Univerza v Ljubljani, Fakulteta za elektrotehniko **REŠITVE**
Modul A: Slikovna informatika (64135) - Laboratorijske vaje

Vaja 7: Prostorsko filtriranje slik

Podrobna navodila za reševanje

S spletne učilnice naložimo sliko cameraman-256x256-08bit.raw, ki jo bomo pri vaji potrebovali. Shranimo jo v novo mapo `Vaja07`. Zaženimo razvojno okolje Spyder, kjer ne pozabimo delovne mape nastaviti na ustvarjeno mapo. V mapi ustvarimo tudi novo `.py` datoteko `vaja07.py`.

Pri tej nalogi bomo uporabljali funkciji `displayImage()` in `loadImage()` iz Vaje 6: Geometrijske preslikave slik, ki jih skopiramo v novo datoteko `vaja07.py`. V testni blok `if __name__ == '__main__'` zapišemo ukaz za nalaganje in prikaz slike cameraman-256x256-08bit.raw. Do te faze naj bo struktura datoteke `vaja07.py` sledeča:

```
import numpy as np
import matplotlib.pyplot as pp

def displayImage(iImage, iTitle, gridX=None, gridY=None):

  pp.figure()
  pp.title(iTitle)
  if gridX is not None and gridY is not None:
    stepX = gridX[1] - gridX[0]
    stepY = gridY[1] - gridY[0]
    extent = (gridX[0] - 0.5*stepX, gridX[-1] + 0.5*stepX,
      gridY[-1] + 0.5*stepY, gridY[0] - 0.5*stepY)
    pp.imshow(iImage, cmap=pp.cm.gray, vmin=0, vmax=255, extent=extent)
  else:
    pp.imshow(iImage, cmap=pp.cm.gray, vmin=0, vmax=255)
  pp.show()

def loadImage(iPath, iSize, iType):

  fid = open(iPath, 'rb')

  oImage = np.ndarray(
    (iSize[1],iSize[0]),
    dtype=iType,
    buffer=fid.read()
    )

  fid.close()

  return oImage

if __name__ == '__main__':

  # naloži in prikaži sliko
  I = loadImage('cameraman-256x256-08bit.raw', (256, 256), np.uint8)
  displayImage(I, 'Originalna slika')
```

Naloga 1

Pri tej nalogi bomo napisali funkcijo, ki glede na vhodno sliko `iImage` in jedro filtra `iKernel` dimenzije $N \times N$ izvede operacijo linearnega filtriranja ter vrne sliko `oImage`, ki naj ima enake dimenzije kot vhodna slika. Pri filtriranju slik z nekim jedrom bomo uporabili princip konvolucije, ki ga v diskretni točki (x, y) v splošnem podaja enačba

$$g(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} w(i,j) \cdot f(x-i, y-j) \ ,$$

in si ga lahko nazorno predstavljamo kot premikanje nekega okna določenega z jedrom filtra po koordinatah vhodne slike (x, y), pri čemer izračunamo in shranimo odziv v novo sliko na mestu (x, y). V našem primeru bomo imeli opravka s kvadratnim jedrom filtra, torej bo veljalo $n = m$. Velja pa tudi $N = 2n + 1$. Potemtakem se zgornja enačba malce poenostavi v:

$$g(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} w(i,j) \cdot f(x-i, y-j) \ . \tag{1}$$

Funkcija za linearno filtriranje s pomočjo nekega jedra izgleda takole:

```
def kernelFiltering(iImage, iKernel):

  '''
  Prostorsko filtriranje slike z jedrom.
  '''

  # inicializacija filtrirane slike in dolžine jedra filtra
  oImage = np.zeros_like(iImage, dtype=np.float)
  n = int((iKernel.shape[0]-1)/2)

  # razširi slikovno domeno
  iImage_padded = np.pad(iImage, n, mode='edge')

  # priredi jedro za konvolucijo
  iKernel = np.rot90(iKernel, 2)

  # filtriranje slike
  for y in range(iImage.shape[0]):
    for x in range(iImage.shape[1]):
      iArea = iImage_padded[y:y+2*n+1, x:x+2*n+1]
      oImage[y, x] = np.sum(iKernel * iArea)

  return oImage
```

Sprva inicializiramo večdimenzionalno polje `oImage`, ki nam predstavlja izhodno filtrirano sliko. Pri tem uporabimo funkcijo `np.zeros_like()`, ki sprejme vhodno večdimenzionalno polje `iImage` in vrne večdimenzionalno polje samih ničel enake dimenzije. Pri tem smo definirali še dodaten vhodni parameter `dtype=np.float`, saj filtriranje na vhodni sliki v splošnem vrne ne-celoštevilske vrednosti. Spomnimo se, da če podatkovnega tipa izhodne slike `oImage` ne bi spremenili v podatkovni tip `np.float`, bi vsako shranjevanje v izhodno sliko rezultiralo v podatkovnem tipu vhodne slike `iImage`. Na primer:

```
# oImage je podatkovnega tipa slike iImage
iImage = np.array([[1,2,3],[1,2,3],[1,2,3]], dtype=np.uint8)
oImage = np.zeros_like(iImage)
oImage[1,1] = 2.5  # priredimo vrednost podatkovnega tipa np.float
print(oImage)  # dobimo na mestu oImage[1,1] vrednost 2
# oImage je podatkovnega tipa np.float
iImage = np.array([[1,2,3],[1,2,3],[1,2,3]], dtype=np.uint8)
oImage = np.zeros_like(iImage, dtype=np.float)
oImage[1,1] = 2.5  # priredimo vrednost podatkovnega tipa np.float
print(oImage)  # dobimo na mestu oImage[1,1] vrednost 2.5
```

Nato definiramo še parameter `n`, ki nosi informacijo o velikosti jedra filtra `iKernel` in ga bomo potrebovali pri izvedbi konvolucije. Definiran je na način, da velja $2n + 1 = N$, kjer je N vedno liho število in podaja velikost jedra v eni dimenziji. Glede na to, da privzamemo kvadratno obliko jedra, bo seveda en parameter `n` dovolj in ga izračunamo iz dolžine jedra v eni dimenziji, npr. `iKernel.shape[0]`. Uporabimo še funkcijo `int()`, ki pretvori dobljeno število v podatkovni tip `int`, saj ga bomo kasneje uporabili pri indeksiranju. Python pri indeksiranju dopušča samo celoštevilske podatkovne tipe, torej število s plavajočo vejico 1.0 ni v redu. Pri jedru dimenzije na primer 3×3 bo parameter `n` zavzemal vrednost 1.

Pri izvedbi konvolucije na diskretni mreži točk naletimo na težave, ko se točka (x, y) nahaja na robu vhodne slike, npr. $(x = 0, y = 0)$. Ko delamo izračun v okolici točke $(x = 0, y = 0)$ se lahko zgodi, da bomo na primer pri velikosti jedra 3×3 in $i = j = -1$ želeli dostopati do elementa $(-1, -1)$ vhodne slike (glejte enačbo (1)). Temu se izognemo tako, da razširimo domeno vhodne slike in sicer jo obdamo z okvirjem širine `n`, ki v primeru jedra velikosti 3×3 znaša 1. Pri tem se je potrebno zavedati, da je širina okvirja ista na vseh štirih straneh vhodne slike zgolj iz dejstva, da je jedro filtra kvadratno. Pri pravokotnem jedru bo seveda v x in y smeri debelina različna.

Sliko okvirimo s pomočjo funkcije `np.pad()`, ki sprejme večdimenzionalno podatkovno polje `iImage`, debelino okvirja `n` in način izvedbe `mode='edge'`. Debelino okvirja `n` smo podali kot konstanto, tako bodo vse štiri strani slike dobile enako debelino okvirja. Lahko bi tudi specificirali debeline za vsako stranico posebej s pomočjo strukture `tuple` in sicer `((rob_y_zgoraj,rob_y_spodaj),(rob_x_levo,rob_x_desno))`, ampak pri tej nalogi to ni potrebno. Za način izvedbe smo izbrali `'edge'`, ki sliko okviri s ponavljanjem vrednosti robnih točk slike. Obstaja še vrsto drugih, omenimo na primer `'reflect'`, ki okvir slike naredi na podlagi zrcaljenja preko robnih točk, `'symmetric'`, ki okvir slike naredi na podlagi zrcaljenja preko stranice in vključuje tudi robne točke ter `'constant'`, ki okvir slike naredi iz samih konstantnih vrednosti, ki jih moramo podati preko dodatnega parametra `constant_values`. Primeri različnih načinov okvirjanja slike so zbrani spodaj:

$$\text{iImage} = \begin{bmatrix} 5 & 0 & 9 \\ 7 & 4 & 3 \\ 2 & 5 & 1 \end{bmatrix} \quad \text{edge} \rightarrow \begin{bmatrix} 5 & 5 & 5 & 0 & 9 & 9 & 9 \\ 5 & 5 & 5 & 0 & 9 & 9 & 9 \\ 5 & 5 & 5 & 0 & 9 & 9 & 9 \\ 7 & 7 & 7 & 4 & 3 & 3 & 3 \\ 2 & 2 & 2 & 5 & 1 & 1 & 1 \\ 2 & 2 & 2 & 5 & 1 & 1 & 1 \\ 2 & 2 & 2 & 5 & 1 & 1 & 1 \end{bmatrix} \quad \text{reflect} \rightarrow \begin{bmatrix} 1 & 5 & 2 & 5 & 1 & 5 & 2 \\ 3 & 4 & 7 & 4 & 3 & 4 & 7 \\ 9 & 0 & 5 & 0 & 9 & 0 & 5 \\ 3 & 4 & 7 & 4 & 3 & 4 & 7 \\ 1 & 5 & 2 & 5 & 1 & 5 & 2 \\ 3 & 4 & 7 & 4 & 3 & 4 & 7 \\ 9 & 0 & 5 & 0 & 9 & 0 & 5 \end{bmatrix}$$

$$\text{symmetric} \rightarrow \begin{bmatrix} 4 & 7 & 7 & 4 & 3 & 3 & 4 \\ 0 & 5 & 5 & 0 & 9 & 9 & 0 \\ 0 & 5 & 5 & 0 & 9 & 9 & 0 \\ 4 & 7 & 7 & 4 & 3 & 3 & 4 \\ 5 & 2 & 2 & 5 & 1 & 1 & 5 \\ 5 & 2 & 2 & 5 & 1 & 1 & 5 \\ 4 & 7 & 7 & 4 & 3 & 3 & 4 \end{bmatrix} \qquad \text{constant} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 5 & 0 & 9 & 1 & 1 \\ 1 & 1 & 7 & 4 & 3 & 1 & 1 \\ 1 & 1 & 2 & 5 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Sledi ukaz s katerim moramo jedro filtra prirediti za konvolucijo. Spomnimo se torej enačbe (1), kjer smo konvolucijo definirali. Vidimo lahko, da se bomo z naraščanjem i in j po slikovnih elementih vhodne slike $f(x-i, y-j)$ dejansko premikali levo navzgor, po elementih jedra filtra $w(i,j)$ pa desno navzdol. Enačbo (1) lahko s pomočjo transformacije $i \rightarrow -i$ in $j \rightarrow -j$ ter upoštevanja komutativnosti seštevanja preoblikujemo v

$$g(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} w(-i,-j) \cdot f(x+i, y+j) \; .$$

Konvolucijo lahko torej izvedemo tako, da se sprehodimo po področju originalne slike, ki ga določa jedro filtra, pri čemer pa filter horizontalno in vertikalno prezrcalimo. Dvema operacijama zrcaljenja pa je ekvivalentna rotacija za 180°. Temu ustrezno zato uporabimo funkcijo `np.rot90()`, kjer prvi vhodni parameter ustreza jedru filtra `iKernel`, drugi pa številu 90° rotacij, ki jih želimo opraviti. Ker želimo doseči rotacijo 180°, bo drugi parameter znašal 2.

Nato se z dvema zankama `for` sprehodimo po celotni sliki, zato definiramo razpon `y` z `range(iImage.shape[0])` ter `x` z `range(iImage.shape[1])`. Znotraj zank `for` najprej naslovimo področje okvirjene slike `iImage_padded`, ki ga definira jedro filtriranja velikosti $(2n+1) \times (2n+1)$, središče tega področja pa je slikovni element originalne slike `iImage` v točki (x, y). Sliko `iImage_padded` naslovimo z rezanjem, torej v smeri y od `y` do `y+2n+1`, pri čemer zadnja točka ne pripada zraven. Podobno tudi za smer x. Področje shranimo v spremenljivko `iArea`.

Nato enostavno zmnožimo področje `iArea` in jedro filtra `iKernel` po istoležnih elementih ter dobljeno večdimenzionalno polje seštejemo s funkcijo `np.sum()` ter jo priredimo izhodnih sliki `oImage` na mestu `[y,x]`.

Izhodno sliko `oImage` nato vrnemo.

Naloga 2

Pri tej nalogi bomo definirali tri različna jedra filtrov in skušali zgladiti originalno sliko. To nalogo bomo rešili z zapisom kode v testni blok `if __name__ == '__main__'`, pri čemer najprej definiramo jedro filtra, nato filtriranje izvedemo s pomočjo funkcije `kernelFiltering()` ter nazadnje sliko prikažemo z ustreznim naslovom. Testni blok za jedra filtrov aritmetičnega povprečja, uteženega povprečja in Gaussovega jedra izgleda takole:

```
# glajenje slike z aritmetičnim povprečjem
avgA_F = 1/9 * np.array([[1, 1, 1], [1, 1, 1],[1, 1, 1]])
avgA_I = kernelFiltering(I, avgA_F)
displayImage(avgA_I, 'Glajenje z aritmetičnim povprečjem')

# glajenje slike z uteženim povprečjem
avgW_F = 1/16 * np.array([[1, 2, 1], [2, 4, 2],[1, 2, 1]])
avgW_I = kernelFiltering(I, avgW_F)
displayImage(avgW_I, 'Glajenje z uteženim povprečjem')

# glajenje slike z Gaussovo funkcijo, sigma=0.5
avgG_F = np.array([[0.01, 0.08, 0.01], [0.08, 0.64, 0.08], [0.01, 0.08, 0.01]])
avgG_I = kernelFiltering(I, avgG_F)
displayImage(avgG_I, 'Glajenje z Gaussovo funkcijo')
```

Posamezna jedra velikosti 3×3, ki jih imamo podane v navodilih, smo definirali kar s konstruktorjem za večdimenzionalno podatkovno polje knjižnice `Numpy`, `np.array()`. Rezultate prikažemo s pomočjo funkcije `displayImage()`.

Naloga 3

Pri tej nalogi bomo najprej ostrenje izvedli s pomočjo Laplaceovega operatorja, ki ima izotropen odziv za inkrementalne 45° rotacije. Laplaceov operator temelji na drugem odvodu, ki je bolj lokalno občutljiv in s tem tudi bolj primeren za ostrenje. Odziv slike na Laplaceov operator bo slika s poudarjenimi področji, kjer konstanto sivinsko področje preide v konstanten naklon. Sliko s poudarjenimi prehodi lahko odštejemo od originalne in s tem dobimo ostrejšo sliko $o(x, y)$:

$$o(x, y) = f(x, y) - c\Big(\nabla^2 f(x, y)\Big)$$

Stopnjo ostrenja c si lahko izbiramo s poskušanjem. Seveda, če z ostrenjem pretiravamo, lahko dobimo povsem neuporabne rezultate. V tem primeru si izberimo stopnjo ostrenja $c = 0.2$.

V kolikor želimo prikazati odziv slike na Laplaceov operator, moramo dinamično območje odziva preslikati na območje med 0 in 255, saj v splošnem ne sovpada z dinamičnim območjem slike, ki jo filtriramo. Tako bomo na tem mestu definirali novo funkcijo `scale2range()`, pri čemer uporabimo linearno sivinsko preslikavo, ki najvišjo vrednosti vhodne slike `iImage.max()` preslika v 255 in najnižjo `iImage.min()` v 0:

```
def scale2range(iImage):

  '''
  Razteg dinamičnega območja slike na 8 bitov
  '''

  oImage = 255/(iImage.max()-iImage.min()) * (iImage - iImage.min())

  return oImage
```

Pri tem izraz `(iImage - iImage.min())` najprej najnižjo vrednost slike `iImage` preslika v 0, nato sledi množenje z razmerjem razponov 8 bitnega dinamičnega območja in dinamičnega območja vhodne slike `255/(iImage.max()-iImage.min())`, s čimer dobimo želeni rezultat `oImage`.

Sedaj lahko najprej izvedemo filtriranje slike z Laplaceovim operatorjem, sledi ostrenje in nazadnje prikaz odziva na Laplaceov operator ter rezultat ostrenja:

```
c = 0.2
lap_F = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])
lap_R = kernelFiltering(I, lap_F)
lap_I = I - c * lap_R
displayImage(scale2range(lap_R), 'Slika odziva Laplaceovega operatorja')
displayImage(lap_I, 'Ostrenje z Laplaceovim operatorjem')
```

Kot lahko vidimo na sliki odziva Laplaceovega operatorja, so področja prehodov zelo izrazita. V postopku ostrenja pa področja prehodov odštejemo od originalne slike, s čimer jih poudarimo.

Drugi del te naloge sestavlja ostrenje slike z maskiranjem neostrih področij

$$m(x,y) = f(x,y) - F(x,y) \qquad \Longrightarrow \qquad o(x,y) = f(x,y) + c\Big(m(x,y)\Big)\,,$$

pri čemer bomo najprej od originalne slike `I` odšteli z Gaussovim filtrom zglajeno sliko `avgG_I`, ki smo jo dobili pri Nalogi 2. Dobljeno masko pa bomo nato množili s konstanto c in jo prišteli originalni sliki. Za vrednost konstante c si tokrat izberemo $c = 2$, izbira pa je spet odvisna od želenega rezultata. Višja kot je vrednost, višja bo stopnja ostrenja.

```
c = 2
msk_R = I - avgG_I
msk_I = I + c*msk_R
displayImage(scale2range(msk_R),
  'Slika maske pri ostrenju z maskiranjem neostrih področij')
displayImage(msk_I, 'Ostrenje z maskiranjem neostrih področij')
```

V primeru prikaza maske za ostrenje `msk_R` smo podobno kot prej dinamično območje prilagodili 8 bitnem področju od 0 do 255 s funkcijo `scale2range()`.

Naloga 4

S pomočjo Sobelovega operatorja bomo določili najprej gradiente slike v smeri x in y, ter nato še amplitudno sliko, ki velja za nelinearno operacijo. Odsek testnega bloka za izračun gradientov s Sobelovim operatorjem izgleda takole:

```
# računanje odziva na Sobelov operator v x smeri
sobX_F = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobX_I = kernelFiltering(I, sobX_F)
displayImage(scale2range(sobX_I), 'Odziv na Sobelov operator v X smeri')

# računanje odziva na Sobelov operator v y smeri
sobY_F = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
sobY_I = kernelFiltering(I, sobY_F)
displayImage(scale2range(sobY_I), 'Odziv na Sobelov operator v Y smeri')

# računanje amplitudne slike gradienta
sobA_I = np.sqrt(sobX_I**2 + sobY_I**2)
displayImage(scale2range(sobA_I), 'Amplitudna slika gradienta')
```

S spremenljivkama `sobX_F` in `sobY_F` definiramo Sobelov operator oziroma jedro filtra v smeri x in y. Uporabimo funkcijo `kernelFiltering()` in izhodne slike gradientov shranimo v

spremenljivki `sobX_I` in `sobY_I` ter ju prikažemo s funkcijo `displayImage()`, pri čemer spet uporabimo funkcijo `scale2range()`, da dinamično območje prilagodimo na interval 0 do 255.

Amplitudno sliko gradienta izračunamo kot velikost vektorja gradienta po enačbi

$$|\nabla f(x,y)| = \sqrt{\Big(\nabla_x f(x,y)\Big)^2 + \Big(\nabla_y f(x,y)\Big)^2},$$

kjer $\nabla_x f(x,y)$ in $\nabla_y f(x,y)$ predstavljata komponenti gradienta v smeri x in y v točki (x,y). Obe komponenti smo izračunali s pomočjo Sobelovega operatorja in jih shranili v spremenljivki `sobX_I` in `sobY_I`. Amplitudo v vsaki točki (x,y) lahko enostavno izračunamo s pomočjo kvadriranja obeh slik gradientov po istoležnih elementih, ter korenjenjem vsote. Vse operacije torej izvedemo na istoležnih elementih slik `sobX_I` in `sobY_I` in povsem neodvisno za vsak slikovni element na položaju (x,y).

Dobljeno sliko amplitude gradienta nato še prikažemo, pri čemer spet prilagodimo dinamično območje na interval [0, 255].

Naloga 5

V tej nalogi bomo napisali funkcijo `statisticalFiltering()` za statistično filtriranje slike, ki sprejme vhodno sliko `iImage`, dolžino `iLength` stranice kvadratnega jedra velikosti $N \times N$ ter statistično funkcijo `iFunc` za izračun statistike znotraj področja vhodne slike, ki ga podaja velikost jedra. Funkcija bo zelo podobna funkciji `kernelFiltering()`, ki smo jo že napisali v okviru Naloge 2. Zapišimo najprej funkcijo ter komentirajmo spremembe:

```
def statisticalFiltering(iImage, iLength, iFunc):

  '''
  Statistično filtriranje.
  '''

  # inicializacija filtrirane slike in dolžine jedra filtra
  oImage = np.zeros_like(iImage)
  n = int((iLength-1)/2)

  # razširi slikovno domeno
  iImage_padded = np.pad(iImage, n, mode='edge')

  # filtriranje slike
  for y in range(iImage.shape[0]):
    for x in range(iImage.shape[1]):
      iArea = iImage_padded[y:y+2*n+1, x:x+2*n+1]
      oImage[y, x] = iFunc(iArea)

  return oImage
```

Manjša sprememba je pri definiciji parametra `n`, saj tokrat dolžino stranice jedra N dobimo v obliki parametra `iLength`, torej klic `iKernel.shape[0]` nadomestimo z `iLength`.

Odstranili smo vrstico rotacije jedra filtra, saj v tem primeru jedra kot linearnega operatorja ne uporabljamo več.

Funkcijia `statisticalFiltering()` se razlikuje le še v jedru zanke `for`, kjer sprva naslovimo področje okvirjene originalne slike `iImage_padded` ter ga shranimo v spremenljivko `iArea`, katero kot vhodni parameter podamo v poljubno statistično funkcijo `iFunc`. S statistično funkcijo `iFunc` torej izvedemo postopek filtriranja.

Naloga 6

Za statistično filtriranje s filtrom mediane, maksimalne vrednosti in minimalne vrednosti v testni blok zapišemo sledeče:

```
# statistično filtriranje z vrednostjo mediane
med_I = statisticalFiltering(I, 3, np.median)
displayImage(med_I, 'Statistično filtriranje - median')
# statistično filtriranje z maksimalno vrednostjo
max_I = statisticalFiltering(I, 3, np.max)
displayImage(max_I, 'Statistično filtriranje - max')
# statistično filtriranje z minimalno vrednostjo
min_I = statisticalFiltering(I, 3, np.min)
displayImage(min_I, 'Statistično filtriranje - min')
```

Glede na to, da statistične operacije izvajamo na sivinskih vrednostih slikovnih elementov, nam dinamičnega območja ne bo potrebno prilagajati, zato enostavno vsako izhodno sliko po filtriranju shranimo v spremenljivko in jo prikažemo s funkcijo `displayImage()`.

Naloga 7

To nalogo smo opravili že v okviru Naloge 3.