ])
```

Ve aşağıdaki sonuca bakın:



Grafikler, artırılmış gerçeklik vb. ile ilgileniyorsanız, daha karmaşık figürler oluşturmak için OpenGL'yi kullanabilirsiniz.

7.3-) Epipolar Geometri Hedef

Bu bölümde,

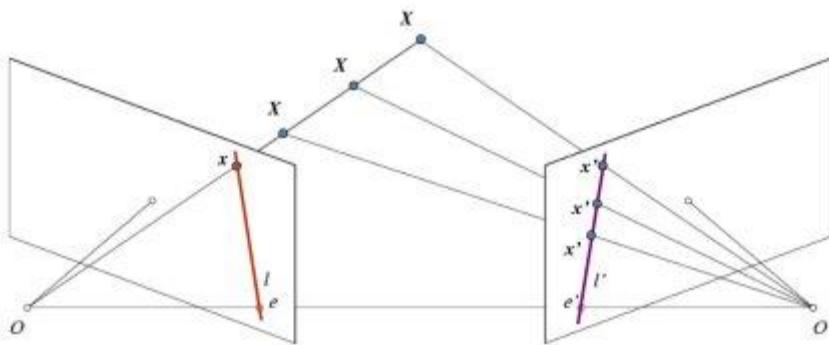
- Çoklu görüntü geometrisinin temellerini öğreneceğiz
- Epipol, epipolar çizgiler, epipolar kısıtlama vb.

Temel konseptler

İğne deliği kamera kullanarak bir görüntü çektiğimizde, önemli bir bilgiyi, yani görüntünün derinliğini kaybederiz. Ya da 3D'den 2D'ye dönüşüm olduğu için görüntüdeki her noktanın kameradan ne kadar uzak olduğu. Bu nedenle, bu kameraları kullanarak derinlik bilgisini bulabileceğimiz önemli bir sorudur. Ve cevap birden fazla kamera kullanmak. Gözlerimiz, stereo görüş adı verilen iki kamera (iki göz) kullandığımız şekilde çalışır. O halde OpenCV'nin bu alanda neler sağladığını görelim.

(Gary Bradsky tarafından *OpenCV öğrenmek* bu alanda birçok bilgiye sahiptir.)

Derinlikli görüntülere gitmeden önce, multiview geometrisindeki bazı temel kavramları anlayalım. Bu bölümde epipolar geometri ile ilgiliyoruz. Aynı sahnenin görüntüsünü çeken iki kamera ile temel bir kurulum gösteren aşağıdaki resme bakın.



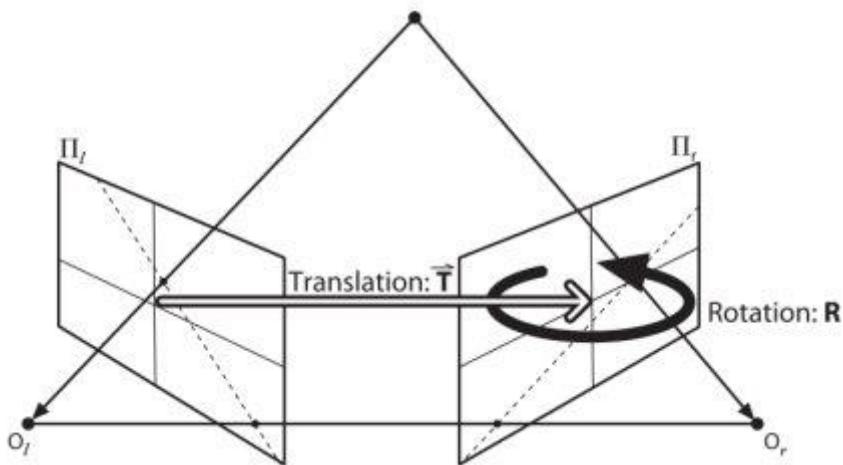
Yalnızca sol kamerayı kullanırsak, X görüntüdeki noktaya karşılık gelen 3B noktasını bulamayız çünkü çizgi üzerindeki her nokta OX görüntü düzleminde aynı noktaya yansır. Ancak doğru görüntüyü de düşünün. Şimdi çizgi üzerinde OX farklı noktalar x' doğru düzlemede farklı noktalara (e') yansıtıyor. Bu iki görüntü ile doğru 3B noktasını üçgenleştirebiliriz. Bütün fikir budur.

Farklı noktaların izdüşümü OX sağ düzlemede bir çizgi oluşturur (çizgi l'). Bu **noktaya** karşılık gelen **epilin** diyoruz X . X Doğu görüntüdeki noktayı bulmak için , bu epilin boyunca arama yapın. Bu çizgide bir yerde olmalı (Bu şekilde düşünün, başka bir görüntüdeki eşleşme noktasını bulmak için, tüm görüntüyü aramanıza gerek yoktur, sadece epilin boyunca arama yapmanız gereklidir). Böylece daha iyi performans ve doğruluk sağlar). Buna **Epipolar Kısıtlama** denir . Benzer şekilde, tüm noktaların diğer görüntüde karşılık gelen epilleri olacaktır. Düzlem XOO' denir **Epipolar Düzlem** .

O ve O' kamera merkezleridir. Yukarıda verilen kurulumdan, sağ kameranın izdüşümünün O' soldaki görüntüde görüldüğünü görebilirsiniz e . Buna **epipol** denir . Epipole çizginin kamera merkezleri ve görüntü düzlemleri ile kesiştiği noktadır. Benzer şekilde e' sol kameranın epipolü. Bazı durumlarda, epipolü görüntüde bulamazsınız, bunlar görüntünün dışında olabilir (yani bir kamera diğerini görmez).

Tüm epiller epipolundan geçer. Epipolun yerini bulmak için birçok epilin bulabilir ve kesişim noktalarını bulabiliriz.

Bu oturumda epipolar çizgiler ve epipoller bulmaya odaklanıyoruz. Ancak onları bulmak için iki **temel** bileşene daha ihtiyacımız var: **Temel Matris (F)** ve **Temel Matris (E)**. Temel Matris, küresel koordinatlarda ikinci kameranın yerini ilkine göre tanımlayan çeviri ve döndürme hakkında bilgi içerir. Aşağıdaki resme bakın (Resim nezaket: Gary Bradsky'nin OpenCV'yi Öğrenme):



Ancak ölçümlerin piksel koordinatlarında yapılmasını tercih ediyoruz, değil mi? Temel Matris, iki kamerası piksel koordinatlarında ilişkilendirebilmemiz için her iki kamerasının iç yapısı hakkındaki bilgilere ek olarak Temel Matris ile aynı bilgileri içerir. (Eğer rektifiye edilmiş görüntüler kullanıyorsanız ve odak uzaklıklarına bölünerek noktayı normalleştirirsek, $F = E$). Basit bir deyişle, Temel Matris F, bir görüntüdeki bir noktayı diğer görüntüdeki bir çizgiye (epilin) eşler. Bu, her iki resimdeki eşleşen noktalardan hesaplanır. Temel matrisi bulmak için en az 8 nokta gereklidir (8 noktalı algoritma kullanılırken). Daha fazla puan tercih edilir ve daha sağlam bir sonuç elde etmek için RANSAC kullanın.

kod

Bu yüzden önce temel matrisi bulmak için iki görüntü arasında mümkün olduğunda çok eşleşme bulmamız gerekiyor. Bunun için FLANN tabanlı eşleştirici ve oran testi ile SIFT tanımlayıcıları kullanıyoruz.

```

import cv2
import numpy as np
from matplotlib import pyplot as plt

img1 = cv2.imread('myleft.jpg',0) #queryimage # Left image
img2 = cv2.imread('myright.jpg',0) #trainimage # right image

sift = cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN parameters
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)

```

```

search_params = dict(checks=50)

flann = cv2.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)

good = []
pts1 = []
pts2 = []

# ratio test as per Lowe's paper
for i,(m,n) in enumerate(matches):
    if m.distance < 0.8*n.distance:
        good.append(m)
        pts2.append(kp2[m.trainIdx].pt)
        pts1.append(kp1[m.queryIdx].pt)

```

Şimdi her iki resimdeki en iyi eşleşmeler listesine sahibiz. Temel Matrisi bulalım.

```

pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
F, mask = cv2.findFundamentalMat(pts1,pts2,cv2.FM_LMEDS)

# We select only inlier points
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]

```

Sonra epilinler buluyoruz. İlk görüntüdeki noktalara karşılık gelen epiller ikinci görüntü üzerine çizilir. Bu yüzden burada doğru görüntülerden bahsetmek önemlidir. Bir dizi çizgi alırız. Bu nedenle, bu çizgileri görüntülere çizmek için yeni bir işlev tanımlarız.

```

def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines '''
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2

```

Şimdi her iki görüntüde de epilleri bulup çiziyoruz.

```

# Find epilines corresponding to points in right image (second image) and
# drawing its lines on left image
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)

# Find epilines corresponding to points in left image (first image) and

```

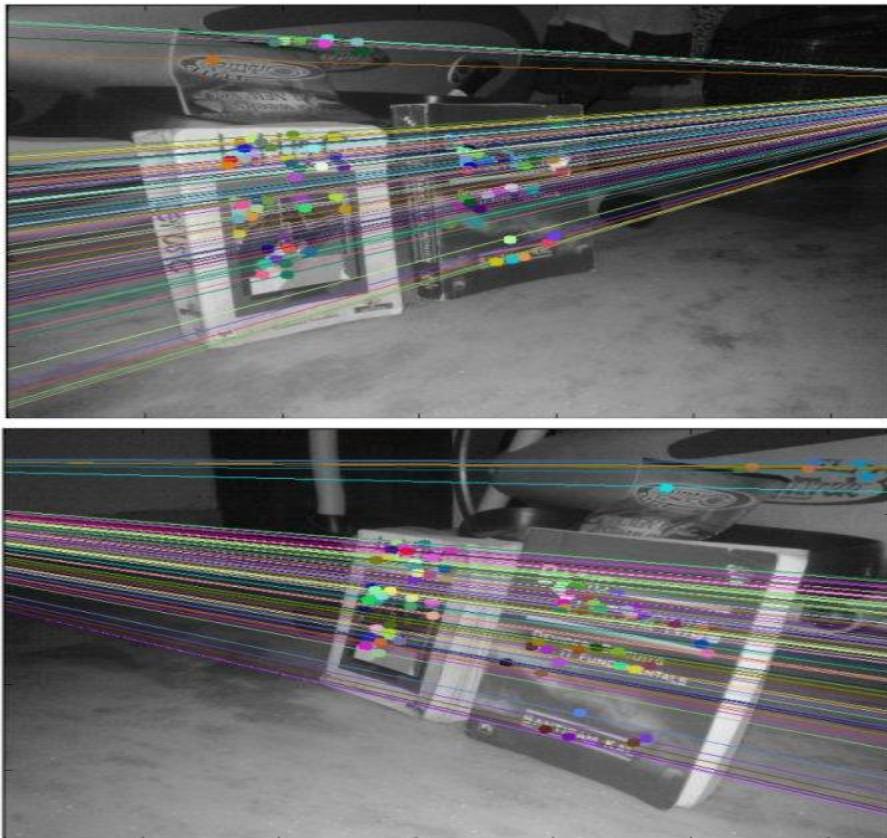
```

# drawing its lines on right image
lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)

plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
plt.show()

```

Aldığımız sonuç aşağıdadır:



Sol görüntüde tüm epillerin sağ taraftaki görüntünün dışında bir noktada birleştiğini görebilirsiniz. Bu buluşma noktası epipoldür.

Daha iyi sonuçlar için, iyi çözünürlüklü ve düzlemsel olmayan birçok noktaya sahip görüntüler kullanılmalıdır.

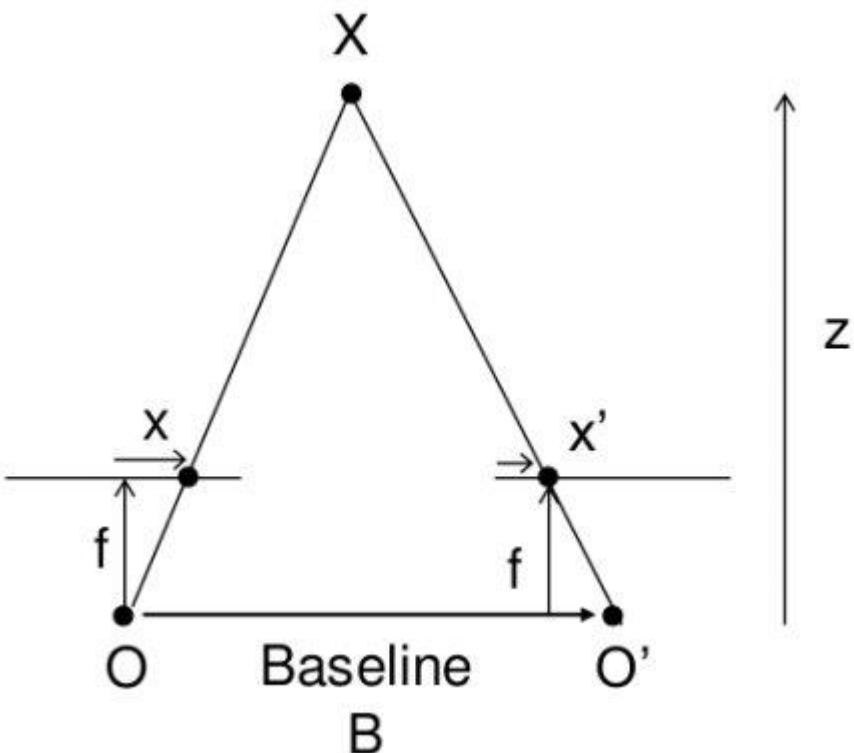
7.4-)Stereo Görüntülerden Derinlik Haritası Hedef

Bu oturumda,

- Stereo görüntülerden derinlik haritası oluşturmayı öğreneceğiz.

temeller

Son oturumda epipolar kısıtlamalar ve diğer ilgili terimler gibi temel kavramları gördük. Aynı sahnenin iki görüntüsüne sahipsek, sezgisel bir şekilde bundan derinlik bilgisi alabileceğimizi de gördük. Aşağıda bu sezgiyi kanıtlayan bir görüntü ve bazı basit matematiksel formüller bulunmaktadır. (Görünüm inceliği :



Yukarıdaki şemada eşdeğer üçgenler bulunmaktadır. Eşdeğer denklemlerini yazmak bize şu sonucu verecektir:

$$\text{disparity} = x - x' = \frac{Bf}{z}$$

x ve x' görüntü düzlemindeki sahne 3B noktasına karşılık gelen noktalar ile kamera merkezleri arasındaki mesafedir. B iki kamera arasındaki mesafe (bildiğimiz) ve f kameranın odak uzaklığı (zaten biliniyor). Kısacası, yukarıdaki denklem, bir sahnedeneki bir noktanın derinliğinin, karşılık gelen görüntü noktalarının ve kamera merkezlerinin mesafesindeki farkla ters orantılı olduğunu söylüyor. Böylece bu bilgilerle bir görüntüdeki tüm piksellerin derinliğini elde edebiliriz.

Böylece iki görüntü arasındaki karşılık gelen eşleşmeleri bulur. Epilin kısıtlamasının bu işlemi nasıl daha hızlı ve doğru hale getirdiğini daha önce görmüştük. Eşleşmeleri bulduktan sonra eşitsizliği bulur. Bakalım bunu OpenCV ile nasıl yapabiliriz.

kod

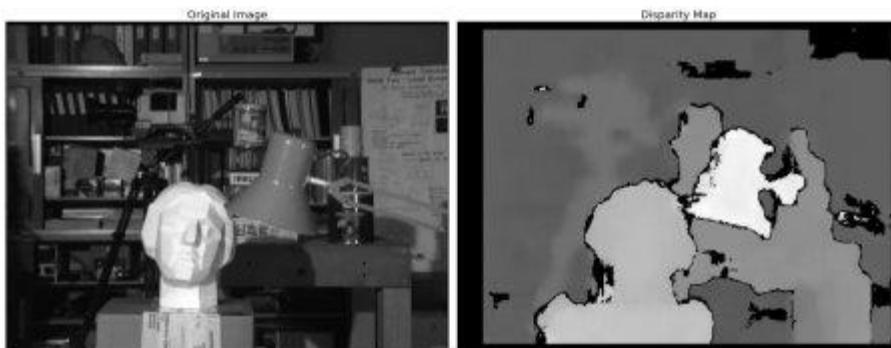
Aşağıdaki kod pasajı, eşitsizlik haritası oluşturmak için basit bir prosedür gösterir.

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

imgL = cv2.imread('tsukuba_l.png',0)
imgR = cv2.imread('tsukuba_r.png',0)

stereo = cv2.createStereoBM(numDisparities=16, blockSize=15)
disparity = stereo.compute(imgL,imgR)
plt.imshow(disparity,'gray')
plt.show()
```

Aşağıdaki görüntü orijinal görüntüyü (solda) ve eşitsizlik harmasını (sağda) içerir. Gördüğünüz gibi, sonuç yüksek derecede gürültü ile kirlenmiştir. NumDisparities ve blockSize değerlerini ayarlayarak daha iyi bir sonuç elde edebilirsiniz.



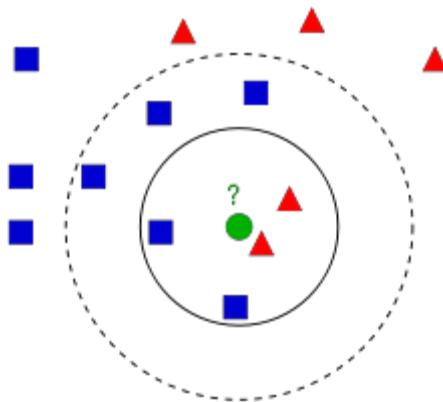
Not Eklenecek daha fazla ayrıntı

8.1.1-)K-En Yakın Komşuyu Anlamak Hedef

Bu bölümde k-En Yakın Komşu (kNN) algoritması kavramlarını anlayacağız.

teori

kNN, denetimli öğrenme için mevcut olan en basit sınıflandırma algoritmalarından biridir. Fikir, özellik alanındaki test verilerinin en yakın eşleşmesini aramaktır. Buna aşağıdaki görüntü ile bakacağız.



Resimde iki aile var: Mavi Kareler ve Kırmızı Üçgenler. Her aileye Sınıf diyoruz. Evleri, özellik alanı dediğimiz kasaba haritalarında gösteriliyor. (Bir özellik alanını tüm verilerin yansıtıldığı bir alan olarak düşünebilirsiniz. Örneğin, bir $2B$ koordinat alanı düşünün. Her verinin iki özelliği vardır, x ve y koordinatları. Bu verileri $2B$ koordinat alanınızda temsil edebilirsiniz, değil mi? üç özellik olup olmadığını hayal edin, $3B$ alana ihtiyacınız var. Şimdi N -boyutlu uzaya ihtiyacınız olan N özelliklerini düşünün, değil mi? Bu N -boyutlu alan onun özellik alanıdır. iki özellik).

Şimdi kasabaya yeni bir üye geliyor ve yeşil daire olarak gösterilen yeni bir ev yaratıyor. Bu Mavi / Kırmızı ailelerinden birine eklenmelidir. Bu sürece Sınıflandırma diyoruz. Ne yapıyoruz? KNN ile uğraştığımızdan, bu algoritmayı uygulayalım.

Bir yöntem, en yakın komşusunun kim olduğunu kontrol etmektir. Görüntüden, Kırmızı Üçgen ailesi olduğu açıktır. Böylece Kırmızı Üçgene eklenir. Bu yönteme basitçe **En Yakın Komşu** denir, çünkü sınıflandırma sadece en yakın komşuya bağlıdır.

Ama bununla ilgili bir sorun var. Kırmızı Üçgen en yakın olabilir. Ama ya yanında çok fazla Mavi Kareler varsa? O zaman Mavi Kareler bu bölgede Kırmızı Üçgenden daha fazla güce sahiptir. Bu yüzden sadece en yakın olanı kontrol etmek yeterli değildir. Bunun yerine bazı kontrol k aileleri en yakın. O zaman kim çoğunlukta olursa, yeni adam o aileye aittir. İmajımıza göre, $k = 3$, yani en yakın 3 aileyi ele alalım. İki Kırmızı ve bir Mavi var (iki Blues eşdeğeri var, ancak $k = 3$ olduğundan sadece bir tanesini alıyoruz), bu yüzden yine Red ailesine eklenmelidir. Peki ya $k = 7$ alırsak? Sonra 5 Mavi ailesi ve 2 Kırmızı ailesi var. Harika!! Şimdi Blue ailesine eklenmeli. Böylece her şey k değeriyle değişir. Daha komik olan şey, $k = 4$ ise? 2 Kırmızı ve 2 Mavi komşusu var. Bir kravat!!! Öyleyse k 'yi tek bir sayı olarak alsak iyi olur. Sınıflandırma en yakın komşulara bağlı olduğundan, bu yönteme **k -En Yakın Komşu** denir.

Yine, kNN'de k komşularını düşündüğümüz doğrudur, ama herkese eşit önem veriyoruz, değil mi? Adalet mi? Örneğin, $k = 4$ durumunu ele alalım. Beraberlik olduğunu söyleyelim. Ama bakın, 2 Kırmızı aile ona diğer 2 Mavi aileden daha yakın. Bu yüzden Red'e eklenmeye daha uygundur. Peki bunu matematiksel olarak nasıl açıklarız? Yeni gelenlere mesafelerine bağlı olarak her aileye biraz ağırlık veriyoruz. Ona yakın olanlar daha yüksek ağırlıklar alırken, onlar uzaktayken daha düşük ağırlıklar alırlar. Sonra her ailenin toplam

ağırlıklarını ayrı ayrı ekliyoruz. Kim en fazla toplam ağırlık alırsa, yeni gelen bu aileye gider. Buna **değiştirilmiş kNN** denir .

Burada gördüğünüz bazı önemli şeyler neler?

- Kasabadaki tüm evler hakkında bilgi sahibi olmalısın, değil mi? Çünkü en yakın komşuyu bulmak için yeni gelen ile mevcut tüm evlere olan mesafeyi kontrol etmeliyiz. Çok sayıda ev ve aile varsa, çok fazla hafıza ve hesaplama için daha fazla zaman alır.
- Her türlü eğitim veya hazırlık için neredeyse sıfır zaman vardır.

Şimdi OpenCV'de görelim.

OpenCV'de kNN

Burada, típkı yukarıdaki gibi iki aile (sınıf) ile basit bir örnek yapacağız. Sonra bir sonraki bölümde daha iyi bir örnek vereceğiz.

Burada Kırmızı aileyi **Sınıf-0** (yani 0 ile gösterilir) ve Mavi aileyi **Sınıf-1** (1 ile gösterilir) olarak etiketliyoruz . 25 aile veya 25 eğitim verisi oluşturuyoruz ve bunları Sınıf-0 veya Sınıf-1 olarak etiketliyoruz. Tüm bunları Numpy'deki Random Number Generator yardımıyla yapıyoruz.

Sonra Matplotlib'in yardımıyla planlıyoruz. Kırmızı aileler Kırmızı Üçgenler ve Mavi aileler Mavi Kareler olarak gösterilir.

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

# Feature set containing (x,y) values of 25 known/training data
trainData = np.random.randint(0,100,(25,2)).astype(np.float32)

# Labels each one either Red or Blue with numbers 0 and 1
responses = np.random.randint(0,2,(25,1)).astype(np.float32)

# Take Red families and plot them
red = trainData[responses.ravel()==0]
plt.scatter(red[:,0],red[:,1],80,'r','^')

# Take Blue families and plot them
blue = trainData[responses.ravel()==1]
plt.scatter(blue[:,0],blue[:,1],80,'b','s')

plt.show()
```

İlk imajımıza benzer bir şey elde edeceksiniz. Rastgele sayı üretici kullandığınız için, kodu her çalıştırıldığınızda farklı veriler elde edersiniz.

Daha sonra kNN algoritmasını başlatın ve kNN'yi eğitmek için *trainData* ve *yanıtları iletin* (Bir arama ağıacı oluşturur).

Sonra bir yeni gelen getireceğiz ve onu OpenCV'deki kNN yardımıyla bir aileye sınıflandıracağız. KNN'ye gitmeden önce, test verilerimiz hakkında bir şeyler bilmemiz gerekiyor (yeni gelenlerin verileri). Verilerimiz, kayan noktalı bir dizi olmalıdır **number of testdata × number of features**. Sonra yeni gelenlerin en yakın komşularını buluruz. Kaç tane komşu istediğimizi belirtebiliriz. Döndürür:

1. Daha önce gördüğümüz kNN teorisine bağlı olarak yeni gelenlere verilen etiket. En Yakın Komşu algoritmasını istiyorsanız, $k = 1$ belirtin, burada k komşu sayısıdır.
2. K-En Yakın Komşuların etiketleri.
3. Yeni gelenlerden en yakın komşulara karşılık gelen mesafeler.

Şimdi nasıl çalıştığını görelim. Yeni gelen yeşil renkle işaretlenmiştir.

```
newcomer = np.random.randint(0,100,(1,2)).astype(np.float32)
plt.scatter(newcomer[:,0],newcomer[:,1],80,'g','o')

knn = cv2.KNearest()
knn.train(trainData,responses)
ret, results, neighbours ,dist = knn.find_nearest(newcomer, 3)

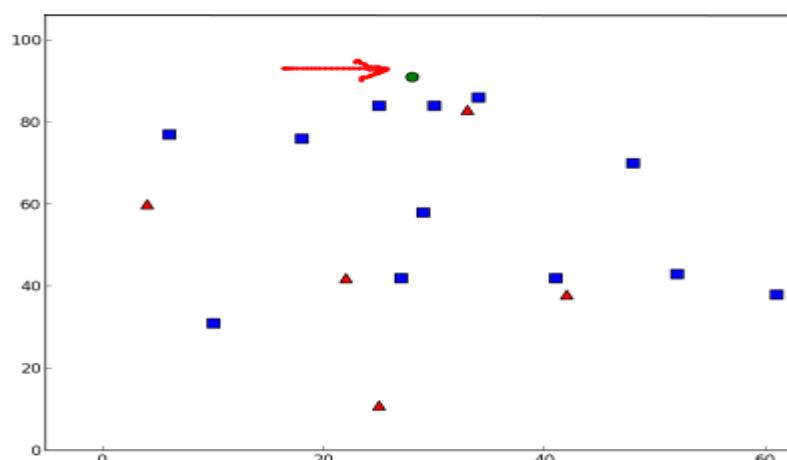
print "result: ", results,"\n"
print "neighbours: ", neighbours,"\n"
print "distance: ", dist

plt.show()
```

Sonucu aşağıdaki gibi aldım:

```
result: [[ 1.]]
neighbours: [[ 1.  1.  1.]]
distance: [[ 53.  58.  61.]]
```

Yeni gelenlerimizin hepsinin Blue ailesinden 3 komşusu olduğunu söylüyor. Bu nedenle, Blue ailesi olarak etiketlenmiştir. Aşağıdaki tablodan açıkça görülmektedir:



Çok sayıda veriniz varsa, bunları dizi olarak iletebilirsiniz. Diziler olarak da karşılık gelen sonuçlar elde edilir.

```
# 10 new comers
newcomers = np.random.randint(0,100,(10,2)).astype(np.float32)
ret, results,neighbours,dist = knn.find_nearest(newcomer, 3)
# The results also will contain 10 labels.
```

7.1.2-)KNN kullanarak El Yazısı Verilerinin OCR'sı Hedef

Bu bölümde

- Temel bir OCR uygulaması oluşturmak için kNN hakkında bilgilerimizi kullanacağımız.
- OpenCV ile birlikte gelen Rakamlar ve Alfabeler verileri ile deneyeceğiz.

Elle Yazılmış Rakamların OCR'sı

Amacımız el yazısı rakamları okuyabilen bir uygulama oluşturmaktır. Bunun için biraz train_data ve test_data'ya ihtiyacımız var. OpenCV, 5000 el yazısı basamağı (her basamak için 500) olan bir görüntü *digits.png* (opencv / samples / python2 / data / klasöründe) ile birlikte gelir. Her basamak 20x20 görüntündür. Yani ilk adımıız bu görüntüyü 5000 farklı basamağa bölmektir. Her basamak için 400 piksel ile tek bir satırda düzlestiriyoruz. Bu özellik setimiz, yani tüm piksellerin yoğunluk değerleri. Yaratabileceğimiz en basit özellik kümesidir. Her basamağın ilk 250 örneğini train_data, sonraki 250 örneğini test_data olarak kullanıyoruz. Önce onları hazırlayalım.

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

img = cv2.imread('digits.png')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Now we split the image to 5000 cells, each 20x20 size
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]

# Make it into a Numpy array. It size will be (50,100,20,20)
X = np.array(cells)

# Now we prepare train_data and test_data.
train = X[:,:50].reshape(-1,400).astype(np.float32) # Size = (2500,400)
test = X[:,50:100].reshape(-1,400).astype(np.float32) # Size = (2500,400)

# Create labels for train and test data
k = np.arange(10)
train_labels = np.repeat(k,250)[:,np.newaxis]
test_labels = train_labels.copy()

# Initiate kNN, train the data, then test it with test data for k=1
knn = cv2.KNearest()
knn.train(train,train_labels)
```

```

ret,result,neighbours,dist = knn.find_nearest(test,k=5)

# Now we check the accuracy of classification
# For that, compare the result with test_labels and check which are wrong
matches = result==test_labels
correct = np.count_nonzero(matches)
accuracy = correct*100.0/result.size
print accuracy

```

Yani temel OCR uygulamamız hazır. Bu özel örnek bana% 91 doğruluk verdi. Doğruluğu artıran seçeneklerden biri, özellikle yanlış olanlar olmak üzere eğitim için daha fazla veri eklemektir. Uygulamayı her başlattığında bu eğitim verilerini bulmak yerine, onu kaydetmem daha iyi olur, böylece bir dahaki sefere, bu verileri doğrudan bir dosyadan okurum ve sınıflandırmaya başlarım. Bunu np.savetxt, np.savez, np.load vb. Gibi bazı Numpy işlevlerinin yardımıyla yapabilirsiniz. Daha fazla bilgi için lütfen belgelerini kontrol edin.

```

# save the data
np.savez('knn_data.npz',train=train, train_labels=train_labels)

# Now Load the data
with np.load('knn_data.npz') as data:
    print data.files
    train = data['train']
    train_labels = data['train_labels']

```

Sistemimde yaklaşık 4.4 MB bellek gerekiyor. Yoğunluk değerlerini (uint8 verileri) özellik olarak kullandığımız için, önce verileri np.uint8'e dönüştürmek ve sonra kaydetmek daha iyi olur. Bu durumda sadece 1,1 MB alır. Ardından yükleme sırasında float32'ye dönüştürebilirsiniz.

OCR İngilizce Alfabeler

Daha sonra İngilizce alfabe için de aynısını yapacağız, ancak veri ve özellik setinde küçük bir değişiklik var. Resimler yerine Burada, OpenCV, bir veri dosyası ile gelir letter-recognition.data içinde opencv / numuneler / cpp / klasör. Açırsanız, ilk bakışta çöp gibi görünebilecek 20000 satır göreceksiniz. Aslında, her satırda ilk sütun bizim etiketimiz olan bir alfabe. Ardından gelen 16 sayı farklı özellikleridir. Bu özellikler [UCI Makine Öğrenim Deposundan](#) elde edilir. Bu özelliklerin ayrıntılarını [bu sayfada bulabilirsiniz](#).

Mevcut 20000 örnek var, bu yüzden eğitim örnekleri olarak ilk 10000 verisi ve test örnekleri olarak kalan 10000'i alıyoruz. Alfabeleri ascii karakterleriyle değiştirmeliyiz çünkü doğrudan alfabe ile çalışmamız.

```

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

```

```

# Load the data, converters convert the letter to a number
data= np.loadtxt('letter-recognition.data', dtype= 'float32', delimiter = ',',
                 converters= {0: lambda ch: ord(ch)-ord('A')})

# split the data to two, 10000 each for train and test
train, test = np.vsplit(data,2)

# split trainData and testData to features and responses
responses, trainData = np.hsplit(train,[1])
labels, testData = np.hsplit(test,[1])

# Initiate the kNN, classify, measure accuracy.
knn = cv2.KNearest()
knn.train(trainData, responses)
ret, result, neighbours, dist = knn.find_nearest(testData, k=5)

correct = np.count_nonzero(result == labels)
accuracy = correct*100.0/10000
print accuracy

```

Bana% 93.22'lik bir doğruluk veriyor. Yine, doğruluğu artırmak istiyorsanız, her düzeyde tekrarlı olarak hata verileri ekleyebilirsiniz.

7.2.1-) SVM'yi anlama

Hedef

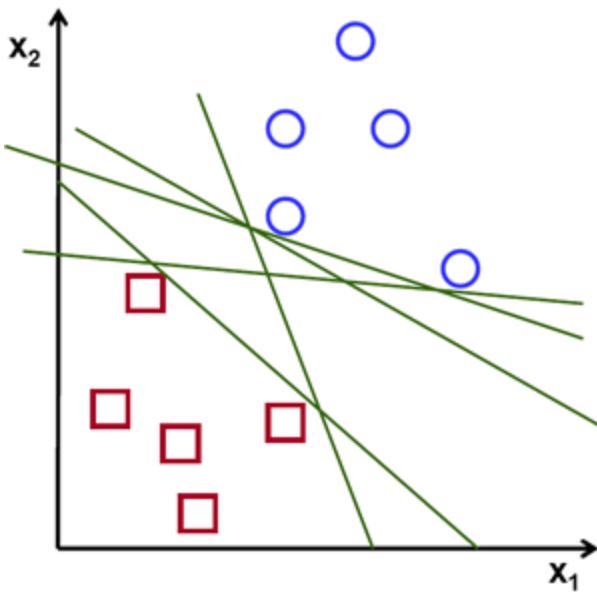
Bu bölümde

- SVM'nin sezgisel bir anlayışını göreceğiz

teori

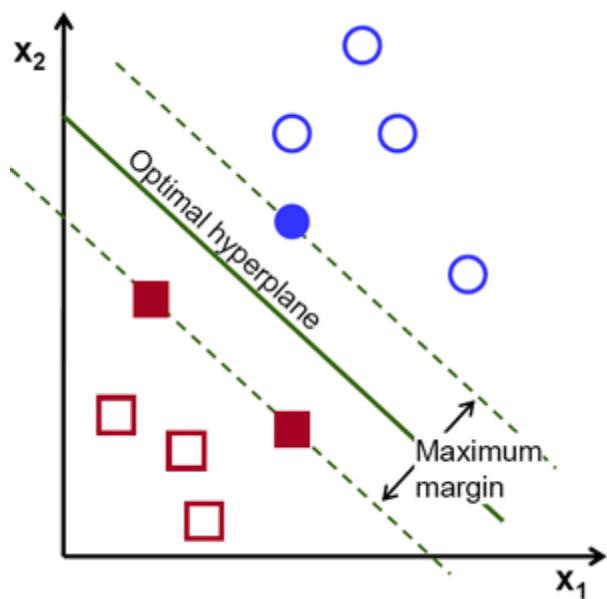
Doğrusal Olarak Ayrılabilir Veriler

Kırmızı ve mavi olmak üzere iki tür veri içeren aşağıdaki resmi düşünün. KNN'de, bir test verisi için, tüm eğitim örneklerine olan mesafesini ölçüyor ve minimum mesafeyle olanı alıyoruz. Tüm eğitim örneklerini saklamak için tüm mesafeleri ve bol miktarda belleği ölçmek çok zaman alır. Ancak görüntüde verilen veriler göz önüne alındığında, buna çok ihtiyacımız var mı?



Başka bir fikir düşünün. Her $f(x) = ax_1 + bx_2 + c$ iki veriyi de iki bölgeye ayıran bir çizgi buluyoruz. Yeni bir test_verisi aldığımızda X , yerine koymaınız yeterlidir $f(x)$. Eğer $f(X) > 0$, mavi grubuna aittir, başka kırmızı grubuna aittir. Bu çizgiye **Karar Sınırı** diyebiliriz. Çok basit ve bellek tasarrufludur. Düz bir çizgiyle (veya daha yüksek boyutlarda hiperplanlar) ikiye ayrılabilen bu verilere **Lineer Ayrılabilir** denir.

Yani yukarıdaki görüntüde, bu tür çizgilerin mümkün olduğunu görebilirsiniz. Hangisini alacağınız? Çok sezgisel olarak, çizginin tüm noktalardan mümkün olduğunda geçmesi gerektiğini söyleyebiliriz. Neden? Çünkü gelen verilerde parazit olabilir. Bu veriler sınıflandırma doğruluğunu etkilememelidir. Böylece en uzak hattı kullanmak gürültüye karşı daha fazla bağışıklık sağlayacaktır. SVM'nin yaptığı şey, eğitim örneklerine en büyük minimum mesafe sahip düz bir çizgi (veya hiper düzlem) bulmaktır. Ortadan geçen görüntüdeki kalın çizgiye bakın.



Dolayısıyla bu Karar Sınırını bulmak için eğitim verilerine ihtiyacınız vardır. Hepsine ihtiyacınız var mı? HAYIR. Sadece karşı gruba yakın olanlar yeterlidir. Bizim görüntümüzde, bunlar mavi dolu bir daire ve iki kırmızı dolu karedir. Onlara **Destek Vektörleri** diyebiliriz ve bunlardan geçen hatlara **Destek Düzlemleri** denir. Karar sınırlarımızı bulmak için yeterlidirler. Tüm veriler için endişelenmememize gerek yok. Veri azaltmaya yardımcı olur.

Olan şey, verileri en iyi temsil eden ilk iki hiper düzlemin bulunmasıdır. Örneğin, istek verileri ile temsil edilir $w^T x + b_0 > 1$ kırmızı veri ile temsil edilmektedir $w^T x + b_0 < -1$ burada w bir **ağırlık vektörü** ($w = [w_1, w_2, \dots, w_n]$) ve x özellik vektörü olan ($x = [x_1, x_2, \dots, x_n]$). b_0 olduğu **önyargı**. Ağırlık vektörü karar sınırının yönüne karar verirken, sapma noktası konumuna karar verir. Şimdi karar sınırı, bu hiper düzlemler arasında ortada olarak tanımlanmaktadır $w^T x + b_0 = 0$. Destek vektöründen karar sınırına minimum mesafe, ile verilir $\text{distance}_{\text{support vectors}} = \frac{1}{\|w\|}$. Marj bu mesafenin iki katıdır ve bu marjı en üst düzeye çıkarmamız gereklidir. yani $L(w, b_0)$, aşağıda ifade edilebilecek bazı kısıtlamalarla yeni bir işlevi en aza indirmemiz gereklidir :

$$\min_{w, b_0} L(w, b_0) = \frac{1}{2} \|w\|^2 \text{ subject to } t_i(w^T x + b_0) \geq 1 \quad \forall i$$

$$t_i \text{ her sınıfın etiketi nerede } t_i \in [-1, 1].$$

Doğrusal Olarak Ayrılamaz Veriler

Düz bir çizgiyle ikiye bölünemeyen bazı verileri düşünün. Örneğin, 'X' -3 ve 'O' -1 ve +1 olduğunda tek boyutlu bir veri düşünün. Açıkçası doğrusal olarak ayrılamaz. Ancak bu tür problemleri çözmenin yöntemleri vardır. Bu veri kümesini bir işlevle eşlestirebilirsek $f(x) = x^2$, 9'da 'X' ve doğrusal olarak ayrılabilir '1'de' O 'elde ederiz.

Aksi takdirde, bu tek boyutlu verileri iki boyutlu verilere dönüştürebiliriz. $f(x) = (x, x^2)$ Bu verileri eşlemek için işlevi kullanabiliriz. Sonra 'X' (-3,9) ve (3,9) olurken 'O' (-1,1) ve (1,1) olur. Bu aynı zamanda doğrusal olarak ayrılabilir. Kısacası, düşük boyutlu uzayda doğrusal olmayan ayrılabilir verilerin yüksek boyutlu uzayda doğrusal ayrılabilir olma şansı daha fazladır.

Genel olarak, $(D > d)$ doğrusal ayrılabilirlik olasılığını kontrol etmek için bir d -boyutlu uzayda noktaları bazı D -boyutlu uzaya eşlemek mümkündür. Düşük boyutlu girdi (özellik) uzayında hesaplamalar yaparak yüksek boyutlu (çekirdek) alanda nokta ürününü hesaplamasına yardımcı olan bir fikir vardır. Aşağıdaki örnekle açıklayabiliriz.

İki boyutlu uzayda iki noktayı düşünün $p = (p_1, p_2)$ ve $q = (q_1, q_2)$. Izin vermek Φ iki boyutlu bir nokta üç boyutlu uzaya aşağıdaki gibi eşleyen bir eşleme fonksiyonu olmak:

$$\phi(p) = (p_1^2, p_2^2, \sqrt{2}p_1p_2) \phi(q) = (q_1^2, q_2^2, \sqrt{2}q_1q_2)$$

$K(p, q)$ Aşağıda iki nokta arasında nokta ürün yapan bir çekirdek işlevi tanımlayalım :

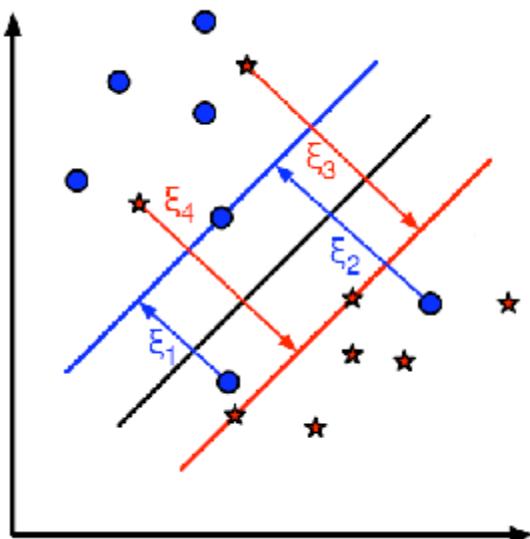
$$\begin{aligned} K(p, q) &= \phi(p) \cdot \phi(q) = \phi(p)^T \phi(q) \\ &= (p_1^2, p_2^2, \sqrt{2}p_1p_2) \cdot (q_1^2, q_2^2, \sqrt{2}q_1q_2) \\ &= p_1q_1 + p_2q_2 + 2p_1q_1p_2q_2 \\ &= (p_1q_1 + p_2q_2)^2 \\ \phi(p) \cdot \phi(q) &= (p \cdot q)^2 \end{aligned}$$

Bu, üç boyutlu uzayda bir nokta ürününün iki boyutlu uzayda kare nokta ürünü kullanılarak elde edilebileceği anlamına gelir. Bu daha yüksek boyutlu uzaya uygulanabilir. Böylece daha yüksek boyutsal özellikleri daha düşük boyutlardan hesaplayabiliyoruz. Onları eşledikten sonra daha yüksek boyutlu bir alan elde ederiz.

Tüm bu kavramlara ek olarak, yanlış sınıflandırma sorunu da ortaya çıkmaktadır. Bu nedenle, sadece maksimum marjla karar sınırını bulmak yeterli değildir. Yanlış sınıflandırma hataları sorununu da dikkate almamız gereklidir. Bazen, daha az marjla, ancak düşük yanlış sınıflandırma ile bir karar sınırı bulmak mümkün olabilir. Her neyse, modelimizi, azami marjla ancak daha az yanlış sınıflandırma ile karar sınırını bulması için değiştirmeliyiz. Minimizasyon kriterleri şu şekilde değiştirilir:

$$\min \|w\|^2 + C(\text{distance of misclassified samples to their correct regions})$$

Aşağıdaki görüntü bu kavramı göstermektedir. Egzersiz verilerinin her bir örneği için yeni bir parametre ξ_i tanımlanır. İlgili eğitim örneğinden doğru karar bölgelerine olan mesafedir. Yanlış sınıflandırılmamış olanlar için, karşılık gelen destek düzlemlerine düşer, böylece mesafeleri sıfırdır.



Yani yeni optimizasyon problemi:

$$\min_{w, b_0} L(w, b_0) = \|w\|^2 + C \sum_i \xi_i \text{ subject to } y_i(w^T x_i + b_0) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \forall i$$

C parametresi nasıl seçilmelidir? Bu sorunun cevabının, eğitim verilerinin nasıl dağıtıldığına bağlı olduğu açıklıktır. Genel bir cevap olmamasına rağmen, bu kuralları dikkate almak yararlıdır:

- Büyük C değerleri, daha az yanlış sınıflandırma hatası ancak daha düşük bir marj ile çözümler sunar. Bu durumda yanlış sınıflandırma hataları yapmanın pahalı olduğunu düşünün. Optimizasyonun amacı argümanı en aza indirmek olduğundan, az sayıda yanlış sınıflandırma hatasına izin verilir.
- Küçük C değerleri daha büyük marj ve daha fazla sınıflandırma hatası olan çözümler sunar. Bu durumda minimizasyon, toplamın bu kadarını dikkate almaz, bu nedenle daha fazla büyük marjlı bir hiper düzlem bulmaya odaklanır.

7.2.2-) SVM kullanarak El Yazısı Verilerinin OCR'si Hedef

Bu bölümde

- Elle yazılmış OCR verilerini tekrar ziyaret edeceğiz, ancak kNN yerine SVM ile.

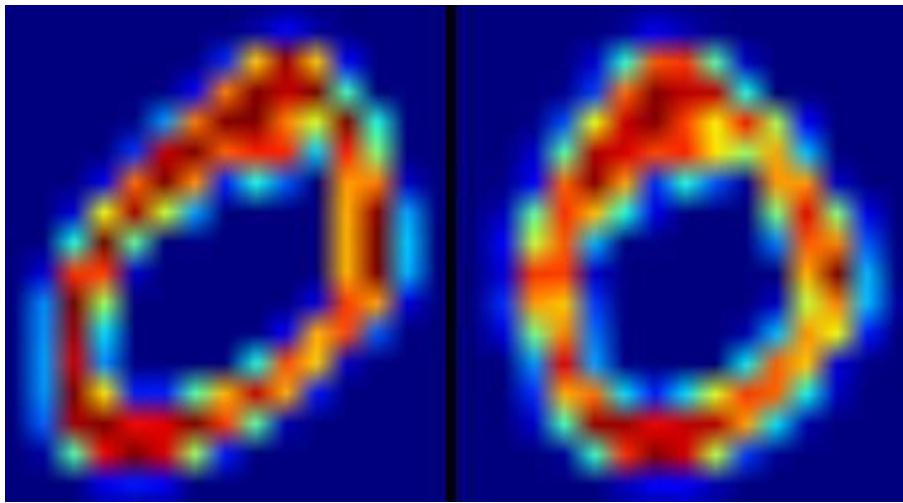
Elle Yazılmış Rakamların OCR'si

KNN'de, özellik vektörü olarak doğrudan piksel yoğunluğunu kullandık. Bu kez, özellik vektörleri olarak [Yönlendirilmiş Degradelerin Histogramını](#) (HOG) kullanacağız .

Burada, HOG'u bulmadan önce, görüntüyü ikinci derece anlarını kullanarak yeniden şekillendiriyoruz. Bu yüzden önce bir rakam görüntüsü alan ve eğrilten bir fonksiyon **deskew()** tanımlarız . Aşağıda deskew () işlevi vardır:

```
def deskew(img):
    m = cv2.moments(img)
    if abs(m['mu02']) < 1e-2:
        return img.copy()
    skew = m['mu11']/m['mu02']
    M = np.float32([[1, skew, -0.5*SZ*skew], [0, 1, 0]])
    img = cv2.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img
```

Aşağıdaki görüntü sıfır görüntüsüne uygulanan eğrilik düzeltme işlevini göstermektedir. Soldaki görüntü orijinal görüntündür ve sağdaki görüntü eğimli görüntündür.



Daha sonra her bir hücrenin HOG Tanımlayıcısını bulmalıyız. Bunun için her bir hücrenin Sobel türevlerini X ve Y yönünde buluyoruz. Ardından her pikselde büyüklüklerini ve eğim yönlerini bulun. Bu gradyan 16 tamsayı değerine niceleştirilir. Bu resmi dört alt kareye bölün. Her bir alt kare için, büyüklükleri ile ağırlıklı yön histogramını (16 kutu) hesaplayın. Böylece her alt kare size 16 değer içeren bir vektör verir. Bu dört vektör (dört alt kareden) birlikte bize 64 değer içeren bir özellik vektörü verir. Bu, verilerimizi eğitmek için kullandığımız özellik vektöridür.

```
def hog(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1)
    mag, ang = cv2.cartToPolar(gx, gy)

    # quantizing binvalues in (0...16)
    bins = np.int32(bin_n*ang/(2*np.pi))

    # Divide to 4 sub-squares
    bin_cells = bins[:10,:10], bins[10:,:10], bins[:10,10:], bins[10:,10:]
    mag_cells = mag[:10,:10], mag[10:,:10], mag[:10,10:], mag[10:,10:]
    hists = [np.bincount(b.ravel(), m.ravel(), bin_n) for b, m in
zip(bin_cells, mag_cells)]
    hist = np.hstack(hists)

    return hist
```

Son olarak, önceki durumda olduğu gibi, büyük veri setimizi tek tek hücrelere bölgerek başlıyoruz. Her basamak için 250 hücre eğitim verileri için ayrıılır ve kalan 250 veri test için ayrıılır. Tam kod aşağıda verilmiştir:

```
import cv2
import numpy as np

SZ=20
bin_n = 16 # Number of bins

svm_params = dict( kernel_type = cv2.SVM_LINEAR,
                   svm_type = cv2.SVM_C_SVC,
                   C=2.67, gamma=5.383 )
```

```

affine_flags = cv2.WARP_INVERSE_MAP|cv2.INTER_LINEAR

def deskew(img):
    m = cv2.moments(img)
    if abs(m['mu02']) < 1e-2:
        return img.copy()
    skew = m['mu11']/m['mu02']
    M = np.float32([[1, skew, -0.5*SZ*skew], [0, 1, 0]])
    img = cv2.warpAffine(img,M,(SZ, SZ),flags=affine_flags)
    return img

def hog(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1)
    mag, ang = cv2.cartToPolar(gx, gy)
    bins = np.int32(bin_n*ang/(2*np.pi))      # quantizing binvalues in (0...16)
    bin_cells = bins[:10,:,:10], bins[10:,:,:10], bins[:10,10:], bins[10:,10:]
    mag_cells = mag[:10,:,:10], mag[10:,:,:10], mag[:10,10:], mag[10:,10:]
    hists = [np.bincount(b.ravel(), m.ravel(), bin_n) for b, m in
    zip(bin_cells, mag_cells)]
    hist = np.hstack(hists)      # hist is a 64 bit vector
    return hist

img = cv2.imread('digits.png',0)

cells = [np.hsplit(row,100) for row in np.vsplit(img,50)]

# First half is trainData, remaining is testData
train_cells = [ i[:50] for i in cells ]
test_cells = [ i[50:] for i in cells ]

#####      Now training      #####
deskewed = [map(deskew,row) for row in train_cells]
hogdata = [map(hog,row) for row in deskewed]
trainData = np.float32(hogdata).reshape(-1,64)
responses = np.float32(np.repeat(np.arange(10),250)[:,np.newaxis])

svm = cv2.SVM()
svm.train(trainData,responses, params=svm_params)
svm.save('svm_data.dat')

#####      Now testing      #####
deskewed = [map(deskew,row) for row in test_cells]
hogdata = [map(hog,row) for row in deskewed]
testData = np.float32(hogdata).reshape(-1,bin_n*4)
result = svm.predict_all(testData)

#####      Check Accuracy      #####
mask = result==responses
correct = np.count_nonzero(mask)
print correct*100.0/result.size

```

Bu teknik bana neredeyse% 94 doğruluk verdi. Daha yüksek doğruluğun mümkün olup olmadığını kontrol etmek için çeşitli SVM parametreleri için farklı değerler deneyebilirsiniz. Veya bu alandaki teknik belgeleri okuyabilir ve uygulamaya çalışabilirsiniz.

7.3.1-) K-Ortalama Kümeleme Anlamak Hedef

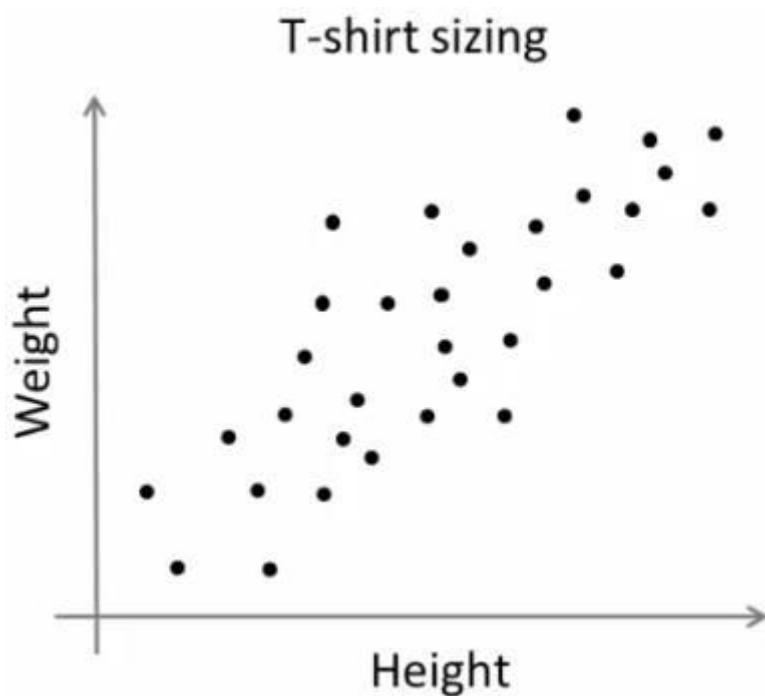
Bu bölümde, K-Ortalama Kümeleme kavramlarını, nasıl çalıştığını vb. Anlayacağız.

teori

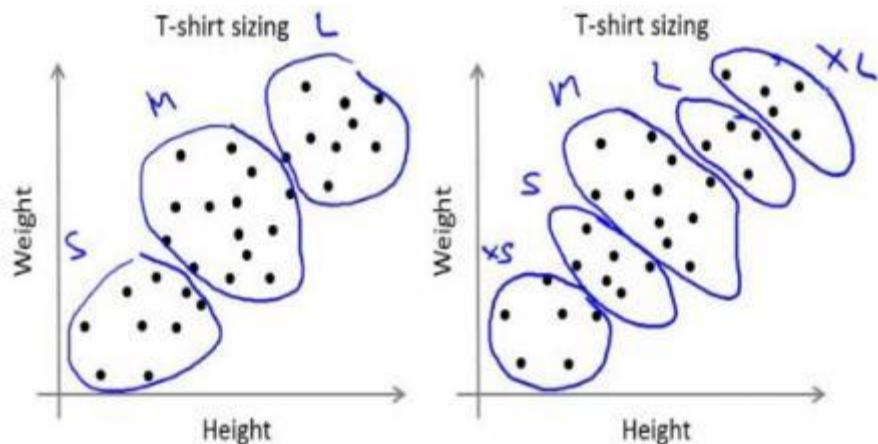
Bunu yaygın olarak kullanılan bir örnekle ele alacağız.

Tişört boyutu sorunu

Piyasaya yeni bir tişört modeli çıkaracak bir şirketi düşünün. Açıkçası, her boyuttaki insanı tatmin etmek için farklı boyutlarda modeller üretmek zorunda kalacaklar. Böylece şirket insanların boy ve kilo verilerini verip aşağıdaki gibi bir grafiğe çiziyor:



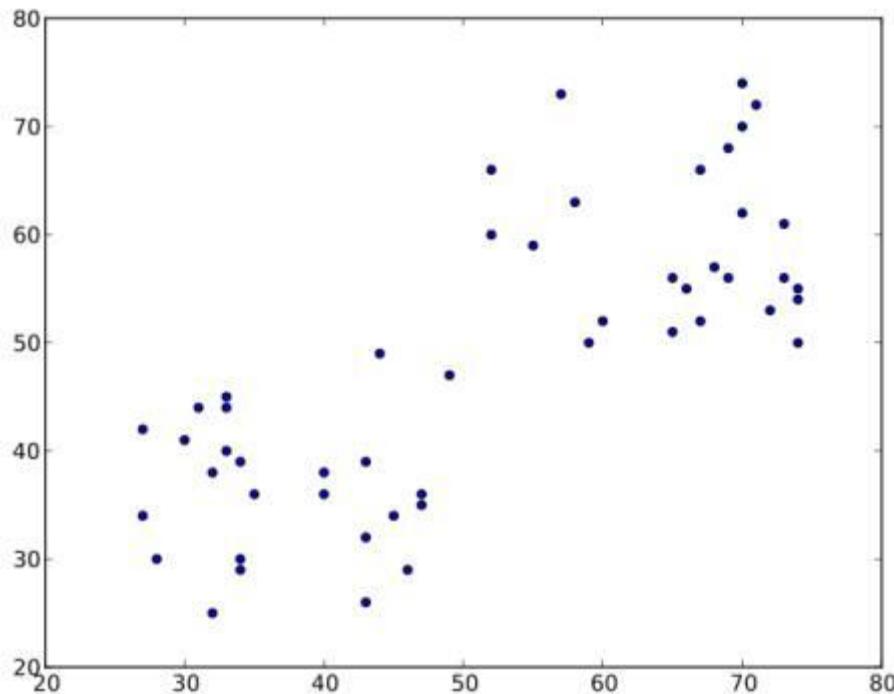
Şirket tüm boyutlarda tişört üretemez. Bunun yerine insanları Küçük, Orta ve Büyük'e bölerler ve sadece tüm insanlara uyacak bu 3 modeli üretirler. İnsanların bu üç gruba gruplanması, k-araçları kümeleme yoluyla yapılabilir ve algoritma bize tüm insanları tatmin edecek en iyi 3 boyutu sağlar. Ve eğer değilse, şirket insanları daha fazla gruba bölebilir, beş olabilir, vb. Aşağıdaki resmi kontrol edin:



O nasıl çalışır ?

Bu algoritma yinelemeli bir süreçtir. Bunu görüntülerin yardımıyla adım adım açıklayacağız.

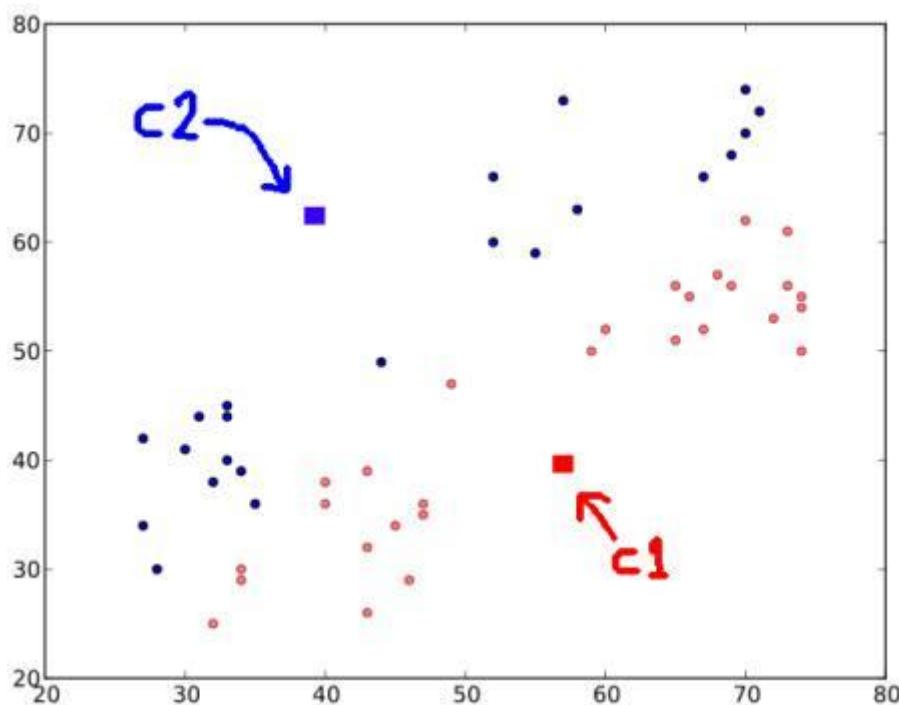
Aşağıdaki gibi bir veri setini düşünün (t-shirt problemi olarak düşünebilirsiniz). Bu verileri iki gruba ayırmamız gerekiyor.



Adım: 1 – Algoritma rastgele iki sentroid seçer C_1 ve C_2 (bazen herhangi iki veri sentroid olarak alınır).

Adım: 2 – Her noktadan her iki merkeze kadar olan mesafeyi hesaplar. Bir test verisi daha yakınsa C_1 , bu veriler '0' ile etiketlenir. Daha yakınsa C_2 , o zaman '1' olarak etiketlenir (Daha fazla centroid varsa, '2', '3' vb. Olarak etiketlenir).

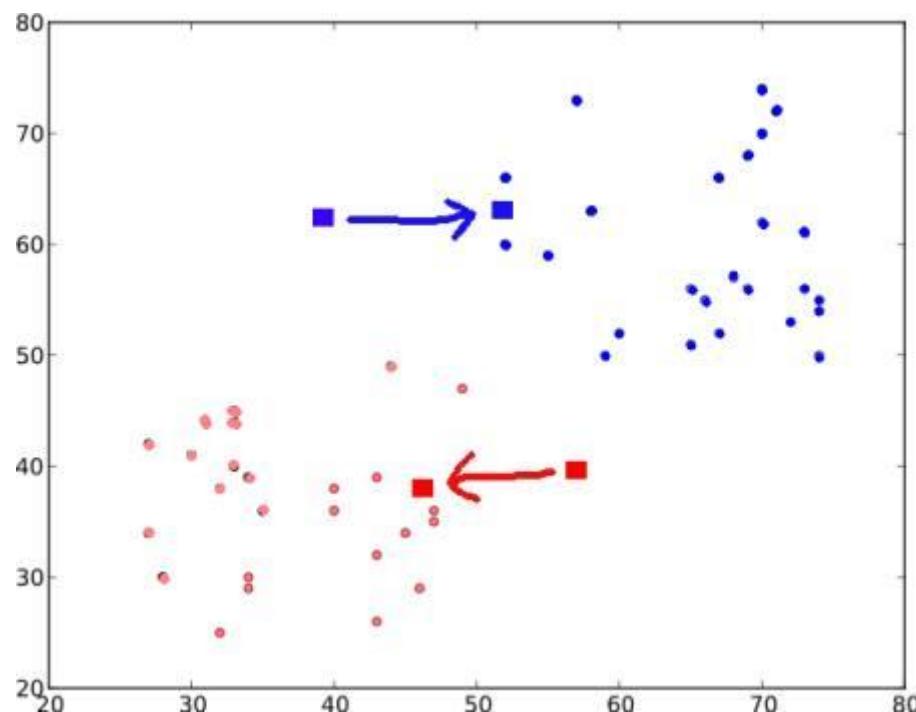
Bizim durumumuzda, kırmızı ile etiketlenmiş tüm '0' ve mavi ile etiketlenmiş '1' renklerini renklendireceğiz. Böylece yukarıdaki işlemlerden sonra aşağıdaki görüntüyü alıyoruz.



Adım: 3 – Sonra tüm mavi noktaların ve kırmızı noktaların ortalamasını ayrı ayrı hesaplıyoruz ve bu yeni sentroidlerimiz olacak. Yani C_1 ve C_2 yeni hesaplanan sentroidlere geçiş. (Unutmayın, gösterilen resimler gerçek değerler değildir ve gerçek ölçek değildir, sadece gösterim içindir).

Ve yine, adım 2'yi yeni sentroidlerle gerçekleştirin ve verileri '0' ve '1' olarak etiketleyin.

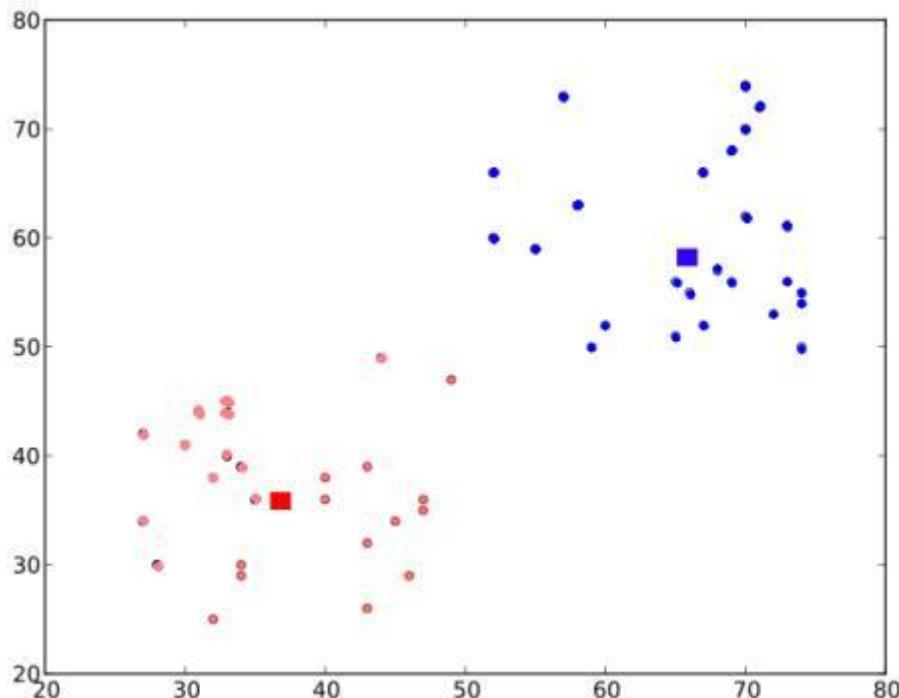
Bu yüzden aşağıdaki gibi sonuç alıyoruz:



Şimdi **Adım - 2** ve **Adım - 3** her iki centroid de sabit noktalara yaklaştırana kadar tekrarlanır. (Veya maksimum yineleme sayısı gibi belirli kriterlere bağlı olarak sağladığımız ölçütlerle bağlı olarak durdurulabilir veya belirli bir doğruluğa ulaşılabilir vb.) Bu noktalar, test verileri ile karşılık gelen centroidleri arasındaki mesafelerin toplamının minimum olacağı şekildedir. Veya basitçe, $C1 \leftrightarrow \text{Red_Points}$ ve arasındaki mesafelerin toplamı $C2 \leftrightarrow \text{Blue_Points}$ minimumdur.

$$\text{minimize} \left[J = \sum_{\text{All Red.Points}} \text{distance}(C1, \text{Red.Point}) + \sum_{\text{All Blue.Points}} \text{distance}(C2, \text{Blue.Point}) \right]$$

Nihai sonuç neredeyse aşağıdaki gibi görünüyor:



Yani bu sadece K-Ortalama Kümelemenin sezgisel bir anlayışıdır. Daha fazla ayrıntı ve matematiksel açıklama için, lütfen standart makine öğrenimi ders kitaplarını okuyun veya ek kaynaklardaki bağlantıları kontrol edin. K-Means kümelemenin sadece üst tabakasıdır. Bu algoritmada, başlangıçtaki sentroidlerin nasıl seçileceği, yineleme sürecinin nasıl hızlandırılacağı gibi birçok değişiklik var.

7.3.2-) K-OpenCV'de Kümeleme Demektir

Hedef

- Veri kümelemesi için OpenCV'de `cv2.kmeans()` işlevini kullanmayı öğrenin

Parametreleri Anlama

Giriş parametreleri

- örnekler** : `np.float32` veri türünde olmalı ve her özellik tek bir sütuna **koyulmalıdır**.

2. **nclusters (K)** : Sonunda gereken küme sayısı
3. **ölçütleri** : Yineleme sonlandırma ölçütleridir. Bu kriterler karşılandığında algoritma yinelemesi durur. Aslında, 3 parametreden oluşan bir demet olmalıdır. Bunlar (tip, max_iter, epsilon) :
 - o 3.a – fesih kriterlerinin türü : Aşağıdaki gibi 3 bayrağı vardır:

cv2.TERM_CRITERIA_EPS – belirli bir doğruluk, eğer algoritma iterasyon durdurma *epsilon* ulaşılır. **cv2.TERM_CRITERIA_MAX_ITER** – belirtilen yineleme sayısından sonra algoritmayı durdurun, *maks_iter*. **cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER** – yukarıdaki durumlardan herhangi biri gerçekleştiğinde yinelemeyi durdurur.
 - o 3.b – max_iter – Maksimum yineleme sayısını belirten bir tam sayı.
 - o 3.c – epsilon – Gerekli doğruluk
4. **teşebbüslər** : Algoritmanın farklı başlangıç etiketlerini kullanarak kaç kez yürütüleceğini belirtmek için işaretleyin. Algoritma, en iyi kompaktlığı veren etiketleri döndürür. Bu kompaktlık çıktı olarak döndürülür.
5. **bayraklar** : Bu bayrak ilk merkezlerin nasıl alınacağını belirtmek için kullanılır. Normalde bunun için iki bayrak kullanılır: **cv2.KMEANS_PP_CENTERS** ve **cv2.KMEANS_RANDOM_CENTERS** .

Çıktı parametreleri

1. **kompaktlik** : Her noktadan karşılık gelen merkezlere kare mesafenin toplamıdır.
2. **etiketleri** : Bu, her ögenin '0', '1' olarak işaretlendiği etiket dizisidir (önceki makaledeki 'kod' ile aynıdır)
3. **centre** : Bu, küme merkezleri dizisidir.

Şimdi K-Means algoritmasının üç örnekle nasıl uygulanacağını göreceğiz.

1. Tek Özellik İceren Veriler

Yalnızca tek bir özelliği, yani tek boyutlu bir veri kümesine sahip olduğunuzu düşünün. Örneğin, tişört boyutumuza karar vermek için yalnızca insanların yüksekliğini kullandığınız tişört sorunumuzu ele alabiliriz.

Bu yüzden veri oluşturarak başlıyoruz ve Matplotlib'de çiziyoruz

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

x = np.random.randint(25,100,25)
```

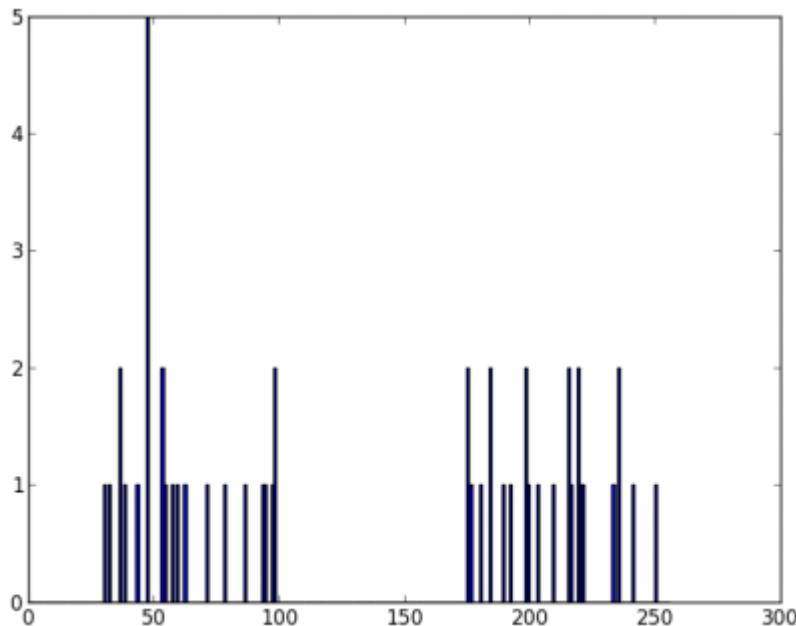
```

y = np.random.randint(175,255,25)
z = np.hstack((x,y))
z = z.reshape((50,1))
z = np.float32(z)
plt.hist(z,256,[0,256]),plt.show()

```

Yani 50 büyüklüğünde bir dizi olan 'z' ve 0 ile 255 arasında değişen değerler var. Birden fazla özellik mevcut olduğunda daha yararlı olacaktır. Sonra np.float32 tipinde veri yaptım.

Aşağıdaki görüntüyü alıyoruz:



Şimdi KMeans fonksiyonunu uyguluyoruz. Bundan önce *kriterLeri* belirtmemiz gerekiyor. Benim ölçütlerim, her 10 algoritma yinelemesi çalıştırıldığında veya *epsilon* = 1.0 doğruluğuna ulaşıldığında, algoritmayı durdurup yanıtı döndürecek şekildedir.

```

# Define criteria = ( type, max_iter = 10 , epsilon = 1.0 )
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Set flags (Just to avoid line break in the code)
flags = cv2.KMEANS_RANDOM_CENTERS

# Apply KMeans
compactness,labels,centers = cv2.kmeans(z,2,None,criteria,10,flags)

```

Bu bize kompaktlığı, etiketleri ve merkezleri verir. Bu durumda, 60 ve 207 olarak merkezler aldım. Etiketler, her bir verinin sentroidlerine bağlı olarak '0', '1', '2' vb. Olarak etiketleneceği test verileriyle aynı boyutta olacaktır. Şimdi verileri etiketlerine bağlı olarak farklı kümelere ayıriyoruz.

```

A = z[labels==0]

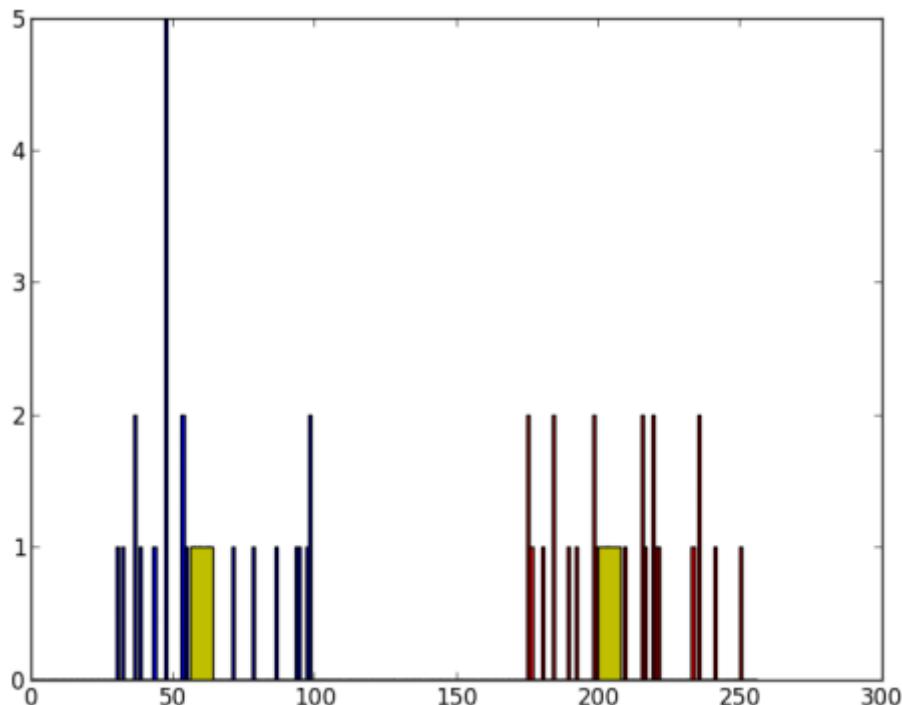
```

```
B = z[labels==1]
```

Şimdi A'yi Kırmızı renkte, B'yi Mavi renkte ve sentroidlerini Sarı renkte çiziyoruz.

```
# Now plot 'A' in red, 'B' in blue, 'centers' in yellow
plt.hist(A,256,[0,256],color = 'r')
plt.hist(B,256,[0,256],color = 'b')
plt.hist(centers,32,[0,256],color = 'y')
plt.show()
```

Aşağıda elde ettiğimiz çıktı:



2. Çoklu Özelliklere Sahip Veriler

Önceki örnekte, t-shirt problemi için sadece yükseklik aldık. Burada, hem boy hem de kilo alacağız, yani iki özellik.

Önceki durumda, verilerimizi tek bir sütun vektörüne yaptığımızı unutmayın. Her özellik bir sütunda düzenlenirken, her satır bir giriş testi örneğine karşılık gelir.

Örneğin, bu durumda, 50×2 büyüklüğünde, 50 kişinin yükseklikleri ve ağırlıkları olan bir test verileri ayarladık. İlk sütun 50 kişinin hepsinin yüksekliğine ve ikinci sütun ağırlıklarına karşılık gelir. İlk satır, birincisinin birinci kişinin yüksekliği ve ikincisinin ağırlığının olduğu iki öğe içerir. Benzer şekilde kalan satırlar diğer insanların boylarına ve ağırlıklarına karşılık gelir. Aşağıdaki resmi kontrol edin:

Features

	Height	Weight
Person 1	H1	W1
Person 2	H2	W2
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
Person 50	H50	W50

Şimdi doğrudan koda geçiyorum:

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

X = np.random.randint(25,50,(25,2))
Y = np.random.randint(60,85,(25,2))
Z = np.vstack((X,Y))

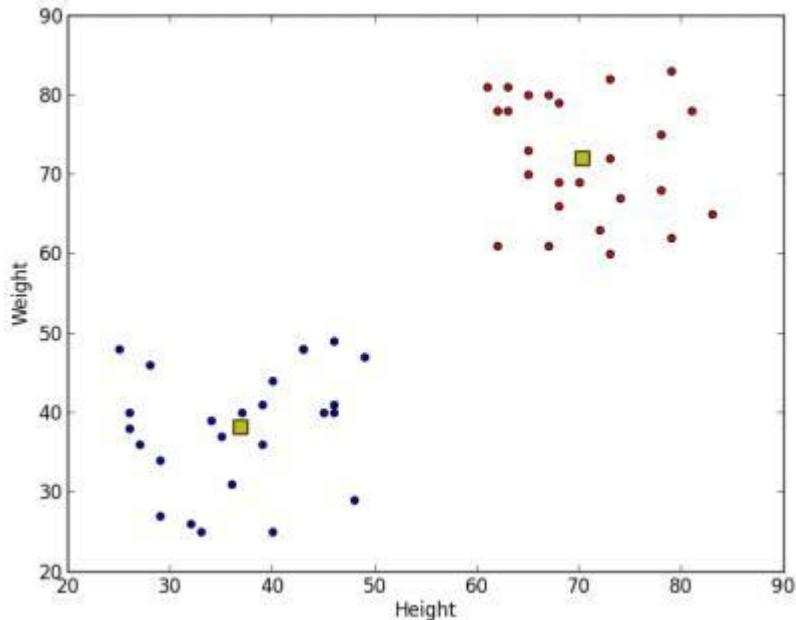
# convert to np.float32
Z = np.float32(Z)

# define criteria and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret,label,center=cv2.kmeans(Z,2,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Now separate the data, Note the flatten()
A = Z[label.ravel()==0]
B = Z[label.ravel()==1]

# Plot the data
plt.scatter(A[:,0],A[:,1])
plt.scatter(B[:,0],B[:,1],c = 'r')
plt.scatter(center[:,0],center[:,1],s = 80,c = 'y', marker = 's')
plt.xlabel('Height'),plt.ylabel('Weight')
plt.show()
```

Aşağıda elde ettiğimiz çıktı:



3. Renk Nicemleme

Renk Nicemleme, bir görüntüdeki renk sayısını azaltma işlemidir. Bunu yapmanın bir nedeni hafızayı azaltmaktadır. Bazen bazı cihazlarda sınırlı sayıda renk üretilebilecek şekilde sınırlama olabilir. Bu durumlarda da renk nicemlemesi yapılır. Burada renk ölçümü için k-ortalamaları kümeleme kullanıyoruz.

Burada açıklanacak yeni bir şey yok. R, G, B gibi 3 özellik vardır. Bu yüzden görüntüyü $M \times 3$ boyutundaki bir diziye yeniden şekillendirmemiz gerekiyor (M görüntüdeki piksel sayısıdır). Kümelemeden sonra, tüm piksellere centroid değerleri (aynı zamanda R, G, B'dir) uygularız, böylece elde edilen görüntü belirli sayıda renge sahip olur. Ve tekrar orijinal görüntünün şeklini yeniden şekillendirmemiz gerekiyor. Kod aşağıdadır:

```

import numpy as np
import cv2

img = cv2.imread('home.jpg')
Z = img.reshape((-1,3))

# convert to np.float32
Z = np.float32(Z)

# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 8
ret,label,center=cv2.kmeans(Z,K,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Now convert back into uint8, and make original image
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((img.shape))

cv2.imshow('res2',res2)
cv2.waitKey(0)

```

```
cv2.destroyAllWindows()
```

K = 8 için aşağıdaki sonuca bakın:



8.1-) Görüntü Denoising Hedef

Bu bölümde,

- Görüntüdeki paraziti gidermek için yerel olmayan araçlar denoising algoritması hakkında bilgi edineceksiniz.
- **Cv2.fastNlMeansDenoising** () , **cv2.fastNlMeansDenoisingColored** () vb. Gibi farklı işlevler göreceksiniz .

teori

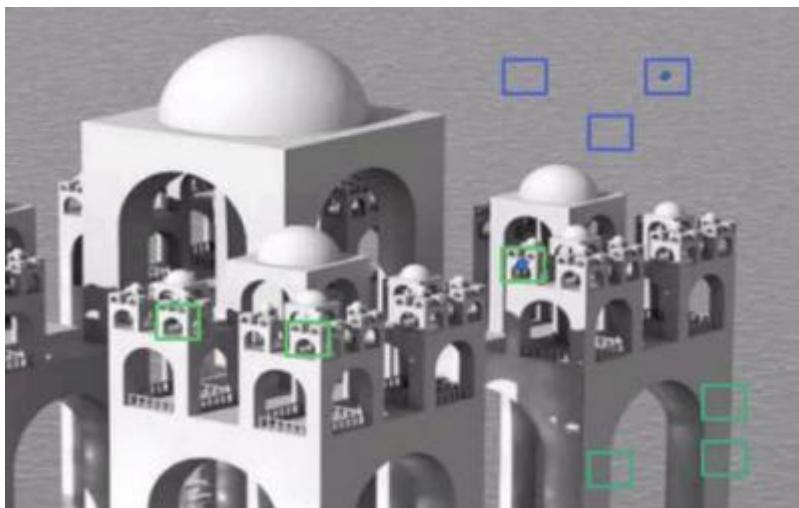
Daha önceki bölümlerde Gauss Bulanıklaştırma, Medyan Bulanıklaştırma vb.Gibi birçok görüntü yumuşatma tekniği gördük ve az miktarda gürültüyü gidermede bir dereceye kadar iyiydiler. Bu tekniklerde, bir piksel etrafında küçük bir mahalle aldık ve merkezi elemanın yerini almak için gauss ağırlıklı ortalama, değerlerin medyanı vb. Kısacası, bir pikseldeki gürültü giderme mahallesinde yereldi.

Gürültü özelliği var. Gürültü genellikle sıfır ortalama ile rastgele bir değişken olarak kabul edilir. Gürültülü bir piksel düşünün, $p = p_0 + n$ burada pikselin p_0 gerçek değeri ve n bu pikseldeki gürültü. Farklı görüntülerden çok sayıda aynı pikseli (örneğin) alabilir ve ortalamasını hesaplayabilirsiniz. İdeal olarak, $p = p_0$ ortalama gürültü sıfır olduğu için almalısınız .

Basit bir kurulumla kendiniz doğrulayabilirsiniz. Statik bir kamerası birkaç saniye boyunca belirli bir yere tutun. Bu size bol miktarda çerçeve veya aynı sahnenin birçok görüntüsünü

verecektir. Ardından, videodaki tüm karelerin ortalamasını bulmak için bir kod parçası yazın (Bu, şimdi sizin için çok basit olmalıdır). Nihai sonucu ve ilk kareyi karşılaşırın. Gürültüde azalma görebilirsiniz. Ne yazık ki bu basit yöntem kamera ve sahne hareketleri için sağlam değildir. Ayrıca genellikle sadece bir gürültülü görüntü vardır.

Fikir basit, gürültüyü ortalamak için bir dizi benzer görüntüye ihtiyacımız var. Görüntüde küçük bir pencere (örneğin 5x5 penceresi) düşünün. Şans aynıdır, aynı yamanın görüntüde başka bir yerde olması mümkündür. Bazen etrafındaki küçük bir semtte. Bu benzer yamaları birlikte kullanmaya ve ortalamalarını bulmaya ne dersiniz? O pencere için, sorun değil. Aşağıdaki örnek resme bakın:



Görüntüdeki mavi yamalar benzer görünüyor. Yeşil yamalar benzer görünüyor. Bu yüzden bir piksel alıyoruz, etrafına küçük bir pencere alıyoruz, görüntüde benzer pencereleri araştırıyoruz, tüm pencereleri ortalıyoruz ve pikseli elde ettiğimiz sonuçla değiştiriyoruz. Bu yöntem Yerel Olmayan Araç Denoising'dir. Daha önce gördüğümüz bulanıklaştırma tekniklerine kıyasla daha fazla zaman alıyor, ancak sonucu çok iyi. Daha fazla ayrıntı ve çevrimiçi demo, ek kaynaklarda ilk bağlantıda bulunabilir.

Renkli görüntüler için, görüntü CIELAB renk aralığına dönüştürülür ve sonra L ve AB bileşenlerini ayrı ayrı kınıyor.

OpenCV'de Görüntü Kınama

OpenCV bu tekniğin dört varyasyonunu sağlar.

1. **cv2.fastNlMeansDenoising ()** – tek bir gri tonlamalı görüntü ile çalışır
2. **cv2.fastNlMeansDenoisingColored ()** – renkli bir görüntüyle çalışır.
3. **cv2.fastNlMeansDenoisingMulti ()** – kısa sürede çekilen görüntü dizisiyle çalışır (gri tonlamalı resimler)
4. **cv2.fastNlMeansDenoisingColoredMulti ()** – yukarıdakiyle aynı, ancak renkli görüntüler için.

Ortak argümanlar:

- h: filtre gücüne karar veren parametre. Daha yüksek h değeri gürültüyü daha iyi kaldırır, ancak görüntünün ayrıntılarını da kaldırır. (10 tamam)
- hForColorComponents: h ile aynı, ancak yalnızca renkli görüntüler için. (normalde h ile aynı)
- templateWindowSize: tuhaf olmalı. (önerilir 7)
- searchWindowSize: tuhaf olmalı. (önerilir 21)

Bu parametreler hakkında daha fazla bilgi için lütfen ek kaynaklardaki ilk bağlantıyı ziyaret edin.

Burada 2 ve 3'ü göstereceğiz. Gerisi size kalmış.

1. cv2.fastNlMeansDenoisingColored ()

Yukarıda belirtildiği gibi, renkli görüntülerden parazit gidermek için kullanılır. (Gürültünün gauss olması bekleniyor). Aşağıdaki örneğe bakın:

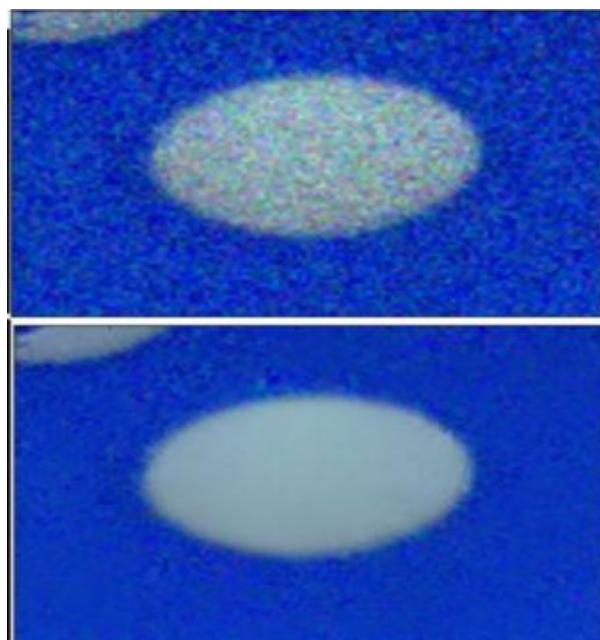
```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('die.png')

dst = cv2.fastNlMeansDenoisingColored(img, None, 10, 10, 7, 21)

plt.subplot(121),plt.imshow(img)
plt.subplot(122),plt.imshow(dst)
plt.show()
```

Aşağıda sonucun yakınlaştırılmış bir versiyonu bulunmaktadır. Giriş görüntümün gauss gürültüsü var $\sigma = 25$. Sonuca bakın:



2. cv2.fastNlMeansDenoisingMulti ()

Şimdi aynı yöntemi bir videoya uygulayacağız. İlk argüman gürültülü çerçevelerin listesidir. İkinci argüman *imgToDenoiseIndex*, hangi çerçeveyi *reddetmemiz gerektiğini* belirtir, çünkü çerçeve listemizi giriş listemize geçiririz. Üçüncü, kinamak için kullanılacak yakındaki karelerin sayısını belirten *temporalWindowSize*'dır. Tuhaf olmalı. Bu durumda, merkezi karenin notalanacak kare olduğu toplam *temporalWindowSize* kareleri kullanılır. Örneğin, giriş olarak 5 karelük bir listeyi geçtiniz. Let *imgToDenoiseIndex* = 2 ve *temporalWindowSize* = 3 . Daha sonra kare-1, kare-2 ve kare-3, kare-2'yi tanımlamak için kullanılır. Bir örnek görelim.

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

cap = cv2.VideoCapture('vtest.avi')

# create a list of first 5 frames
img = [cap.read()[1] for i in xrange(5)]

# convert all to grayscale
gray = [cv2.cvtColor(i, cv2.COLOR_BGR2GRAY) for i in img]

# convert all to float64
gray = [np.float64(i) for i in gray]

# create a noise of variance 25
noise = np.random.randn(*gray[1].shape)*10

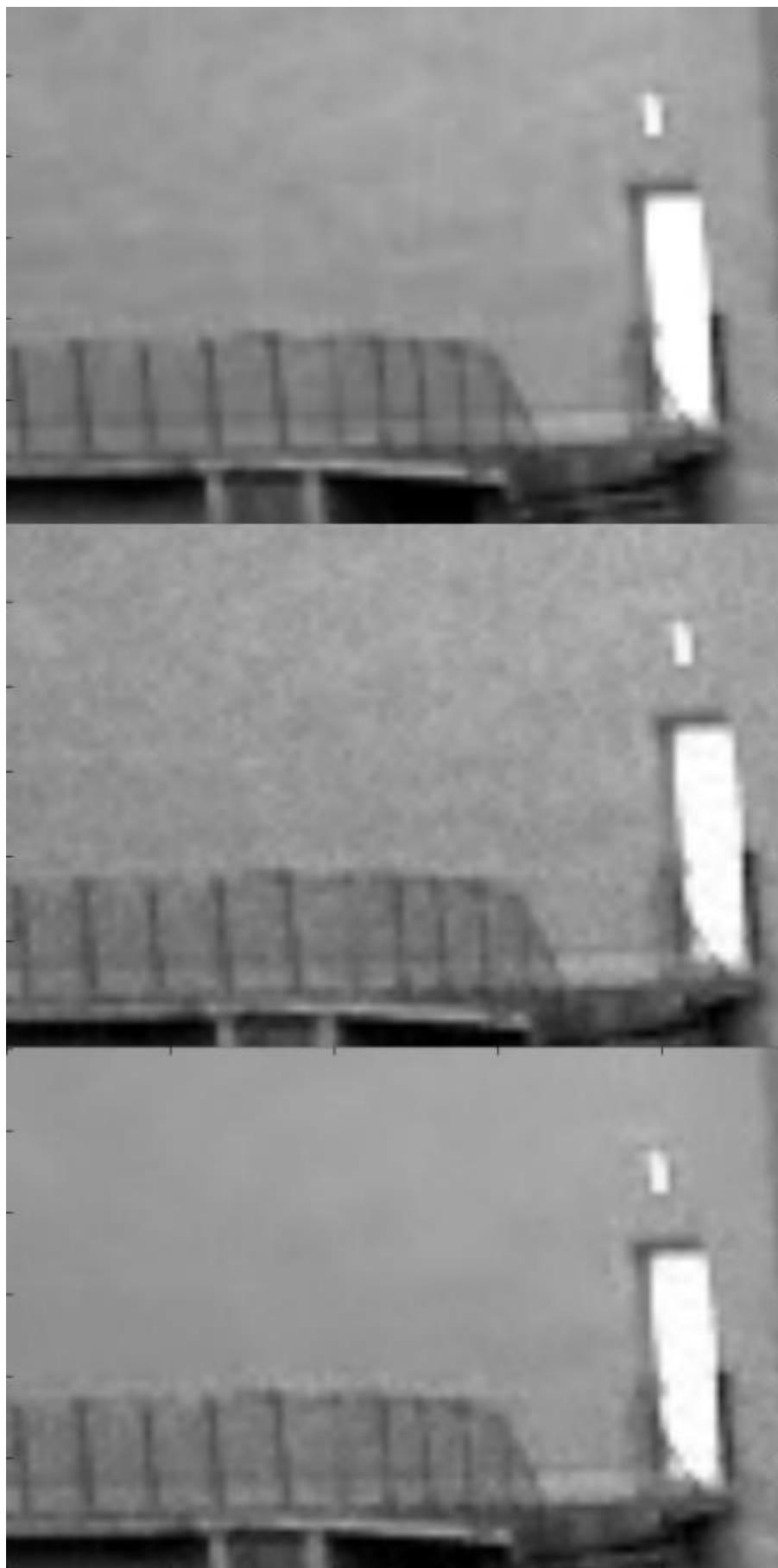
# Add this noise to images
noisy = [i+noise for i in gray]

# Convert back to uint8
noisy = [np.uint8(np.clip(i,0,255)) for i in noisy]

# Denoise 3rd frame considering all the 5 frames
dst = cv2.fastNlMeansDenoisingMulti(noisy, 2, 5, None, 4, 7, 35)

plt.subplot(131),plt.imshow(gray[2],'gray')
plt.subplot(132),plt.imshow(noisy[2],'gray')
plt.subplot(133),plt.imshow(dst,'gray')
plt.show()
```

Aşağıdaki resimde elde ettiğimiz sonucun yakınlaştırılmış bir versiyonu gösterilmektedir:



Hesaplama için oldukça fazla zaman gereklidir. Sonuçta, ilk görüntü orijinal çerçeveye, ikincisi gürültülü, üçüncüsü denoised görüntündür.

8.2-) Görüntü Boyama Hedef

Bu bölümde,

- Eski fotoğraflarda küçük sesleri, konturları vb. Boyamayı denen bir yöntemle nasıl kaldıracağımızı öğreneceğiz
- OpenCV'de boyama işlevlerini göreceğiz.

temeller

Çoğunuz evinizde bazı siyah lekeler, bazı vuruşlar vb. Hiç geri yüklemeyi düşündün mü? Onları bir boyalar aracında basitçe silemeyez, çünkü siyah yapıları hiçbir işe yaramayan beyaz yapılarla değiştirecektir. Bu durumlarda, resim boyama adı verilen bir teknik kullanılır. Temel fikir basittir: Bu kötü işaretleri komşu piksellerle değiştirin, böylece komşuluk gibi görünür. Aşağıda gösterilen resmi düşünün ([Wikipedia'dan](#) alınmıştır):



Bu amaçla çeşitli algoritmalar tasarlandı ve OpenCV bunlardan ikisini sunuyor. Her ikisine de aynı işlevle erişilebilir, `cv2.inpaint()`

İlk algoritma “**Hızlı Yürüyüş Yöntemine Dayalı Bir Görüntü Inpainting Tekniği**” makalesine dayanmaktadır. 2004 yılında Alexandru Telea tarafından hazırlanmıştır. Hızlı Yürüme Yöntemi'ne dayanmaktadır. Resimde boyanacak bir bölge düşünün. Algoritma bu bölgenin sınırlarından başlar ve önce sınırındaki her şeyi yavaş yavaş doldurarak bölgenin içine girer. Boyamak için komşuluktaki pikselin etrafında küçük bir mahalle gereklidir. Bu piksel, komşulukta bilinen tüm piksellerin normalleştirilmiş ağırlıklı toplamı ile değiştirilir. Ağırlık seçimi önemli bir konudur. Noktanın yakınında, sınırın normaline yakın olan piksellere ve sınır konturlarında yatan piksellere daha fazla ağırlık verilir. Bir piksel boyandığında, Hızlı Yürüyüş Yöntemi kullanılarak bir sonraki en yakın piksele geçer. FMM önce bilinen piksellerin yakınındaki piksellerin boyanmasını sağlar, böylece sadece manüel bir sezgisel işlem gibi çalışır. `cv2.INPAINT_TELEA`.

İkinci algoritma 2001 yılında Bertalmio, Marcelo, Andrea L. Bertozzi ve Guillermo Sapiro'nun “**Navier-Stokes, Akışkanlar Dinamiği ve Görüntü ve Video Inpainting**” adlı makalesine dayanmaktadır. Bu algoritma sıvı dinamiklerine dayanmaktadır ve kısmi diferansiyel denklemleri kullanmaktadır. Temel ilke bulusaldır. İlk olarak bilinen

bölgelerden bilinmeyen bölgelere kenarlar boyunca ilerler (çünkü kenarlar sürekli olmalıdır). Boyayan bölgenin sınırdaki gradyan vektörleri eşleştirilirken izoforlar (aynı yoğunluktaki noktaları birleştiren çizgiler, tıpkı konturlar aynı yüksekliğe sahip noktaları birleştirir) devam eder. Bunun için sıvı dinamiklerinden bazı yöntemler kullanılır. Elde edildikten sonra, o alandaki minimum varyansı azaltmak için renk doldurulur. Bu algoritma cv2.INPAINT_NS bayrağı kullanılarak etkinleştirilir.

kod

Sıfır olmayan piksellerin boyanacak alana karşılık geldiği giriş görüntüsüyle aynı boyutta bir maske oluşturmamız gereklidir. Geri kalan her şey basit. Görüntüm bazı siyah konturlarla bozuluyor (manuel olarak ekledim). Paint aracıyla ilgili konturlar oluştururdum.

```
import numpy as np
import cv2

img = cv2.imread('messi_2.jpg')
mask = cv2.imread('mask2.png', 0)

dst = cv2.inpaint(img, mask, 3, cv2.INPAINT_TELEA)

cv2.imshow('dst', dst)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Aşağıdaki sonuca bakın. İlk resim bozulmuş girişi gösterir. İkinci görüntü maskedir. Üçüncü görüntü birinci algoritmanın sonucudur ve son görüntü ikinci algoritmanın sonucudur.



8-) Haar Cascades ile Yüz Tanıma Hedef

Bu oturumda,

- Haar Feature-based Cascade Classifiers kullanarak yüz tespitinin temellerini göreceğiz
- Göz tespiti vb. İçin aynısını uzatacağız.

temeller

Haar özellik tabanlı basamaklı sınıflandırıcılar kullanarak Nesne Algılama, 2001 yılında Paul Viola ve Michael Jones tarafından önerilen “Basit Özelliklerin Arttırılmış Bir Kaskadını Kullanarak Hızlı Nesne Algılama” adlı makalesinde önerilen etkili bir nesne algılama yöntemidir. kaskat işlevi birçok olumlu ve olumsuz görüntünden eğitilir. Daha sonra diğer görüntülerdeki nesneleri algılamak için kullanılır.

Burada yüz tanıma ile çalışacağız. Başlangıçta, algoritmanın sınıflandırıcıyı eğitmek için çok sayıda olumlu görüntüye (yüz görüntülerleri) ve negatif görüntülere (yüzsüz görüntüler) ihtiyacı vardır. O zaman ondan özellikler çıkarmamız gerekiyor. Bunun için, aşağıdaki resimde gösterilen tavşan özellikleri kullanılır. Típki evrimsel çekirdeğimiz gibi. Her özellik, beyaz dikdörtgen altındaki piksellerin toplamının, siyah dikdörtgen altındaki piksellerin toplamından çıkarılmasıyla elde edilen tek bir değerdir.



(a) Edge Features



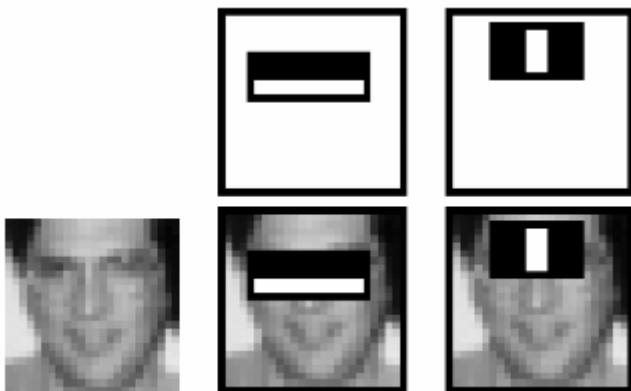
(b) Line Features



(c) Four-rectangle features

Artık her bir çekirdeğin tüm olası boyutları ve konumları, birçok özelliği hesaplamak için kullanılabilir. (Sadece ne kadar hesaplama gerektirdiğini hayal edin? 24×24 penceresi bile 160000 'ün üzerinde özelliğe neden olur). Her özellik hesaplaması için, beyaz ve siyah dikdörtgenlerin altındaki piksellerin toplamını bulmamız gereklidir. Bunu çözmek için ayrılmaz görüntülerini tanıttılar. Sadece dört piksel içeren bir işlem için piksel toplamının, piksel sayısının ne kadar büyük olabileceğini hesaplamayı kolaylaştırır. Güzel, değil mi? İşleri süper hızlı yapar.

Ancak hesapladığımız tüm bu özellikler arasında, çoğu ilgisiz. Örneğin, aşağıdaki resmi düşünün. Üst sıra iki iyi özellik gösterir. Seçilen ilk özellik, gözlerin bölgesinin genellikle burun ve yanak bölgelerinden daha koyu olduğu özelliğine odaklanıyor gibi görünüyor. Seçilen ikinci özellik, gözlerin burun köprüsünden daha koyu olma özelliğine dayanır. Ancak yanaklara veya başka herhangi bir yere uygulanan aynı pencereler önemsizdir. Peki 160000'den fazla özellik arasından en iyi özellikleri nasıl seçeriz? **Adaboost** tarafından elde edilir .



Bunun için her bir özelliği tüm eğitim resimlerine uyguluyoruz. Her özellik için, yüzleri pozitif ve negatif olarak sınıflandıracak en iyi eşiği bulur. Ancak açıkçası, hatalar veya yanlış sınıflamalar olacaktır. Minimum hata oranına sahip özellikleri seçiyoruz, yani yüz ve yüz olmayan görüntülerin en iyi sınıflandıran özellikler bunlar. (İşlem bu kadar basit değildir. Her görüntüye başlangıçta eşit ağırlık verilir. Her sınıflandırmadan sonra, yanlış sınıflandırılan görüntülerin ağırlıkları artırılır. Daha sonra aynı işlem yapılır. Yeni hata oranları hesaplanır. Ayrıca yeni ağırlıklar. gerekli doğruluk veya hata oranı elde edilene veya gerekli sayıda özellik bulunana kadar işleme devam edilir).

Son sınıflandırıcı, bu zayıf sınıflandırıcıların ağırlıklı toplamıdır. Zayıf denir çünkü tek başına görüntüyü sınıflandıramaz, ancak diğerleriyle birlikte güçlü bir sınıflandırıcı oluşturur. Makale, 200 özelliğin bile% 95 doğrulukla algılama sağladığını söylüyor. Son kurulumları yaklaşık 6000 özelliğe sahipti. (160000'den fazla özelliğe 6000 özelliğe bir düşüş düşünün. Bu büyük bir kazançtır).

Şimdi bir resim çekiyorsunuz. Her 24x24 penceresini alın. 6000 özellik uygulayın. Yüzünün olup olmadığını kontrol edin. Wow .. Wow .. Biraz verimsiz ve zaman alıcı değil mi? Evet öyle. Yazarlar bunun için iyi bir çözüm var.

Bir görüntüde, görüntü bölgesinin çoğu yüz olmayan bölgedir. Bu nedenle, bir pencerenin bir yüz bölgesi olup olmadığını kontrol etmek için basit bir yönteme sahip olmak daha iyi bir fikirdir. Değilse, tek bir çekimde atın. Tekrar işlemeyin. Bunun yerine bir yüzün olabileceği bölgeye odaklanın. Bu şekilde, olası bir yüz bölgesini kontrol etmek için daha fazla zaman bulabiliriz.

Bunun için **Sınıflandırıcıların Çağlayanı** kavramını tanıttılar . 6000 özelliğin tümünü bir pencereye uygulamak yerine, özellikleri sınıflandırıcıların farklı aşamalarında gruplayın ve tek

tek uygulayın. (Normalde ilk birkaç aşama çok daha az sayıda özellik içerir). Bir pencere ilk aşamada başarısız olursa, atın. Üzerinde kalan özellikleri dikkate alımıyoruz. Başarılı olursa, özelliklerin ikinci aşamasını uygulayın ve işleme devam edin. Tüm aşamalardan geçen pencere bir yüz bölgesidir. Plan nasıl !!!

Yazarların dedektörü, ilk beş aşamada 1, 10, 25, 25 ve 50 özellikli 38 aşamalı 6000+ özelliğe sahipti. (Yukarıdaki görüntüdeki iki özellik aslında Adaboost'tan en iyi iki özellik olarak elde edilmiştir). Yazarlara göre, ortalama olarak, alt pencere başına 6000'den 10'un özelliği değerlendirilmektedir.

Bu, Viola-Jones'un yüz tespitinin nasıl çalıştığını dair basit, sezgisel bir açıklamadır. Daha fazla bilgi için makaleyi okuyun veya Ek Kaynaklar bölümündeki referanslara göz atın.

OpenCV'de Haar-basamaklı Algılama

OpenCV, bir eğitmen ve dedektör ile birlikte gelir. Araba, uçak vb. Gibi herhangi bir nesne için kendi sınıflandırıcısını eğitmek istiyorsanız, bir tane oluşturmak için OpenCV'yi kullanabilirsiniz. Tüm ayrıntıları burada verilmiştir: [Kademeli Sınıflandırıcı Eğitimi](#).

Burada algılama ile ilgiliyoruz. OpenCV, yüz, gözler, gülümseme vb. İçin önceden eğitilmiş birçok sınıflandırıcı içerir. Bu XML dosyaları opencv / data / haarcascades / klasöründe saklanır. OpenCV ile yüz ve göz dedektörü oluşturalım.

Öncelikle gerekli XML sınıflandırıcılarını yüklememiz gerekiyor. Ardından giriş görüntümüzü (veya videoyu) gri tonlamalı modda yükleyin.

```
import numpy as np
import cv2

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

img = cv2.imread('sachin.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

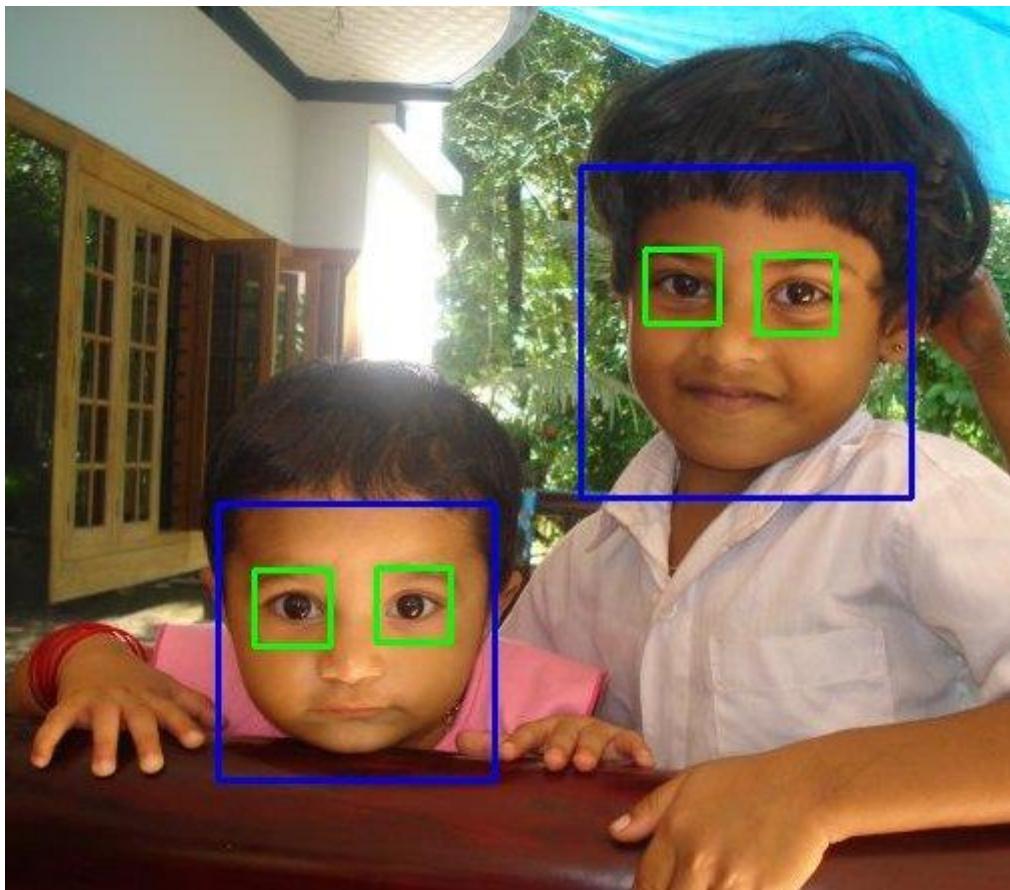
Şimdi görüntüdeki yüzleri buluyoruz. Yüzler bulunursa, algılanan yüzlerin konumlarını Rect (x, y, w, h) olarak döndürür. Bu konumları aldıktan sonra, yüz için bir ROI oluşturabilir ve bu ROI'ye göz algılama uygulayabiliriz (gözler her zaman yüzündeyken !!!).

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
for (x,y,w,h) in faces:
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

cv2.imshow('img',img)
cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
```

Sonuç aşağıdaki gibi görünür:



9-)OpenCV-Python Bağlamaları Nasıl Çalışır? Hedef

bilgi:

- OpenCV-Python bağlamaları nasıl oluşturulur?
- Yeni OpenCV modülleri Python'a nasıl genişletilir?

OpenCV-Python bağlamaları nasıl oluşturulur?

OpenCV'de, tüm algoritmalar C ++ ile uygulanır. Ancak bu algoritmalar Python, Java vb. Gibi farklı dillerden kullanılabilir. Bu, bağlantı oluşturucular tarafından mümkün kılmıştır. Bu jeneratörler, kullanıcıların C ++ işlevlerini Python'dan çağırmasını sağlayan C ++ ve Python arasında bir köprü oluşturur. Arka planda olup bitenlerin tam bir resmini elde etmek için iyi bir Python / C API bilgisi gereklidir. C ++ işlevlerinin Python'a genişletilmesine ilişkin basit bir örnek resmi Python belgelerinde bulunabilir [1]. Bu nedenle, sarma işlevlerini elle yazarak OpenCV'deki tüm işlevleri Python'a genişletmek zaman alıcı bir iştir. Bu yüzden OpenCV bunu daha akıllı bir şekilde yapıyor. OpenCV, bu sarmalayıcı işlevlerini / python / src2 modüllerinde bulunan bazı Python komut dosyalarını kullanarak C ++ başlıklarından otomatik olarak oluşturur. Ne yaptıklarına bakacağız.

İlk olarak, `modules / python / CMakeFiles.txt`, Python'a genişletilecek modüllerin kontrol eden bir CMake betigidir. Genişletilecek tüm modüller otomatik olarak kontrol eder ve başlık dosyalarını alır. Bu başlık dosyaları, söz konusu modüller için tüm sınıfların, işlevlerin, sabitlerin vb. Listesini içerir.

İkincisi, bu başlık dosyaları, Python komut geçirilen modüller / `piton / src2 / gen2.py`. Bu Python bağlama jeneratörü betigidir. Başka bir Python kod modülünü / `python / src2 / hdr_parser.py` olarak adlandırır. Bu başlık ayırtıcı komut dosyasıdır. Bu üstbilgi ayırtıcı tüm üstbilgi dosyasını küçük Python listelerine böler. Böylece bu listeler belirli bir işlev, sınıf vb. ile ilgili tüm ayrıntıları içerir. Örneğin, işlev adı, dönüş türü, girdi bağımsız değişkenleri, bağımsız değişken türleri vb. İçeren bir liste almak için bir işlev ayırtılacaktır. Son liste tüm işlevlerin, yapılarının ayrıntılarını içerir, sınıflar vb.

Ancak başlık ayırtıcısı, başlık dosyasındaki tüm işlevleri / sınıfları ayırtırmaz. Geliştirici, hangi işlevlerin Python'a aktarılacağını belirtmelidir. Bunun için, başlık ayırtıcısının ayırtılacak işlevleri tanımlamasını sağlayan bu bildirimlerin başına eklenen bazı makrolar vardır. Bu makrolar, belirli bir işlevi programlayan geliştirici tarafından eklenir. Kısacası, geliştirici hangi işlevlerin Python'a genişletilmesi gereğine karar verir. Bu makroların detayları bir sonraki oturumda verilecektir.

Böylece başlık ayırtıcı, ayırtılmış işlevlerin son büyük bir listesini döndürür. Jeneratör komut dosyamız (`gen2.py`), başlık ayırtıcısı tarafından ayırtılan tüm işlevler / sınıflar / numaralar / yapılar için sarmalayıcı işlevleri oluşturur (Bu başlık dosyalarını derleme sırasında `build / modules / python /` klasöründe `pyopencv_generated_*` .h dosyaları olarak bulabilirsiniz.). Ancak Mat, Vec4i, Size gibi bazı temel OpenCV veri türleri olabilir. Manuel olarak genişletilmeleri gereklidir. Örneğin, bir Mat türü Numpy dizisine genişletilmeli, Boyut iki tamsayı vb. Bir gruba genişletilmelidir. Benzer şekilde, manuel olarak genişletilmesi gereken bazı karmaşık yapılar / sınıflar / işlevler vb. Olabilir. Bu tür tüm manuel sarma işlevleri / `python / src2 / pcv2.hpp` modüllerine yerleştirilir.

Şimdi kalan tek şey, bize `cv2` modülünü veren bu sarıcı dosyaların derlenmesidir. Yani bir işlevi çağrıdığınızda, diyelim ki Python'da `res = equalizeHist (img1, img2)`, iki numpy dizisi geçirirsiniz ve çıktı olarak başka bir numpy dizisi beklersiniz. Böylece bu numpy dizileri `cv :: Mat`'a dönüştürülür ve sonra C++'da `equalizeHist ()` işlevini çağrırlar. Nihai sonuç, `res` tekrar Numpy dizisine dönüştürülecektir. Kısacası, neredeyse tüm işlemler C++'da yapılır, bu da bize C++ ile neredeyse aynı hızı verir.

Bu OpenCV-Python bağlarının nasıl üretildiğinin temel versiyonudur.

Yeni modüller Python'a nasıl genişletilir?

Üstbilgi ayırtıcısı, üstbilgi dosyalarını işlev bildirimine eklenen bazı sarıcı makrolara göre ayırtır. Numeralandırma sabitleri herhangi bir sarıcı makroya ihtiyaç duymaz. Otomatik olarak sarılırlar. Ancak kalan işlevler, sınıflar vb. Sarıcı makrolara ihtiyaç duyar.

Fonksiyonlar `CV_EXPORTS_W` makrosu kullanılarak genişletilir. Aşağıda bir örnek gösterilmiştir.

```
CV_EXPORTS_W void equalizeHist( InputArray src, OutputArray dst );
```

Üstbilgi ayırtıcısı, `InputArray`, `OutputArray` vb. Anahtar sözcüklerden gelen girdi ve çıktı bağımsız değişkenlerini anlayabilir. Ancak bazen, girişleri ve çıktıları sabit kodlamamız gerekebilir. Bunun için `CV_OUT`, `CV_IN_OUT` vb. Gibi makrolar kullanılır.

```
CV_EXPORTS_W void minEnclosingCircle( InputArray points,
                                     CV_OUT Point2f& center, CV_OUT float&
radius );
```

Büyük sınıflar için de `CV_EXPORTS_W` kullanılır. Sınıf yöntemlerini genişletmek için `CV_WRAP` kullanılır. Benzer şekilde, sınıf alanları için `CV_PROP` kullanılır.

```
class CV_EXPORTS_W CLAHE : public Algorithm
{
public:
    CV_WRAP virtual void apply(InputArray src, OutputArray dst) = 0;

    CV_WRAP virtual void setClipLimit(double clipLimit) = 0;
    CV_WRAP virtual double getClipLimit() const = 0;
}
```

Aşırı yüklenmiş fonksiyonlar `CV_EXPORTS_AS` kullanılarak genişletilebilir. Ancak her fonksiyonun Python'da bu isimle çağrıması için yeni bir isim geçmemiz gerekiyor. Aşağıdaki integral fonksiyonunu ele alalım. Üç işlev mevcuttur, bu nedenle her biri Python'da bir sonekle adlandırılır. Benzer şekilde, aşırı yüklenmiş yöntemleri sarmak için `CV_WRAP_AS` kullanılabilir.

```
/// computes the integral image
CV_EXPORTS_W void integral( InputArray src, OutputArray sum, int sdepth = -1 );

/// computes the integral image and integral for the squared image
CV_EXPORTS_AS(integral2) void integral( InputArray src, OutputArray sum,
                                         OutputArray sqsum, int sdepth = -1, int
sqdepth = -1 );

/// computes the integral image, integral for the squared image and the tilted
integral image
CV_EXPORTS_AS(integral3) void integral( InputArray src, OutputArray sum,
                                         OutputArray sqsum, OutputArray tilted,
                                         int sdepth = -1, int sqdepth = -1 );
```

Küçük sınıflar / yapılar `CV_EXPORTS_W_SIMPLE` kullanılarak genişletilir. Bu yapılar değere göre C ++ işlevlerine geçirilir. Örnekler `KeyPoint`, `Match` vs.'dir. Yöntemleri `CV_WRAP` tarafından genişletilir ve alanlar `CV_PROP_RW` tarafından genişletilir .

```
class CV_EXPORTS_W_SIMPLE DMatch
{
public:
    CV_WRAP DMatch();
```

```

CV_WRAP DMatch(int _queryIdx, int _trainIdx, float _distance);
CV_WRAP DMatch(int _queryIdx, int _trainIdx, int _imgIdx, float _distance);

CV_PROP_RW int queryIdx; // query descriptor index
CV_PROP_RW int trainIdx; // train descriptor index
CV_PROP_RW int imgIdx; // train image index

CV_PROP_RW float distance;
};

```

Diğer bazı küçük sınıflar / yapılar, bir Python yerel sözlüğüne aktarıldığı `CV_EXPORTS_W_MAP` kullanılarak dışa aktarılabilir. `Moments()` bunun bir örneğidir.

```

class CV_EXPORTS_W_MAP Moments
{
public:
    //! spatial moments
    CV_PROP_RW double m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
    //! central moments
    CV_PROP_RW double mu20, mu11, mu02, mu30, mu21, mu12, mu03;
    //! central normalized moments
    CV_PROP_RW double nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};

```

Yani bunlar OpenCV'de bulunan ana uzantı makroları. Tipik olarak, bir geliştirici uygun makroları uygun konumlarına koymalıdır. Dinlenme jeneratör komut dosyaları tarafından yapılır. Bazen, jeneratör komut dosyalarının sarmalayıcıları oluşturamayacağı istisnai durumlar olabilir. Bu tür işlevlerin manuel olarak ele alınması gereklidir. Ancak çoğu zaman, OpenCV kodlama yönergelerine göre yazılan bir kod otomatik olarak jeneratör komut dosyaları tarafından sarılır.