Technická univerzita Košice
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a umelej inteligencie

Zadanie z predmetu Počítačové videnie

Lukáš Kladný, Michaela Pekárová, Ľubomír Rusnák
Ročník: štvrtý
Študijný odbor: Inteligentné systémy
2019/2020

Obsah

Úvod 3

1. Tenzor gradientovej štruktúry 4

2. Implementácia 6

3. Výsledky 9

Záver 13

Úvod

Cieľom tohto zadania je vytvoriť program ktorý bude anizotropne segmentovať obrázok za použitia tenzora gradientovej štruktúry, využiť pritom čo najviac vlastných funkcii a následne porovnať efektivitu dvoch kódov, prvého v ktorom sú použité opencv funkcie a druhého v ktorý používa funkcie vlastné. Efektivitu určujeme pomocou času za ktorý daný program prebehne.

1. Tenzor gradientovej štruktúry

V matematike je tenzor gradientovej štruktúry (tiež označovaný ako matica druhého momentu, tenzor momentu druhého rádu, tenzor zotrvačnosti atď.) matica odvodená z gradientu funkcie. Sumarizuje dominantné smery gradientu v určenom susedstve bodu a stupeň, v akom sú tieto smery koherentné (koherentnosť). Tenzor gradientovej štruktúry sa široko používa v spracovaní obrazu a počítačovom videní pre segmentáciu 2D / 3D obrazu, detekciu pohybu, adaptívnu filtráciu, lokálnu detekciu obrazových prvkov atď.

Medzi dôležité vlastnosti anizotropných obrazov patrí orientácia a koherencia lokálnej anizotropie. V tomto článku ukážeme, ako odhadnúť orientáciu a koherenciu a ako segmentovať anizotropný obraz s jednou lokálnou orientáciou tenzorom gradientnej štruktúry.

Tenzor gradientovej štruktúry obrazu je symetrická matica 2x2. Vlastné vektory tenzora gradientovej štruktúry naznačujú lokálnu orientáciu, zatiaľ čo vlastné hodnoty dávajú koherenciu (miera anizotropismu).

Tenzor gradientovej štruktúry J, ktorý pochádza zo Z (náhodný obrázok), môže byť zapísaný ako:

$$J = \begin{bmatrix} J_{11} & J_{12} \\ J_{12} & J_{22} \end{bmatrix}$$

Kde $J_{11} = M[Z_x^2], J_{22} = M\left[Z_y^2\right], J_{12} = M\left[Z_x Z_y\right]$ - komponenty tenzora, *M[]* je symbolom matematického očakávania (túto operáciu môžeme považovať za priemernú hodnotu v okne w), Z_xa Z_y sú čiastkové derivácie obrazu *Z* vzhľadom na *x* a *y*.

Vlastné hodnoty tenzora možno nájsť v nasledujúcom vzorci:

$$\lambda_{1,2} = J_{11} + J_{22} \pm \sqrt{(J_{11} - J_{22})^2 + 4J_{12}^2}$$

Kde λ_1 je najväčšia vlastná hodnota a λ_2 je najmenšia vlastná hodnota.

Ako odhadnúť orientáciu a koherenciu anizotropného obrazu pomocou tenzora gradientovej štruktúry?

Orientácia anizotropného obrazu:

$$\alpha = 0.5 arctg \frac{2J_{12}}{J_{22} - J_{11}}$$

Koherencia:

$$C = \frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$$

Koherencia sa pohybuje od 0 do 1. Pre ideálnu miestnu orientáciu ($\lambda_2 = 0$, $\lambda_1 > 0$) je jedna, pre štruktúru izotropnej šedej hodnoty ($\lambda_1 = \lambda_2 > 0$) je nula.

2. Implementácia

V tejto časti si prejdeme časti originálneho programu ktorý využíva opencv a a našej implementacie ktorá ma dané funkcie nahradené. Sústredili sme sa na funkciu Calcgst ktorá vyzerá takto:

```
def calcGST(inputIMG, w):
    img = inputIMG.astype(np.float32)
    # GST components calculation (start)
    # J =  (J11 J12; J12 J22) - GST
    imgDiffX = cv.Sobel(img, cv.CV_32F, 1, 0, 3)
    imgDiffY = cv.Sobel(img, cv.CV_32F, 0, 1, 3)
    imgDiffXY = cv.multiply(imgDiffX, imgDiffY)
    imgDiffXX = cv.multiply(imgDiffX, imgDiffX)
    imgDiffYY = cv.multiply(imgDiffY, imgDiffY)
    J11 = cv.boxFilter(imgDiffXX, cv.CV_32F, (w,w))
    J22 = cv.boxFilter(imgDiffYY, cv.CV_32F, (w,w))
    J12 = cv.boxFilter(imgDiffXY, cv.CV_32F, (w,w))
    # GST components calculations (stop)
    # eigenvalue calculation (start)
    # lambda1 = J11 + J22 + sqrt((J11-J22)^2 + 4*J12^2)
    # lambda2 = J11 + J22 - sqrt((J11-J22)^2 + 4*J12^2)
    tmp1 = J11 + J22
    tmp2 = J11 - J22
    tmp2 = cv.multiply(tmp2, tmp2)
    tmp3 = cv.multiply(J12, J12)
    tmp4 = np.sqrt(tmp2 + 4.0 * tmp3)
    lambda1 = tmp1 + tmp4    # biggest eigenvalue
    lambda2 = tmp1 - tmp4    # smallest eigenvalue
    # eigenvalue calculation (stop)
    # Coherency calculation (start)
    # Coherency = (lambda1 - lambda2)/(lambda1 + lambda2)) - measure of anisotropism
    # Coherency is anisotropy degree (consistency of local orientation)
    imgCoherencyOut = cv.divide(lambda1 - lambda2, lambda1 + lambda2)
    # Coherency calculation (stop)
    # orientation angle calculation (start)
    # tan(2*Alpha) = 2*J12/(J22 - J11)
    # Alpha = 0.5 atan2(2*J12/(J22 - J11))
    imgOrientationOut = cv.phase(J22 - J11, 2.0 * J12, angleInDegrees = True)
    imgOrientationOut = 0.5 * imgOrientationOut
    # orientation angle calculation (stop)
    return imgCoherencyOut, imgOrientationOut
```

Vstup do funkcie je typu obraz a parameter W ktory signalizuje veľkosť strany štvorcového obrazu. Výstupom je obraz koherencie a orientácie vyrátaný z parametrov gradientového výpočtu.Tuto funkciu sme si rozobrali na menšie kusy a postupne nahradzovali za vlastný kód. V prvom kroku sme si vytvorili dva funkcie ktoré budeme ďalej používať a to:

```
def naveenConvolve(img,kernel):
    row1total = img[0,1]*kernel[0,1] + img[0,2]*kernel[0,2] + img[0,0]*kernel[0,0]
    row2total = 0
    row3total = img[2,1]*kernel[2,1] + img[2,2]*kernel[2,2] + img[2,0]*kernel[2,0]
    return row1total + row2total + row3total
```

Konvolučna funkcia ktorá convolvuje obrázok podľa daného kernelu.

```
def takePartImage(inpimg,i,j):
    image = np.zeros((3,3))
    a = i
    b = j
    for k in range(0,3):
        b = j
        for l in range(0,3):
            image[k,l] = inpimg[a,b]
            b = b+1
        a = a +1
    return image
```

A takePartImage funkcia ktorá vyberie časť daného obrázku podľa daných vstupných parametrov.

```
    imgDiffX = cv.Sobel(img, cv.CV_32F, 1, 0, 3)
    imgDiffY = cv.Sobel(img, cv.CV_32F, 0, 1, 3)
```

Volanie sobelu cez cv sme si nahradili týmito našimi funkciami:

```
#a function for finding X gradient
def naveenSobelXgradient(inputimg):
    rows = len(inputimg)
    cols = len(inputimg[0])
    Gx = np.array(np.mat('1 0 -1; 2 0 -2; 1 0 -1'))
    outputimg = np.zeros((rows,cols))
    for i in range(0,rows-3):
        for j in range(0,cols-3):
            # retreve the part of image of 3 x 3 dimension from inputimage
            image = takePartImage (inputimg, i, j)
            outputimg[i,j] = naveenConvolve(image,Gx)
    return outputimg

#a function for finding Y gradient
def naveenSobelYgradient(inputimg):
    rows = len(inputimg)
    cols = len(inputimg[0])
    #print(rows,cols)
    Gy = np.array(np.mat('1 2 1; 0 0 0; -1 -2 -1'))
    outputimg = np.zeros((rows,cols))
    for i in range(0,rows-3):
        for j in range(0,cols-3):
            # retreve the part of image of 3 x 3 dimension from inputimage
            image = takePartImage (inputimg, i, j)
            outputimg[i,j] = naveenConvolve(image,Gy)
    return outputimg
```

V daných funkciách čiastočne prechádzame obrázok a vypratávame matice pre parametre obrázku.

```
    imgDiffXY = cv.multiply(imgDiffX, imgDiffY)
    imgDiffXX = cv.multiply(imgDiffX, imgDiffX)
    imgDiffYY = cv.multiply(imgDiffY, imgDiffY)
```

Ďalej nasleduje jednoduchá matematika v ktorej funkcia vyratava časti matice gradientového tenzora podľa už spomínaného vzorcu ktorým je vyjadrený tenzor(viď. Definicia tenzoru). Túto funkciu sme nahradili jednoduchým for() cyklom.

```
diffxy = [sobelimagex[i]*sobelimagey[i] for i in range(len(sobelimagex))] # mutiply
diffxx = [sobelimagex[i]*sobelimagex[i] for i in range(len(sobelimagex))] # mutiply
diffyy = [sobelimagey[i]*sobelimagey[i] for i in range(len(sobelimagey))] # mutiply
```

Následne sa jednotlivé zložky tenzoru filtruju cez boxfilter:

```
    J11 = cv.boxFilter(imgDiffXX, cv.CV_32F, (w,w))
    J22 = cv.boxFilter(imgDiffYY, cv.CV_32F, (w,w))
    J12 = cv.boxFilter(imgDiffXY, cv.CV_32F, (w,w))
```

Túto funkciu sme nahradili našou funkciou blur ktorá aplikuje filter na vstupný obrázok podľa preddefinovaného kernelu:

```
def blur(a):
    kernel = np.ones((3,3))
    kernel = kernel / np.sum(kernel)
    arraylist = []
    for y in range(3):
        temparray = np.copy(a)
        temparray = np.roll(temparray, y - 1, axis=0)
        for x in range(3):
            temparray_X = np.copy(temparray)
            temparray_X = np.roll(temparray_X, x - 1, axis=1)*kernel[y,x]
            arraylist.append(temparray_X)

    arraylist = np.array(arraylist)
    arraylist_sum = np.sum(arraylist, axis=0)
    return arraylist_sum
```

Ďalej nasleduje samotný výpočet gradientu.

```
    tmp1 = J11 + J22
    tmp2 = J11 - J22
    tmp2 = cv.multiply(tmp2, tmp2)
    tmp3 = cv.multiply(J12, J12)
    tmp4 = np.sqrt(tmp2 + 4.0 * tmp3)
    lambda1 = tmp1 + tmp4    # biggest eigenvalue
    lambda2 = tmp1 - tmp4    # smallest eigenvalue
```

Nahradená u nás vyzerá takto

```
tmp1 = J11 + J22
tmp2 = J11 - J22
tmp2 = [tmp2[i]*tmp2[i] for i in range(len(tmp2))]
tmp3 = [J12[i]*J12[i] for i in range(len(J12))]
tmp4 = [tmp2[i] + 4.0*tmp3[i] for i in range(len(tmp2))]
tmp4 = np.sqrt(tmp4)
# Calculation of lambda
lambda1 = tmp1 + tmp4    # biggest eigenvalue
lambda2 = tmp1 - tmp4    # smallest eigenvalue
```

Nasleduje výpočet samotného výstupu:

```
    imgCoherencyOut = cv.divide(lambda1 - lambda2, lambda1 + lambda2)
    imgOrientationOut = cv.phase(J22 - J11, 2.0 * J12, angleInDegrees = True)
    imgOrientationOut = 0.5 * imgOrientationOut
```

Ktorý v našom kóde vyzeral takto:

```
#Coherency
imgCoherencyOut=[((lambda1[i]-lambda2[i])/(lambda1[i]+lambda2[i])) for i in range(len(lambda1))] # devide

# Orietation Phase
imgOrientationOutDiff=[(J22[i]-J11[i]) for i in range(len(J22))]
imgOrientationOutMlt=[(2*J12[i]) for i in range(len(J12))]
imgOrientationDiv=[(imgOrientationOutDiff[i]/imgOrientationOutMlt[i]) for i in range(len(imgOrientationOutMlt))]
imgOrientationOut=np.arctan(imgOrientationDiv)
imgOrientationOut=imgOrientationOut*57.295779513
imgOrientationOut=0.5*imgOrientationOut
```

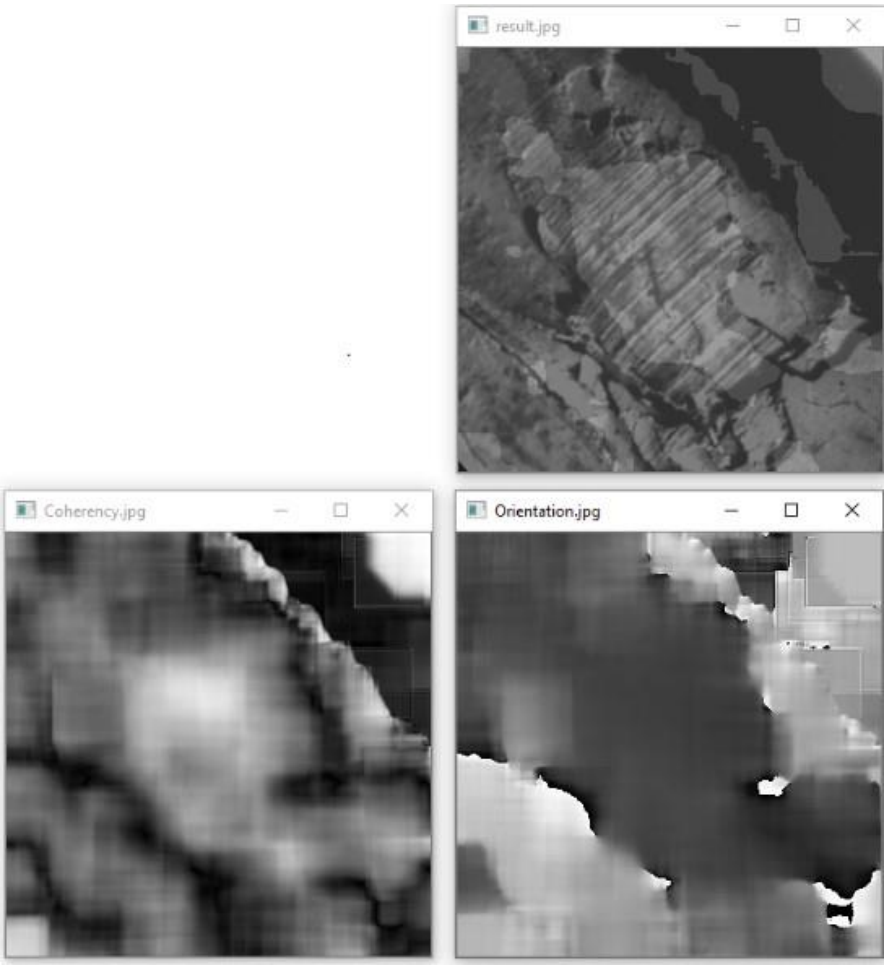
Násobenie v danej funkcii je premena radiánov na stupne.

Zvyšok sme neupravovali nakoľko je to len aplikovanie tresholdu na daný vystup a zlúčenie obrazov na binárnej úrovni.

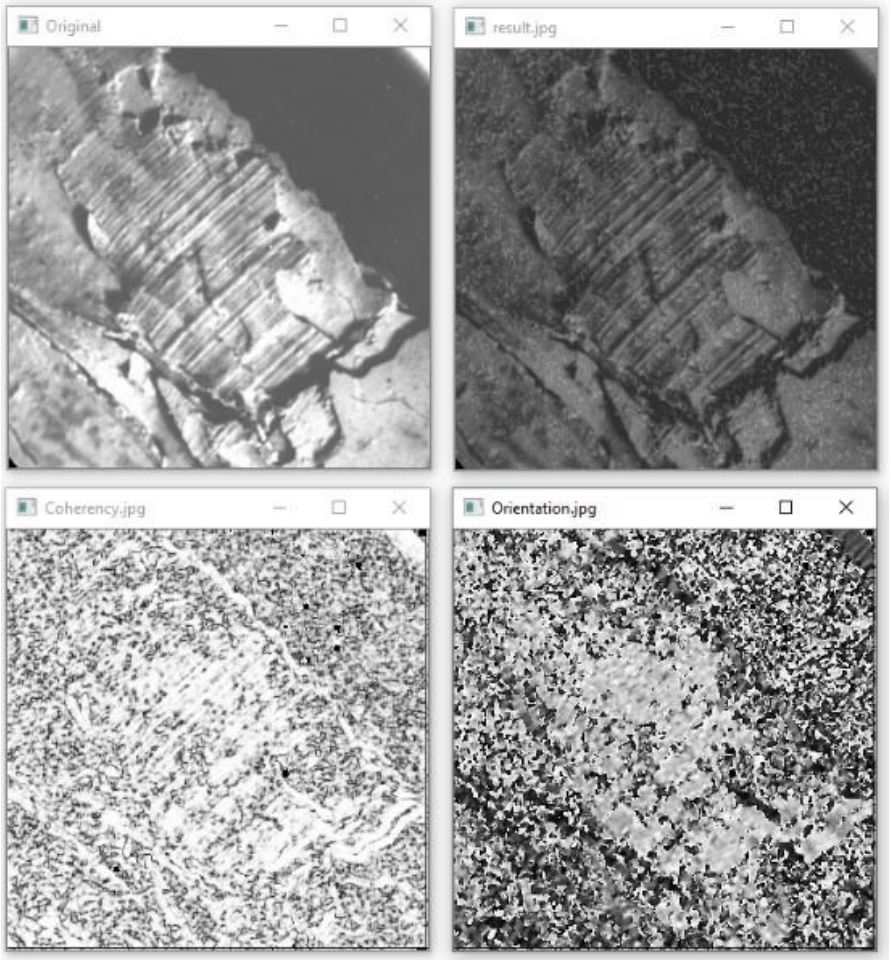
3. Výsledky

Pri výstupoch sme zistili odlišnosť v našom blur od boxfilteru a preto sme výsledky upravili nasledovne:

Originál:

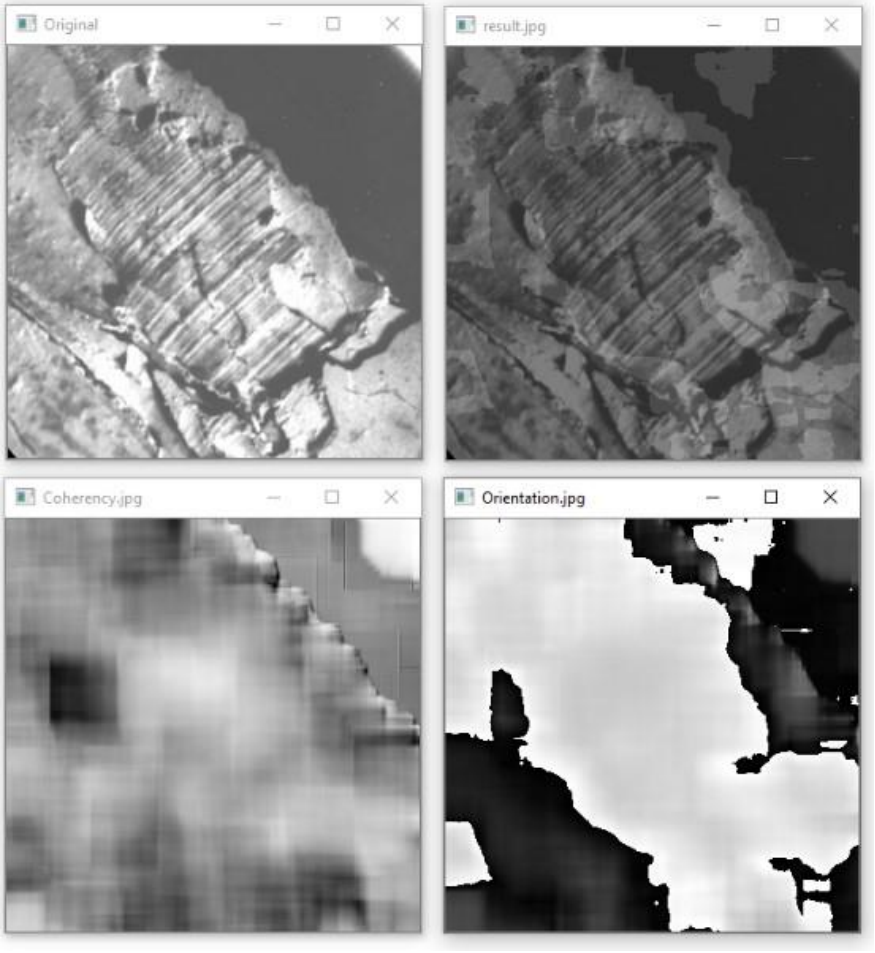


Náš program :

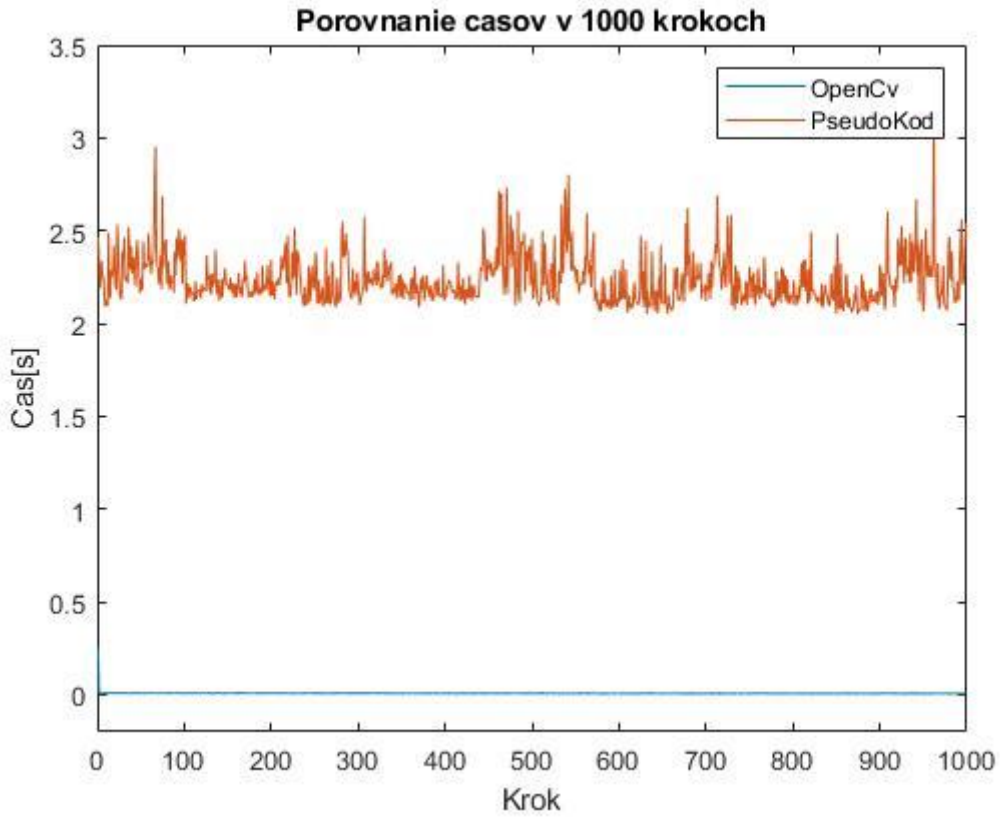


Využili sme metódu gausian blur pre mierne rozmazanie obrazka ale výsledok sa nezhodoval s originálom, takže sme ešte vyskúšali obyčajný blur ktorý bol definovaný ako matica 3*3, ale nanešťastie výsledky boli rovnaké. Takže sme tento krok nechali na knižnici opencv, kde sa tento filter volá boxfilter9().

Náš program s použitim boxfilter():



Ako vidíme na hornom obrázku ešte stále sa nedostávame ku rovnakému výsledku ako v originali, ktorý bol naprogramovaný cez opencv knižnice. Náš výsledok obkresľuje oblasť, kde sa nachádza reliéf, ktorý hľadáme.



Výsledky z meraní ukázali že rýchlosť zbehnutia nášho kódu je 2.2342080134134132 s a rýchlosť zbehnutia pôvodného kódu ktorý využíva opencv je 0.007385100600600601 s. Z tohto vyplýva že náš kód je výrazne neefektívnejší než pôvodný.

Záver

Cieľom zadania bolo vytvoriť program ktorý bude segmentovať obrázok na základe tenzora gradientovej štruktúry za použitia vlastných funkcii a následne porovnať s pôvodným kódom, ktorý využíva opencv funkcie. Merania ukázali na to že opencv kód je viacej optimalizovaný a tým pádom rýchlejší než naše vlastné funkcie.