domaci3_19_135

January 5, 2023

[1]:
```
USE_WIDGETS = False
if USE_WIDGETS:
    %matplotlib widget
else:
    %matplotlib inline

import scipy
import skimage
from pylab import *
import numpy as np
from skimage import exposure
```

1 Zadatak 1

Da bismo detektovali rezultat dobijen bacanjem kockica, potrebno je izvršiti segmentaciju. Potrebno je izdvojiti samo plave i samo crvene delove slike. Kada bismo to pokušali da uradimo nad **RGB** sistemom, izvukli bismo i piksele pozadine, jer se i u njima nalaze i crvena i plava boja. Najjednostavnije je preći iz **RGB** sistema u **HSV** sistem.

S komponenta nam govori nivo saturacije bojom. Kako je jedino obojen rezultat, poređenjem **S** komponente sa pragom izdvajamo obojene delove slike. Da bismo razlikovali boje potrebno je primeniti poređenje sa pragom na **H** komponentu koja sadrži boje.

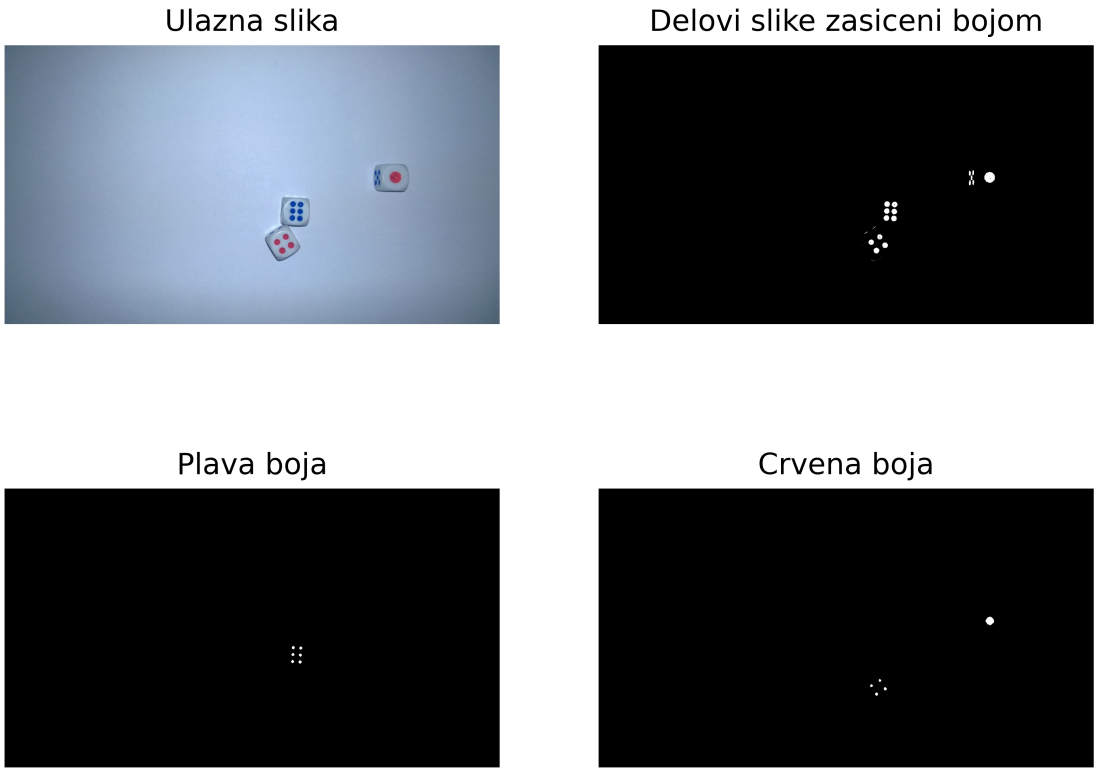
Problem koji može da se pojavi prilikom detekcije saturacije jeste **"lažni"** rezultat. Zbog toga što kamera gleda na kockicu pod nekim uglom, kamera može detektovati i tačkice na bočnoj strani kockice, kao što je prikazano na testnoj slici 6. Za otklanjanje ovakvog problema se koriste morfološke transformacije. Ako bismo samo primenili eroziju da uklonimo **"lažni"** rezultat, mogli bismo da narušimo **"pravi"** rezultat. Zato je pre erozije izvršena dilatacija. Paramteri erozije i dilatacije su izabrani tako da algoritam radi za što veći broj testnih slika.

Za detektovanje povezanih piksela na slici se koriste funkcije **skimage.measure.label** i **skimage.measure.regionprops**. Druga funkcija vraća niz. Svaki element niza predstavlja detektovan objekat. Dužina niza odgovara broju detektovanih objekata, u ovom slučaju kružića koji čine rezultat.

[2]:
```
import cv2

imgRGB = skimage.img_as_float(imread('sekvence/dices6.jpg'))
# Converting RGB to HSV color system
```

```
imgHSV = skimage.color.rgb2hsv(imgRGB)

# Thresholding S component to get color saturated parts of an image
score = np.zeros(imgHSV[:, :, 1].shape)
score[imgHSV[:, :, 1] > 0.55] = 1

# Thresholding H component, to separate blue and red parts of an image
blue_score = score.copy()
red_score = score.copy()
blue_score[imgHSV[:, :, 0] > 0.7] = 0
red_score[imgHSV[:, :, 0] <= 0.7] = 0

# Defining kernels for dilatation and erosion
kernel1 = np.ones((4, 4))
kernel2 = np.ones((9, 9))

blue_score = cv2.dilate(blue_score, kernel1)
blue_score = cv2.erode(blue_score, kernel2)

red_score = cv2.dilate(red_score, kernel1)
red_score = cv2.erode(red_score, kernel2)

# Labeling conected structures in an image
labeled_blue_score = skimage.measure.label(blue_score, background = 0)
regions_blue_score = skimage.measure.regionprops(labeled_blue_score)
print("Blue score: ", len(regions_blue_score))

# Labeling conected structures in an image
labeled_red_score = skimage.measure.label(red_score, background = 0)
regions_red_score = skimage.measure.regionprops(labeled_red_score)
print("Red score: ", len(regions_red_score))

# Ploting images
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (8, 6), dpi = 600)
ax = axes.ravel()
ax[0].imshow(imgRGB); ax[0].set_axis_off(); ax[0].set_title('Ulazna slika')
ax[1].imshow(score, cmap = 'gray'); ax[1].set_axis_off(); ax[1].
 ↪set_title("Delovi slike zasiceni bojom")
ax[2].imshow(blue_score, cmap = 'gray'); ax[2].set_axis_off(); ax[2].
 ↪set_title('Plava boja');
ax[3].imshow(red_score, cmap = 'gray'); ax[3].set_axis_off(); ax[3].
 ↪set_title('Crvena boja');
```

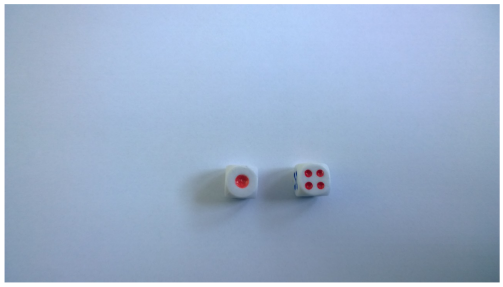
```
Blue score:  6
Red score:  5
```



Testiranjem se može videti da algoritam ne radi kako treba na trećoj slici. Algoritam detektuje jednu plavu tačku, dok plavih kockica nema na slici. Plotovanjem nulte ravni **HSV** slike vidi se da unutar crvenih tačkica postoje plave boje. Unutar crvene tačkice, koja odgovara kockici čiji je razultat 1, se nalazi veliki broj plavih piksela. Zadati parametri erozije ne mogu potisnuti te piksele. Ukoliko bi se parametri povećali, narušilo bi se detektovanje u ostalim slikama.

[6]:
```
imgRGB = skimage.img_as_float(imread('sekvence/dices3.jpg'))
# Converting RGB to HSV color system
imgHSV = skimage.color.rgb2hsv(imgRGB)

fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (8, 6), dpi = 600)
ax = axes.ravel()
ax[0].imshow(imgRGB); ax[0].set_axis_off(); ax[0].set_title('Ulazna slika')
ax[1].imshow(imgHSV[390:500, 500:900, 0], cmap = 'jet'); ax[1].set_axis_off();␣
 ↪ax[1].set_title("Delovi slike zasiceni bojom");
```

Ulazna slika



Delovi slike zasiceni bojom

Funkcija **skimage.measure.regionprops** vraća niz objekata klase koja, između ostalog, ima osobinu nalaženja površine detektovane strukture na slici. Kako je površina **"lažnog rezultata"** dosta manja od površina **"pravih rezultata"**, možemo porediti tačke sa još jednim pragom. Ukoliko je površina detektovane strukture manja od definisane vrednosti, u ovom slučaju 10, taj objekat se izbacuje iz liste.

[7]:
```
def extract_dice_score(imgRGB):
    '''
        Extracting, separately, blue and red dice score.

        This function supports RGB images.

        Input
            I     : Input image, double ([0,1]), RGB.

        Output
            blue_score    : Number of blue dots.
            red_score     : Number of red dots.
    '''
    # Converting RGB to HSV color system
    imgHSV = skimage.color.rgb2hsv(imgRGB)

    # Thresholding S component to get color saturated parts of an image
    score = np.zeros(imgHSV[:, :, 1].shape)
    score[imgHSV[:, :, 1] > 0.55] = 1

    # Thresholding H component, to separate blue and red parts of an image
    blue_score = score.copy()
    red_score = score.copy()
    blue_score[imgHSV[:, :, 0] > 0.7] = 0
    red_score[imgHSV[:, :, 0] <= 0.7] = 0

    # Defining kernels for dilatation and erosion
    kernel1 = np.ones((4, 4))
    kernel2 = np.ones((9, 9))
```

```
    blue_score = cv2.dilate(blue_score, kernel1)
    blue_score = cv2.erode(blue_score, kernel2)

    red_score = cv2.dilate(red_score, kernel1)
    red_score = cv2.erode(red_score, kernel2)

    # Labeling conected structures in an image
    labeled_blue_score = skimage.measure.label(blue_score, background = 0)
    regions_blue_score = skimage.measure.regionprops(labeled_blue_score)

    # Labeling conected structures in an image
    labeled_red_score = skimage.measure.label(red_score, background = 0)
    regions_red_score = skimage.measure.regionprops(labeled_red_score)

    # Checking if area of detected blue object in an image is large enough,
    # if it is not that object is rejected.
    for region in regions_blue_score:
        if region.area < 10:
            regions_blue_score = list(regions_blue_score)
            regions_blue_score.remove(region)

    # Checking if area of detected red object in an image is large enough,
    # if it is not that object is rejected.
    for region in regions_red_score:
        if region.area < 10:
            regions_red_score = list(regions_red_score)
            regions_red_score.remove(region)

    blue_score = len(regions_blue_score)
    red_score = len(regions_red_score)

    return blue_score, red_score
```

Rezultati testiranja.

[8]:
```
images = ['sekvence/dices1.jpg', 'sekvence/dices2.jpg', 'sekvence/dices3.jpg',␣
↪'sekvence/dices4.jpg', 'sekvence/dices5.jpg', 'sekvence/dices6.jpg',
             'sekvence/dices7.jpg', 'sekvence/dices8.jpg', 'sekvence/dices9.
↪jpg', 'sekvence/dices10.jpg', 'sekvence/dices11.jpg', 'sekvence/dices12.jpg']

for image in images:
    imgRGB = skimage.img_as_float(imread(image))
    blue_score, red_score = extract_dice_score(imgRGB)
    print("Test image:", image)
    print("Blue score:", blue_score)
    print("Red score:", red_score, '\n');
```

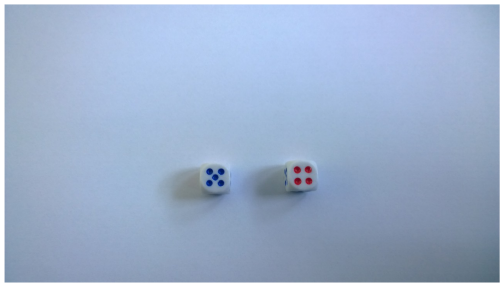
```
Test image: sekvence/dices1.jpg
Blue score: 8
Red score: 0

Test image: sekvence/dices2.jpg
Blue score: 5
Red score: 4

Test image: sekvence/dices3.jpg
Blue score: 0
Red score: 5

Test image: sekvence/dices4.jpg
Blue score: 10
Red score: 0

Test image: sekvence/dices5.jpg
Blue score: 6
Red score: 4

Test image: sekvence/dices6.jpg
Blue score: 6
Red score: 5

Test image: sekvence/dices7.jpg
Blue score: 6
Red score: 0

Test image: sekvence/dices8.jpg
Blue score: 9
Red score: 4

Test image: sekvence/dices9.jpg
Blue score: 8
Red score: 0

Test image: sekvence/dices10.jpg
Blue score: 10
Red score: 1

Test image: sekvence/dices11.jpg
Blue score: 5
Red score: 0

Test image: sekvence/dices12.jpg
Blue score: 5
Red score: 4
```

Potrebno je detektovati svaku kockicu zasebno. To se može postići morfolškom transformacijom dilatacije sa takvim parametrima da se tačkice koje pripadaju jednoj kockici spoje. Funkcija **skimage.measure.label** će detektovati sve spojene piksele, što su u ovom slučaju kockice.

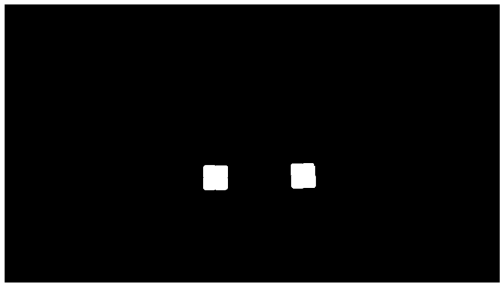
Objekat koji vraća funkcija **skimage.measure.regionprops** pripada klasi koja ima osobinu da nađe koordinate kojima pripada detektovana struktura. Na taj način se originalna slika može iseći tako da sadrži samo jednu kockicu.

[10]:
```
imgRGB = skimage.img_as_float(imread('sekvence/dices2.jpg'))
# Converting RGB to HSV color system
imgHSV = skimage.color.rgb2hsv(imgRGB)

# Thresholding S component to get color saturated parts of an image
dices = np.zeros(imgHSV[:, :, 1].shape)
dices[imgHSV[:, :, 1] > 0.6] = 1

# Defining kernels for dilatation and erosion
kernel1 = np.ones((4, 4))
kernel2 = np.ones((9, 9))
dices = cv2.dilate(dices, kernel1)
dices = cv2.erode(dices, kernel2)

# Defining kernel for dilatation that big to connect
# every dot on on dice
kernel3 = np.ones((25, 25))
dices = cv2.dilate(dices, kernel3)

# Labeling conected structures in an image
labeled_dices = skimage.measure.label(dices, background=0)
regions_dices = skimage.measure.regionprops(labeled_dices)

slice_1 = regions_dices[0].slice
slice_2 = regions_dices[1].slice

# Ploting images
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (8, 6), dpi = 600)
ax = axes.ravel()
ax[0].imshow(imgRGB); ax[0].set_axis_off(); ax[0].set_title('Ulazna slika')
ax[1].imshow(dices, cmap = 'gray'); ax[1].set_axis_off(); ax[1].
 ↪set_title("Binarizacija slike")
ax[2].imshow(imgRGB[slice_1]); ax[2].set_axis_off(); ax[2].set_title('Isecena␣
 ↪jedna kockica iz slike');
ax[3].imshow(imgRGB[slice_2]); ax[3].set_axis_off(); ax[3].set_title('Isecena␣
 ↪jedna kockica iz slike');
```

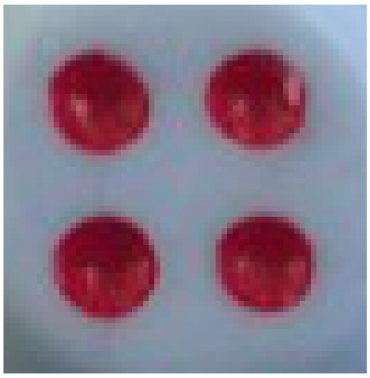
Ulazna slika



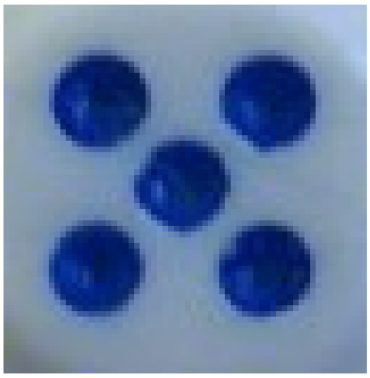
Binarizacija slike



Isecena jedna kockica iz slike



Isecena jedna kockica iz slike



Ukoliko se funkciji **extract_dice_score** prosledi tako isečena slika, ona će vratiti rezultat plave i crvene kockice zasebno. Kako se funkciji prosleđuje slika na kojoj se nalazi samo jedna kockica, jedan vraćeni podatak će biti 0. Na taj način ćemo detektovani rezultat svake kockice zasebno, i pritom ćemo znati da li je detektovana plava ili crvena kockica.

[11]:
```
def extract_dice_score_bonus(imgRGB):
    '''
        Extracting score, separately for every dice.

        This function supports RGB images.

        Input
            I     : Input image, double ([0,1]), RGB.

        Output
            dices_score    : 2D array that contains score for every dice.
                             First column represents blue score and second red␣
↪score.
    '''
    # Converting RGB to HSV color system
    imgHSV = skimage.color.rgb2hsv(imgRGB)
```

```
    # Thresholding S component to get color saturated parts of an image
    dices = np.zeros(imgHSV[:, :, 1].shape)
    dices[imgHSV[:, :, 1] > 0.6] = 1

    # Defining kernels for dilatation and erosion
    kernel1 = np.ones((4, 4))
    kernel2 = np.ones((9, 9))
    dices = cv2.dilate(dices, kernel1)
    dices = cv2.erode(dices, kernel2)

    # Defining kernel for dilatation that big to connect
    # every dot on on dice
    kernel3 = np.ones((25, 25))
    dices = cv2.dilate(dices, kernel3)

    # Labeling conected structures in an image
    labeled_dices = skimage.measure.label(dices, background=0)
    regions_dices = skimage.measure.regionprops(labeled_dices)

    # Score of every dice
    dices_score = []

    # Calling extract_dice_score function for every detected object.
    # Cropped original image is sent to extract_dice_score function to
    # separate only svore on one dice
    for i in range(len(regions_dices)):
        i_slice = regions_dices[i].slice
        dices_score.append(extract_dice_score(imgRGB[i_slice]))

    return dices_score
```

Rezultati testiranja.

[14]:
```
images = ['sekvence/dices1.jpg', 'sekvence/dices2.jpg', 'sekvence/dices3.jpg',␣
 ↪'sekvence/dices4.jpg', 'sekvence/dices5.jpg', 'sekvence/dices6.jpg',
             'sekvence/dices7.jpg', 'sekvence/dices8.jpg', 'sekvence/dices9.
 ↪jpg', 'sekvence/dices10.jpg', 'sekvence/dices11.jpg', 'sekvence/dices12.jpg']

for image in images:
    imgRGB = skimage.img_as_float(imread(image))
    dices_score = extract_dice_score_bonus(imgRGB)
    print("Test image:", image)
    for i in range(len(dices_score)):
        if dices_score[i][0] != 0:
            print("Blue score:", dices_score[i][0])
        elif dices_score[i][1] != 0:
```

```
            print("Red score:", dices_score[i][1])
    print('\n');
```

```
Test image: sekvence/dices1.jpg
Blue score: 6
Blue score: 2


Test image: sekvence/dices2.jpg
Red score: 4
Blue score: 5


Test image: sekvence/dices3.jpg
Red score: 4
Red score: 1


Test image: sekvence/dices4.jpg
Blue score: 5
Blue score: 5


Test image: sekvence/dices5.jpg
Blue score: 6
Red score: 4


Test image: sekvence/dices6.jpg
Red score: 1
Blue score: 6
Red score: 4


Test image: sekvence/dices7.jpg
Blue score: 6


Test image: sekvence/dices8.jpg
Red score: 4
Blue score: 6
Blue score: 3


Test image: sekvence/dices9.jpg
Blue score: 3
Blue score: 3
```

```
Blue score: 2


Test image: sekvence/dices10.jpg
Blue score: 5
Blue score: 5
Red score: 1


Test image: sekvence/dices11.jpg
Blue score: 3
Blue score: 2


Test image: sekvence/dices12.jpg
Red score: 4
Blue score: 5
```

2 Zadatak 2

Pomoćne funkcije. Prva funkcija proširuje matricu kako bi se odradio proces **potiskivanja lokalnih ne maksimuma** i proces **histerezisnog poređenja sa pragom**.

Druga funkcija predstavlja **Sobelov operator**, tj. izvlači gradijent slike, kao i njegovu magnitudu i ugao.

$$H_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad H_y = \begin{bmatrix} -2 & 1 & 2 \\ 0 & 0 & 0 \\ -2 & 1 & 2 \end{bmatrix}$$

Treća funkcija **kvantizuje orijentaciju gradijenta**. Koristi se maska 3x3 tako da su moguće orijentacije vertikalna, horizontalna, $+45$ stepeni i -45 stepeni.

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

[3]:
```python
def resize_array(x, radius):

    M, N = x.shape[:2]     # M je broj vrsta, N je broj kolona

    # Inicijalizacija prosirene matrice, tako da filtriranje bude moguce i na␣
↪ivicama slike
    resized_x = np.zeros((M + 2*radius, N + 2*radius), dtype = float)
```

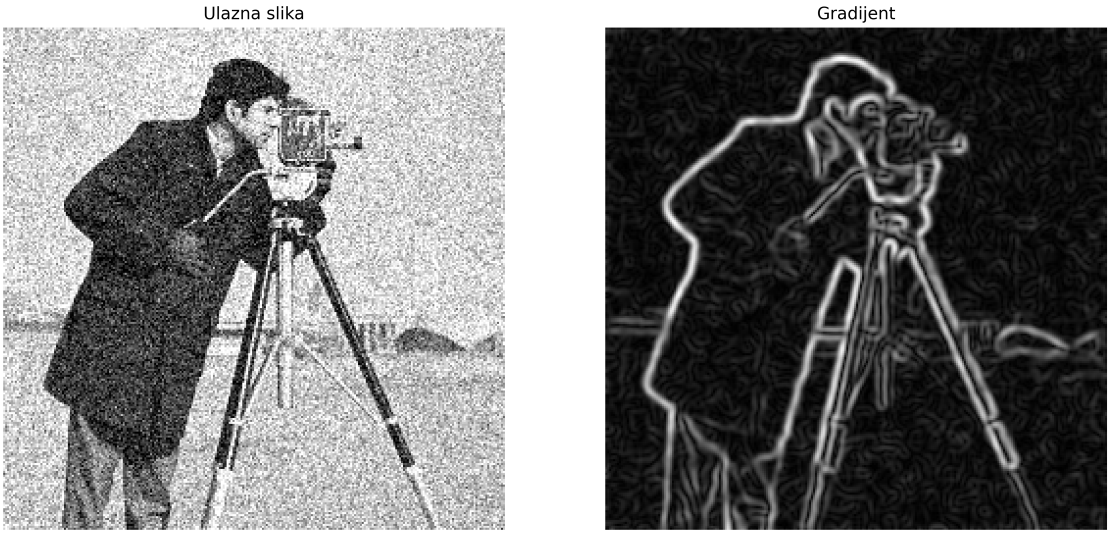
```
    # Popunjavanje coskova prosirene matrice elementima sa coskova originalne␣
↪matrice x[0,0], x[M-1,0], x[0,N-1], x[M-1,N-1]
    resized_x[0:radius, 0:radius] = x[0, 0]
    resized_x[radius+M : 2*radius+1+M, 0:radius] = x[M-1, 0]
    resized_x[0:radius, radius+N : 2*radius+1+N] = x[0, N-1]
    resized_x[radius+M : 2*radius+1+M, radius+N : 2*radius+1+N] = x[M-1, N-1]

    # Popunjavanje sredine prosirene matrice elementima originalne matrice
    resized_x[radius : radius+M, radius : radius+N] = x

    # Popunjavanje ivica prosirene matrice elementima sa ivica originalne␣
↪matrice tako da se slikaju kao u ogledalu
    # Simetry
    resized_x[0:radius, radius:radius+N] = x[radius-1::-1][:]
    resized_x[radius:radius+M, 0:radius] = x[:,radius-1::-1]
    resized_x[radius+M:, radius:radius+N] = x[:M-radius-1:-1, :]
    resized_x[radius:radius+M, radius+N: ] = x[:, :N-radius-1:-1]

    return resized_x

def sobel_operator(image):
    Hx = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
    Hy = np.transpose(Hx)

    # Horisontal and vertical gradient
    Gx = scipy.ndimage.convolve(image, Hx, mode='nearest')
    Gy = scipy.ndimage.convolve(image, Hy, mode='nearest')

    # Magnitude and angle of gradient
    magnitude = np.sqrt(np.square(Gx) + np.square(Gy))
    angle = np.rad2deg(np.arctan2(Gy, Gx))

    return (Gx, Gy, magnitude, angle)

def check_orientation(angle):
    # Qantifying angle in 4 directions, vertical, horisontal, -45 and 45 degrees
    if (angle >= -22.5 and angle < 22.5) or ((angle >= -180 and angle < -157.5)␣
↪or (angle >= 157.5 and angle <= 180)):
        orientation = 'h'
    elif (angle >= 22.5 and angle < 67.5) or (angle >= -157.5 and angle < -112.
↪5):
        orientation = "q_n"
    elif (angle >= 67.5 and angle < 112.5) or (angle >= -112.5 and angle < -67.
↪5):
        orientation = 'v'
    elif (angle >= 112.5 and angle < 157.5) or (angle >= -67.5 and angle < -22.
↪5):
```

```
        orientation = "q_p"

    return orientation
```

Optimalan postupak detekcije ivica zašumljene slike se dobija numeričkim metodama. On je približan prvom izvodu Gausove funkcije pa će taj postupak i biti korišćen. On je ekvivalentan filtriranju Gausovom funkcijom pa onda rađenju gradijenta. Ukoliko bismo samo to uradili, ivice bi bile vrlo debele i na taj način se detektuju neke ivice koje ne postoje na slici.

Pojava lažnih ivica se najbolje može uočiti na primeru slike kamermana, s obzirom da je ta slika zašumljena velikim stepenom šuma. Lažne ivice su posledica šuma. Takođe se mogu uočiti debele ivice. Ta dva problema rešava **Kanijev** algoritam.

[4]:
```
image = skimage.img_as_float(imread('sekvence/camerman.tif'))
I_filtered = skimage.filters.gaussian(image, sigma = 1.6, truncate = 7/1.6)
I_Gx, I_Gy, I_magnitude, I_angle = sobel_operator(I_filtered)

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14,12), dpi=500)
ax = axes.ravel()
ax[0].imshow(image, cmap='gray'); ax[0].set_axis_off(); ax[0].set_title('Ulazna␣
 ↪slika')
ax[1].imshow(I_magnitude, cmap='gray'); ax[1].set_axis_off(); ax[1].
 ↪set_title('Gradijent');
```



Da bismo istanjili ivice, Kanijev algoritam kvantizuje orijentaciju gradijenta. Ukoliko se koristi maska $3x3$, postoje četiri orijentacije gradijenta, vertikalna, horizontalna, $+45$ i -45 stepeni. Ukoliko je orijentacija gradijenta vertikalna, orijentacija ivice kojoj odgovara taj gradijent je horizontalna. Postupak je sledeći. Ukoliko se u orijentaciji gradijenta u lokalnom suisedstvu ($3x3$) nalazi neki piksel čiji je gradijent veći od centralnog piksela, za vrednost centralnog piksela se uzima 0, u suprotnom se uzima magnituda gradijenta koja odgovara centralnom pikselu.

Ostaje problem detekcije lažnih ivica. Ako bismo uzeli samo jedan prag sa kojim bismo poredili magnitude gradijenta, izgubili bismo neke prave ivice, ili bismo propustili neke lažne. **Kanijev** algoritam koristi dva praga, jedan veći tako da sigurno propustimo samo prave ivice, i jedan niži, koji propušta i prave ivice, koje ne priradaju izlazu poređenja većim pragom, i neke lažne ivice. Ivice dobijene poređenjem sa većim pragom se automacki proglašavaju validnim ivicama. Da bismo odredili validne ivice iz mape slabih ivica, potrebno je za svaki piksel vrednosti 1 u mapi slabih ivica proveriti da li se u lokalnom susedstvu $3x3$ nalazi i neka od jakih ivica, ako se nalazi, dati piksel je validan, briše se iz mape slabih ivica i upisuje se u mapu jakih ivica. Ovaj postupak se ponavlja sve dok broj ivica u mapi jakih ivica ne ostane konstantan u celoj iteraciji.

[5]:
```
def canny_edge_detection(I, sigma, TL, TH):

    '''
        Canny Edge Detection implementation. This algorithm detects edges in␣
↪image.

        This function supports Gray images.

        Input
            I     : Input image, uint8 ([0,255]) or double ([0,1]).

            sigma : Standard deviation of gaussian function. Gausiian function␣
↪is used
                    for clearing an image from noise because gradient is␣
↪sensitive to noise.

            TL    : Absolut higher threshold.

            TH    : Absolut lower threshold.

        Output
            Iout : Output image same size as input image with only true edges␣
↪in original image.
    '''

    # Calculating size of Gaussian filter
    if(np.ceil(6 * sigma) % 2 == 0):
        size = np.ceil(6 * sigma) + 1
    else:
        size = np.ceil(6 * sigma)

    # Gaussian filtering
    I_filtered = skimage.filters.gaussian(I, sigma = sigma, truncate = size /␣
↪sigma)
    # Gradient
    I_Gx, I_Gy, I_magnitude, I_angle = sobel_operator(I_filtered)
```

```
    # Dimensions
    N, M = I.shape
    # Initialisation of matrix that contains thin edges
    I_thin = np.zeros((N, M))
    # Resised matrix that contains gradient magnitude of an inage
    I_magnitude_resised = resize_array(I_magnitude, 1)

    for i in range(N):
        for j in range(M):
            # Qantifying orientation
            orientation = check_orientation(I_angle[i, j])

            # If orientation is horisontal. Every pixel that is not local␣
↪maximum
            # in horisontal orientation, gets replaced by 0, otherwise it's␣
↪value is kept.
            if (orientation == 'h'):
                if(I_magnitude_resised[i + 1, j + 1] ==␣
↪max(I_magnitude_resised[i + 1, j + 1], I_magnitude_resised[i, j + 1],␣
↪I_magnitude_resised[i + 2, j + 1])):
                    I_thin[i, j] = I_magnitude_resised[i + 1, j + 1]
                else:
                    I_thin[i, j] = 0

            # If orientation is vertical. Every pixel that is not local maximum
            # in vertical orientation, gets replaced by 0, otherwise it's value␣
↪is kept.
            if (orientation == 'v'):
                if (I_magnitude_resised[i + 1, j + 1] ==␣
↪max(I_magnitude_resised[i + 1, j + 1], I_magnitude_resised[i + 1, j],␣
↪I_magnitude_resised[i + 1, j + 2])):
                    I_thin[i, j] = I_magnitude_resised[i + 1, j + 1]
                else:
                    I_thin[i, j] = 0

            # If orientation is -45 degrees. Every pixel that is not local␣
↪maximum
            # in -45 degrees orientation, gets replaced by 0, otherwise it's␣
↪value is kept.
            if (orientation == "q_n"):
                if (I_magnitude_resised[i + 1, j + 1] ==␣
↪max(I_magnitude_resised[i + 1, j + 1], I_magnitude_resised[i, j],␣
↪I_magnitude_resised[i + 2, j + 2])):
                    I_thin[i, j] = I_magnitude_resised[i + 1, j + 1]
                else:
                    I_thin[i, j] = 0
```

```
            # If orientation is +45 degrees. Every pixel that is not local␣
↪maximum
            # in +45 degrees orientation, gets replaced by 0, otherwise it's␣
↪value is kept.
            if (orientation == "q_p"):
                if (I_magnitude_resised[i + 1, j + 1] ==␣
↪max(I_magnitude_resised[i + 1, j + 1], I_magnitude_resised[i + 2, j],␣
↪I_magnitude_resised[i, j + 2])):
                    I_thin[i, j] = I_magnitude_resised[i + 1, j + 1]
                else:
                    I_thin[i, j] = 0

    # Strong edge pixels
    I_H = np.zeros(I_thin.shape)
    # Weak edge pixels
    I_L = np.zeros(I_thin.shape)


    I_H[I_thin >= TH] = 1
    I_L[I_thin >= TL] = 1
    I_L = I_L - I_H

    r = 1
    I_H_resized = resize_array(I_H, r)
    I_L_resized = resize_array(I_L, r)

    # Checking if there are strong edges in local area of weak edges and
    # traversing until the number of edges is unchanged in iteratuiron
    cnt = 0
    while(True):
        tmp_outer = 0
        for i in range(N):
            for j in range(M):
                if (I_L_resized[i + r, j + r] == 1):
                    tmp_iner = 0
                    for k in range(i, i + 2 * r + 1):
                        for p in range(j, j + 2 * r + 1):
                            if (I_H_resized[k, p] == 1):
                                # Updating strong edges matrix
                                I_H_resized[i + r, j + r] = 1
                                # Updating weak edges matrix
                                I_L_resized[i + r, j + r] = 0
                                tmp_iner = 1
                                tmp_outer = 1
                                break
                        if tmp_iner == 1:
```

```
                            break
        cnt += 1
        if tmp_outer == 0:
            print("Number of iterations in which weak edges are recognized as␣
↪valid:", cnt)
            break

    edges = I_H_resized[r : N + r + 1, r : M + r + 1]

    return edges, I_H, I_L, I_magnitude, I_filtered
```

[6]:
```
def print_images(image, I_magnitude, I_filtered,  I_out, I_L, I_H):
    # Iscrtavanje slika i njohovih frekvencijskih karakteristika.
    fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(14,18), dpi=600)
    ax = axes.ravel()
    ax[0].imshow(image, cmap='gray'); ax[0].set_axis_off(); ax[0].
↪set_title('Ulazna slika')
    ax[1].imshow(I_filtered, cmap='gray'); ax[1].set_axis_off(); ax[1].
↪set_title('Gausov filtar');
    ax[2].imshow(I_L, cmap='gray'); ax[2].set_axis_off(); ax[2].
↪set_title('Slabe ivice')
    ax[3].imshow(I_H, cmap='gray'); ax[3].set_axis_off(); ax[3].set_title('Jake␣
↪ivice');
    ax[4].imshow(I_magnitude, cmap='gray'); ax[4].set_axis_off(); ax[4].
↪set_title('Sobelov operator')
    ax[5].imshow(I_out, cmap='gray'); ax[5].set_axis_off(); ax[5].
↪set_title('Kanijev algoritam');
```

S obzirom da slika, Lena, nije zašumljena, parametri se mogu relaksirano birati. Izabrani su tako da se što više ivica vidi.

[19]:
```
TH = 0.25
TL = TH / 3

image = skimage.img_as_float(imread('sekvence/lena.tif'))
I_out, I_H, I_L, I_magnitude, I_filtered = canny_edge_detection(image, 1.1, TL,␣
↪TH)
print_images(image, I_magnitude, I_filtered, I_out, I_L, I_H)
```

```
Number of iterations in which weak edges are recognized as valid: 66
```

Ulazna slika

Gausov filtar

Slabe ivice

Jake ivice

Sobelov operator

Kanijev algoritam

Na slici postoji ivica između dimljaka i kuće ali intenzitet gradijenta te ivice previše mali. Ako

bismo želeli i tu ivicu da detektujemo, morali bismo da smanjimo viši prag poređenja do te mere da bismo propustili i lažne ivice koje su posledica šuma. Za ovako izabrane parametre se na iylaynoj slici vide granice između senki. Parametri su birani tako da se što više pravih ivica vidi na izlazu, a da se pritom ne vide lažne ivice.

[20]:
```
TH = 0.3
TL = TH / 3

image = skimage.img_as_float(imread('sekvence/house.tif'))
I_out, I_H, I_L, I_magnitude, I_filtered = canny_edge_detection(image, 1.1, TL,␣
↪TH)
print_images(image, I_magnitude, I_filtered, I_out, I_L, I_H);
```

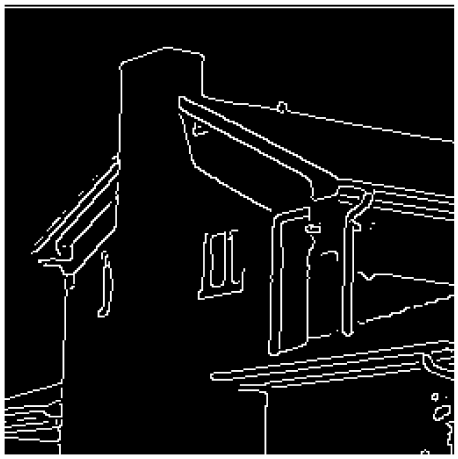
```
Number of iterations in which weak edges are recognized as valid: 12
```

Ulazna slika

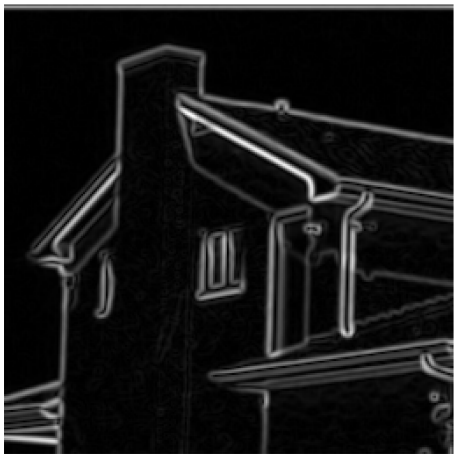
Gausov filtar

Slabe ivice

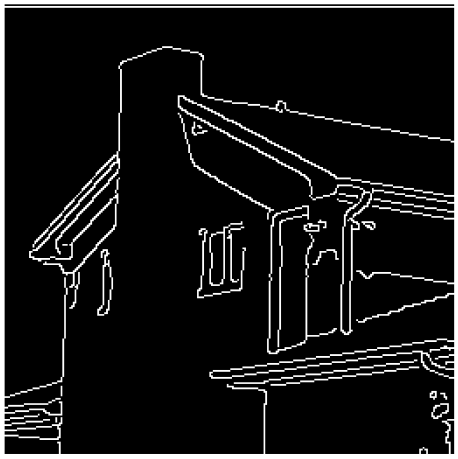
Jake ivice



Sobelov operator



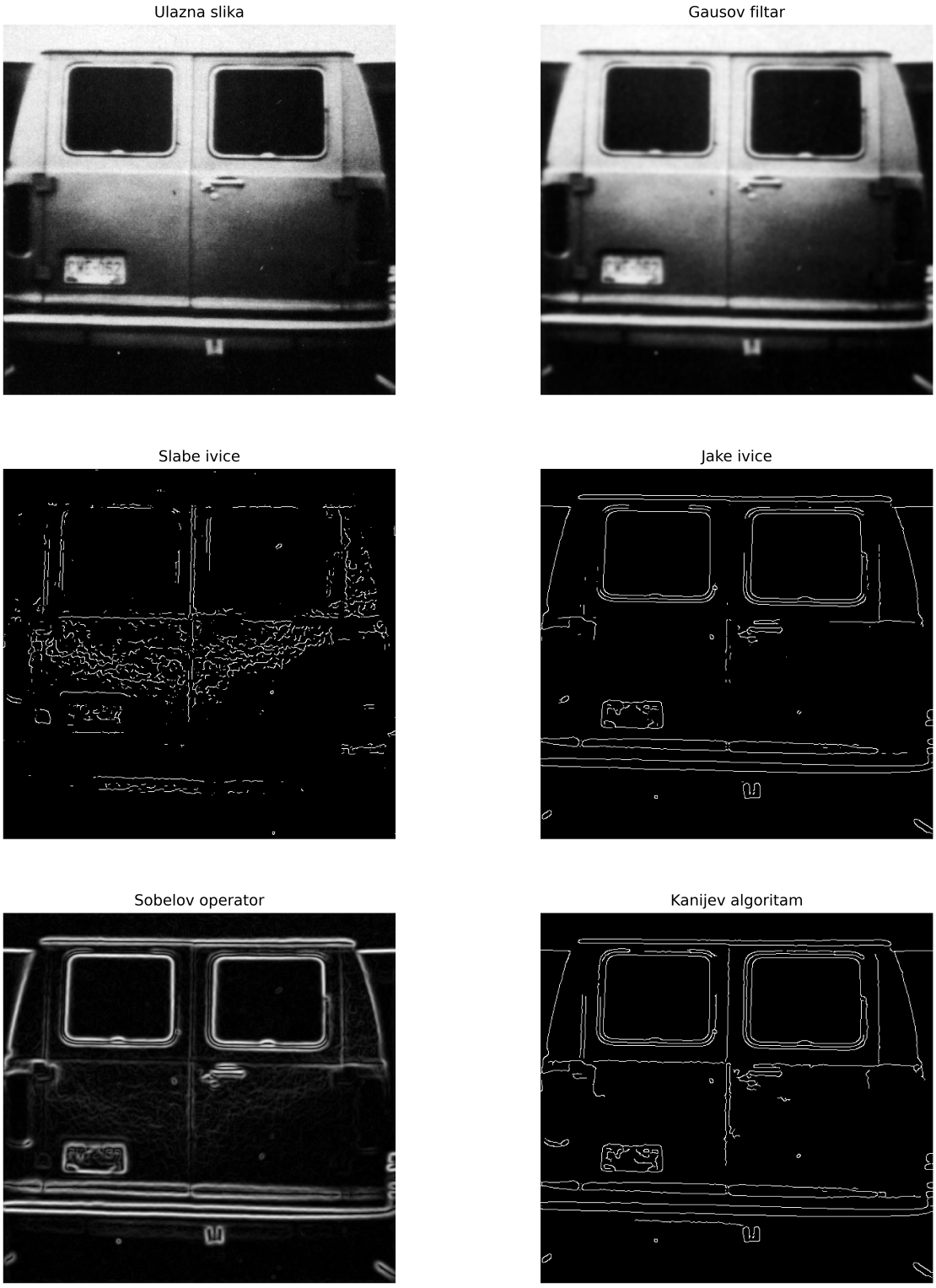
Kanijev algoritam



Slika kombija je zašumljena. Potrebno je povećati standardnu devijaciju Gausovog filtra.

[21]:
```
TH = 0.25
TL = TH / 3

image = skimage.img_as_float(imread('sekvence/van.tif'))
I_out, I_H, I_L, I_magnitude, I_filtered = canny_edge_detection(image, 1.6, TL,␣
 ↪TH)
print_images(image, I_magnitude, I_filtered, I_out, I_L, I_H);
```

```
Number of iterations in which weak edges are recognized as valid: 144
```



Slika kamermana je najzašumljenija. Potrebna je velika standardna devijacija Gausovog filtra kako bi se što veći deo šuma potisnuo. Prag je izabran tako da se u što većoj meri otklone lažne ivice.

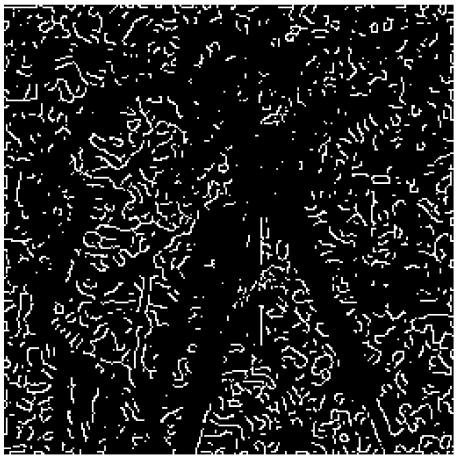
[7]:
```
TH = 0.3
TL = TH / 3

image = skimage.img_as_float(imread('sekvence/camerman.tif'))
I_out, I_H, I_L, I_magnitude, I_filtered = canny_edge_detection(image, 1.8, TL,␣
 ↪TH)
print_images(image, I_magnitude, I_filtered, I_out, I_L, I_H)
```

```
Number of iterations in which weak edges are recognized as valid: 36
```

Ulazna slika

Gausov filtar

Slabe ivice



Jake ivice

Sobelov operator

Kanijev algoritam