Izveštaj trećeg domaćeg zadatka

iz predmeta

Digitalna obrada slike

Katarina Petrović 2021/0068

1. Zadatak:

U prvom zadatku realizovan je **Canny algoritam** za detekciju ivica koji prati korake (1-7) iz postavke prvog zadatka.

1) Filtriranje ulazne slike Gausovom funkcijom:

```
#Filtriranje Gausovim filtrom - suzbijamo sum
img_gauss = filters.gaussian(img_in, sigma, truncate=3, mode='nearest')
```

2) i 3) Određivanje horizontalnih i vertikalnih gradijenta (Sobelovim operatorom), magnitude i ugla gradijenta:

```
#Maske Sobelovog filtra
Hx = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
Hy = np.transpose(Hx)

#Primenjujemo Sobelov filtar na zasumljenu sliku - dobijamo gradijente po x i y osi
Gx = ndimage.correlate(img_gauss, Hx, mode='nearest')
Gy = ndimage.correlate(img_gauss, Hy, mode='nearest')

#Magnituda i ugao gradijenta
mag = np.sqrt(np.square(Gx) + np.square(Gy))
angle = np.degrees(np.arctan2(Gy, Gx))
```

4) i 5) Kvantizacija gradijenta i potiskivanje lokalnih nemaksimuma (po pravcu gradijenta):

```
#Potiskivanje lokalnih nemaksimuma za svaki piksel i definisanje slabih i jakih ivica
for i in range(M):
    for j in range(N):
        grad_ang = angle[i, j]

        #Zavisno od ugla gradijenta definisemo koje piksele zelimo da razmatramo kao susede
        if (grad_ang<= 22.5 and grad_ang>-22.5) or grad_ang<= -157.5 or grad_ang>157.5:
            #Gornji i donji sused
            neighb_1_i, neighb_1_j = i-1, j
            neighb_2_i, neighb_2_j = i + 1, j

        elif (grad_ang>22.5 and grad_ang<=67.5) or (grad_ang>-157.5 and grad_ang<=-112.5):
            #Dijagonalni susedi
            neighb_1_i, neighb_1_j = i-1, j-1
            neighb_2_i, neighb_2_j = i + 1, j + 1


        elif (grad_ang>67.5 and grad_ang<=112.5) or (grad_ang>-112.5 and grad_ang<=-67.5):
            #Levi i desni sused
            neighb_1_i, neighb_1_j = i, j-1
            neighb_2_i, neighb_2_j = i, j + 1


        elif (grad_ang>112.5 and grad_ang<=157.5) or (grad_ang>-67.7 and grad_ang<=-22.5):
            #Dijagonalni susedi
            neighb_1_i, neighb_1_j = i-1, j + 1
            neighb_2_i, neighb_2_j = i + 1, j-1

        #Proveravamo da li je piksel "jaca ivica" od suseda -> ako nije, postavljamo njegovu vrednost na 0
        if M>neighb_1_i>= 0 and N>neighb_1_j>= 0:
            if mag[i, j]<mag[neighb_1_i, neighb_1_j]:
                mag[i, j]= 0

        if M>neighb_2_i>= 0 and N>neighb_2_j>= 0:
            if mag[i, j]<mag[neighb_2_i, neighb_2_j]:
                mag[i, j]= 0
```

6) Određivanje mapa jakih i slabih ivica:

```
#Matrica slabih ivica, jakih ivica i neivicnih piksela
weak_ids = np.zeros_like(img_in)
strong_ids = np.zeros_like(img_in)
no_edge_ids = np.zeros_like(img_in)
```

*naredni isečak koda se nalazi u istoj for petlji kao delovi 4) i 5) dok se prethodni (definisanje matrica) nalazi pre for petlje – preglednije se može videti u jupyter svesci

```
#Piksel svrstavamo u jaku ili slabu ivicu ili u neivične piksele
if mag[i, j] > thr_high:
    strong_ids[i, j] = 1
elif mag[i,j] >= thr_low and mag[i, j] <= thr_high:
    weak_ids[i, j] = 1
else:
    no_edge_ids[i, j] = 1
```

7) Uključivanje u izlaznu mapu ivica onih slabih ivica koje su povezane sa jakim ivicama:

```
#Dodajemo slabe ivice koje su povezane sa jakim ivicama u jake ivice
changes = True

#Sve dok ne udjemo u stacionarno stanje
while changes:
    changes = False

    weak_indices = np.argwhere(weak_ids)
    for i, j in weak_indices:
        #Posmatrmo okolinu slabo ivicnog piksela i u njoj trazimo jaku ivicu
        #Ako ona postoji prebacujemo piksel u jake ivice
        if np.any(strong_ids[max(0, i - 1):min(M, i + 2), max(0, j - 1):min(N, j + 2)]):
            strong_ids[i, j] = 1
            weak_ids[i, j] = 0
            changes = True


img_edges = strong_ids
```

Kao što se može pročitati iz koda, ovaj korak je iterativan i izvršava se sve dok postoje promene u matricama slabih i jakih ivica. U svakoj iteraciji, za svaki piksel iz slabih ivica proveravamo da li se u njegovoj okolini nalazi piksel koji predstavlja jaku ivicu. Ukoliko se nalazi, trenutni piksel je takođe ivični piksel i smeštamo ga u matricu jakih ivica. Ukoliko ne, ostavljamo ga u matrici slabih ivica (jer će možda u nekoj od narednih iteracija on ispuniti ovaj uslov). Nakon ovog iterativnog postupka, dobijena matrica jakih ivica predstavlja izlaznu mapu ivica.

Na kraju samog algoritma vraćamo sliku img_edges kao izlaz funkcije.

Testiranje i međurezultati funkcije <u>canny_edge_detection:</u>

Na primeru slike lena.tif sa parametrima:

- sigma = 0.8,
- thr_low = 0.3,
- thr_high = 0.5

dobijeni su međurezultati:

Ulazna slika

Slika nakon Gaussian filtra

Gradijent po x-osi

Gradijent po y-osi

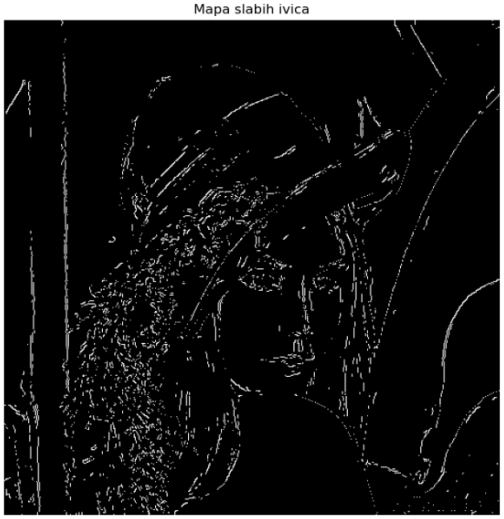
Magnituda gradijenta

Ugao gradijenta

Magnituda gradijenta nakon suzbijanja nelokalnih maksimuma

Mapa jakih ivica

Mapa slabih ivica



Izlazna slika - mapa ivica slike

Na slici lena.tif sa ovim parametrima možemo uočiti da je većina ivica (koje čovek detektuje na slici) obuhvaćena algoritmom. U algoritmu je primenjeno blago zamućenje Gausovim filtrom (sigma = 0.8) kako bi se suzbio početni šum, a vrednosti thr_low i thr_high su podešene tako da se detektuju najznačajnije ivice i suzbiju pikseli koji nisu ivični.

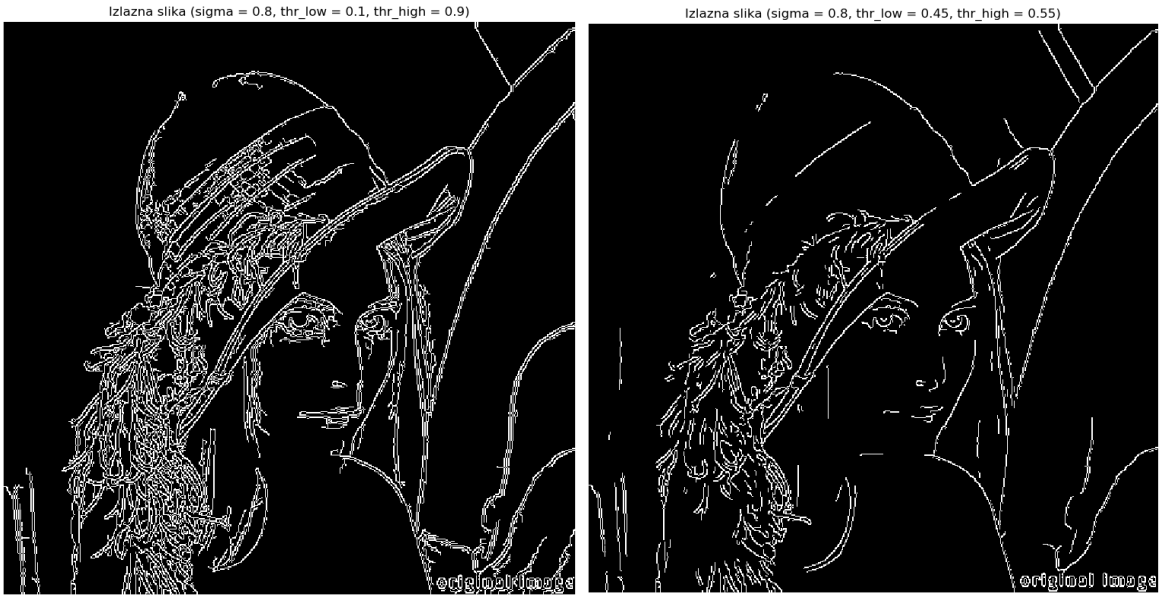
U nastakvu su prikazani rezultati algoritma sa drugim parametrima:

U slučaju **velikog zamućenja** ulazne slike, dobro suzbijamo šum, ali gubimo veliku količinu ivičnih piksela. Primećujemo sa ove slike da većina perja sa šešira, kosa, pa čak ni usta i nos nisu detektovani kada imamo veliko zamućenje.

U slučaju **malog zamućenja**, ukoliko na slici postoji šum može doći do lažno detektovanih ivičnih piksela, s obzirom da su gradijenti (prvi izvod) osetljivi na šum. Konkretno, u slučaju

slike lena.tif, malo zamućenje neće značajno štetiti, osim u delovima slike gde su neke ivice nepotrebno detektovane (na primer dosta detektovanog perja na šeširu i nasumične tačkaste ivice po slici).

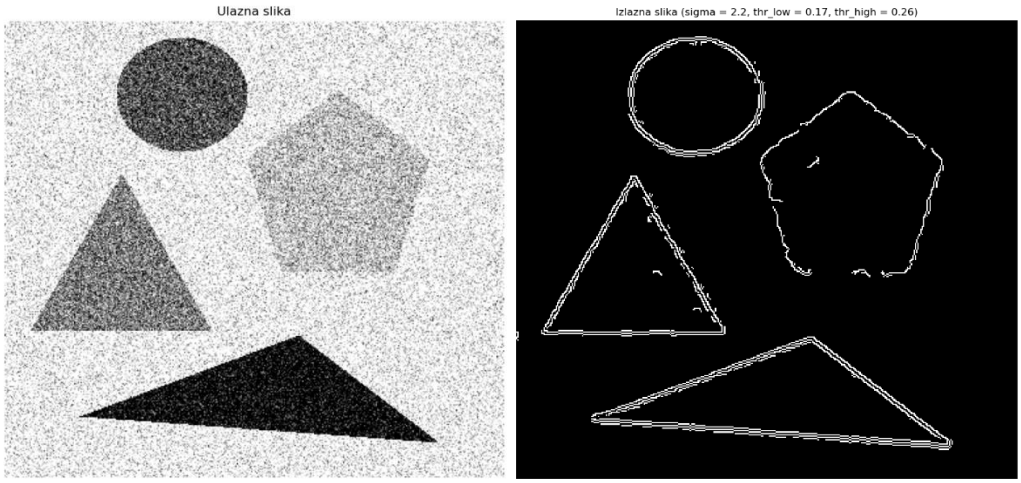
Ako u algoritmu postavimo **visoke vrednosti** za thr_low i thr_high, mnogi ivični pikseli neće biti detektovani s obzirom da nisu prešli granicu thr_low, dok u suprotnom slučaju ako ih postavimo na **niske vrednosti** dobijamo ogroman broj ivičnih piksela i gubi se odnos korisnih i beskorisnih informacija.



Ako thr_low i thr_high izaberemo tako da je njihova **razlika velika**, algoritam radi značajno sporije (zbog velikog broja slabih ivica koje iterativno povezujemo sa jakim ivicama), ali dobro izdvaja veliku količinu korisnih informacija. U suprotnom slučaju, kada je **razlika mala**, efekat povezivanja jakih i slabih ivica ne postoji i algoritam se svodi na poređenje magnitude gradijenta

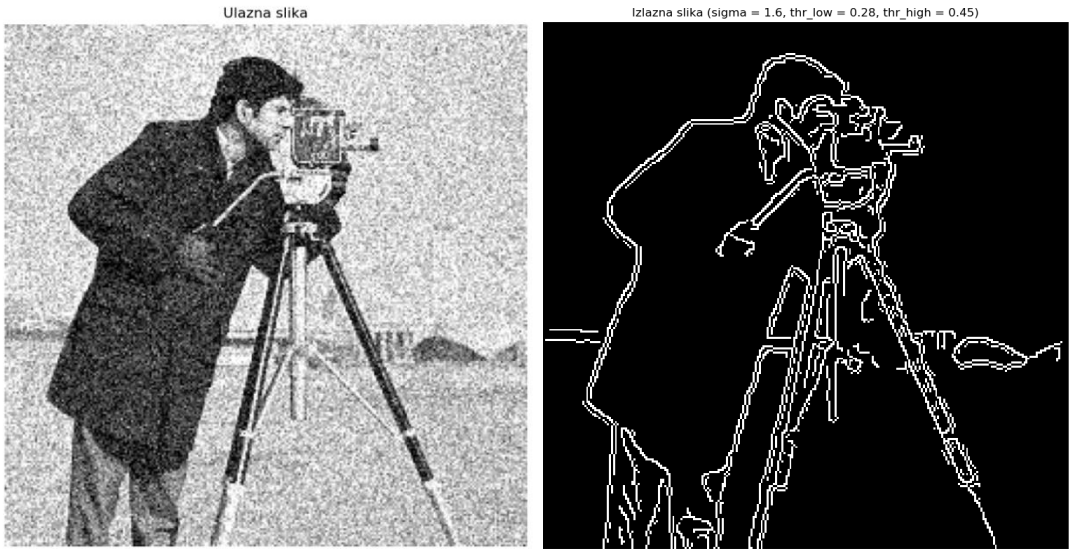
sa pragom. Tada kvalitet izdvajanja ivica zavisi od toga da li je uzet viši ili niži prag, ali generalno loše izdvaja „korisne ivice“.

Na primeru **shapes_noisy.tif** nije moguće podesiti parametre tako da se dobiju samo ivice geometrijskih oblika, bez da se uhvati i relativno velik deo šuma. Razlog za ovo je impulsni šum koji ne možemo adekvatno obraditi Gausovom funkcijom, već bi bilo neophodno izvršiti medijan filtriranje unutar funkcije. Ipak, najboljim mogućim podešavanjima slike, dobijen je sledeći rezultat:



Za ostala podešavanja parametra dolazi do različitih problema. U slučaju manjeg sigma, funkcija detektuje sve veći broj ivica koje potiču od impulsnog šuma, dok je za veliko sigma petougao na slici neprepoznatljiv zbog nedovoljnog gradijenta. Za veće vrednosti thr_low i thr_high ivice petougla na slici se ne detektuju, dok se za manje vrednosti threshold-a detektuje i šum.

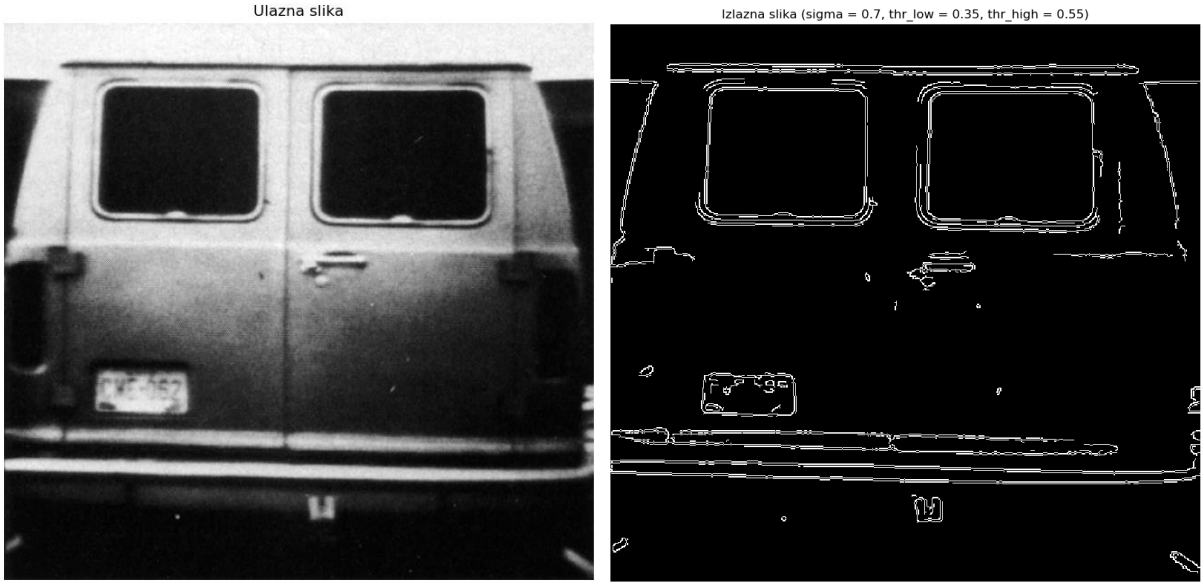
Jako sličan je problem pri obradi slike **camerman.tif** jer se i na ovoj slici pojavljuje impulsni šum koji otežava obradu i za koji opisana funkcija nema adekvatno rešenje (median filtar). Algoritam sa optimalno podešenim parametrima daje sledeći rezultat:



Takođe, kao probleme pri obradi ovih slika uočavamo loše lokalizovane ivice, odnosno pojavljuju se duple ivice na mestima gde postoji samo jedna ivica (npr. rukav i glava kamermana, ivice geometrijskih tela). Ovo je posledica velikog zamućenja slike pri Gausovom filtriranju.

Na primeru slike **house.tif**, moguće je dobiti dobro izdvojene ivice kuće. Ova slika nema veliku količinu šuma, pa je moguće staviti malo zamućenje (kako ne bi detektovali cigle na kući kao ivice), i veliki opseg threshold-a. U nastavku je prikaz detekcije ivica na ovoj slici:

Na slici **van.tif**, šum postoji, ali je moguće suzbiti ga Gausovim filtriranjem. U nastavku je prikaz detekcije ivica na ovoj slici sa optimalnim parametrima:



Na ovoj slici primećujemo neke crne i bele tačke po slici koje su verovatno posledica šuma (ili sitnih tufni po vozilu), koje ne predstavljaju ivice od značaja, ali su detektovane kao visoke

vrednosti gradijenta. Uočavamo da su obod, prozori vozila i kvaka detektovani kao ivice, a čak su detektovane i ivice slova na tablicama. U slučaju drugačije odabranih parametara, gubi se kompromis između količine šuma i korisnih ivica koje detektujemo.

2. Zadatak:

U okviru algoritma funkcije **get_line_segments** uočava se nekoliko celina koje rešavaju određene delove funkcije.

1) Obrada ulaznih podataka – obrađujemo parameter tako da u slučaju pogrešnog unosa algoritam adekvatno odreaguje:

```
#Ekstrakcija ulaznih parametara - theta je ugao, rho je najkrace rastojanje duzi od koord pocetka
theta, rho = line

#Specificni slucajevi i obrada ulaznih parametara
if theta < -np.pi/2 or theta > np.pi/2:
    print('Greska: Ugao mora biti u opsegu [-pi/2, pi/2]')
    return (0, [], [])
if min_length <= 0:
    print('Greska: Minimalna duzina segmenta mora biti pozitivna vrednost')
    return (0, [], [])
if max_gap < 0:
    print('Greska: Maksimalna rupa u segmentu mora biti pozitivna vrednost')
    return (0, [], [])
if tolerancy <= 0:
    print('Greska: Tolerancija mora biti pozitivna vrednost')
    return (0, [], [])
```

2) Za zadati pravac (preko rho i theta) proveravamo piksele na tom pravcu i u tolerancy okolini pravca da li imamo ivične piksele. Za svaki piksel proveravamo da li se nalazi na zadatom pravcu (preko distance), zatim da li je ivični piksel - ako da ubacujemo ga u segment_pot_arr niz, ako ne proveravamo da li se u njegovoj okolini nalazi ivični piksel – ako da ubacujemo ga u niz segment_pot_arr. Na kraju ovog procesa, dobijamo sve piksele koji se nalaze na zadatom pravcu segmenta (nismo ispitali za sad da li su povezani i dovoljne dužine da bi bili segment).

```
#Segment pot su svi pikseli koji su potencijalno segmenti - nalaze se na pravcu (sa tolerancijom) - za vizuelni prikaz
segment_pot = np.zeros_like(img_edges)

#Segment pot arr su svi ti pikseli samo uzete njihove (i, j) koordinate kako bi kasnije lakse manipulisali podacima
segment_pot_arr = []
M, N = img_edges.shape
#Ubacujemo piksele iz slike img_edges koji se nalaze na pravcu (sa tolerancijom)
for i in range(M):
    for j in range(N):

        #Distance je rastojanje od zadatog pravca
        distance = abs(np.round(i * np.sin(theta) + j * np.cos(theta)) - rho)

        #Ako je distanca manja od 1 - na pravcu smo
        if distance < 1:
            #Provera da li je piksel ivicni - ako jeste stavimo da pripada potencijalnom segmentu
            if img_edges[i,j] == 1:
                segment_pot[i, j] = 1
                segment_pot_arr.append([i, j])
            #Ako nije ivicni, proverimo da li u njegovoj tolerancy okolini postoji ivicni
            #Ako da, stavimo da pripada potencijalnom segmentu
            else:
                i_min = max(0, i - tolerancy)
                i_max = min(M, i + tolerancy + 1)
                j_min = max(0, j - tolerancy)
                j_max = min(N, j + tolerancy + 1)
                window = img_edges[i_min:i_max, j_min:j_max]
                if np.any(window) == True:
                    segment_pot[i,j] = 1
                    segment_pot_arr.append([i, j])
```

3) Definišemo max_gap i min_length po x i y osi i inicijalizujemo segment_coords niz:

```
#Ako nemamo nijedan piksel koji potencijalno pripada segmentu, vracamo (0, [], [])
if not segment_pot_arr:
    return (0, [], [])
#Ako imamo piksele koji ce potencijalno pripadati pikselu, inicijalizujemo segment_coords
#Prvi (potencijalni segment) ce poceti prvim pikselom iz segment_pot_arr
else:
    segment_coords = [segment_pot_arr[0][0], segment_pot_arr[0][1]]

#Vrednosti max_gap i min_length skalirane na x i y osu zavisno od ugla theta
#Koristimo abs za sin/cos da ne bi razmisljali o znakovima, bitno je samo rastojanje piksela
max_gap_x = max_gap*abs(np.cos(theta))
max_gap_y = max_gap*abs(np.sin(theta))
min_length_x = min_length*abs(np.cos(theta))
min_length_y = min_length*abs(np.sin(theta))


#Dodatak za sin/cos priblizno jednak 0
if min_length_x < 1:
    min_length_x = 0
if min_length_y < 1:
    min_length_y = 0
if max_gap_x < 1:
    max_gap_x = 0
if max_gap_y < 1:
    max_gap_y = 0
```

4) Za svaka dva susedna piksela iz segment_pot_arr proveravamo njihovo međusobno rastojanje – ako je ono veće od max_gap, oni pripadaju različitim segmentima i tada pamtimo kraj trenutnog segmenta (koordinate trenutnog piksela) i početak narednog segmenta (koordinate narednog segmenta). Posebno, kada je zadati pravac paralelan sa nekom od osa, moramo odvojeno obrađivati ove slučaje po obe ose. Početak i kraj segmenta ubacujemo u segment_coords. Takođe, na kraju poslednji piksel iz niza postavljamo kao kraj poslednjeg segmenta i ubacujemo njegove koordinate u segment_coords.

```
#Piksele koji su na vecem rastojanju od max_gap uzimamo kao da su delovi razlicitih segmenata
#Pamtimo koordinate pocetka i kraja svakog (i za sada potencijalnog) segmenta
#Trenutno ne uzimamo u obzir min_length, kasnije cemo izbacivati segmente koji su manji od min_length
for elem in range(0,len(segment_pot_arr)-1):
    #(i,j) koordinate trenutnog piksela i narednog
    i_coord = segment_pot_arr[elem][0]
    j_coord = segment_pot_arr[elem][1]
    i_next = segment_pot_arr[elem+1][0]
    j_next = segment_pot_arr[elem+1][1]

    #Rastojanje trenutnog piksela i narednog
    dist_x = abs(i_next-i_coord)
    dist_y = abs(j_next-j_coord)

    #U slucaju da je neki sin ili cos = 0 zelimo da proveravamo rastojanje samo po horizontali/vertikali
    if max_gap_x == 0:
        if dist_y > max_gap_y:
            segment_coords.append(i_coord)
            segment_coords.append(j_coord)
            segment_coords.append(i_next)
            segment_coords.append(j_next)

    elif max_gap_y == 0:
        if dist_x > max_gap_x:
            segment_coords.append(i_coord)
            segment_coords.append(j_coord)
            segment_coords.append(i_next)
            segment_coords.append(j_next)

    #Ako ugao nije nula, proveramo da li je naredni piksel dovoljno daleko po oba pravca
    #Ako da, dodajemo koordinate trenutnog piksela u segment_coords
    #To ce biti kraj trenutnog segmenta i koordinate narednog piksela - to ce biti pocetak narednog piksela
    #Ako ne, ne dodajemo ga - on je deo segmenta
    elif dist_x > max_gap_x or dist_y > max_gap_y:
        segment_coords.append(i_coord)
        segment_coords.append(j_coord)
        segment_coords.append(i_next)
        segment_coords.append(j_next)

#Dodajemo poslednji piksel iz potencijalnih segmenata - on predstavlja kraj poslednjeg segmenta
segment_coords.append(segment_pot_arr[-1][0])
segment_coords.append(segment_pot_arr[-1][1])
#Segment_coords je niz od 4*(broj_segmenata) za sad - za svaki segment imamo pocetne i krajnje koordinate
```

5) Proveravamo da li su potencijalni segmenti dovoljno dugački da bi bili detektovani kao segmenti – u segment_coords_copy ubacujemo samo one segmente koji zadovoljavaju uslov da su veći od minimalne dužine (skalirano sa sinusom i kosinusom po obe ose). Takođe pamtimo dužinu segmeta u nizu segments_size:

```python
#Segment_coords_copy ce biti lista iz koje su izbaceni elementi koji su manji od zadate duzine
segment_coords_copy = []
#Pamtimo velicine segmenata ukoliko su one vece od min_length
segments_size = []

#Proveravamo duzine segmenata
for i in range(0, len(segment_coords), 4):
    #Ako je duzina veca od zadate, u segment_coords_copy dodajemo koordinate segmenta i pamtimo duzinu segmenta u segments_size
    if abs(segment_coords[i]-segment_coords[i+2]) >= min_length_x and abs(segment_coords[i+1]-segment_coords[i+3]) >= min_lengt
        segment_coords_copy.append(segment_coords[i])
        segment_coords_copy.append(segment_coords[i+1])
        segment_coords_copy.append(segment_coords[i+2])
        segment_coords_copy.append(segment_coords[i+3])

        seg_len = np.sqrt((segment_coords[i]-segment_coords[i+2])**2 + (segment_coords[i+1]-segment_coords[i+3])**2)
        segments_size.append(np.round(seg_len))
```

6) Računamo broj segmenata kao dužinu niza segments_size i u segments_coord ubacujemo elemente niza segment_coords_copy tako da formatiramo izlaz funkcije na traženi način:

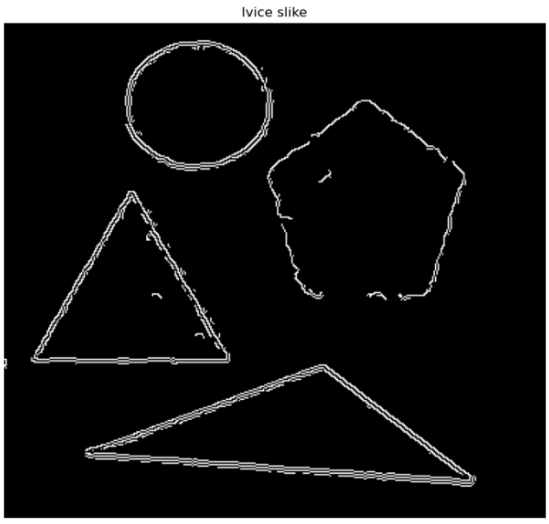
```python
segments_num = len(segments_size)
segments_coord = []
for i in range(0, len(segment_coords_copy), 4):
    segments_coord.append([(segment_coords_copy[i], segment_coords_copy[i + 1]), (segment_coords_copy[i + 2], segment_coords_co
```

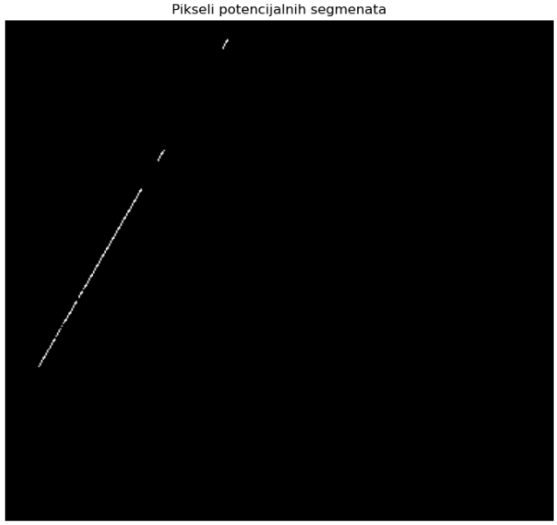
Izlaz iz ove funkcije predstavljaju segments_num, segments_size i segments_coord.

Napomena: ugao theta se računa u odnosu na vertikalnu osu na slici (kasnije će biti prikaz i na slikama sata – npr. brojevi 1 i 2 su pozitivni uglovi kazaljki, dok su 10 i 11 negativni uglovi, a 12 sati predstavlja theta = 0)

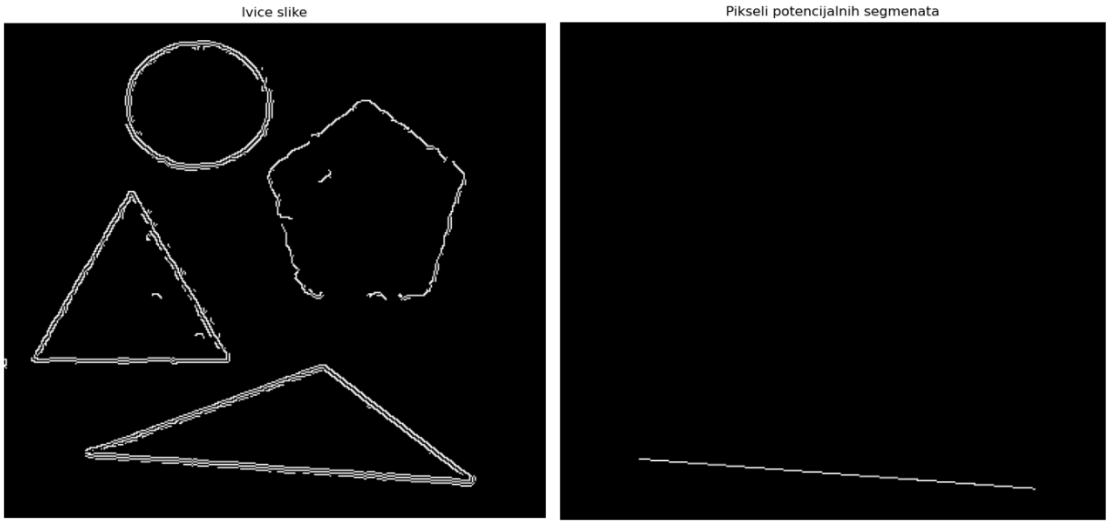
Testiranje i međurezultati funkcije get_line_segments:

Funkcija je testirana na slici shapes_noisy.tif i u nastavku su prikazani rezultati i međurezultati za različite parametre. Na slici desno nalazi se prikaz promenljive segment_pot, dok su ispod slika ulazni i izlazni parametri funkcije.

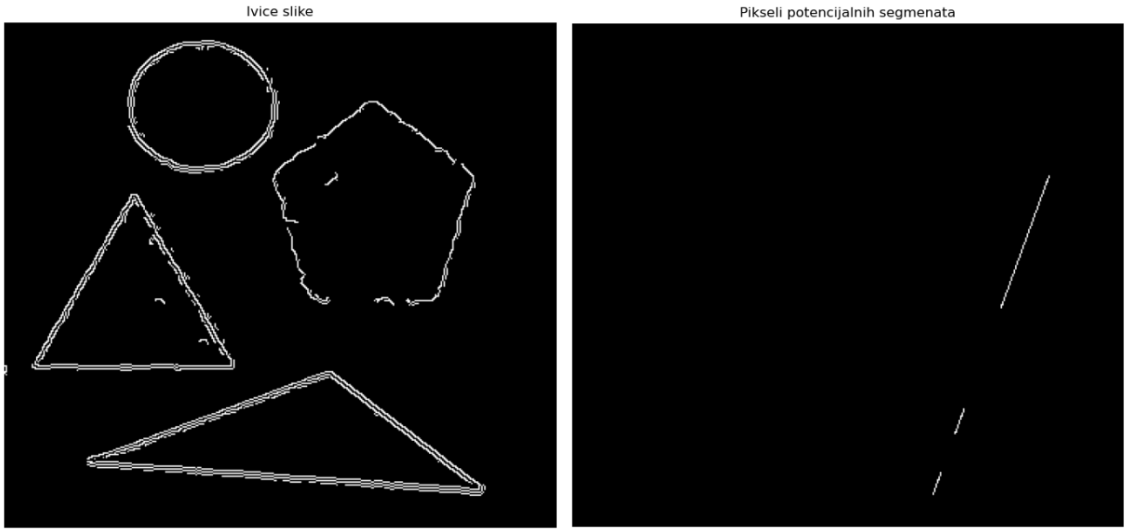





```
Parametri: theta: 0.5235987755982988 , rho: 140 , min_length: 50 , max_gap: 5 , tolerancy: 2
Broj segmenata: 1
Duzine segmenata: [142.0]
Koordinate segmenata: [[(117, 94), (240, 23)]]
```



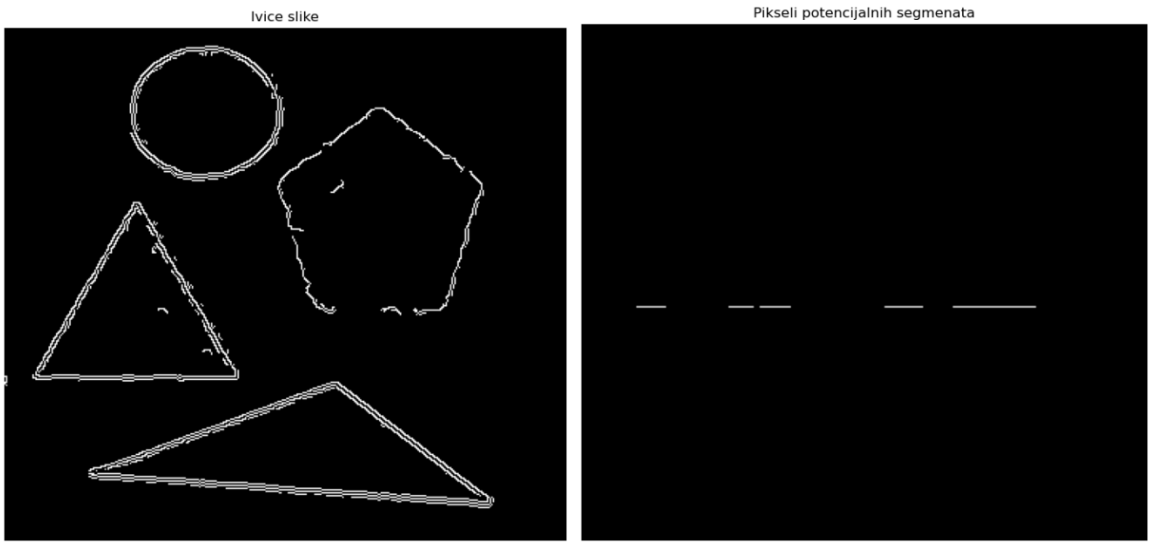

```
Parametri: theta: -1.4959965017094252 , rho: -300 , min_length: 50 , max_gap: 5 , tolerancy: 2
Broj segmenata: 1
Duzine segmenata: [277.0]
Koordinate segmenata: [[(305, 55), (326, 331)]]
```




```
Parametri: theta: 0.3490658503988659 , rho: 345 , min_length: 50 , max_gap: 5 , tolerancy: 5
Broj segmenata: 1
Duzine segmenata: [97.0]
Koordinate segmenata: [[(105, 329), (196, 296)]]
```

Na ova tri primera uočljivo je da algoritam dobro lokalizuje segmente – sa slike potencijalnih segmenata vidimo da su ivice geometrijskih tela uhvaćene, a iz broja, dužine i koordinata segmenata vidimo da su za segmente detektovane samo ivice, dok ostali pikseli koji pripadaju tom pravcu nisu.

Za ispitivanje uticaja drugih parametara, uzimamo pravac horizontalne linije koja obuhvata petougao (pogodno za prikaz različitih vrednosti).



U slučaju promene parametra **min_length** algoritam može ili hvatati više piksela kao segmente (za malo min_length) ili nedetektovati ni dugačke segmente (za veliko min_length). Demonstracija ovoga je prikazana u nastavku:

```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 15 , max_gap: 2 , tolerancy: 5
Broj segmenata: 5
Duzine segmenata: [19.0, 16.0, 20.0, 25.0, 55.0]
Koordinate segmenata: [[(190, 37), (190, 56)], [(190, 99), (190, 115)], [(190, 120), (190, 140)], [(190, 204), (190, 229)],
[(190, 250), (190, 305)]]
```

```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 25 , max_gap: 2 , tolerancy: 5
Broj segmenata: 2
Duzine segmenata: [25.0, 55.0]
Koordinate segmenata: [[(190, 204), (190, 229)], [(190, 250), (190, 305)]]
```

```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 70 , max_gap: 2 , tolerancy: 5
Broj segmenata: 0
Duzine segmenata: []
Koordinate segmenata: []
```

Uočavamo da što više povećavamo min_length broj detektovanih segmenata opada. Na slici vidimo 5 delova geometrijskih tela po ovom pravcu, a od njihove dužine zavisi (tj da li je ona veća ili manja od min_length) zavisi da li će on biti detektovan kao segment.

Ukoliko menjamo vrednosti **max_gap** menja se povezanost segmenata. Prikaz je u nastavku (odnosi se na isti horizontalni pravac sa slike):

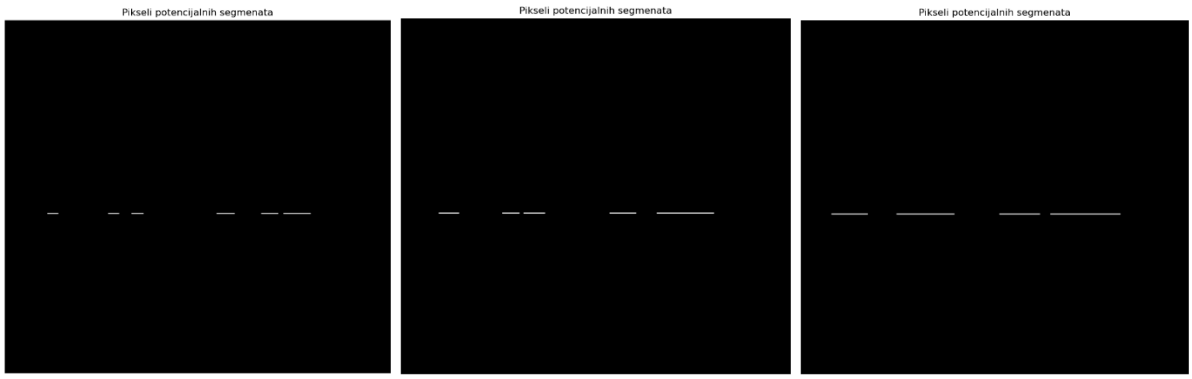
```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 15 , max_gap: 2 , tolerancy: 5
Broj segmenata: 5
Duzine segmenata: [19.0, 16.0, 20.0, 25.0, 55.0]
Koordinate segmenata: [[(190, 37), (190, 56)], [(190, 99), (190, 115)], [(190, 120), (190, 140)], [(190, 204), (190, 229)],
[(190, 250), (190, 305)]]
```

```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 15 , max_gap: 5 , tolerancy: 5
Broj segmenata: 4
Duzine segmenata: [19.0, 41.0, 25.0, 55.0]
Koordinate segmenata: [[(190, 37), (190, 56)], [(190, 99), (190, 140)], [(190, 204), (190, 229)], [(190, 250), (190, 305)]]
```

```
Parametri: theta: 1.5707963267948966 , rho: 190 , min_length: 15 , max_gap: 30 , tolerancy: 5
Broj segmenata: 3
Duzine segmenata: [19.0, 41.0, 101.0]
Koordinate segmenata: [[(190, 37), (190, 56)], [(190, 99), (190, 140)], [(190, 204), (190, 305)]]
```

Ostavili smo vrednosti min_length na 15 da bi detektovali sve delove segmenata radi prikaza. Uočavamo da sa porastom max_gap imamo manji broj segmenata, tj. za veliko max_gap se pojedinačni segmenti računaju spojeno sa nekim u svojoj okolini. Vidimo i da su dužine segmenata sa porastom max_gap veće – više delova je obuhvaćeno jednim segmentom. (Treći primer nema preterano smisla jer je uzeto min_length = 15, a max_gap = 30, ali je to samo radi demonstriracije).

Sa promenom parametra **tolerancy**, dolazi do promene već u detekciji potencijalnih segmenata. Naredne tri slike su za vrednosti tolerancy = 2, 5 i 10 respektivno.



Sa ovih slika uočljivo je da su za malo tolerancy potencijalni segmenti kraći, ali verniji slici ivica. Razlog tome je to što sa povećanjem tolerancy, povećavamo okolinu piksela u kojoj tražimo ivični piksel – za veliko tolerancy je veća okolina u kojoj tražimo pa je i veća verovatnoća da se uhvati neki ivični piksel. Ovo dodatno produžava segmente jer se svakom potencijalnom segmentu u produžetku dodaje broj piksela srazmeran vrednosti tolerancy (jer se u njihovoj okolini nalazi ivični piksel).

Zavisno od primene i kvaliteta dobijenih ivica slike treba pažljivo birati optimalne vrednosti pri obradi.

3. Zadatak:

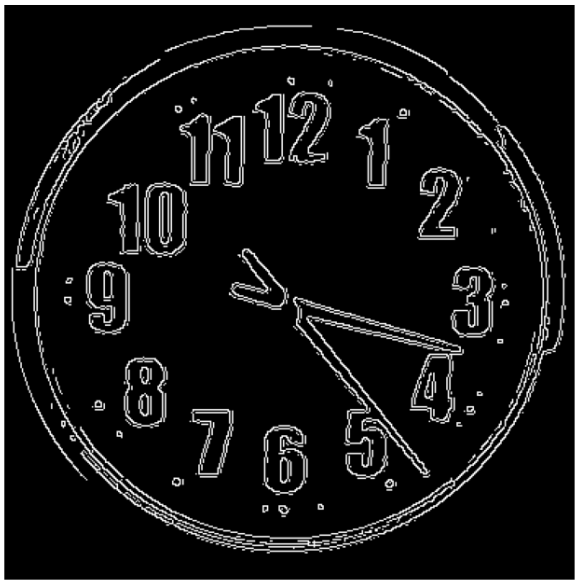
U okviru trećeg zadatka realizovana je funkcija **extract_time**. Generalno, ideja je detektovanje pravaca na kojima se nalazi najveći broj piksela, kako bi se detektovale mala i velika kazaljka, a zatim na osnovu pozicije i ugla kazaljki određivanje vremena. U ovom algoritmu bitna je i obrada treće kazaljke (za sekunde) kojom su obrađene slike iz dostupne baze slika, ali ova obrada nije univerzalna za sve satove. Algoritam je realizovan na sledeći način (u pregledu koraka će biti prikazivani delovi koda i međurezultati radi preglednosti):

1) Predprocesiranje slike – konverzija u grayscale, brisanje crvenih delova slike (kazaljka za sekunde) i primena funkcije canny_edge_detection:

```python
#Konvertujemo sliku u grayscale
img_shape = img_in.shape
M = img_shape[0]
N = img_shape[1]

if len(img_shape) == 2:
    img_gray = img_in
    img_hsv = img_in
elif img_shape[2] == 3:
    img_gray = color.rgb2gray(img_in)
    img_hsv = color.rgb2hsv(img_in)
elif img_shape[2] == 4:
    img_gray = color.rgb2gray(color.rgba2rgb(img_in))
    img_hsv = color.rgb2hsv(color.rgba2rgb(img_in))


#Ako postoje crveni pikseli na slici pretvorimo ih u bele, to je crvena kazaljka
for i in range(0,M):
    for j in range(0,N):
        if img_hsv[i,j,1] > 0.2 and img_hsv[i,j,2] > 0.2:
            if img_hsv[i,j,0] > 0.85 or img_hsv[i,j,0] < 0.15:
                img_gray[i,j] = 1

#Primenjujemo realizovanu funkciju za detektovanje ivica
img_edges = canny_edge_detection(img_gray, 0.8, 0.5, 0.6)
```

Ulazna slika

Ivice slike



2) Primena Hafove transformacije za linije i definisanje neophodnih promenljivih – dobijamo maksimume pravaca na slici ivica:

```python
#Hafova transformacija za linije - dobijamo pikove uglova i rastojanja
tested_angles = np.linspace(-np.pi / 2, np.pi / 2, 100, endpoint=False)
out, angles, d = transform.hough_line(img_edges, tested_angles)
[intensities, peak_angles, peak_distances] = transform.hough_line_peaks(out, angles, d, min_distance=1, min_angle=10, thresh

#Centar sata
i_center = round(M/2)
j_center = round(N/2)

#U dobijenim nizovima peak_angles i peak_distances se nalaze mnoge nezeljene vrednosti (npr. ponovljene slicne vrednosti)
#Pravimo nove nizove peak_angles_real i peak_distances_real da bi ocitali vreme
peak_angles_real = []
peak_distances_real = []

#Pamtimo i gde se nalazi sredina segmenta da bi znali da li je kazaljka u gornjoj ili donjoj polovini sata
segment_middles = []
segment_lengths = []
```

Izlaz Hafove transformacije su prvih 10 pravaca koji su detektovani kao maksimalni. Međutim, nisu sve vrednosti korisne, zbog čega je neophodno dodatno obraditi izlaz Hafove transformacije – to će biti vrednosti peak_angles_real i peak_distances_real (u ovim

nizovima želimo da čuvamo samo pravce kazaljki). Neophodan nam je centar sata – i_center i j_center (on se može odrediti i preko Hafove transformacije za krugove, ali u ovoj bazi slika to nije potrebno i dodatno će usporiti program). Takođe ćemo pamtiti i dužine segmenata koje budemo detektovali da bi ih kasnije sortirali po veličini i tako odredili koja kazaljka je za sate, a koja za minute.

Primer za sliku 1 (12h 15min):

```
Peak uglovi: [-1.57079633 -1.57079633 -1.57079633 -1.57079633 -1.57079633  0.18849556
  1.5393804  -1.57079633  1.5393804  -1.57079633]
Peak rastojanja: [-351. -349. -341. -343. -189.  413.  363. -157.  355. -283.]
```

Naredni proces (korake 3, 4 i 5) izvršavamo za svaki pravac (i) dobijen Hafovom transformacijom:

3) Primena get_line segments i zaustavljanje petlje ako imamo više od 3 elementa u nizu:

```
#Zelimo samo prva tri max elementa da uzmemo - tri kazaljke (sati, minuti i sekunde ako postoje)
if len(peak_angles_real) == 3:
    break

theta = peak_angles[i]
rho = peak_distances[i]

#Primenjujemo realizovanu funkciju da bi detektovali da li postoje kazaljke na pravcima
(segments_num, segments_size, segments_coord) = get_line_segments(img_edges, (theta, rho), 75, 7, 1)
```

U nizu peak_angles_real želimo da na kraju imamo 1, 2 ili 3 elementa (uglove) koji će definisati naše kazaljke (detaljnije o tome šta se dešava kada imamo 1, 2 ili 3 elementa kasnije). Zbog ovoga, želimo da se for petlja zaustavi nakon što niz peak_angles_real dobije 3 vrednosti.
Primenjujemo get_line_segments za pravac koji trenutno obrađujemo – parametri su ručno testirani i određeni.

4) Proveravamo da li je segment dovoljno blizu centra slike:

```
#Proveravamo da li je segment dovoljno dugacak i da li je dovoljno blizu centra sata
far = False
short = False

if segments_num > 1:

    #Ako se desi da postoji vise od jednog segmenta koji je veci od zadate duzine
    max_seg = max(segments_size)
    seg_length = max_seg
    index = segments_size.index(max_seg)
    i_begin = segments_coord[index][0][0]
    j_begin = segments_coord[index][0][1]
    i_end = segments_coord[index][1][0]
    j_end = segments_coord[index][1][1]
    i_middle = (i_begin + i_end)/2


    if abs(i_middle-i_center) > min(M,N)/6:
        far = True

elif segments_num == 1:
    seg_length = segments_size[0]
    i_begin = segments_coord[0][0][0]
    j_begin = segments_coord[0][0][1]
    i_end = segments_coord[0][1][0]
    j_end = segments_coord[0][1][1]
    i_middle = (i_begin + i_end)/2

    if abs(i_middle-i_center) > min(M,N)/6:
        far = True
else:
    #Nemamo nijedan segment - svi segmenti su kratki da bi predstavljali kazaljku
    short = True
```

Prvo proveravamo da li imamo 0, 1 ili više segmenata – ako imamo 0 znači da nijedan segment nije dovoljno dugačak da bi bio detektovan kao kazaljka (postavljamo short na vrednost True), ako imamo 1 proveravamo da li je dovoljno blizu centra kazaljke (u

suprotnom to su neki delovi brojeva, npr. detektovan segment na pravcu kod broja 11 na osnovama brojeva 1 ili duži pravac broja 7), a ako ih imamo više od 1 onda biramo onaj koji je maksimalne dužine i za njega takođe proveravamo da li je dovoljno blizu centra. Ukoliko je (maksimalni) segment daleko od centra, postavljamo far na True.

5) Ubacujemo elemente u nizove peak_angles_real i peak_distances_real:

```
#Prvo ubacimo najveci peak
if len(peak_angles_real) == 0:
    if far == False and short == False:

        peak_angles_real.append(peak_angles[i])
        peak_distances_real.append(peak_distances[i])
        segment_middles.append(i_middle)
        segment_lengths.append(seg_length)

#Zatim ostale koji su razliciti od vec ubacenih (sa malom tolerancijom jer je problematicno sa uglovima oko 0)
else:
    exists = False
    for j in range(len(peak_angles_real)):
        if abs(peak_angles[i]-peak_angles_real[j])<0.15:
            exists = True
    if exists == False and far == False and short == False:
        peak_angles_real.append(peak_angles[i])
        peak_distances_real.append(peak_distances[i])
        segment_middles.append(i_middle)
        segment_lengths.append(seg_length)
```

Ako nemamo ubačene elemente, ubacujemo detektovani segment u nizove peak_angles_real i peak_distances_real ukoliko su far i short vrednosti False. Takođe pamtimo dužinu segmenta koji ubacujemo jer on ne mora biti maksimalan po dužini (iako je po Hafovoj transformaciji po tom pravcu bilo najviše piksela – tome nam i služi funkcija get_line_segments u suštini).
Ako već imamo neke ubačene elemente, ispitujemo da li je segment koji posmatramo dovoljno različit od nekog već postojećeg (da li je razlika u uglovima veća od vrednosti 0.15 – dobijeno testiranjem) jer su se u nizovima peak_angles i peak_distances pojavljivale jako slične vrednosti što je posledica detektovanja više ivica na mestu gde realno postoji samo jedna ivica ili više ivica jedne kazaljke.

6) Nakon obrade svih pravaca dobijenih Hafovom transformacijom, sortiramo uglove i rastojanja po dužinama segmenata:

```
#Zelimo da sortiramo opadajuce po duzinama segmenata
sorting = sorted(zip(segment_lengths, peak_angles_real, peak_distances_real, segment_middles), key=lambda x: x[0], reverse=True)

segment_lengths, peak_angles_real, peak_distances_real, segment_middles = zip(*sorting)
```

Primer za sliku 1 (12h 15min):

```
Peak uglovi (nakon obrade): (-1.5707963267948966, 0.18849555921538785)
Peak rastojanja (nakon obrade): (-351.0, 413.0)
```

7) Na osnovu broja elemenata niza i sadržaja čitamo vreme:

```
#Na osnovu niza peak_angles_real i funkcija hours i minutes dobijamo vreme
if len(peak_angles_real) == 0:
    print('Greska: nema detektovanih kazaljki')
if len(peak_angles_real) == 1:
    #Preklopljene kazaljke sata i minuta
    h = hours(peak_angles_real[0], segment_middles[0], i_center)
    m = minutes(peak_angles_real[0], segment_middles[0], i_center)
if len(peak_angles_real) == 2:
    #Detektovali smo dve kazaljke - sate i minute
    #Veca je za minute
    h = hours(peak_angles_real[1], segment_middles[1], i_center)
    m = minutes(peak_angles_real[0], segment_middles[0], i_center)
if len(peak_angles_real) == 3:
    #Detektovali smo 3 kazaljke, od kojih je jedna sekundara
    #Ovde treba neka provera koja od ovih je sekundara pa onda, za sad pretpostavimo:
    #Za minute je najveca, pa onda za sekunde, pa onda za sate
    h = hours(peak_angles_real[2], segment_middles[2], i_center)
    m = minutes(peak_angles_real[1], segment_middles[1], i_center)
return [h,m]
```

- Ukoliko nemamo detektovane kazaljke, korisniku vraćamo grešku.
- Ukoliko imamo samo jedan detektovan pravac znači da su kazaljke preklopljene.
- Ukoliko imamo dva detektovana pravca imamo kazaljku za minute i sate – maksimalan detektovan pravac su minuti, a manji su sati.
- Ukoliko su detektovana tri pravca imamo i sekunde i to je obično najduža kazaljka, zatim je kraća za minute i najkraća za sate.

***Ova podela je odgovarajuća na ovom skupu slika, ali generalno bi bila neophodna bolja obrada koja se zasniva na debljini kazaljki i slicno. Takođe, detektovana najduža kazaljka zavisi od toga na kojoj poziciji se nalaze kazaljke, tj. kazaljka može biti preklopljena sa nekim brojem i tako ispasti kraća nego što zapravo jeste.

U prethodnom koraku korišćene su funkcije hours i minutes koje na osnovu ugla i sredine segmenta i slike vraćaju sate i minute. U nastavku je realizacija ovih funkcija:

```python
def hours(theta, i_middle, i_center):
    if theta >= -np.pi/2 and theta < -np.pi/3:
        if i_middle < i_center:
            h = 9
        else:
            h = 3
    elif theta >= -np.pi/3 and theta < -np.pi/6:
        if i_middle < i_center:
            h = 10
        else:
            h = 4
    elif theta >= -np.pi/6 and theta < 0:
        if i_middle < i_center:
            h = 11
        else:
            h = 5
    elif theta >= 0 and theta < np.pi/6:
        if i_middle < i_center:
            h = 12
        else:
            h = 6
    elif theta >= np.pi/6 and theta < np.pi/3:
        if i_middle < i_center:
            h = 1
        else:
            h = 7
    elif theta > np.pi/3 and theta < np.pi/2:
        if i_middle < i_center:
            h = 2
        else:
            h = 8
    return h
```

```python
def minutes(theta, i_middle, i_center):
    if theta >= 0:
        if i_middle < i_center:
            m = round(theta/np.pi*2*15)
        else:
            m = round(theta/np.pi*2*15)+30
    elif theta < 0:
        if i_middle < i_center:
            m = round(15/np.pi*2*(theta+np.pi/2)+45)
        else:
            m = round(15/np.pi*2*(theta+np.pi/2)+15)

    return m
```

S obzirom da su dobijeni uglovi u odnosu na vertikalnu osu (u odnosu na pravac 12-6) i kreću se od -pi/2 do pi/2 bitno je uzeti u obzir da li je segment ispod ili iznad centra slike, tj. da li je na primer za pravac pi/6 u pitanju 1 sat ili 7 sati. Takođe, za minute su određene funkcije koje vraćaju minute na osnovu pravca zavisno od “kvadranta” slike u kom se nalazi segment.

Testiranje funkcije extract_time:

U nastavku je izlaz testiranja funkcije na svim primerima baze uz ovaj zadatak (kod koji prati ovo testiranje ostavljen je u jupyter notebooku). Funkcija vraća tačno vreme za svih 12 slika.

```
Slika 1 - vreme: 12 h 15 min
Slika 2 - vreme: 9 h 12 min
Slika 3 - vreme: 1 h 25 min
Slika 4 - vreme: 4 h 35 min
Slika 5 - vreme: 11 h 30 min
Slika 6 - vreme: 12 h 0 min
Slika 7 - vreme: 10 h 7 min
Slika 8 - vreme: 8 h 22 min
Slika 9 - vreme: 3 h 24 min
Slika 10 - vreme: 12 h 48 min
Slika 11 - vreme: 10 h 8 min
Slika 12 - vreme: 8 h 16 min
```