

Kesit Seviyeleri, Kenar Bazlı İmaj Gruplamak

Bir dijital imajı renklere, objelere göre belli parçalara bölmek (segmentation) için, matematiksel bir formül kullanmak iyi çözümlerden biridir. Bunu yapmanın bazı yolları var. Basitleştirerek bir örnek verelim: diyelim ki gruplama için elimizdeki formül bir yuvarlak formülü $x^2 + y^2 - c = 0$, ki c bir sabit. Bu formülü x ve y koordinatları üzerinde bastığımız zaman radius'u \sqrt{c} olan bir çember elde ederiz. Gruplama için bu çemberi büyütüp küçülttüğümüzü farzedelim, çember imaj üzerindeki istediğimiz bölme en iyi uyduğu anda gruplamayı başarılı olarak kabul ediyoruz.

Fakat problem burada: eğer imajda birden fazla grup var ise, o zaman birden fazla çember gerekecektir, bu sefer algoritmik olarak üstteki formülü ikinci, üçüncü kere yaratmamız, ve o formların o gruplara uyumunu ayrı ayrı takip etmemiz gerekirdi. Ya da diyelim ki özyineli (iterative) bir uydurma işlemi takip ediyoruz, bu işlem sırasında belki iki çemberin birleşmesi gerekse, o zaman iki formülü silip, yerine yenisini oluşturmakla uğrasmak gerekli olacaktı. Bunlar hem matematiksel, hem kodlama açısından kulfet oluşturmaktadır.

Kesit Seviyeleri kavramını kullanarak bu işi daha basitleştirebiliriz. Diyelim ki bölme görevini yapan ϕ adlı fonksiyonumuzu 2 boyutlu olmak yerine 3 boyutlu eksende tanımladık, ve, 2 boyutta bölme yapma görevini onun bir kesitine verdik. Kesit derken, alttaki üç boyutlu fonksiyonu yatay olarak bir noktadan “kestiğimizi” farz ediyoruz, ve o kesit üzerinde geçen ϕ değerlerine bakıyoruz.

Bakıcı acisimizi, tanımlamamızı değiştirerek, bazı avantajlar elde etmeyi umuyoruz aslında. Altta iki tane ϕ fonksiyonu ve onların altında kesitlerini görebiliriz.

Kesit Seviyeleri tekniğini kullanarak elde ettiğimiz avantaj nedir? Artık sadece **tek** bir ϕ fonksiyonu kullanarak 2 boyutlu imajımız üzerinde birbirinden ayrı gruplamalar yaratabiliyoruz. Bu gruplar birbiri ile birleşebilir, ayrılabilir, bu artık bizi ilgilendirmiyor. Biz sadece 3. boyuttaki ϕ fonksiyonunu değiştirmekle uğrascagız, imaj üzerindeki gruplamalar ise o fonksiyonun 2. boyuta yansımaları (projection) üzerinden kendiliğinden gerçekleşecekler.

Matematiksel olarak ϕ fonksiyonunu nasıl temsil ederiz? ϕ fonksiyonu x , y , boyutlarını alıp bize bir üçüncü z boyutu dondurmeli, ayrıca bu fonksiyonu imaj parçalarına ayırma işlemini gerçekleştirmek için kademeli olarak değiştirmeyi planladığımıza göre, o zaman bir t değişkeni de gerekiyor. Yani $\phi(x, y, t)$ fonksiyonu. Gruplama için kullanılacak kesiti ise sıfır kesiti olarak alalım, yani $\phi(x, y, t) = 0$. Doğal olarak

$$\frac{d}{dt}(\phi(x, y, t) = 0) = 0$$

Şimdi x , ve y değişkenlerinin zaman göre değişimini formüle bir şekilde dahil etmek lazım. Bunun için sıfır kesit seviyesi üzerinde bir parçacık hayal edilir, ve bu parçacığın gittiği yol $x(t)$, ve $y(t)$ olarak tanımlanır. O zaman

$$\frac{d}{dt}(\phi(x(t), y(t), t) = 0) = 0$$

Tam diferansiyel formülünden hareketle:

$$\begin{aligned}
d(\phi(x(t), y(t), t)) &= \frac{\partial \phi}{\partial x} dx + \frac{\partial \phi}{\partial y} dy + \frac{\partial \phi}{\partial t} dt = 0 \\
\frac{d(\phi(x(t), y(t), t))}{dt} &= \frac{\partial \phi}{\partial x} \frac{dx}{dt} + \frac{\partial \phi}{\partial y} \frac{dy}{dt} + \frac{\partial \phi}{\partial t} = 0 \\
\frac{d(\phi(x(t), y(t), t))}{dt} &= \frac{\partial \phi}{\partial x} \frac{dx}{dt} + \frac{\partial \phi}{\partial y} \frac{dy}{dt} + \phi_t = 0
\end{aligned} \tag{1} \quad (\text{eq1})$$

Temsilen daha kısa bir isaret kullanmak gerekirse, ∇ ile ϕ 'nin gradyanini (gradient) alarak, elde edilecek vektörün nokta carpımını kullanabiliriz. O zaman formül [1](#) daha kısa olarak:

$$\phi_t + \nabla \phi \cdot \vec{V} = 0$$

olarak temsil edilebilir, ki

$$\nabla \phi = \left(\frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y} \right)$$

$$\vec{V} = \left(\frac{dx}{dt}, \frac{dy}{dt} \right)$$

İki vektörün nokta carpımı bilindiği gibi sırayla her iki vektörün sırasıyla uyan elemanlarının birbirleri ile carpılması ve o carpımların toplanmasıdır.

\vec{V} vektörü neyi temsil eder? Formüle göre bu vektör ϕ 'nin üzerindeki değişimi etkiliyor, ve bu değişimler t 'nin değişimine göre tanımlandığına göre bu değerler “hız” olarak tanımlanabilir. İmaj bağlamında düşünürsek mesela ϕ renklerin aynı olduğu yerlerde yüksek hızda, renklerin değiştiği yerler düşük hızda değişebilir şeklinde bir kurgu yapılabilir, iste bu bölgelerde değişiminin hızını \vec{V} ile gösterebiliriz.

\vec{V} yerine kesit seviyelerine dik olan (normal) vektörler ile çalışmak isteseydik, \vec{V} 'yi dik ve teğet bileşenlerine ayırarak tekrar temsil edebiliriz: $\vec{V} = V_N \vec{N} + V_T \vec{T}$. Bu formülde \vec{T} teğet, \vec{N} dik vektörler, N ve T skalar. Yerine koyalım:

$$\phi_t + \nabla \phi \cdot (V_N \vec{N} + V_T \vec{T}) = 0$$

ϕ 'ye göre dik vektörün diğer bir formülü $\vec{N} = \frac{\nabla \phi}{|\nabla \phi|}$ olduğuna göre

$$\phi_t + (\nabla \phi \cdot V_N \frac{\nabla \phi}{|\nabla \phi|} + \nabla \phi \cdot V_T \vec{T}) = 0$$

Devam edelim: $\nabla\phi$ yuzeye dik olduguna gore, bu dik vektorun teget olan \vec{T} ile noktasal carpimi sifir degerini verecektir, o carpim formulden atilabilir. Kalanlar:

$$\phi_t + (\nabla\phi \cdot V_N \frac{\nabla\phi}{|\nabla\phi|}) = 0$$

Daha da kisaltabiliriz: $\nabla\phi \cdot \nabla\phi = |\nabla\phi|^2$ oldugunu biliyoruz, gradyanin kendisi ile noktasal carpimi, o gradyan vektorunun uzunlugunun karesidir. Daha genel olarak, bir vektorun uzunlugu, o vektorun kendisi ile noktasal carpiminin karekokudur. Ayni sey. O zaman en son formilde bu carpimi gerceklestirip, uzunluk olarak yazalim:

$$\phi_t + V_N \frac{|\nabla\phi|^2}{|\nabla\phi|} = 0$$

$$\phi_t + V_N |\nabla\phi| = 0$$

Simdi bu formul hakkinda biraz anlayis gelistirelim. Eger elimizdeki bir ϕ seviye kesitinin seklen oldugu gibi kalmasini ama sadece kuculmesini isteseydik, ϕ 'nin normalinin tersi yonunde bir buyume tanimlamamiz gerekirdi. Normal vektor disa dogru isaret ettigine gore ustteki formilde mesela $V_N = -1$ tanimlayabilirdik. O zaman

$$\phi_t + -1 |\nabla\phi| = 0$$

$$\phi_t = |\nabla\phi|$$

Hesapsal olarak bunu nasil gerceklestiririz? 80 x 80 boyutunda bir matris icinde ϕ fonksiyonu ayriksal olarak tutalim. Yani 80 tane x, 80 tane ayri y degeri var, her x ve y degerlerin kombinasyonlarına tekabül eden ϕ degerleri bu matris icinde. Gradyanin ne oldugunu hatirlayalim. Gradyan

$$\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y} \right)$$

olarak tanimlidir, ve her (x_i, y_i) noktasindaki $\phi(x_i, y_i)$ degerine gore degisik bir vektor sonucunu getirecektir. Bilgisayar dunyasinda parcali turevler hesapsal “farkliliklara” donusurler, `phi` matrisindeki farkliliklari Python ile

```
gradPhiY, gradPhiX = np.gradient(phi)
```

olarak hesaplayabiliriz. Ustte elimize gecen gradyan dizinlerindeki degerler ile $|\nabla\phi|$ buyuklugunu hesaplayabiliriz, ve bu sonucu ϕ uzerindeki degisim orani ϕ_t olarak kabul ederiz. O zaman ϕ_t ile zaman t degimi dt carptigimiz zaman ele gececek

olan ϕ 'nin deşisimidir. Dongunun her basamagında eski **phi** deęerlerine bu farkları eklediğimiz zaman ϕ fonksiyonu istediğimiz gibi evrilecektir.

Alttaki kodda bizim başlangıç ϕ 'miz kenarlardan w uzaklığında içi boş bir kutu olacak.

Ortalama Egim (Mean Curvature) Kullanmak

Eğer sabit hız yerine sıfır kesit seviyesinin herhangi bir noktada ne kadar “egri” olduğuna göre ilerlemesini isletseydik ne olurdu? Diyelim ki çok eğri bölgelerde çok hızlı, az eğik (düz, düze yakın) bölgelerde ilerleme az hız istiyoruz. O zaman hangi şekille başlarsa başlasındalar ϕ kesiti sonucta bir çember şekline doğru evrilecektir. Ortalama egim (mean curvature) hesabi için şu denklem kullanılır:

$$\kappa = -div\left(\frac{\nabla\phi}{|\nabla\phi|}\right)$$

```
f, ax = plt.subplots()

# initial function phi - level set is a square 4 pixels
# away from borders on each side, in 3D it looks like an empty
# box
c0=2; w=2
nrow, ncol= (30,30)
phi=c0*np.ones((nrow,ncol))
phi[w+1:-w-1, w+1:-w-1]=-c0

dt=1.

phiOld=np.zeros((nrow,ncol))

iter=0

while iter < 50:
    # gradient of phi
    gradPhiY, gradPhiX = np.gradient(phi)
    # magnitude of gradient of phi
    absGradPhi=np.sqrt(gradPhiX**2+gradPhiY**2)

    # normalized gradient of phi - eliminating singularities
    normGradPhiX=gradPhiX/(absGradPhi+(absGradPhi==0))
    normGradPhiY=gradPhiY/(absGradPhi+(absGradPhi==0))

    divYnormGradPhiX, divXnormGradPhiX=np.gradient(normGradPhiX)
    divYnormGradPhiY, divXnormGradPhiY=np.gradient(normGradPhiY)

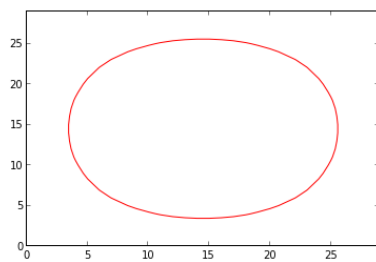
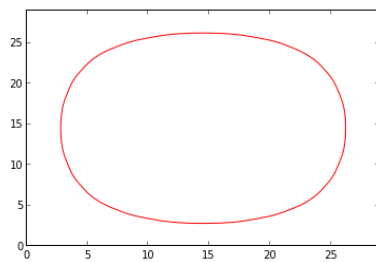
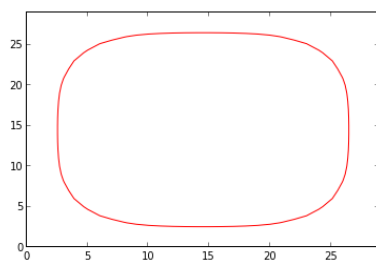
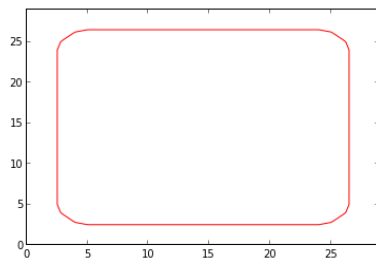
    # curvature is the divergence of normalized gradient of phi
```

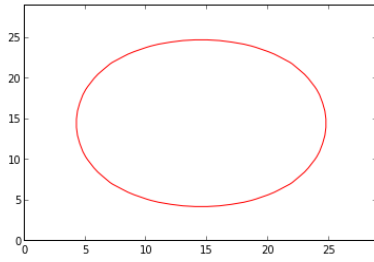
```

K = divXnormGradPhiX + divYnormGradPhiY
dPhiBydT = K * absGradPhi # makes everything circle

# level set evolution equation
phi = phi + ( dt * dPhiBydT )
if iter % 10 == 0:
    CS = plt.contour(phi,0, colors='r')
    plt.hold(False)
    plt.savefig('level_1_'+ str(iter) + '.png')
    iter += 1

```





Imaj Gruplamak

Imaji bolumlere ayirmak icin (segmentation) birkac faktorun bilesimi kullaniliyor. Koseleri kullanan aktif kontr (edge based active contour) yonteminde ortalama egim ve imajin piksel degerlerinin farkliliklari (image gradient) ayni anda kullanilir. Yani kesit seviyesini ilerletirken hizi hem egime oranliyoruz, hem de imaj piksel renk degerleri arasindaki farka ters oranda hizlandiriyor, ya da yavaslatiyoruz. Boylece kesit seviyemiz renk farkliliklari cok olmayan yani buyuk bir ihtimalle tek bir objeye ait bir bolgede hizla ilerliyor, buyuk renk farkinin oldugu buyuk bir ihtimalle bir kenar noktasina gelince ise yavasliyor. O sirada kesit seviyesinin geri kalan taraflari tabii ki baska hizlarda hareket ediyor olabilirler, zaten isin puf noktası burada, sonunda resim bolgelere ayrilmis oluyor.

Bitirirken onemli gozlemi vurgulayalım. Problemi matematiksel olarak temsil ederken, hedefe dogru turetirken surekli (continous) alemde, surekli, kesintisiz fonksiyonlarla is yapıyoruz. Hesaplama ani gelince surekli fonksiyonlari ayriksal (discrete) hale ceviriyoruz, iste uygulamali matematigin hesapsal kısmi burada devreye giriyor. Fakat diferansiyel denklemler, fonksiyonlar, turevler gibi surekli matematigin kavramlari cok onemli, bunlar olmasa problemi soyut bir sekilde temsil edemez, ve basitlestiremezdik. Temel matematigin kavramlarini kullanirken yuzyıllarin matematiksel bilgisi devreye girebiliyor, matematigin en yogun sekilde kullanildigi fizikten bol bol teknik alınabilir. Yani soylemek istedigimiz problemi cozmek icin hemen kodlamaya baslamıyoruz, dusunsel eylemin onemli bir kısmi matematiksel formullerle (belki kalem kagitla) yapıyor.

```
import scipy.signal as signal
import scipy.ndimage as image
from scipy import ndimage

def bwdist(a):
    """
    this is an intermediary function, 'a' has only True, False vals,
    so we convert them into 0, 1 values -- in reverse. True is 0,
    False is 1, distance_transform_edt wants it that way.
    """
    b = np.ones(a.shape)
    b[a==True] = 0.
    return ndimage.distance_transform_edt(b)
```

```

def gauss_kern():
    """ Returns a normalized 2D gauss kernel array for convolutions """
    h1 = 15
    h2 = 15
    x, y = np.mgrid[0:h2, 0:h1]
    x = x-h2/2
    y = y-h1/2
    sigma = 1.5
    g = np.exp( -( x**2 + y**2 ) / (2*sigma**2) );
    return g / g.sum()

Img = plt.imread("twoObj.bmp")
Img = Img[::-1]
g = gauss_kern()
Img_smooth = signal.convolve(Img,g,mode='same')
Iy,Ix=np.gradient(Img_smooth)
absGradI=np.sqrt(Ix**2+Iy**2);
rows, cols = Img.shape

# initial function phi - level set is a square 4 pixels
# away from borders on each side, in 3D it looks like an empty
# box
c0=4
w=4
nrow, ncol=Img.shape
phi=c0*np.ones((nrow,ncol))
phi[w+1:-w-1, w+1:-w-1]=-c0

# edge-stopping function
g = 1 / (1+absGradI**2)

# gradient of edge-stopping function
gy,gx = np.gradient(g)

# gradient descent step size
dt=1.

# number of iterations after which we reinitialize the surface
num_reinit=10

phiOld=np.zeros((rows,cols))

# number of iterations after which we reinitialize the surface
iter=0

```

```

while iter < 70:
    # gradient of phi
    gradPhiY, gradPhiX = np.gradient(phi)
    # magnitude of gradient of phi
    absGradPhi=np.sqrt(gradPhiX**2+gradPhiY**2)
    # normalized gradient of phi - eliminating singularities
    normGradPhiX=gradPhiX/(absGradPhi+(absGradPhi==0))
    normGradPhiY=gradPhiY/(absGradPhi+(absGradPhi==0))

    divYnormGradPhiX, divXnormGradPhiX=np.gradient(normGradPhiX)
    divYnormGradPhiY, divXnormGradPhiY=np.gradient(normGradPhiY)

    # curvature is the divergence of normalized gradient of phi
    K = divXnormGradPhiX + divYnormGradPhiY
    tmp1 = g * K * absGradPhi
    tmp2 = g * absGradPhi
    tmp3 = gx * gradPhiX + gy*gradPhiY
    dPhiBydT =tmp1 + tmp2 + tmp3

    phiOld=phi
    # level set evolution equation
    phi = phi + ( dt * dPhiBydT )
    iter=iter+1
    if np.mod(iter,num_reinit)==0:
        # reinitialize the embedding function
        # after num_reinit iterations
        phi=np.sign(phi)
        phi = (phi > 0) * (bwdist(phi < 0)) - \
            (phi < 0) * (bwdist(phi > 0))

    if np.mod(iter,10)==0:
        plt.imshow(Img, cmap='gray')
        plt.hold(True)
        plt.contour(phi,0, colors='r')
        plt.savefig('level_2_' + str(iter) + '.png')
        plt.hold(False)

```

